This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, docs.sun.com, AnswerBook, AnswerBook2, and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun’s licensees who implement OPEN LOOK GUIs and otherwise comply with Sun’s written license agreements.

Federal Acquisitions: Commercial Software—Government Users Subject to Standard License Terms and Conditions.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.
Contents

Preface 13

Basic Library Functions 19
a64l(3C) 20
abort(3C) 21
abs(3C) 22
addsev(3C) 23
addseverity(3C) 24
assert(3C) 26
atexit(3C) 27
attropen(3C) 28
basename(3C) 29
bsdmalloc(3MALLOC) 30
bsd_signal(3C) 32
bsearch(3C) 33
bstring(3C) 35
btowc(3C) 36
catgets(3C) 37
catopen(3C) 38
cfgetispeed(3C) 41
cfsetispeed(3C) 42
clock(3C) 43
closedir(3C) 44
closefrom(3C) 45
confstr(3C) 47
crypt(3C) 51
Contents 5
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>isatty(3C)</td>
<td>286</td>
</tr>
<tr>
<td>isnan(3C)</td>
<td>287</td>
</tr>
<tr>
<td>iswalpha(3C)</td>
<td>289</td>
</tr>
<tr>
<td>iswctype(3C)</td>
<td>291</td>
</tr>
<tr>
<td>killpg(3C)</td>
<td>293</td>
</tr>
<tr>
<td>lckpwdf(3C)</td>
<td>294</td>
</tr>
<tr>
<td>ldexp(3C)</td>
<td>295</td>
</tr>
<tr>
<td>lfmt(3C)</td>
<td>296</td>
</tr>
<tr>
<td>localeconv(3C)</td>
<td>300</td>
</tr>
<tr>
<td>lockf(3C)</td>
<td>304</td>
</tr>
<tr>
<td>_longjmp(3C)</td>
<td>307</td>
</tr>
<tr>
<td>lsearch(3C)</td>
<td>308</td>
</tr>
<tr>
<td>madvise(3C)</td>
<td>310</td>
</tr>
<tr>
<td>makecontext(3C)</td>
<td>313</td>
</tr>
<tr>
<td>makedev(3C)</td>
<td>316</td>
</tr>
<tr>
<td>malloc(3C)</td>
<td>317</td>
</tr>
<tr>
<td>malloc(3MALLOCC)</td>
<td>320</td>
</tr>
<tr>
<td>mapmalloc(3MALLOCC)</td>
<td>323</td>
</tr>
<tr>
<td>mblen(3C)</td>
<td>325</td>
</tr>
<tr>
<td>mbrlen(3C)</td>
<td>326</td>
</tr>
<tr>
<td>mbtowc(3C)</td>
<td>328</td>
</tr>
<tr>
<td>mbsinit(3C)</td>
<td>330</td>
</tr>
<tr>
<td>mbstowcs(3C)</td>
<td>331</td>
</tr>
<tr>
<td>mbtowcs(3C)</td>
<td>333</td>
</tr>
<tr>
<td>mbtowc(3C)</td>
<td>334</td>
</tr>
<tr>
<td>mctl(3UCB)</td>
<td>335</td>
</tr>
<tr>
<td>memory(3C)</td>
<td>337</td>
</tr>
<tr>
<td>mkfifo(3C)</td>
<td>339</td>
</tr>
<tr>
<td>mkstemp(3C)</td>
<td>341</td>
</tr>
<tr>
<td>mktime(3C)</td>
<td>342</td>
</tr>
<tr>
<td>mlock(3C)</td>
<td>343</td>
</tr>
<tr>
<td>mlockall(3C)</td>
<td>348</td>
</tr>
<tr>
<td>modf(3C)</td>
<td>350</td>
</tr>
<tr>
<td>monitor(3C)</td>
<td>351</td>
</tr>
<tr>
<td>msync(3C)</td>
<td>353</td>
</tr>
<tr>
<td>mtmalloc(3MALLOCC)</td>
<td>355</td>
</tr>
<tr>
<td>ndbm(3C)</td>
<td>358</td>
</tr>
</tbody>
</table>
nice(3UCB) 362
nlist(3UCB) 363
nl_langinfo(3C) 364
offsetof(3C) 365
opendir(3C) 366
perror(3C) 368
printf(3C) 369
plock(3C) 372
popen(3C) 373
printf(3UCB) 385
printf(3UCB) 385
pset_getloadavg(3C) 389
psignal(3C) 390
psignal(3UCB) 391
pthread_atfork(3C) 392
ptsname(3C) 394
putenv(3C) 395
putpwent(3C) 396
puts(3C) 397
putspent(3C) 398
putw(3C) 399
qsort(3C) 400
raise(3C) 402
rand(3C) 403
rand(3UCB) 404
random(3C) 405
rctlblk_set_value(3C) 408
rctl_walk(3C) 412
readdir(3C) 414
readdir(3UCB) 418
realpath(3C) 420
reboot(3C) 422
re_comp(3C) 423
regcmp(3C) 424
regcomp(3C) 426
remove(3C) 432
rewind(3C) 433
rewinddir(3C) 434
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>scandir(3UCB)</td>
<td>435</td>
</tr>
<tr>
<td>scanf(3C)</td>
<td>436</td>
</tr>
<tr>
<td>seekdir(3C)</td>
<td>443</td>
</tr>
<tr>
<td>select(3C)</td>
<td>444</td>
</tr>
<tr>
<td>setbuf(3C)</td>
<td>448</td>
</tr>
<tr>
<td>setbuffer(3C)</td>
<td>450</td>
</tr>
<tr>
<td>setcat(3C)</td>
<td>451</td>
</tr>
<tr>
<td>setjmp(3C)</td>
<td>452</td>
</tr>
<tr>
<td>setjmp(3UCB)</td>
<td>455</td>
</tr>
<tr>
<td>setkey(3C)</td>
<td>458</td>
</tr>
<tr>
<td>setlabel(3C)</td>
<td>459</td>
</tr>
<tr>
<td>setlocale(3C)</td>
<td>460</td>
</tr>
<tr>
<td>sigblock(3UCB)</td>
<td>463</td>
</tr>
<tr>
<td>sigfpe(3C)</td>
<td>464</td>
</tr>
<tr>
<td>siginterrupt(3UCB)</td>
<td>466</td>
</tr>
<tr>
<td>signal(3C)</td>
<td>467</td>
</tr>
<tr>
<td>signal(3UCB)</td>
<td>469</td>
</tr>
<tr>
<td>sigsetops(3C)</td>
<td>471</td>
</tr>
<tr>
<td>sigstack(3C)</td>
<td>473</td>
</tr>
<tr>
<td>sigstack(3UCB)</td>
<td>475</td>
</tr>
<tr>
<td>sigvec(3UCB)</td>
<td>476</td>
</tr>
<tr>
<td>sleep(3C)</td>
<td>481</td>
</tr>
<tr>
<td>sleep(3UCB)</td>
<td>482</td>
</tr>
<tr>
<td>ssignal(3C)</td>
<td>483</td>
</tr>
<tr>
<td>stack_getbounds(3C)</td>
<td>484</td>
</tr>
<tr>
<td>_stack_grow(3C)</td>
<td>485</td>
</tr>
<tr>
<td>stack_inbounds(3C)</td>
<td>486</td>
</tr>
<tr>
<td>stack_setbounds(3C)</td>
<td>487</td>
</tr>
<tr>
<td>stackViolation(3C)</td>
<td>488</td>
</tr>
<tr>
<td>stdio(3C)</td>
<td>490</td>
</tr>
<tr>
<td>str2sig(3C)</td>
<td>494</td>
</tr>
<tr>
<td>strcoll(3C)</td>
<td>495</td>
</tr>
<tr>
<td>strerror(3C)</td>
<td>496</td>
</tr>
<tr>
<td>strfmon(3C)</td>
<td>497</td>
</tr>
<tr>
<td>strftime(3C)</td>
<td>501</td>
</tr>
<tr>
<td>string(3C)</td>
<td>506</td>
</tr>
<tr>
<td>string_to_decimal(3C)</td>
<td>510</td>
</tr>
<tr>
<td>strptime(3C)</td>
<td>513</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>strsignal(3C)</td>
<td>518</td>
</tr>
<tr>
<td>strtod(3C)</td>
<td>519</td>
</tr>
<tr>
<td>strtol(3C)</td>
<td>522</td>
</tr>
<tr>
<td>strtoull(3C)</td>
<td>525</td>
</tr>
<tr>
<td>strtows(3C)</td>
<td>527</td>
</tr>
<tr>
<td>strxfrm(3C)</td>
<td>528</td>
</tr>
<tr>
<td>swab(3C)</td>
<td>530</td>
</tr>
<tr>
<td>sync_instruction_memory(3C)</td>
<td>531</td>
</tr>
<tr>
<td>syscall(3UCB)</td>
<td>532</td>
</tr>
<tr>
<td>sysconf(3C)</td>
<td>533</td>
</tr>
<tr>
<td>syslog(3C)</td>
<td>540</td>
</tr>
<tr>
<td>system(3C)</td>
<td>544</td>
</tr>
<tr>
<td>tcdrain(3C)</td>
<td>545</td>
</tr>
<tr>
<td>tcflow(3C)</td>
<td>546</td>
</tr>
<tr>
<td>tcflush(3C)</td>
<td>547</td>
</tr>
<tr>
<td>tcgetattr(3C)</td>
<td>548</td>
</tr>
<tr>
<td>tcgetpgrp(3C)</td>
<td>549</td>
</tr>
<tr>
<td>tcgetsid(3C)</td>
<td>550</td>
</tr>
<tr>
<td>tcsendbreak(3C)</td>
<td>551</td>
</tr>
<tr>
<td>tcsetattr(3C)</td>
<td>552</td>
</tr>
<tr>
<td>tcsetpgrp(3C)</td>
<td>554</td>
</tr>
<tr>
<td>tell(3C)</td>
<td>555</td>
</tr>
<tr>
<td>telldir(3C)</td>
<td>556</td>
</tr>
<tr>
<td>termios(3C)</td>
<td>557</td>
</tr>
<tr>
<td>times(3UCB)</td>
<td>558</td>
</tr>
<tr>
<td>tmpfile(3C)</td>
<td>559</td>
</tr>
<tr>
<td>tmpnam(3C)</td>
<td>560</td>
</tr>
<tr>
<td>toascii(3C)</td>
<td>562</td>
</tr>
<tr>
<td>_tolower(3C)</td>
<td>563</td>
</tr>
<tr>
<td>tolower(3C)</td>
<td>564</td>
</tr>
<tr>
<td>_toupper(3C)</td>
<td>565</td>
</tr>
<tr>
<td>toupper(3C)</td>
<td>566</td>
</tr>
<tr>
<td>towctrans(3C)</td>
<td>567</td>
</tr>
<tr>
<td>towlower(3C)</td>
<td>568</td>
</tr>
<tr>
<td>towupper(3C)</td>
<td>569</td>
</tr>
<tr>
<td>truncate(3C)</td>
<td>570</td>
</tr>
<tr>
<td>tsearch(3C)</td>
<td>573</td>
</tr>
<tr>
<td>ttyname(3C)</td>
<td>576</td>
</tr>
</tbody>
</table>
ttyslot(3C)  578
ualarm(3C)  579
umem_alloc(3MALLO)  580
umem_cache_create(3MALLO)  585
umem_debug(3MALLO)  593
ungetc(3C)  596
ungetwc(3C)  597
unlockpt(3C)  598
usleep(3C)  599
vfwprintf(3C)  600
vlfmt(3C)  601
vprintf(3C)  603
vprintf(3C)  605
vsyslog(3C)  607
wait3(3C)  608
wait(3UCB)  611
walkcontext(3C)  615
watchmalloc(3MALLO)  617
wcrtomb(3C)  620
wcscoll(3C)  622
wcsftime(3C)  623
wcsrtombs(3C)  624
wcsstr(3C)  626
wcstod(3C)  627
wcstol(3C)  629
wcstombs(3C)  631
wcstoul(3C)  632
wcstring(3C)  634
wcswidth(3C)  639
wcsxfrm(3C)  640
wctob(3C)  642
wctomb(3C)  643
wctype(3C)  644
wctype(3C)  645
wcwidth(3C)  646
wmemchr(3C)  647
wmemcmp(3C)  648
wmemcpy(3C)  649
wmemmove(3C) 650
wmemset(3C) 651
wordexp(3C) 652
wsprintf(3C) 656
wsscanf(3C) 657
wstring(3C) 658

Index 659
Preface

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

Overview

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

- Section 1 describes, in alphabetical order, commands available with the operating system.
- Section 1M describes, in alphabetical order, commands that are used chiefly for system maintenance and administration purposes.
- Section 2 describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible returned value.
- Section 3 describes functions found in various libraries, other than those functions that directly invoke UNIX system primitives, which are described in Section 2.
- Section 4 outlines the formats of various files. The C structure declarations for the file formats are given where applicable.
- Section 5 contains miscellaneous documentation such as character-set tables.
- Section 6 contains available games and demos.
- Section 7 describes various special files that refer to specific hardware peripherals and device drivers. STREAMS software drivers, modules and the STREAMS-generic set of system calls are also described.
Section 9 provides reference information needed to write device drivers in the kernel environment. It describes two device driver interface specifications: the Device Driver Interface (DDI) and the Driver/Kernel Interface (DKI).

Section 9E describes the DDI/DKI, DDI-only, and DKI-only entry-point routines a developer can include in a device driver.

Section 9F describes the kernel functions available for use by device drivers.

Section 9S describes the data structures used by drivers to share information between the driver and the kernel.

Below is a generic format for man pages. The man pages of each manual section generally follow this order, but include only needed headings. For example, if there are no bugs to report, there is no BUGS section. See the intro pages for more information and detail about each section, and man(1) for more information about man pages in general.

NAME

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

SYNOPSIS

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

The following special characters are used in this section:

[ ] Brackets. The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.

... Ellipses. Several values can be provided for the previous argument, or the previous argument can be specified multiple times, for example, "filename ...".

| Separator. Only one of the arguments separated by this character can be specified at a time.

{ } Braces. The options and/or arguments enclosed within braces are interdependent, such that everything enclosed must be treated as a unit.
PROTOCOL
This section occurs only in subsection 3R to indicate the protocol description file.

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

IOCTL
This section appears on pages in Section 7 only. Only the device class that supplies appropriate parameters to the ioctl(2) system call is called ioctl and generates its own heading. ioctl calls for a specific device are listed alphabetically (on the man page for that specific device). ioctl calls are used for a particular class of devices all of which have an io ending, such as mtcio(7I).

OPTIONS
This section lists the command options with a concise summary of what each option does. The options are listed literally and in the order they appear in the SYNOPSIS section. Possible arguments to options are discussed under the option, and where appropriate, default values are supplied.

OPERANDS
This section lists the command operands and describes how they affect the actions of the command.

OUTPUT
This section describes the output – standard output, standard error, or output files – generated by the command.

RETURN VALUES
If the man page documents functions that return values, this section lists these values and describes the conditions under which they are returned. If a function can return only constant values, such as 0 or -1, these values are listed in tagged paragraphs. Otherwise, a single paragraph describes the return values of each function. Functions declared void do not return values, so they are not discussed in RETURN VALUES.

ERRORS
On failure, most functions place an error code in the global variable errno indicating why they failed. This section lists alphabetically all error codes a function can generate and describes the conditions that cause each error. When more than
one condition can cause the same error, each condition is described in a separate paragraph under the error code.

**USAGE**

This section lists special rules, features, and commands that require in-depth explanations. The subsections listed here are used to explain built-in functionality:

- Commands
- Modifiers
- Variables
- Expressions
- Input Grammar

**EXAMPLES**

This section provides examples of usage or of how to use a command or function. Wherever possible a complete example including command-line entry and machine response is shown. Whenever an example is given, the prompt is shown as `example%`, or if the user must be superuser, `example#`. Examples are followed by explanations, variable substitution rules, or returned values. Most examples illustrate concepts from the SYNOPSIS, DESCRIPTION, OPTIONS, and USAGE sections.

**ENVIRONMENT VARIABLES**

This section lists any environment variables that the command or function affects, followed by a brief description of the effect.

**EXIT STATUS**

This section lists the values the command returns to the calling program or shell and the conditions that cause these values to be returned. Usually, zero is returned for successful completion, and values other than zero for various error conditions.

**FILES**

This section lists all file names referred to by the man page, files of interest, and files created or required by commands. Each is followed by a descriptive summary or explanation.

**ATTRIBUTES**

This section lists characteristics of commands, utilities, and device drivers by defining the attribute type and its corresponding value. See attributes(5) for more information.

**SEE ALSO**

This section lists references to other man pages, in-house documentation, and outside publications.
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIAGNOSTICS</td>
<td>This section lists diagnostic messages with a brief explanation of the condition causing the error.</td>
</tr>
<tr>
<td>WARNINGS</td>
<td>This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.</td>
</tr>
<tr>
<td>NOTES</td>
<td>This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.</td>
</tr>
<tr>
<td>BUGS</td>
<td>This section describes known bugs and, wherever possible, suggests workarounds.</td>
</tr>
</tbody>
</table>
Basic Library Functions
a64l(3C)

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

SYNOPSIS  #include <stdlib.h>

    long a64l(const char *s);
    char *l64a(long l);

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

The characters used to represent “digits” are as follows:

<table>
<thead>
<tr>
<th>Character</th>
<th>Digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>/</td>
<td>1</td>
</tr>
<tr>
<td>0-9</td>
<td>2-11</td>
</tr>
<tr>
<td>A-Z</td>
<td>12-37</td>
</tr>
<tr>
<td>a-z</td>
<td>38-63</td>
</tr>
</tbody>
</table>

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

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

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

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  attributes(5)
NAME  abort – terminate the process abnormally

SYNOPSIS  #include <stdlib.h>

    void abort(void);

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

    The status made available to wait(2) or waitpid(2) by abort will be that of a
    process terminated by the SIGABRT signal. abort will override blocking or ignoring
    the SIGABRT signal.

RETURN VALUES  The abort() function does not return.

ERRORS  No errors are defined.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  exit(2), getrlimit(2), kill(2), wait(2), waitpid(2), fclose(3C), raise(3C),
    signal(3C), attributes(5)
abs(3C)

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

SYNOPSIS  
#include <stdlib.h>

int abs(int val);
long labs(long lval);
long long llabs(long long llval);

DESCRIPTION  
The abs() function returns the absolute value of its int operand.
The labs() function returns the absolute value of its long operand.
The llabs() function returns the absolute value of its long long operand.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  attributes(5)
addsev – define additional severities

SYNOPSIS
#include <pfmt.h>

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

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

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

RETURN VALUES
Upon successful completion, addsev() returns 0. Otherwise it returns −1.

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

EXAMPLES
EXAMPLE 1 Example of addsev() function.

The following example
#define Panic 5
setLabel("APPL");
setcat("my_appl");
addsev(Panic, gettext("*:26", "PANIC"));
/*...*/
lfmt(stderr, MM_SOFT|MM_APPL|PANIC, "*:12:Cannot locate database\n");

will display the message to stderr and forward to the logging service
APPL: PANIC: Cannot locate database

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-safe</td>
</tr>
</tbody>
</table>

SEE ALSO
getxt(3C), lfmt(3C), pfmt(3C), attributes(5)
addseverity(3C)

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

SYNOPSIS
#include <fmtmsg.h>

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

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

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

MM_HALT    Indicates that the application has encountered a severe fault and is
           halting. Produces the print string HALT.

MM_ERROR   Indicates that the application has detected a fault. Produces the
           print string ERROR.

MM_WARNING Indicates a condition that is out of the ordinary, that might be a
           problem, and should be watched. Produces the print string
           WARNING.

MM_INFO    Provides information about a condition that is not in error.
           Produces the print string INFO.

MM_NOSEV   Indicates that no severity level is supplied for the message.

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

RETURN VALUES
Upon successful completion, addseverity() returns MM_OK. Otherwise it returns
MM_NOTOK.

EXAMPLES
EXAMPLE 1 Example of addseverity() function.

When the function call

addseverity(7,"ALERT")

is followed by the call

fmtmsg(MM_PRINT, "UX:cat", 7, "invalid syntax", "refer to manual",
"UX:cat:001")
EXAMPLE 1  Example of addseverity() function.  (Continued)

the resulting output is
UX:cat: ALERT: invalid syntax
TO FIX: refer to manual  UX:cat:001

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  fmtmsg(1), fmtmsg(3C), gettext(3C), printf(3C), attributes(5)
NAME
assert – verify program assertion

SYNOPSIS
#include <assert.h>

void assert(int expression);

DESCRIPTION
The assert() macro inserts diagnostics into applications. When executed, if
expression is FALSE (zero), assert() prints the error message
Assertion failed: expression, file xyz, line nnn
on the standard error output and aborts. In the error message, xyz is the name of the
source file and nnn the source line number of the assert() statement. These are
respectively the values of the preprocessor macros __FILE__ and __LINE__.

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

Compiling with the preprocessor option -DNDEBUG (see cc(1B)), or with the
preprocessor control statement #define NDEBUG ahead of the #include
<assert.h> statement, will stop assertions from being compiled into the program.

If the application is linked with -lint1, messages printed from this function are in
the native language specified by the LC_MESSAGES locale category; see
setlocale(3C).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
cc(1B), abort(3C), gettext(3C), setlocale(3C), attributes(5)
NAME | atexit – register a function to run at process termination or object unloading

SYNOPSIS | #include <stdlib.h>

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

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

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

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

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

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

RETURN VALUES | Upon successful completion, the atexit() function returns 0. Otherwise, it returns a non-zero value.

ERRORS | The atexit() function may fail if:

    ENOMEM Insufficient storage space is available.

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

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

ATTRIBUTES | See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO | exec(2), dlclose(3DL), exit(3C), sysconf(3C), attributes(5)
attropen(3C)

NAME
attropen – open a file

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

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

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

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

RETURN VALUES
Refer to open(2).

ERRORS
Refer to open(2).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
open(2), attributes(5), fsattr(5)
### NAME
basename – return the last element of a path name

### SYNOPSIS
#include <libgen.h>

char *basename(char *path);

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

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

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

### RETURN VALUES
The basename() function returns a pointer to the final component of path.

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

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

### EXAMPLES
**EXAMPLE 1** Examples for Input String and Output String

<table>
<thead>
<tr>
<th>Input String</th>
<th>Output String</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;/usr/lib&quot;</td>
<td>&quot;lib&quot;</td>
</tr>
<tr>
<td>&quot;/usr/&quot;</td>
<td>&quot;usr&quot;</td>
</tr>
<tr>
<td>&quot;/&quot;</td>
<td>&quot;/&quot;</td>
</tr>
</tbody>
</table>

### ATTRIBUTES
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

### SEE ALSO
basename(1), dirname(3C), attributes(5)
NAME  bsdmalloc – memory allocator

SYNOPSIS  cc [ flag ... ] file ... -lsdmalloc [ library ... ]

char *malloc(size);
unsigned size;
int free( ptr);
char *ptr;
char *realloc( ptr, size);
char *ptr;
unsigned size;

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

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

The free() function releases a previously allocated block. Its argument is a pointer to
a block previously allocated by malloc() or realloc().

The realloc() function changes the size of the block referenced 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 unable to honor a reallocation request,
realloc() leaves its first argument unaltered. For backwards compatibility,
realloc() accepts a pointer to a block freed since the most recent call to malloc() or
realloc().

RETURN VALUES  The malloc() and realloc() functions return a null pointer if there is not enough
available memory. When realloc() returns NULL, the block pointed to by ptr is left
intact.

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

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

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

SEE ALSO  brk(2), malloc(3C), malloc(3MALLOC), mapmalloc(3MALLOC)

WARNINGS  Use of libbsdmalloc renders an application non-SCD compliant.
The libbsdmalloc routines are incompatible with the memory allocation routines in the standard C-library (libc): malloc(3C), alloca(3C), calloc(3C), free(3C), memalign(3C), realloc(3C), and valloc(3C).

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

The malloc() and realloc() functions return a non-null pointer if size is 0. These pointers should not be dereferenced.

Always cast the value returned by malloc() and realloc().

Comparative features of bsdmalloc, malloc(3MALLOCC), and malloc(3C):

- The bsdmalloc() routines afford better performance but are space-inefficient.
- The malloc(3MALLOCC) routines are space-efficient but have slower performance.
- The standard, fully SCD-compliant malloc(3C) routines are a trade-off between performance and space-efficiency.

The free() function does not set errno.
The `bsd_signal` function provides a partially compatible interface for programs written to historical system interfaces (see `USAGE` below).

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

```c
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:

```c
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.

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

This function is a direct replacement for the BSD `signal(3UCB)` function for simple applications that are installing a single-argument signal handler function. If a BSD signal handler function is being installed that expects more than one argument, the application has to be modified to use `sigaction(2)`. The `bsd_signal()` function differs from `signal(3UCB)` 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(3UCB)`.

**SEE ALSO**

`sigaction(2), sigaddset(3C), sigemptyset(3C), signal(3UCB)`
### NAME
bsearch – binary search a sorted table

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

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

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

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

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

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

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

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

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

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

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

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

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct node { /* these are stored in the table */
    char *string;
    int length;
};

static struct node table[] = { /* table to be searched */
```
EXAMPLE 1 Examples for searching a table containing pointers to nodes.

Continued

```
{ "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
",
                node_ptr->string, node_ptr->length);
        } else {
            (void) printf("not found: %20s
", node.string);
        }
    }
    return(0);
}

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

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

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO hsearch(3C), lsearch(3C), qsort(3C), tsearch(3C), attributes(5)
bstring(3C)

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 bcopy(), bcmp(), and bzero() functions operate on variable length strings of bytes. They do not check for null bytes as do the functions described on the string(3C) manual page.

The bcopy() function copies n bytes from string s1 to the string s2. Overlapping strings are handled correctly.

The bcmp() function compares byte string s1 against byte string s2, returning 0 if they are identical, 1 otherwise. Both strings are assumed to be n bytes long. The bcmp() function always returns 0 when n is 0.

The bzero() function places n null bytes in the string s.

WARNINGS  The bcmp() and bcopy() routines take parameters backwards from strcmp() and strcpy(), respectively. See string(3C).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  memory(3C), string(3C), attributes(5)
btowc(3C)

NAME
btowc – single-byte to wide-character conversion

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

wint_t btowc(int c);

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

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

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

ERRORS
No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
</tbody>
</table>

SEE ALSO
setlocale(3C), wctob(3C), attributes(5), environ(5)

NOTES
The btowc() function can be used safely in multithreaded applications, as long as setlocale(3C) is not being called to change the locale.
catgets – read a program message

#include <nl_types.h>

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

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

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

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_id and msg_id is not in the message catalog.

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

ATRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
gencat(1), catclose(3C), catopen(3C), gettext(3C), setlocale(3C), attributes(5)

*International Language Environments Guide*
NAME
    catopen, catclose – open/close a message catalog

SYNOPSIS
    #include <nl_types.h>

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

    int catclose(nl_catd catd);

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

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

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

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

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

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

The full set of metacharacters is:
%N    The value of the name parameter passed to catopen().
%L    The value of LANG or LC_MESSAGES.
%\    The value of the language element of LANG or LC_MESSAGES.
%t    The value of the territory element of LANG or LC_MESSAGES.
%c    The value of the codeset element of LANG or LC_MESSAGES.
%\     A single %.
The LANG environment variable provides the ability to specify the user's requirements for native languages, local customs and character set, as an ASCII string in the form

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

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

```
LANG=De_A.88591
```

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

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

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

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

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

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

**RETURN VALUES**

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

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

**ERRORS**

The `catopen()` function may fail if:

- **EACCES** Search permission is denied for the component of the path prefix of the message catalogue or read permission is denied for the message catalogue.
- **EMFILE** There are OPEN_MAX file descriptors currently open in the calling process.
- **ENAMETOOLONG** The length of the pathname of the message catalogue exceeds PATH_MAX, or a pathname component is longer than NAME_MAX.
- **ENAMETOOLONG** Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
catopen(3C)

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.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO gencat(1), catgets(3C), gettext(3C), nl_types(3HEAD), setlocale(3C), attributes(5), environ(5)
NAME

cfgetispeed, cfgetospeed – get input and output baud rate

SYNOPSIS

#include <termios.h>

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

DESCRIPTION

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

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

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

RETURN VALUES

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

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

ERRORS

No errors are defined.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe, and Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

cfgetospeed(3C), tcgetattr(3C), attributes(5), termio(7I)
### NAME

cfsetispeed, cfsetospeed – set input and output baud rate

### SYNOPSIS

```c
#include <termios.h>

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

### DESCRIPTION

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

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

### RETURN VALUES

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

### ERRORS

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

- **EINVAL** The `speed` value is not a valid baud rate.
- **EINVAL** The value of `speed` is outside the range of possible speed values as specified in `<termios.h>`.

### ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe, and Async-Signal-Safe</td>
</tr>
</tbody>
</table>

### SEE ALSO

`cfgetispeed(3C)`, `tcsetattr(3C)`, `attributes(5)`, `termio(7I)`
clock — report CPU time used

#include <time.h>

clock_t clock(void);

DESCRIPTION
The clock() function returns the amount of CPU time (in microseconds) used since the first call to clock() in the calling process. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed the wait(2) function, the pclose(3) function, or the system(3) function.

RETURN VALUES
Dividing the value returned by clock() by the constant _CLOCKS_PER_SEC, defined in the <time.h> header, will give the time in seconds. If the process time used is not available or cannot be represented, clock returns the value (clock_t) -1.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
times(2), wait(2), pclose(3), system(3), attributes(5)
closedir(3C)

NAME  closedir – close a directory stream

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

int closedir(DIR *dirp);

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

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

ERRORS  The closedir() function may fail if:

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

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  opendir(3C), attributes(5)
#include <stdlib.h>

void closefrom(int lowfd);

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

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

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

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

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

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

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

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

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

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

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

The `fdwalk()` function can then be used to count the number of open files in the process.
closefrom(3C)

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

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

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

FILES
/proc/self/fd directory (list of open files)

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Unsafe</td>
</tr>
</tbody>
</table>

SEE ALSO
close(2), getrlimit(2), proc(4), attributes(5)
confstr – get configurable variables

#include <unistd.h>

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

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

The name argument represents the system variable to be queried.

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

If len is 0, and buf is a null pointer, then confstr() still returns the integer value as defined below, but does not return the string. If len is 0 but buf is not a null pointer, the result is unspecified.

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

_CS_LFS64_CFLAGS
   If _LFS64_LARGEFILE is defined in <unistd.h>, this value is the set of initial options to be given to the cc and c89 utilities to build an application using the Large File Summit transitional compilation environment (see lfstream64(5)).

_CS_LFS64_LDFLAGS
   If _LFS64_LARGEFILE is defined in <unistd.h>, this value is the set of final options to be given to the cc and c89 utilities to build an application using the Large File Summit transitional compilation environment (see lfstream64(5)).

_CS_LFS64_LIBS
   If _LFS64_LARGEFILE is defined in <unistd.h>, this value is the set of libraries to be given to the cc and c89 utilities to build an application using the Large File Summit transitional compilation environment (see lfstream64(5)).

_CS_LFS64_LINTFLAGS
   If _LFS64_LARGEFILE is defined in <unistd.h>, this value is the set of options to be given to the lint utility to check application source using the Large File Summit transitional compilation environment (see lfstream64(5)).

_CS_LFS_CFLAGS
   If _LFS_LARGEFILE is defined in <unistd.h>, this value is the set of initial options to be given to the cc and c89 utilities to build an application using the Large File Summit large file compilation environment for 32-bit applications (see lfstream(5)).
_CS_LFS_LDFLAGS
If _LFS_LARGEFILE is defined in <unistd.h>, this value is the set of final options to be given to the cc and c89 utilities to build an application using the Large File Summit large file compilation environment for 32-bit applications (see lfcompile(5)).

_CS_LFS_LIBS
If _LFS_LARGEFILE is defined in <unistd.h>, this value is the set of libraries to be given to the cc and c89 utilities to build an application using the Large File Summit large file compilation environment for 32-bit applications (see lfcompile(5)).

_CS_LFS_LINTFLAGS
If _LFS_LARGEFILE is defined in <unistd.h>, this value is the set of options to be given to the lint utility to check application source using the Large File Summit large file compilation environment for 32-bit applications (see lfcompile(5)).

_CS_PATH
If the ISO POSIX.2 standard is supported, this is the value for the PATH environment variable that finds all standard utilities. Otherwise the meaning of this value is unspecified.

_CS_XBS5_ILP32_OFF32_CFLAGS
If sysconf(_SC_XBS5_ILP32_OFF32) returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the cc and c89 utilities to build an application using a programming model with 32-bit int, long, pointer, and off_t types.

_CS_XBS5_ILP32_OFF32_LDFLAGS
If sysconf(_SC_XBS5_ILP32_OFF32) returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the cc and c89 utilities to build an application using a programming model with 32-bit int, long, pointer, and off_t types.

_CS_XBS5_ILP32_OFF32_LIBS
If sysconf(_SC_XBS5_ILP32_OFF32) returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the cc and c89 utilities to build an application using a programming model with 32-bit int, long, pointer, and off_t types.

_CS_XBS5_ILP32_OFF32_LINTFLAGS
If sysconf(_SC_XBS5_ILP32_OFF32) returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the lint utility to check application source using a programming model with 32-bit int, long, pointer, and off_t types.

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

If `sysconf(_SC_XBS5_ILP32_OFFBIG_LIBS)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the `cc` and `c89` utilities to build an application using a programming model with 32-bit `int`, `long`, and `pointer` types, and an `off_t` type using at least 64 bits.

If `sysconf(_SC_XBS5_ILP32_OFFBIG_LINTFLAGS)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the `lint` utility to check an application using a programming model with 32-bit `int`, `long`, and `pointer` types, and an `off_t` type using at least 64 bits.

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

If `sysconf(_SC_XBS5_LP64_OFF64_CFLAGS)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the `cc` and `c89` utilities to build an application using a programming model with 64-bit `int`, `long`, `pointer`, and `off_t` types.

If `sysconf(_SC_XBS5_LP64_OFF64_LDFLAGS)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the `cc` and `c89` utilities to build an application using a programming model with 64-bit `int`, `long`, `pointer`, and `off_t` types.

If `sysconf(_SC_XBS5_LP64_OFF64_LIBS)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the `cc` and `c89` utilities to build an application using a programming model with 64-bit `int`, `long`, `pointer`, and `off_t` types.

If `sysconf(_SC_XBS5_LP64_OFF64_LINTFLAGS)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the `lint` utility to check application source using a programming model with 64-bit `int`, `long`, `pointer`, and `off_t` types.

If `sysconf(_SC_XBS5_LPBIG_OFFBIG_CFLAGS)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of initial options to be given to the `cc` and `c89` utilities to build an application using a programming model with an `int` type using at least 32 bits and `long`, `pointer`, and `off_t` types using at least 64 bits.
CS_XBS5_LPBIG_OFFBIG_LDFLAGS
If `sysconf(_SC_XBS5_LPBIG_OFFBIG)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of final options to be given to the `cc` and `c89` utilities to build an application using a programming model with an `int` type using at least 32 bits and `long`, `pointer`, and `off_t` types using at least 64 bits.

CS_XBS5_LPBIG_OFFBIG_LIBS
If `sysconf(_SC_XBS5_LPBIG_OFFBIG)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of libraries to be given to the `cc` and `c89` utilities to build an application using a programming model with an `int` type using at least 32 bits and `long`, `pointer`, and `off_t` types using at least 64 bits.

CS_XBS5_LPBIG_OFFBIG_LINTFLAGS
If `sysconf(_SC_XBS5_LPBIG_OFFBIG)` returns −1 the meaning of this value is unspecified. Otherwise, this value is the set of options to be given to the `lint` utility to check application source using a programming model with an `int` type using at least 32 bits and `long`, `pointer`, and `off_t` types using at least 64 bits.

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

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

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

**ERRORS**
The `confstr()` function will fail if:

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Mt-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**
`pathconf(2)`, `sysconf(3C)`, `attributes(5)`, `lfcompile(5)`, `lfcompile64(5)`
crypt(3C)

NAME
  crypt – string encoding function

SYNOPSIS
  #include <crypt.h>
  char *crypt(const char *key, const char *salt);

  #include <unistd.h>
  char *crypt(const char *key, const char *salt);

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

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

  The crypt() function calls crypt_gensalt(3C) to generate the salt. If the first character of salt is "$", crypt() uses crypt.conf(4) to determine which shared module to load for the encryption algorithm. If the first character of salt is not "$", the algorithm described on crypt_unix(5) is used.

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

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

ERRORS
  The crypt() function will fail if:

  EINVAL    An entry in crypt.conf is invalid.
  ELIBACC   The required shared library was not found.
  ENOMEM    There is insufficient memory to generate the hash.
  ENOSYS    The functionality is not supported on this system.

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

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
  passwd(1), crypt_genhash_impl(3C), crypt_gensalt(3C),
  crypt_gensalt_impl(3C), getpassphrase(3C), pam(3PAM), passwd(4),
  attributes(5), crypt_unix(5), standards(5)
crypto_genhash_impl(3C)

NAME    crypt_genhash_impl – generate encrypted password

SYNOPSIS

#include <crypt.h>

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

DESCRIPTION

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

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

The salt argument is the salt used in encoding.

The params argument is an argv-like null-terminated vector of type char *.
The first element of params represents the mechanism token name from crypt.conf(4). The
remaining elements of params represent strings of the form <parameter>=value

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

RETURN VALUES

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

ERRORS

The crypt_genhash_impl() function will fail if:

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

ELIBACC The required shared library was not found.

ENOMEM There is insufficient memory to perform hashing.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

passwd(1), crypt(3C), crypt_gensalt_impl(3C), free(3C),
getpassphrase(3C), crypt.conf(4), passwd(4), attributes(5)
The `crypt_gensalt()` function generates the salt string required by `crypt(3C)`.

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

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

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

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

The `crypt_gensalt()` function will fail if:

- **EINVAL** The configuration file `crypt.conf` contains an invalid entry.
- **ELIBACC** The required shared library was not found.
- **ENOMEM** There is insufficient memory to perform hashing.

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

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

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

<table>
<thead>
<tr>
<th>Attribute Type</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

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

NAME  crypt_gensalt_impl – generate salt for password encryption

SYNOPSIS  
#include <crypt.h>

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

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

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

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

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

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

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

ERRORS  The crypt_gensalt_impl() function will fail if:

EINVAL  The configuration file crypt.conf contains an invalid entry.
ELIBACC  The required crypt shared library was not found.
ENOMEM  There is insufficient memory to perform hashing.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  passwd(1), crypt(3C), crypt_genhash_impl(3C), crypt_gensalt(3C),
getpassphrase(3C), crypt.conf(4), passwd(4), attributes(5)
cset(3C)

NAME

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

SYNOPSIS

```c
#include <euc.h>

int csetlen(int codeset);
int csetcol(int codeset);
int csetno(unsigned char c);
#include <widec.h>
int wcsetno(wchar_t pc);
```

DESCRIPTION

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

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

```c
#include <euc.h>
...

x+=csetcol(csetno(c));
```

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

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

```c
#include <euc.h>
#include <widec.h>
...

x+=csetcol(wcsetno(pc));
```

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

USAGE

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

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
</tbody>
</table>

SEE ALSO

setlocale(3C) euclen(3C), attributes(5)
ctermid(3C)

NAME
ctermid, ctermid_r – generate path name for controlling terminal

SYNOPSIS
#include <stdio.h>

char *ctermid(char *s);
char *ctermid_r(char *s);

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

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

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

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

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

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>ctermid() is Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>ctermid() is Unsafe; ctermid_r() is MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
ttyname(3C), attributes(5)
NAME  ctime, ctime_r, localtime, localtime_r, gmtime, gmtime_r, asctime, asctime_r, tzset –
convert date and time to string

SYNOPSIS
#include <time.h>

char *ctime(const time_t *clock);
struct tm *localtime(const time_t *clock);
struct tm *gmtime(const time_t *clock);
char *asctime(const struct tm *tm);

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

void tzset(void);

char *ctime_r(const time_t *clock, char *buf, int buflen);
struct tm *localtime_r(const time_t *clock, struct tm *res);
struct tm *gmtime_r(const time_t *clock, struct tm *res);
char *asctime_r(const struct tm *tm, char *buf, int buflen);

DESCRIPTION
The ctime() function converts the time pointed to by clock, representing the time in
seconds since the Epoch (00:00:00 UTC, January 1, 1970), to local time in the form of a
26-character string, as shown below. Time zone and daylight savings corrections are
made before string generation. The fields are in constant width:

Fri Sep 13 00:00:00 1986

The ctime() function is equivalent to:

asctime(localtime(clock))

The ctime(), asctime(), 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 can overwrite the information returned in either of
these objects by any of the other functions.

The ctime_r() function has the same functionality as ctime() except that the caller
must supply a buffer buf with length buflen to store the result; buf must be at least 26
bytes. The POSIX ctime_r() function does not take a buflen parameter.
The `localtime()` and `gmtime()` functions return pointers to `tm` structures (see below). The `localtime()` function corrects for the main time zone and possible alternate ("daylight savings") time zone; the `gmtime()` function converts directly to Coordinated Universal Time (UTC), which is what the UNIX system uses internally.

The `localtime_r()` and `gmtime_r()` functions have the same functionality as `localtime()` and `gmtime()` respectively, except that the caller must supply a buffer `res` to store the result.

The `asctime()` function converts a `tm` structure to a 26-character string, as shown in the previous example, and returns a pointer to the string.

The `asctime_r()` function has the same functionality as `asctime()` except that the caller must supply a buffer `buf` with length `buflen` for the result to be stored. The `buf` argument must be at least 26 bytes. The POSIX `asctime_r()` function does not take a `buflen` parameter. The `asctime_r()` function returns a pointer to `buf` upon success. In case of failure, `NULL` is returned and `errno` is set.

Declarations of all the functions and externals, and the `tm` structure, are in the `<time.h>` header. The members of the `tm` structure are:

```c
int tm_sec; /* seconds after the minute — [0, 61] */
int tm_min; /* minutes after the hour — [0, 59] */
int tm_hour; /* hour since midnight — [0, 23] */
int tm_mday; /* day of the month — [1, 31] */
int tm_mon; /* months since January — [0, 11] */
int tm_year; /* years since 1900 */
int tm_wday; /* days since Sunday — [0, 6] */
int tm_yday; /* days since January 1 — [0, 365] */
int tm_isdst; /* flag for alternate daylight savings time */
```

The value of `tm_isdst` is positive if daylight savings time is in effect, zero if daylight savings time is not in effect, and negative if the information is not available. Previously, the value of `tm_isdst` was defined as non-zero if daylight savings was in effect.

The external `time_t` variable `altzone` contains the difference, in seconds, between Coordinated Universal Time and the alternate time zone. The external variable `timezone` contains the difference, in seconds, between UTC and local standard time. The external variable `daylight` indicates whether time should reflect daylight savings time. Both `timezone` and `altzone` default to 0 (UTC). The external variable `daylight` is non-zero if an alternate time zone exists. The time zone names are contained in the external variable `tzname`, which by default is set to:

```c
char *tzname[2] = { "GMT","
```

These functions know about the peculiarities of this conversion for various time periods for the U.S. (specifically, the years 1974, 1975, and 1987). They start handling the new daylight savings time starting with the first Sunday in April, 1987.
The `tzset()` function uses the contents of the environment variable `TZ` to override the value of the different external variables. It is called by `asctime()` and can also be called by the user. See `environ(5)` for a description of the `TZ` environment variable.

Starting and ending times are relative to the current local time zone. If the alternate time zone start and end dates and the time are not provided, the days for the United States that year will be used and the time will be 2 AM. If the start and end dates are provided but the time is not provided, the time will be 2 AM. The effects of `tzset()` change the values of the external variables `timezone`, `altzone`, `daylight`, and `tzname`.

Note that in most installations, `TZ` is set to the correct value by default when the user logs on, using the local `/etc/default/init` file (see `TIMEZONE(4)`).

**ERRORS**

The `ctime_r()` and `asctime_r()` functions will fail if:

- `ERANGE` The length of the buffer supplied by the caller is not large enough to store the result.

**USAGE**

These functions do not support localized date and time formats. The `strftime(3C)` function can be used when localization is required.

The `localtime()`, `localtime_r()`, `gmtime()`, `gmtime_r()`, `ctime()`, and `ctime_r()` functions assume Gregorian dates. Times before the adoption of the Gregorian calendar will not match historical records.

**EXAMPLES**

**EXAMPLE 1** Examples of the `tzset()` function.

The `tzset()` function scans the contents of the environment variable and assigns the different fields to the respective variable. For example, the most complete setting for New Jersey in 1986 could be:

```
EST5EDT4,116/2:00:00,298/2:00:00
```
or simply

```
EST5EDT
```

An example of a southern hemisphere setting such as the Cook Islands could be

```
EDT9:30KST10:00,63/5:00,302/20:00
```

In the longer version of the New Jersey example of `TZ`, `tzname[0]` is EST, `timezone` is set to $5^\text{h}60^\text{m}60^\text{s}$, `tzname[1]` is EDT, `altzone` is set to $4^\text{h}60^\text{m}60^\text{s}$, the starting date of the alternate time zone is the 117th day at 2 AM, the ending date of the alternate time zone is the 299th day at 2 AM (using zero-based Julian days), and `daylight` is set positive. Starting and ending times are relative to the current local time zone. If the alternate time zone start and end dates and the time are not provided, the days for the United States that year will be used and the time will be 2 AM. If the start and end dates are provided but the time is not provided, the time will be 2 AM. The effects of `tzset()` are thus to change the values of the external variables `timezone`, `altzone`, `daylight`, and `tzname`. The `ctime()`, `localtime()`, `mktime()`, and `strftime()` functions also update these external variables as if they had called `tzset()` at the
EXAMPLE 1 Examples of the tzset() function.  (Continued)

time specified by the time_t or struct tm value that they are converting.

BUGS

The zoneinfo timezone data files do not transition past Tue Jan 19 03:14:07 2038 UTC. Therefore for 64-bit applications using zoneinfo timezones, calculations beyond this date might not use the correct offset from standard time, and could return incorrect values. This affects the 64-bit version of localtime(), localtime_r(), ctime(), and ctime_r().

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO

time(2), Intro(3), getenv(3C), mktime(3C), printf(3C), putenv(3C), setlocale(3C), strftime(3C), TIMEZONE(4), attributes(5), environ(5)

NOTES

When compiling multithreaded programs, see Intro(3), Notes On Multithreaded Applications.

The return values for ctime(), localtime(), and gmtime() point to static data whose content is overwritten by each call.

Setting the time during the interval of change from timezone to altzone or vice versa can produce unpredictable results. The system administrator must change the Julian start and end days annually.

The asctime(), ctime(), gmtime(), and localtime() functions are unsafe in multithread applications. The asctime_r() and gmtime_r() functions are MT-Safe. The localtime_r(), localtime_r(), and tzset() functions are MT-Safe in multithread applications, as long as no user-defined function directly modifies one of the following variables: timezone, altzone, daylight, and tzname. These four variables are not MT-Safe to access. They are modified by the tzset() function in an MT-Safe manner. The mktime(), localtime_r(), and ctime_r() functions call tzset().

Solaris 2.4 and earlier releases provided definitions of the ctime_r(), localtime_r(), gmtime_r(), and asctime_r() functions as specified in POSIX.1c Draft 6. The final POSIX.1c standard changed the interface for ctime_r() and asctime_r(). Support for the Draft 6 interface is provided for compatibility only and might not be supported in future releases. New applications and libraries should use the POSIX standard interface.
For POSIX.1c-compliant applications, the \_POSIX_PTHREAD_SEMANTICS and \_REENTRANT flags are automatically turned on by defining the \_POSIX_C_SOURCE flag with a value \( \geq 199506L \).
NAME

cctype, isdigit, isxdigit, islower, isupper, isalpha, isalnum, isspace, iscntrl, ispunct, isprint, isgraph, 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, 0 for false. The behavior of these macros, except isascii(), is affected by the current locale (see setlocale(3C)). To modify the behavior, change the LC_TYPE category in setlocale(), that is, setlocale(LC_CTYPE, newlocale). In the "C" locale, or in a locale where character type information is not defined, characters are classified according to the rules of the US-ASCII 7-bit coded character set.

The macro isascii() is defined on all integer values; the rest are defined only where the argument is an int, the value of which is representable as an unsigned char, or EOF, which is defined by the <stdio.h> header and represents end-of-file.

Functions exist for all the macros defined below. To get the function form, the macro name must be undefined (for example, #undef isdigit).

For macros described with Default and Standard conforming versions, standard-conforming behavior will be provided for standard-conforming applications (see standards(5)) and for applications that define __XPG4_CHAR_CLASS__ before including <ctype.h>.

Default

isalpha() Tests for any character for which isupper() or islower() is true.

Standard conforming

isalpha() Tests for any character for which isupper() or islower() is true, or any character that is one of the current locale-defined set of characters for which none of iscntrl(), isdigit(),
ispunct(), or isspace() is true. In "C" locale, isalpha() returns true only for the characters for which isupper() or islower() is true.

isupper() Tests for any character that is an upper-case letter or is one of the current locale-defined set of characters for which none of iscntrl(), isdigit(), ispunct(), isspace(), or islower() is true. In the "C" locale, isupper() returns true only for the characters defined as upper-case ASCII characters.

islower() Tests for any character that is a lower-case letter or is one of the current locale-defined set of characters for which none of iscntrl(), isdigit(), ispunct(), isspace(), or isupper() is true. In the "C" locale, islower() returns true only for the characters defined as lower-case ASCII characters.

isdigit() Tests for any decimal-digit character.

Default isxdigit() Tests for any hexadecimal-digit character ([0-9], [A-F], or [a-f]).

Standard conforming isxdigit() Tests for any hexadecimal-digit character ([0-9], [A-F], or [a-f] or the current locale-defined sets of characters representing the hexadecimal digits 10 to 15 inclusive). In the "C" locale, only 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f are included.

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 the current locale-defined set of characters for which isalnum() is false. In the C locale, isspace() returns true only for the standard white-space characters.

ispunct() Tests for any printing character which is neither a space (" ") nor a character for which isalnum() or iscntrl() is true.

Default isprint() Tests for any character for which ispunct(), isupper(), islower(), isdigit(), and the space character (" ") is true.

Standard conforming isprint() Tests for any character for which iscntrl() is false, and isalnum(), isgraph(), ispunct(), the space character (" "), and the characters in the current locale-defined "print" class are true.

Default isgraph() Tests for any character for which ispunct(), isupper(), islower(), and isdigit() is true.
isgraph() Tests for any character for which isalnum() and ispunct() are true, or any character in the current locale-defined "graph" class which is neither a space (" ") nor a character for which iscntrl() is true.

iscntrl() Tests for any “control character” as defined by the character set.

isascii() Tests for any ASCII character, code between 0 and 0177 inclusive.

RETURN VALUES If the argument to any of the character handling macros is not in the domain of the function, the result is undefined. Otherwise, the macro/function will return non-zero if the classification is TRUE, and 0 for FALSE.

USAGE The isdigit(), isxdigit(), islower(), isupper(), isalpha(), isalnum(), isspace(), iscntrl(), ispunct(), isprint(), isgraph(), and isascii() macros can be used safely in multithreaded applications, as long as setlocale(3C) is not being called to change the locale.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO setlocale(3C), stdio(3C), ascii(5), environ(5), standards(5)
NAME
cuserid – get character login name of the user

SYNOPSIS
#include <stdio.h>

char *cuserid(char *s);

DESCRIPTION
The cuserid() function generates a character-string representation of the login name under which the owner of the current process is logged in. If s is a null pointer, this representation is generated in an internal static area whose address is returned. Otherwise, s is assumed to point to an array of at least L_cuserid characters; the representation is left in this array. The constant L_cuserid is defined in the <stdio.h> header.

In multithreaded applications, the caller must always supply an array s for the return value.

RETURN VALUES
If the login name cannot be found, cuserid() returns a null pointer. If s is not a null pointer, the null character ‘\0’ will be placed at s[0].

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
getlogin(3C), getpwnam(3C), attributes(5)
The `dbm()` library has been superseded by `ndbm` (see `ndbm(3C)`).

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.

`key/dat` and their content 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 `dbminit()`. 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 \texttt{fetch()} and data is placed under a key by \texttt{store}. A key (and its associated contents) is deleted by \texttt{delete()}. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of \texttt{firstkey()} and \texttt{nextkey()}. \texttt{firstkey()} will return the first key in the database. With any key \texttt{nextkey()} will return the next key in the database. This code will traverse the data base:

\begin{verbatim}
for (key = firstkey; key.dptr != NULL; key = nextkey(key))
\end{verbatim}

### RETURN VALUES

All functions that return an \texttt{int} indicate errors with negative values. A zero return indicates no error. Routines that return a \texttt{datum} indicate errors with a \texttt{NULL (0) dptr.}

### SEE ALSO

\texttt{ar(1), cat(1), cp(1), tar(1), ndbm(3C)}

### NOTES

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

The `.pag` file will contain holes so that its apparent size may be larger than its actual content. Older versions of the UNIX operating system may create real file blocks for these holes when touched. These files cannot be copied by normal means ( \texttt{cp(1), cat(1), tar(1), ar(1)} ) without filling in the holes.

\texttt{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. \texttt{store} will return an error in the event that a disk block fills with inseparable data.

\texttt{delete()} does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by \texttt{firstkey()} and \texttt{nextkey()} depends on a hashing function, not on anything interesting.

There are no interlocks and no reliable cache flushing; thus concurrent updating and reading is risky.

The database files (\texttt{file.dir} and \texttt{file.pag}) are binary and are architecture-specific (for example, they depend on the architecture's byte order.) These files are not guaranteed to be portable across architectures.
decimal_to_floating(3C)

NAME  decimal_to_floating, decimal_to_single, decimal_to_double, decimal_to_extended, decimal_to_quadruple – convert decimal record to floating-point value

SYNOPSIS  

```c
#include <floatingpoint.h>

void decimal_to_single(s去了, 维度 *pm,
  decimal_record *pd, fp_exception_field_type *ps);

void decimal_to_double(double *px, decimal_mode *pm,
  decimal_record *pd, fp_exception_field_type *ps);

void decimal_to_extended(extended *px, decimal_mode *pm,
  decimal_record *pd, fp_exception_field_type *ps);

void decimal_to_quadruple(quadruple *px, decimal_mode *pm,
  decimal_record *pd, fp_exception_field_type *ps);
```

DESCRIPTION  

The decimal_to_floating() functions convert the decimal record at *pd into a floating-point value at *px, observing the modes specified in *pm and setting exceptions in *ps. If there are no IEEE exceptions, *ps will be zero.

*pd->sign and *pd->fpclass are always taken into account. *pd->exponent, *pd->dp and *pd->ndigits are used when *pd->fpclass is fp_normal or fp_subnormal. In these cases *pd->dp must contain one or more ascii digits followed by a NULL and *pd->ndigits is assumed to be the length of the string *pd->dp. Notice that for efficiency reasons, the assumption that *pd->ndigits == strlen(*pd->dp) is NEVER verified.

On output, *px is set to a correctly rounded approximation to

\[(pd->sign)*(pd->dp)*10**(pd->exponent)\]

Thus if *pd->exponent == -2 and *pd->dp == "1234", *px will get 12.34 rounded to storage precision. *pd->dp cannot have more than DECIMAL_STRING_LENGTH-1 significant digits because one character is used to terminate the string with a NULL. If *pd->more != 0 on input then additional nonzero digits follow those in *pd->dp; fp_inexact is set accordingly on output in *ps.

*px is correctly rounded according to the IEEE rounding modes in *pm->rd. *ps is set to contain fp_inexact, fp_underflow, or fp_overflow if any of these arise.

*pm->df and *pm->ndigits are not used.

strtod(3C), scanf(3C), fscanf(3C), and sscanf(3C) all use decimal_to_double().

ATTRIBUTES  

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>
SEE ALSO  fscanf(3C), scanf(3C), sscanf(3C), strtof(3C), attributes(5)
The `difftime()` function computes the difference between two calendar times.

The `difftime()` function returns the difference `(time1-time0)` expressed in seconds as a double.

The `difftime()` function is provided because there are no general arithmetic properties defined for type `time_t`.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

See also `ctime(3C), attributes(5)`
#include <sys/types.h>
#include <sys/fcntl.h>

int directio(int fd, int advice);

The `directio()` function provides advice to the system about the expected behavior of the application when accessing the data in the file associated with the open file descriptor `fd`. The system uses this information to help optimize accesses to the file's data. The `directio()` function has no effect on the semantics of the other operations on the data, though it may affect the performance of other operations.

The `advice` argument is kept per file; the last caller of `directio()` sets the `advice` for all applications using the file associated with `fd`.

Values for `advice` are defined in `<sys/fcntl.h>`.

**DIRECTIO_OFF** Applications get the default system behavior when accessing file data.

When an application reads data from a file, the data is first cached in system memory and then copied into the application's buffer (see `read(2)`). If the system detects that the application is reading sequentially from a file, the system will asynchronously "read ahead" from the file into system memory so the data is immediately available for the next `read(2)` operation.

When an application writes data into a file, the data is first cached in system memory and is written to the device at a later time (see `write(2)`). When possible, the system increases the performance of `write(2)` operations by caching the data in memory pages. The data is copied into system memory and the `write(2)` operation returns immediately to the application. The data is later written asynchronously to the device. When possible, the cached data is "clustered" into large chunks and written to the device in a single `write` operation.

The system behavior for `DIRECTIO_OFF` can change without notice.

**DIRECTIO_ON** The system behaves as though the application is not going to reuse the file data in the near future. In other words, the file data is not cached in the system's memory pages.

When possible, data is read or written directly between the application's memory and the device when the data is accessed with `read(2)` and `write(2)` operations. When such transfers are not possible, the system switches back to the default behavior, but just for that operation. In general, the transfer is possible when the...
directio(3C)

application’s buffer is aligned on a two-byte (short) boundary, the offset into the file is on a device sector boundary, and the size of the operation is a multiple of device sectors.

This advisory is ignored while the file associated with fildes is mapped (see mmap(2)).

The system behavior for DIRECTIO_ON can change without notice.

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

ERRORS
The directio() function will fail if:

EBADF The fildes argument is not a valid open file descriptor.
ENOTTY The fildes argument is not associated with a file system that accepts advisory functions.
EINVAL The value in advice is invalid.

USAGE
Small sequential I/O generally performs best with DIRECTIO_OFF.

Large sequential I/O generally performs best with DIRECTIO_ON, except when a file is sparse or is being extended and is opened with O_SYNC or O_DSYNC (see open(2)).

The directio() function is supported for the ufs file system type (see fstyp(1M)).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
fstyp(1M), mmap(2), open(2), read(2), write(2), attributes(5), fcntl(3HEAD)

WARNINGS
Switching between DIRECTIO_OFF and DIRECTIO_ON can slow the system because each switch to DIRECTIO_ON might entail flushing the file’s data from the system’s memory.
dirname (3C)

NAME
dirname – report the parent directory name of a file path name

SYNOPSIS
#include <libgen.h>

char *dirname(char *path);

DESCRIPTION
The dirname() function takes a pointer to a character string that contains a
pathname, and returns a pointer to a string that is a pathname of the parent directory
of that file. Trailing ‘/’ characters in the path are not counted as part of the path.

If path does not contain a ‘/’, then dirname() returns a pointer to the string “.”. If
path is a null pointer or points to an empty string, dirname() returns a pointer to the
string ”.”.

RETURN VALUES
The dirname() function returns a pointer to a string that is the parent directory of
path. If path is a null pointer or points to an empty string, a pointer to a string ”.” is
returned.

ERRORS
No errors are defined.

EXAMPLES
EXAMPLE 1 A sample code using the dirname() function.

<table>
<thead>
<tr>
<th>Input String</th>
<th>Output String</th>
</tr>
</thead>
<tbody>
<tr>
<td>“/usr/lib”</td>
<td>“/usr”</td>
</tr>
<tr>
<td>“/usr/”</td>
<td>“/”</td>
</tr>
<tr>
<td>“usr”</td>
<td>“/”</td>
</tr>
<tr>
<td>“/”</td>
<td>“/”</td>
</tr>
<tr>
<td>“.”</td>
<td>“.”</td>
</tr>
<tr>
<td>“..”</td>
<td>“..”</td>
</tr>
</tbody>
</table>

The following code fragment reads a path name, changes directory to the parent
directory of the named file (see chdir(2)), and opens the file.

char path[100], *pathcopy;
int fd;
gets (path);
pathcopy = strdup (path);
chdir (dirname (pathcopy) );
free (pathcopy);
fd = open (basename (path), O_RDONLY);
dirname(3C)

The dirname() and basename(3C) functions together yield a complete pathname. The expression dirname(path) obtains the pathname of the directory where basename(path) is found.

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

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

basename(1), chdir(2), basename(3C), attributes(5)
NAME | div, ldiv, lldiv – compute the quotient and remainder
SYNOPSIS | #include <stdlib.h>

    div_t div(int numer, int denom);
    ldiv_t ldiv(long int numer, long int denom);
    lldiv_t lldiv(long long numer, long long denom);

DESCRIPTION | The div() function computes the quotient and remainder of the division of the numerator numer by the denominator denom. It provides a well-defined semantics for the signed integral division and remainder operations, unlike the implementation-defined semantics of the built-in operations. The sign of the resulting quotient is that of the algebraic quotient, and if the division is inexact, the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, quotient * denom + remainder will equal numer.

    The ldiv() and lldiv() functions are similar to div(), except that the arguments and the members of the returned structure are different. The ldiv() function returns a structure of type ldiv_t and has type long int. The lldiv() function returns a structure of type lldiv_t and has type long long.

RETURN VALUES | The div() function returns a structure of type div_t, comprising both the quotient and remainder:

    int quot; /*quotient*/
    int rem; /*remainder*/

    The ldiv() function returns a structure of type ldiv_t and lldiv() returns a structure of type lldiv_t, comprising both the quotient and remainder:

    long int quot; /*quotient*/
    long int rem; /*remainder*/

ATTRIBUTES | See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO | attributes(5)
NAME dladdr, dladdr1 – translate address to symbolic information

SYNOPSIS
#include <dlfcn.h>

int dladdr(void *address, Dl_info *dli);

int dladdr1(void *address, Dl_info *dli, void **info, int flags);

DESCRIPTION
The dladdr() and dladdr1() functions determine if the specified address is located within one of the mapped objects that make up the current applications address space. An address is deemed to fall within a mapped object when it is between the base address, and the _end address of that object. If a mapped object fits this criteria, the symbol table made available to the runtime linker is searched to locate the nearest symbol to the specified address. The nearest symbol is one that has a value less than or equal to the required address.

The Dl_info 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;

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.

The dladdr1() function provides for addition information to be returned as specified by the flags argument:

RTLD_DL_SYMENT Obtain the ELF symbol table entry for the matched symbol. The info argument points to a symbol pointer as defined in <sys/elf.h> (Elf32_Sym **info or Elf64_Sym **info).

RTLD_DL_LINKMAP Obtain the Link_map for the matched file. The info argument points to a Link_map pointer as defined in <sys/link.h> (Link_map **info).

RETURN VALUES
If the specified address cannot be matched to a mapped object, a 0 is returned. Otherwise, a non-zero return is made and the associated Dl_info elements are filled.
The `dladdr()` and `dladdr1()` functions are one of a family of functions that give the user direct access to the dynamic linking facilities (see *Linker and Libraries Guide*) and are available to dynamically-linked processes only.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`ld(1), dlclose(3DL), dldump(3DL), dlerror(3DL), dlopen(3DL), dlSYM(3DL), attributes(5)`

*Linker and Libraries Guide*

**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()`).

If no symbol is found to describe the specified address, both the `dli_sname` and `dli_saddr` members are set to 0.
dlclose(3DL)

NAME
dlclose – close a shared object

SYNOPSIS
c c [ flag ... ] file ... -ldl [ library ... ]
#include <dlfcn.h>

int dlclose(void *handle);

DESCRIPTION
The dlclose() function disassociates a shared object previously opened by
dlopen() from the current process. Once an object has been closed using
dlclose(), 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. handle is the value returned by a previous invocation of
dlopen().

RETURN VALUES
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().

USAGE
The dlclose() function is one of a family of functions that give the user direct access
to the dynamic linking facilities (see Linker and Libraries Guide) and are available to
dynamically-linked processes only.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
ld(1), dladdr(3DL), dldump(3DL), dlerror(3DL), dlopen(3DL), dlsym(3DL), attributes(5)

Linker and Libraries Guide

NOTES
A successful invocation of dlclose() does not guarantee that the objects associated
with handle will actually be removed from the address space of the process. Objects
loaded by one invocation of dlopen() may also be loaded by another invocation of
dlopen(). The same object may also be opened multiple times. An object will not be
removed from the address space until all references to that object through an explicit
dlopen() invocation have been closed and all other objects implicitly referencing that
object have also been closed.

Once an object has been closed by dlclose(), referencing symbols contained in that
object can cause undefined behavior.
dldump – create a new file from a dynamic object component of the calling process

SYNOPSIS

cc [ flag ... ] file ... -ldl [ library ... ]
#include <dlfcn.h>

int dldump(const char *ipath, const char *opath, int flags);

DESCRIPTION

The dldump() function creates a new dynamic object opath from an existing dynamic object ipath that is bound to the current process. An ipath value of 0 is interpreted as the dynamic object that started the process. The new object is constructed from the existing objects’ disc file. Relocations can be applied to the new object to pre-bind it to other dynamic objects, or fix the object to a specific memory location. In addition, data elements within the new object may be obtained from the objects’ memory image as it exists in the calling process.

These techniques allow the new object to be executed with a lower startup cost, either because there are less relocations required to load the object, or because of a reduction in the data processing requirements of the object. However, it is important to note that limitations may exist in using these techniques. Applying relocations to the new dynamic object opath may restrict its flexibility within a dynamically changing environment. In addition, limitations regarding data usage may make dumping a memory image impractical (see EXAMPLES).

The runtime linker verifies that the dynamic object ipath is mapped as part of the current process. Thus, the object must either be the dynamic object that started the process (see exec(2)), one of the process’s dependencies, or an object that has been preloaded (see ld.so.1(1)).

As part of the runtime processing of a dynamic object, relocation records within the object are interpreted and applied to offsets within the object. These offsets are said to be relocated. Relocations can be categorized into two basic types: non-symbolic and symbolic.

The non-symbolic relocation is a simple relative relocation that requires the base address at which the object is mapped to perform the relocation. The symbolic relocation requires the address of an associated symbol, and results in a binding to the dynamic object that defines this symbol. This symbol definition may originate from any of the dynamic objects that make up the process, that is, the object that started the process, one of the process’s dependencies, an object that has been preloaded, or the dynamic object being relocated.

The flags parameter controls the relocation processing and other attributes of producing the new dynamic object opath. Without any flags, the new object is constructed solely from the contents of the ipath disc file without any relocations applied.

Various relocation flags may be oz’ed into the flags parameter to affect the relocations applied to the new object. Non-symbolic relocations can be applied using the following:

RTLD_REL_RELATIVE Relocation records from the object ipath, that define relative relocations, are applied to the object opath.
A variety of symbolic relocations can be applied using the following flags (each of these flags also implies RTLD_REL_RELATIVE is in effect):

- **RTLD_REL_EXEC**
  Symbolic relocations that result in binding *ipath* to the dynamic object that started the process (commonly a dynamic executable) are applied to the object *opath*.

- **RTLD_REL_DEPENDS**
  Symbolic relocations that result in binding *ipath* to any of the dynamic dependencies of the process are applied to the object *opath*.

- **RTLD_REL_PRELOAD**
  Symbolic relocations that result in binding *ipath* to any objects preloaded with the process are applied to the object *opath*. (See LD_PRELOAD in ld.so.1).

- **RTLD_REL_SELF**
  Symbolic relocations that result in binding *ipath* to itself are applied to the object *opath*.

- **RTLD_REL_WEAK**
  Weak relocations that remain unresolved are applied to the object *opath* as 0.

- **RTLD_REL_ALL**
  *All* relocation records defined in the object *ipath* are applied to the new object *opath* (this is basically a concatenation of all the above relocation flags).

Note that for dynamic executables, RTLD_REL_RELATIVE, RTLD_REL_EXEC, and RTLD_REL_SELF have no effect (see EXAMPLES).

If relocations, knowledgeable of the base address of the mapped object, are applied to the new object *opath*, then the new object will become fixed to the location that the *ipath* image is mapped within the current process.

Any relocations applied to the new object *opath* will have the original relocation record removed so that the relocation will not be applied more than once. Otherwise, the new object *opath* will retain the relocation records as they exist in the *ipath* disc file.

The following additional attributes for creating the new dynamic object *opath* can be specified using the flags parameter:

- **RTLD_MEMORY**
  The new object *opath* is constructed from the current memory contents of the *ipath* image as it exists in the calling process. This option allows data modified by the calling process to be captured in the new object. Note that not all data modifications may be applicable for capture; significant restrictions exist in using this technique (see EXAMPLES). By default, when processing a dynamic executable, any allocated memory that follows the end of the data segment is captured in the new object (see malloc (3C) and brk(2)). This data, which represents the process heap, is saved as a new .SUNW_heap section in the object.
opath. The objects’ program headers and symbol entries, such as _end, are adjusted accordingly. See also RTLD_NOHEAP. When using this attribute, any relocations that have been applied to the ipath memory image that do not fall into one of the requested relocation categories are undone, that is, the relocated element is returned to the value as it existed in the ipath disc file.

RTLD_STRIP

Only collect allocatable sections within the object opath; sections that are not part of the dynamic objects’ memory image are removed. This parameter reduces the size of the opath disc file and is comparable to having run the new object through strip(1).

RTLD_NOHEAP

Do not save any heap to the new object. This option is only meaningful when processing a dynamic executable with the RTLD_MEMORY attribute and allows for reducing the size of the opath disc file. In this case, the executable must confine its data initialization to data elements within its data segment and must not use any allocated data elements that comprise the heap.

It should be emphasized that an object created by dldump() is simply an updated ELF object file. No additional state regarding the process at the time dldump() is called is maintained in the new object. dldump() does not provide a panacea for checkpoint/resume. A new dynamic executable, for example, will not start where the original executable called dldump(); it will gain control at the executable’s normal entry point (see EXAMPLES).

RETURN VALUES

On successful creation of the new object, dldump() returns 0. Otherwise, a non-zero value is returned and more detailed diagnostic information is available through dlerror().

EXAMPLES

**EXAMPLE 1** Sample code using dldump().

The following code fragment, which can be part of a dynamic executable a.out, can be used to create a new shared object from one of the dynamic executables’ dependencies libfoo.so.1:

```c
const char * ipath = "libfoo.so.1";
const char * opath = "/tmp/libfoo.so.1";
...
if (dldump(ipath, opath, RTLD_REL_RELATIVE) != 0)
    (void) printf("dldump failed: %s\n", dlerror( ));
```

The new shared object opath is fixed to the address of the mapped ipath bound to the dynamic executable a.out. All relative relocations are applied to this new shared object, which will reduce its relocation overhead when it is used as part of another process.
dldump(3DL)

EXAMPLE 1 Sample code using dldump().  (Continued)

By performing only relative relocations, any symbolic relocation records remain
deﬁned within the new object, and thus the dynamic binding to external symbols will
be preserved when the new object is used.

Use of the other relocation ﬂags can ﬁx speciﬁc relocations in the new object and thus
can reduce even more the runtime relocation startup cost of the new object. However,
this will also restrict the ﬂexibility of using the new object within a dynamically
changing environment, as it will bind the new object to some or all of the dynamic
objects presently mapped as part of the process.

For example, the use of RTLD_REL_SELF will cause any references to symbols from
ipath to be bound to deﬁnitions within itself if no other preceding object deﬁned the
same symbol. In other words, a call to foo() within ipath will bind to the deﬁnition foo
within the same object. Therefore, opath will have one less binding that must be
computed at runtime. This reduces the startup cost of using opath by other
applications; however, interposition of the symbol foo will no longer be possible.

Using a dumped shared object with applied relocations as an applications dependency
normally requires that the application have the same dependencies as the application
that produced the dumped image. Dumping shared objects, and the various
ﬂags associated with relocation processing, have some specialized uses. However, the
technique is intended as a building block for future technology.

The following code fragment, which is part of the dynamic executable a.out, can be
used to create a new version of the dynamic executable:

```c
static char * dumped = 0;
const char * opath = "/a.out.new";
...
if (dumped == 0) {
    char buffer[100];
    int size;
    time_t seconds;
    ...
    /* Perform data initialization */
    seconds = time((time_t *)0);
    size = cftime(buffer, (char *)0, &seconds);
    if ((dumped = (char *)malloc(size + 1)) == 0) {
        (void) printf("malloc failed: %s\n", strerror(errno));
        return (1);
    }
    (void) strcpy(dumped, buffer);
    ...
    /*
    * Tear down any undesirable data initializations and
    * dump the dynamic executables memory image.
    */
    _exithandle();
    _exit(dldump(0, opath, RTLD_MEMORY));
}
```
EXAMPLE 1 Sample code using dldump().  (Continued)

```c
(void) printf("Dumped: %s\n", dumped);
```

Any modifications made to the dynamic executable, up to the point the `dldump()` call is made, are saved in the new object `a.out.new`. This mechanism allows the executable to update parts of its data segment and heap prior to creating the new object. In this case, the date the executable is dumped is saved in the new object. The new object can then be executed without having to carry out the same (presumably expensive) initialization.

For greatest flexibility, this example does not save any relocated information. The elements of the dynamic executable `ipath` that have been modified by relocations at process startup, that is, references to external functions, are returned to the values of these elements as they existed in the `ipath` disc file. This preservation of relocation records allows the new dynamic executable to be flexible, and correctly bind and initialize to its dependencies when executed on the same or newer upgrades of the OS. Fixing relocations by applying some of the relocation flags would bind the new object to the dependencies presently mapped as part of the process calling `dldump()`. It may also remove necessary copy relocation processing required for the correct initialization of its shared object dependencies. Therefore, if the new dynamic executables’ dependencies have no specialized initialization requirements, the executable may still only interact correctly with the dependencies to which it binds if they were mapped to the same locations as they were when `dldump()` was called.

Note that for dynamic executables, `RTLD_REL_RELATIVE`, `RTLD_REL_EXEC`, and `RTLD_REL_SELF` have no effect, as relocations within the dynamic executable will have been fixed when it was created by `ld(1)`.

When `RTLD_MEMORY` is used, care should be taken to insure that dumped data sections that reference external objects are not reused without appropriate re-initialization. For example, if a data item contains a file descriptor, a variable returned from a shared object, or some other external data, and this data item has been initialized prior to the `dldump()` call, its value will have no meaning in the new dumped image.

When `RTLD_MEMORY` is used, any modification to a data item that is initialized via a relocation whose relocation record will be retained in the new image will effectively be lost or invalidated within the new image. For example, if a pointer to an external object is incremented prior to the `dldump()` call, this data item will be reset to its disc file contents so that it can be relocated when the new image is used; hence, the previous increment is lost.

Non-idempotent data initializations may prevent the use of `RTLD_MEMORY`. For example, the addition of elements to a linked-list via `init` sections can result in the linked-list data being captured in the new image. Running this new image may result in `init` sections continuing to add new elements to the list without the prerequisite
EXAMPLE 1 Sample code using dldump(). (Continued)

initialization of the list head. It is recommended that _exithandle(3C) be called before dldump() to tear down any data initializations established via initialization code. Note that this may invalidate the calling image; thus, following the call to dldump(), only a call to _exit(2) should be made.

USAGE

The dldump() function is one of a family of functions that give the user direct access to the dynamic linking facilities (see Linker and Libraries Guide) and are available to dynamically-linked processes only.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>SUNWcsu</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

ld(1), ld.so.1(1), strip(1), _exit(2), brk(2), exec(2), _exithandle(3C), dladdr(3DL), dlclose(3DL), dlerror(3DL), dlopen(3DL), dlsym(3DL), end(3C), malloc(3C), attributes(5)

Linker and Libraries Guide

NOTES

These functions are available to dynamically-linked processes only.

Any NOBITS sections within the ipath are expanded to PROGBITS sections within the opath. NOBITS sections occupy no space within an ELF file image. They declare memory that must be created and zero-filled when the object is mapped into the runtime environment. .bss is a typical example of this section type. PROGBITS sections, on the other hand, hold information defined by the object within the ELF file image. This section conversion reduces the runtime initialization cost of the new dumped object but increases the objects’ disc space requirement.

When a shared object is dumped, and relocations are applied which are knowledgeable of the base address of the mapped object, the new object is fixed to this new base address and thus its ELF type is reclassified to be a dynamic executable. This new object can be processed by the runtime linker, but is not valid as input to the link-editor.

If relocations are applied to the new object, any remaining relocation records will be reorganized for better locality of reference. The relocation sections are renamed to _SUNW_reloc and the association to the section they were to relocate is lost. Only the offset of the relocation record itself is meaningful. This change does not make the new object invalid to either the runtime linker or link-editor, but may reduce the objects analysis with some ELF readers.
NAME
dlerror – get diagnostic information

SYNOPSIS
cce [ flag ... ] file ... -ldl [ library ... ]
#include <dlfcn.h>
char *dlerror(void);

DESCRIPTION
The dlerror() function 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.

USAGE
The dlerror() function is one of a family of functions that give the user direct access to the dynamic linking facilities (see Linker and Libraries Guide) and are available to dynamically-linked processes only.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
ld(1), dladdr(3DL), dlclose(3DL), dldump(3DL), dlopen(3DL), dlsym(3DL), attributes(5)

Linker and Libraries Guide

NOTES
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.
The `dlinfo()` function extracts information about a dynamically-loaded object. This function is loosely modeled after the `ioctl()` function. The `request` argument and a third argument of varying type are passed to `dlinfo()`. The action taken by `dlinfo()` depends on the value of the `request` provided.

A `handle` argument, required for all requests except `RTLD_DI_CONFIGADDR`, is either the value returned from a `dlopen()` or `dlmopen()` call, or the special handle `RTLD_SELF`. If `handle` is the value returned from a `dlopen()` or `dlmopen()` call, the information returned by the `dlinfo()` call pertains to the specified object. If `handle` is the special handle `RTLD_SELF`, the information returned by the `dlinfo()` call pertains to the caller itself.

The following are possible values for `request` to be passed into `dlinfo()`:

**RTLD_DI_CONFIGADDR**

Obtain the configuration file name and the address at which it has been loaded. The `p` argument is a `Dl_info` pointer (`Dl_info *p`). The following elements from this structure are initialized:

- `dli_fname` The full name of the configuration file.
- `dli_fbase` The base address of the configuration file loaded into memory.

**RTLD_DI_LINKMAP**

Obtain the `Link_map` for the `handle` specified. The `p` argument points to a `Link_map` pointer (`Link_map **p`). The actual storage for the `Link_map` structure is maintained by `ld.so.1`.

The `Link_map` structure includes the following members:

- `unsigned long l_addr;` /* base address */
- `char *l_name;` /* object name */
- `Elf32_Dyn *l_ld;` /* .dynamic section */
- `Link_map *l_next;` /* next link object */
- `Link_map *l_prev;` /* previous link object */
- `char *l_refname;` /* filter reference name */
- `l_addr` The base address of the object loaded into memory.
dlinfo(3DL)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>l_name</td>
<td>The full name of the loaded object. This is the filename of the object as referenced by ld.so.1.</td>
</tr>
<tr>
<td>l ld</td>
<td>Points to the SHT_DYNAMIC structure.</td>
</tr>
<tr>
<td>l next</td>
<td>The next Link_map on the link-map list, other objects on the same link-map list as the current object may be examined by following the and l_prev fields.</td>
</tr>
<tr>
<td>l prev</td>
<td>The previous Link_map on the link-map list.</td>
</tr>
<tr>
<td>l refname</td>
<td>If the object referenced is a filter this field points to the name of the object being filtered. If the object is not a filter, this field will be 0. See Linker and Libraries Guide.</td>
</tr>
</tbody>
</table>

**RTLD_DI_LMID**

Obtain the ID for the link-map list upon which the handle is loaded. The p argument is a Lmid_t pointer (Lmid_t *p).

**RTLD_DI_SERINFO**

Obtain the library search paths for the handle specified. The p argument is a Dl_serinfo pointer (Dl_serinfo *p). A user must first initialize the Dl_serinfo structure with a RTLD_DI_SERINFOSIZE request. See EXAMPLES.

The returned Dl_serinfo structure contains dls_cnt Dl_serpath entries. Each entry’s dlp_name field points to the search path. The corresponding dlp_info field contains one or more flags indicating the origin of the path (see the LA_SER_ * flags defined in <link.h>).

**RTLD_DI_SERINFOSIZE**

Initialize a Dl_serinfo structure for use in a RTLD_DI_SERINFO request. Both the dls_cnt and dls_size fields are returned to indicate the number of search paths applicable to the handle, and the total size of a Dl_serinfo buffer required to hold dls_cnt Dl_serpath entries and the associated search path strings.
To obtain the complete path information, a new Dl_serinfo buffer of size dls_size should be allocated, initialized with the dls_cnt and dls_size entries, and passed to a RTLD_DI_SERINFO request. See EXAMPLES.

RTLD_DI_ORIGIN Obtain the origin of the dynamic object associated with the handle. The p argument is a char pointer (char *p). The dirname(3C) of the associated object’s realpath(3C), which can be no bigger than PATH_MAX, is copied to the pointer p.

RETURN VALUES If the request is invalid, the parameter p is null, handle does not refer to a valid object opened by dlopen() or is not the special handle RTLD_SELF, or the Dl_serinfo structure is uninitialized for a RTLD_DI_SERINFO request, then dlinfo() returns -1. More detailed diagnostic information is available through dlerror(3DL).

EXAMPLES

EXAMPLE 1 Using dlinfo() to obtain the library search paths

The following example shows how a dynamic object can inspect the library search paths that would be used to locate a simple filename with dlopen(). For simplicity, error checking has been omitted.

```c
Dl_serinfo _info, *info = &_info;
Dl_serpath *path;
uint_t cnt;

/* determine search path count and required buffer size */
dlinfo(RTLD_SELF, RTLD_DI_SERINFOSIZE, (void *)info);

/* allocate new buffer and initialize */
info = malloc(_info.dls_size);
info->dls_size = _info.dls_size;
info->dls_cnt = _info.dls_cnt;

/* obtain search path information */
dlinfo(RTLD_SELF, RTLD_DI_SERINFO, (void *)info);

path = &info->dls_serpath[0];
for (cnt = 1; cnt <= info->dls_cnt; cnt++, path++) {
    (void) printf("%2d: %s\n", cnt, path->dls_name);
}
```

USAGE The dlinfo() function is one of a family of functions that give the user direct access to the dynamic linking facilities (see Linker and Libraries Guide) and are available to dynamically-linked processes only.
ATTRIBUTES See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO ld(1), ioctl(2), dirname(3C), dlclose(3DL), dldump(3DL), dlerror(3DL), dlmopen(3DL), dlopen(3DL), dlSYM(3DL), realpath(3C), attributes(5)

*Linker and Libraries Guide*
dlopen, dlmopen

NAME
dlopen, dlmopen – gain access to an executable object file

SYNOPSIS
cce [ flag...] file... -ldl [ library...] 
#include <dlfcn.h>
#include <link.h>

void * dlopen(const char * pathname, int mode);
void * dlmopen(Lmid_t lmid, const char * pathname, int mode);

DESCRIPTION
The dlopen() function makes an executable object file available to a running process. It returns to the process a handle which the process may use on subsequent calls to dl sym() and dl close(). The value of this handle should not be interpreted in any way by the process. The pathname argument is the path name of the object to be opened. A path name containing an embedded ‘/ ’ is interpreted as an absolute path or relative to the current directory; otherwise, the set of search paths currently in effect by the runtime linker will be used to locate the specified file. See NOTES below.

Any dependencies recorded within pathname are also loaded as part of the dlopen(). These dependencies are searched, in the order they are loaded, to locate any additional dependencies. This process will continue until all the dependencies of pathname are loaded. This dependency tree is referred to as a group.

If the value of pathname is 0, dlopen() provides a handle on a 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 dependencies loaded at program startup, and any objects that were loaded using dlopen() together with the RTLD_GLOBAL flag. As the latter set of objects can change during process execution, the set identified by handle can also change dynamically.

The dlmopen() function is identical to the dlopen() routine, except that an identifying link-map id ( lmid) is passed into it. This link-map id informs the dynamic linking facilities upon which link-map list to load the object. See Linker and Libraries Guide.

The mode argument describes how dlopen() will operate upon pathname with respect to the processing of reference relocations and the scope of visibility of the symbols provided by pathname and its dependencies.

When an object is brought into the address space of a process, it can contain references to symbols whose addresses are not known until the object is loaded. These references must be relocated before the symbols can be accessed and can be categorized as either immediate or lazy references. immediate references are typically to data items used by the object code, pointers to functions, and even calls to functions made from position dependent shared objects. lazy references are typically calls to global functions made from a position independent shared objects. For more information on these types of reference see Linker and Libraries Guide. The mode argument governs when these references take place and can have the following values:

| RTLD_LAZY | Only immediate symbol references are relocated when the object is first loaded. lazy references are not relocated until a given function |
is invoked for the first time. This mode should improve performance, since a process cannot require all lazy references in any given object. This behavior mimics the normal loading of dependencies during process initialization.

**RTLD_NOW**

All necessary relocations are performed when the object is first loaded. This may waste some processing, if relocations are performed for lazy references that are never used. This behavior can be useful for applications that need to know as soon as an object is loaded that all symbols referenced during execution will be available. This option mimics the loading of dependencies when the environment variable `LD_BIND_NOW` is in effect.

To determine the scope of visibility for 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 global symbols are made available for the relocation processing of any other object. In addition, symbol lookup using `dlopen(0, mode)` and an associated `dlsym()`, allows objects loaded with `RTLD_GLOBAL` to be searched.

- **RTLD_LOCAL**
  The object’s globals symbols are only available for the relocation processing of other objects that comprise the same group.

The program image file, and any objects loaded at program startup, have the mode `RTLD_GLOBAL`. The mode `RTLD_LOCAL` is the default mode for any objects acquired with `dlopen()`. A local object may be a dependency of more then one group. Any object of mode `RTLD_LOCAL` that is referenced as a dependency of an object of mode `RTLD_GLOBAL` will be promoted to `RTLD_GLOBAL`. In other words, the `RTLD_LOCAL` mode is ignored.

Any object loaded by `dlopen()` that requires relocations against global symbols can reference the symbols in any `RTLD_GLOBAL` object, which are at least the program image file and any objects loaded at program startup, from the object itself, and from any dependencies the object references. However, the *mode* parameter may also be bitwise OR-ed with the following values to affect the scope of symbol availability:

- **RTLD_GROUP**
  Only symbols from the associated group are made available for relocation. A group is established from the defined object and all the dependencies of that object. A group must be completely self-contained. All dependency relationships between the members of the group must be sufficient to satisfy the relocation requirements of each object that comprises the group.

- **RTLD_PARENT**
  The symbols of the object initiating the `dlopen()` call are made available to the objects obtained by `dlopen()` itself. This option is useful when hierarchical `dlopen()` families are created. Note that although the parent object can supply symbols for the relocation of this object, the parent object is not available to `dlsym()` through the returned `handle`. 
dlopen(3DL)

**RTLD_WORLD** Only symbols from RTLD_GLOBAL objects are made available for relocation.

The default modes for dlopen() are both RTLD_WORLD and RTLD_GROUP. These modes are or’ed together if an object is required by different dependencies specifying differing modes.

The following modes provide additional capabilities outside of relocation processing:

**RTLD_NODELETE** The specified object will not be deleted from the address space as part of a dlclose().

**RTLD_NOLOAD** The specified object is not loaded as part of the dlopen(), but a valid handle is returned if the object already exists as part of the process address space. Additional modes can be specified and will be or’ed with the present mode of the object and its dependencies. The RTLD_NOLOAD mode provides a means of querying the presence, or promoting the modes, of an existing dependency.

The lmid passed to dlmopen() identifies the link-map list where the object will be loaded. This can be any valid lmid_t returned by dlinfo() or one of the following special values:

- **LM_ID_BASE** Load the object on the applications link-map list.
- **LM_ID_LDSO** Load the object on the dynamic linkers (ld.so.1) link-map list.
- **LM_ID_NEWLM** Causes the object to create a new link-map list as part of loading. It is vital that any object opened on a new link-map list have all of its dependencies expressed because there will be no other objects on this link-map.

**RETURN VALUES** If pathname cannot be found, cannot be opened for reading, is not a shared or relocatable object, or if an error occurs during the process of loading pathname or relocating its symbolic references, dlopen() will return NULL. More detailed diagnostic information will be available through dlerror().

**USAGE** The dlopen() and dlmopen() functions are members of a family of functions that give the user direct access to the dynamic linking facilities (see Linker and Libraries Guide) and are available to dynamically-linked processes only.

**ATTRIBUTES** See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO** ld(1), ld.so.1(1), dladdr(3DL), dlclose(3DL), dldebug(3DL), dlerror(3DL), dlinfo(3DL), dltsym(3DL), attributes(5)

92 man pages section 3: Basic Library Functions • Last Revised 8 Nov 2001
If other objects were link-edited with *pathname* when *pathname* was built, that is, the *pathname* has dependencies on other objects, those objects will automatically be loaded by *dlopen()*.

The directory search path used to find both *pathname* and the other needed objects may be affected by setting the environment variable *LD_LIBRARY_PATH*, which is analyzed once at process startup, and from a runpath setting within the object from which the call to *dlopen()* originated. These search rules will only be applied to path names that do not contain an embedded ‘/’.

Objects whose names resolve to the same absolute or relative path name may be opened any number of times using *dlopen()*; however, the object referenced will only be loaded once into the address space of the current process.

When loading shared objects the application should open a specific version of the shared object, as opposed to relying on the version of the shared object pointed to by the symbolic link.

When building objects that are to be loaded on a new link-map list (see *LM_ID_NEWLM*), some precautions need to be taken. In general, all dependencies must be included when building an object. Also, include */usr/lib/libmalloc.so.1* before */usr/lib/libc.so.1* when building an object.

When an object is loaded into memory on a new link-map list, it is isolated from the main running program. There are certain global resources that are only usable from one link-map list. A few examples of these would be the *sbrk()* based *malloc()*, *libthread()*, and the signal vectors. Because of this, care must be taken not to use any of these resources on any but the primary link-map list. These issues are discussed in further detail in the *Linker and Libraries Guide*.

Some symbols defined in dynamic executables or shared objects may not be available to the runtime linker. The symbol table created by *ld* for use by the runtime linker might contain only a subset of the symbols defined in the object.
**NAME**
dlsym – get the address of a symbol in a shared object or executable

**SYNOPSIS**
```
#include <dlfcn.h>

void *dlsym(void *handle, const char *name);
```

**DESCRIPTION**
The `dlsym()` function allows a process to obtain the address of a symbol defined within a shared object or executable. The `handle` argument is either the value returned from a call to `dlopen()` or one of the special handles `RTLD_DEFAULT`, `RTLD_NEXT`, or `RTLD_SELF`. The `name` argument is the symbol’s name as a character string.

In the case of a handle returned from `dlopen()`, the corresponding shared object must not have been closed using `dlclose()`. The `dlsym()` function searches for the named symbol in all shared objects loaded automatically as a result of loading the object referenced by `handle`. See `dlopen(3DL)`.

In the case of the special handle `RTLD_DEFAULT`, `dlsym()` searches for the named symbol starting with the first object loaded and proceeding through the list of initial loaded objects, and any global objects obtained with `dlopen(3DL)`, until a match is found. This search follows the default model employed to relocate all objects within the process.

In the case of the special handle `RTLD_NEXT`, `dlsym()` searches for the named symbol in the objects that were loaded following the object from which the `dlsym()` call is being made.

In the case of the special handle `RTLD_SELF`, `dlsym()` searches for the named symbol in the objects that were loaded starting with the object from which the `dlsym()` call is being made.

In the case of `RTLD_DEFAULT`, `RTLD_NEXT`, and `RTLD_SELF`, if the objects being searched have been loaded from `dlopen()` calls, `dlsym()` searches the object only if the caller is part of the same `dlopen()` dependency hierarchy, or if the object was given global search access. See `dlopen(3DL)` for a discussion of the `RTLD_GLOBAL` mode.

**RETURN VALUES**
If `handle` does not refer to a valid object opened by `dlopen()`, is not the special handle `RTLD_DEFAULT`, `RTLD_NEXT`, or `RTLD_SELF`, or if the named symbol cannot be found within any of the objects associated with `handle`, `dlsym()` will return `NULL`. More detailed diagnostic information is available through `dlerror(3DL)`.

**EXAMPLES**

### Example 1
Using `dlopen()` and `dlsym()` to access a function or data objects.

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.

```c
void *handle;
int *iptr, (*fptr)(int);

/* open the needed object */
handle = dlopen("/usr/home/me/libfoo.so.1", RTLD_LAZY);
```
EXAMPLE 1 Using dlopen() and dlsym() to access a function or data objects.

(Continued)

/* find the address of function and data objects */
fptr = (int (*)(int))dlsym(handle, "my_function");
iptr = (int *)dlsym(handle, "my_object");

/* invoke function, passing value of integer as a parameter */
(*fptr)(*iptr);

EXAMPLE 2 Using dlsym() to verify that a particular function is defined.

The following code fragment shows how dlsym() can be used to verify that a
particular function is defined and to call it only if it is.

int (*fptr)();

if ((fptr = (int (*)())dlsym(RTLD_DEFAULT,
    "my_function") != NULL) {
    (*fptr)();
}

USAGE
The dlsym() function is one of a family of functions that give the user direct access to
the dynamic linking facilities (see Linker and Libraries Guide) and are available to
dynamically-linked processes only.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
ld(1), dladdr(3DL), dlclose(3DL), dldump(3DL), dlerror(3DL), dlopen(3DL),
attributes(5)

Linker and Libraries Guide
This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions `drand48()` and `erand48()` return non-negative double-precision floating-point values uniformly distributed over the interval \([0.0, 1.0)\).

Functions `lrand48()` and `nrand48()` return non-negative long integers uniformly distributed over the interval \([0, 2^{31}]\).

Functions `mrand48()` and `jrand48()` return signed long integers uniformly distributed over the interval \([-2^{31}, 2^{31}]\).

Functions `srand48()`, `seed48()`, and `lcong48()` are initialization entry points, one of which should be invoked before either `drand48()`, `lrand48()`, or `mrand48()` is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if `drand48()`, `lrand48()`, or `mrand48()` is called without a prior call to an initialization entry point.) Functions `erand48()`, `nrand48()`, and `jrand48()` do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, \(X_i\), according to the linear congruential formula

\[ X_{n+1} = (aX_n + c) \mod m, \quad n \geq 0. \]

The parameter \(m = 2^{48}\); hence 48-bit integer arithmetic is performed. Unless `lcong48()` has been invoked, the multiplier value \(a\) and the addend value \(c\) are given by

\[ a = 5DECE66D_{16} = 2736731631558_8 \]
The value returned by any of the functions `drand48()`, `erand48()`, `lrand48()`, `nrand48()`, `mrand48()`, or `jrand48()` is computed by first generating the next 48-bit $X_i$ in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of $X_i$ and transformed into the returned value.

The functions `drand48()`, `lrand48()`, and `mrand48()` store the last 48-bit $X_i$ generated in an internal buffer. $X_i$ must be initialized prior to being invoked. The functions `erand48()`, `nrand48()`, and `jrand48()` require the calling program to provide storage for the successive $X_i$ values in the array specified as an argument when the functions are invoked. These routines do not have to be initialized; the calling program must place the desired initial value of $X_i$ into the array and pass it as an argument. By using different arguments, functions `erand48()`, `nrand48()`, and `jrand48()` allow separate modules of a large program to generate several independent streams of pseudo-random numbers, that is, the sequence of numbers in each stream will not depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function `srand48()` sets the high-order 32 bits of $X_i$ to the 32 bits contained in its argument. The low-order 16 bits of $X_i$ are set to the arbitrary value $330E_{16}$.

The initializer function `seed48()` sets the value of $X_i$ to the 48-bit value specified in the argument array. In addition, the previous value of $X_i$ is copied into a 48-bit internal buffer, used only by `seed48()`, and a pointer to this buffer is the value returned by `seed48()`. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last $X_i$ value, and then use this value to reinitialize using `seed48()` when the program is restarted.

The initialization function `lcong48()` allows the user to specify the initial $X_i$, the multiplier value $a$, and the addend value $c$. Argument array elements `param[0-2]` specify $X_i$, `param[3-5]` specify the multiplier $a$, and `param[6]` specifies the 16-bit addend $c$. After `lcong48()` has been called, a subsequent call to either `srand48()` or `seed48()` will restore the “standard” multiplier and addend values, $a$ and $c$, specified above.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`rand(3C)`, attributes(5)
dup2(3C)

NAME dup2 – duplicate an open file descriptor

SYNOPSIS #include <unistd.h>

    int dup2(int fildes, int fildes2);

DESCRIPTION The dup2() function causes the file descriptor fildes2 to refer to the same file as fildes. The fildes argument is a file descriptor referring to an open file, and fildes2 is a non-negative integer less than the current value for the maximum number of open file descriptors allowed the calling process. See getrlimit(2). If fildes2 already refers to an open file, not fildes, it is closed first. If fildes2 refers to fildes, or if fildes is not a valid open file descriptor, fildes2 will not be closed first.

The dup2() function is equivalent to fcntl(fildes, F_DUP2FD, fildes2).

RETURN VALUES Upon successful completion a non-negative integer representing the file descriptor is returned. Otherwise, −1 is returned and errno is set to indicate the error.

ERRORS The dup2() function will fail if:

EBADF The fildes argument is not a valid open file descriptor.

EBADF The fildes2 argument is negative or is not less than the current resource limit returned by getrlimit(RLIMIT_NOFILE, ...).

EINTR A signal was caught during the dup2() call.

EMFILE The process has too many open files. See fcntl(2).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO close(2), creat(2), exec(2), fcntl(2), getrlimit(2), open(2), pipe(2), lockf(3C), attributes(5)
The `econvert()` function converts the value to a null-terminated string of `ndigit` ASCII digits in `buf` and returns a pointer to `buf`. `buf` should contain at least `ndigit+1` characters. The position of the decimal point relative to the beginning of the string is stored indirectly through `decpt`. Thus `buf == "314"` and `*decpt == 1` corresponds to the numerical value 3.14, while `buf == "0314.0"` and `*decpt == -1` corresponds to the numerical value .0314. If the sign of the result is negative, the word pointed to by `sign` is nonzero; otherwise it is zero. The least significant digit is rounded.

The `fconvert()` function works much like `econvert()`, except that the correct digit has been rounded as if for `sprintf(%w.nf)` output with `n=ndigit` digits to the right of the decimal point. `ndigit` can be negative to indicate rounding to the left of the decimal point. The return value is a pointer to `buf`. `buf` should contain at least `310+max(0,ndigit)` characters to accommodate any double-precision value.

The `gconvert()` function converts the value to a null-terminated ASCII string in `buf` and returns a pointer to `buf`. It produces `ndigit` significant digits in fixed-decimal format, like `sprintf(%w.nf)`, if possible, and otherwise in floating-decimal format, like `sprintf(%w.ne)`; in either case `buf` is ready for printing, with sign and exponent. The result corresponds to that obtained by

```c
(void) sprintf(buf, "%w.ng", value);
```

If `trailing = 0`, trailing zeros and a trailing point are suppressed, as in `sprintf(%g)`. If `trailing != 0`, trailing zeros and a trailing point are retained, as in `sprintf(%#g)`.
The `seconvert()`, `sfconvert()`, and `sgconvert()` functions are single-precision versions of these functions, and are more efficient than the corresponding double-precision versions. A pointer rather than the value itself is passed to avoid C’s usual conversion of single-precision arguments to double.

The `qeconvert()`, `qfconvert()`, and `qgconvert()` functions are quadruple-precision versions of these functions. The `qfconvert()` function can overflow the `decimal_record` field `ds` if `value` is too large. In that case, `buf[0]` is set to zero.

The `ecvt()`, `fcvt()` and `gcvt()` functions are versions of `econvert()`, `fconvert()`, and `gconvert()`, respectively, that are documented on the `ecvt(3C)` manual page. They constitute the default implementation of these functions and conform to the X/Open CAE Specification, System Interfaces and Headers, Issue 4, Version 2.

**IEEE Infinities and NaNs** are treated similarly by these functions. “NaN” is returned for NaN, and “Inf” or “Infinity” for Infinity. The longer form is produced when `ndigit` >= 8.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**See also** `ecvt(3C)`, `sprintf(3C)`, attributes(5)
The `ecvt()`, `fcvt()` and `gcvt()` functions convert floating-point numbers to null-terminated strings.

**ecvt()**

The `ecvt()` function converts `value` to a null-terminated string of `ndigit` digits (where `ndigit` is reduced to an unspecified limit determined by the precision of a `double`) and returns a pointer to the string. The high-order digit is non-zero, unless the value is 0. The low-order digit is rounded. The position of the radix character relative to the beginning of the string is stored in the integer pointed to by `decpt` (negative means to the left of the returned digits). The radix character is not included in the returned string. If the sign of the result is negative, the integer pointed to by `sign` is non-zero, otherwise it is 0.

If the converted value is out of range or is not representable, the contents of the returned string are unspecified.

**fcvt()**

The `fcvt()` function is identical to `ecvt()` except that `ndigit` specifies the number of digits desired after the radix point. The total number of digits in the result string is restricted to an unspecified limit as determined by the precision of a `double`.

**gcvt()**

The `gcvt()` function converts `value` to a null-terminated string (similar to that of the `%g` format of `printf(3C)`) in the array pointed to by `buf` and returns `buf`. It produces `ndigit` significant digits (limited to an unspecified value determined by the precision of a `double`) in `%f` if possible, or `%e` (scientific notation) otherwise. A minus sign is included in the returned string if `value` is less than 0. A radix character is included in the returned string if `value` is not a whole number. Trailing zeros are suppressed where `value` is not a whole number. The radix character is determined by the current locale. If `setlocale(3C)` has not been called successfully, the default locale, POSIX, is used. The default locale specifies a period (.) as the radix character. The `LC_NUMERIC` category determines the value of the radix character within the current locale.

**RETURN VALUES**

The `ecvt()` and `fcvt()` functions return a pointer to a null-terminated string of digits.

The `gcvt()` function returns `buf`.

**ERRORS**

No errors are defined.

**USAGE**

The return values from `ecvt()` and `fcvt()` may point to static data which may be overwritten by subsequent calls to these functions.

For portability to implementations conforming to earlier versions of this document, `sprintf(3C)` is preferred over this function.
ecvt(3C)

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Unsafe</td>
</tr>
</tbody>
</table>

SEE ALSO
printf(3C), setlocale(3C), sprintf(3C), attributes(5)
The `encrypt()` function provides (rather primitive) access to the hashing algorithm employed by the `crypt(3C)` function. The key generated by `setkey(3C)` is used to encrypt the string `block` with `encrypt()`. The `block` argument to `encrypt()` is an array of length 64 bytes containing only the bytes with numerical value of 0 and 1. The array is modified in place to a similar array using the key set by `setkey(3C)`. If `edflag` is 0, the argument is encoded. If `edflag` is 1, the argument may be decoded (see the `USAGE` section below); if the argument is not decoded, `errno` will be set to `ENOSYS`.

The `encrypt()` function returns no value.

The `encrypt()` function will fail if:

- `ENOSYS` The functionality is not supported on this implementation.

In some environments, decoding may not be implemented. This is related to U.S. Government restrictions on encryption and decryption routines: the DES decryption algorithm cannot be exported outside the U.S.A. Historical practice has been to ship a different version of the encryption library without the decryption feature in the routines supplied. Thus the exported version of `encrypt()` does encoding but not decoding.

Because `encrypt()` does not return a value, applications wishing to check for errors should set `errno` to 0, call `encrypt()`, then test `errno` and, if it is non-zero, assume an error has occurred.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO** `crypt(3C), setkey(3C), attributes(5)`
### NAME
end, _end, etext, _etext, edata, _edata – last locations in program

### SYNOPSIS
```c
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 location after the program text.
- `_edata` The address of `_edata` is the first location after the initialized data region.
- `_end` The address of `_end` is the first location after the uninitialized data region.

When execution begins, the program break (the first location beyond the data) coincides with `_end`, but the program break may be reset by the `brk(2)`, `malloc(3C)`, and the standard input/output library (see `stdio(3C)`), functions by the profile (-p) option of `cc(1B)`, and so on. Thus, the current value of the program break should be determined by `sbrk ((char *)0)`.

References to `end`, `etext`, and `edata`, without a preceding underscore will be aliased to the associated symbol that begins with the underscore.

### SEE ALSO
`cc(1B), brk(2), malloc(3C), stdio(3C)`
NAME
euclen, euccol, eucscol – get byte length and display width of EUC characters

SYNOPSIS
#include <euc.h>

int euclen(const unsigned char *s);
int euccol(const unsigned char *s);
int eucscol(const unsigned char *str);

DESCRIPTION
The euclen() function returns the length in bytes of the Extended Unix Code (EUC) character pointed to by s, including single-shift characters, if present.

The euccol() function returns the screen column width of the EUC character pointed to by s.

The eucscol() function returns the screen column width of the EUC string pointed to by str.

For the euclen() and euccol(), functions, s points to the first byte of the character. This byte is examined to determine its codeset. The character type table for the current locale is used for codeset byte length and display width information.

USAGE
These functions will work only with EUC locales.

These functions can be used safely in multithreaded applications, as long as setlocale(3C) is not called to change the locale.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
</tbody>
</table>

SEE ALSO
gewidth(3C), setlocale(3C), attributes(5)
NAME  
exit, _exithandle – terminate process

SYNOPSIS  
#include <stdlib.h>

void exit(int status);
void _exithandle(void);

DESCRIPTION  
The exit() function terminates a process by calling first _exithandle() and then _exit() (see exit(2)).

The _exithandle() function calls any functions registered through the atexit(3C) function in the reverse order of their registration. This action includes executing all finalization code from the _fini sections of all objects that are part of the process.

The _exithandle() function is intended for use only with _exit(), and allows for specialized processing such as dldump(3DL) to be performed. Normal process execution should not be continued after a call to _exithandle() has occurred, as internal data structures may have been torn down due to atexit() or _fini processing.

The symbols EXIT_SUCCESS and EXIT_FAILURE are defined in the header <stdlib.h> and may be used as the value of status to indicate successful or unsuccessful termination, respectively.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  
exit(2), atexit(3C), dldump(3DL), attributes(5)
NAME  
fattach – attach a STREAMS-based file descriptor to an object in the file system name space

SYNOPSIS  
#include <stropts.h>

int fattach(int fildes, const char *path);

DESCRIPTION  
The fattach() function attaches a STREAMS-based file descriptor to an object in the file system name space, effectively associating a name with fildes. fildes 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. fildes can be attached to more than one path, that is, 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 fildes. If any attributes of the named stream are subsequently changed (for example, chmod(2)), the attributes of the underlying object are not affected.

RETURN VALUES  
Upon successful completion, fattach() returns 0. Otherwise it returns −1 and sets errno to indicate an error.

ERRORS  
The fattach() function will fail if:

EACCES  
The user is the owner of path but does not have write permissions on path or fildes is locked.

EBADF  
The fildes argument is not a valid open file descriptor.

EBUSY  
The path argument is currently a mount point or has a STREAMS file descriptor attached it.

EINVAL  
The path argument is a file in a remotely mounted directory.

EINVAL  
The fildes argument does not represent a STREAMS file.

ELOOP  
Too many symbolic links were encountered in translating path.

ENAMETOOLONG  
The size of path exceeds {PATH_MAX}, or the component of a path name is longer than {NAME_MAX} while {__POSIX_NO_TRUNC} is in effect.

ENOENT  
The path argument does not exist.

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

EPERM  
The effective user ID is not the owner of path or a user with the appropriate privileges.
fattach(3C)

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  fdetach(1M), chmod(2), mount(2), stat(2), fdetach(3C), isastream(3C), attributes(5), streamio(7I)

STREAMS Programming Guide
### NAME
__fbufsize__, __flbf__, __fpending__, __fpurge__, __freadable__, __freading__, __fsetlocking__, __fwritable__, __fwriting__, _flushlbf_ - interfaces to stdio FILE structure

### SYNOPSIS

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

size_t __fbufsize(FILE *stream);
int __flbf(FILE *stream);
size_t __fpending(FILE *stream);
void __fpurge(FILE *stream);
int __freadable(FILE *stream);
int __freading(FILE *stream);
int __fsetlocking(FILE *stream, int type);
int __fwritable(FILE *stream);
int __fwriting(FILE *stream);
void _flushlbf(void);
```

### DESCRIPTION
These functions provide portable access to the members of the stdio(3C) FILE structure.

The __fbufsize() function returns in bytes the size of the buffer currently in use by the given stream.

The __flbf() function returns non-zero if the stream is line-buffered.

The __fpending function returns in bytes the amount of output pending on a stream.

The __fpurge() function discards any pending buffered I/O on the stream.

The __freadable() function returns non-zero if it is possible to read from a stream.

The __freading() function returns non-zero if the file is open readonly, or if the last operation on the stream was a read operation such as fread(3C) or fgetc(3C). Otherwise it returns 0.

The __fsetlocking() function allows the type of locking performed by stdio on a given stream to be controlled by the programmer.

If type is FSETLOCKING_INTERNAL, stdio performs implicit locking around every operation on the given stream. This is the default system behavior on that stream.

If type is FSETLOCKING_BYCALLER, stdio assumes that the caller is responsible for maintaining the integrity of the stream in the face of access by multiple threads. If there is only one thread accessing the stream, nothing further needs to be done. If multiple threads are accessing the stream, then the caller can use the flockfile().
funlockfile(), and ftrylockfile() functions described on the flockfile(3C) manual page to provide the appropriate locking. In both this and the case where type is FSETLOCKING_INTERNAL, __fsetlocking() returns the previous state of the stream.

If type is FSETLOCKING_QUERY, __fsetlocking() returns the current state of the stream without changing it.

The __fwriteable() function returns non-zero if it is possible to write on a stream.

The __fwriteing() function returns non-zero if the file is open write-only or append-only, or if the last operation on the stream was a write operation such as fwrite(3C) or fputc(3C). Otherwise it returns 0.

The _flushlbf() function flushes all line-buffered files. It is used when reading from a line-buffered file.

**USAGE**

Although the contents of the stdio FILE structure have always been private to the stdio implementation, some applications have needed to obtain information about a stdio stream that was not accessible through a supported interface. These applications have resorted to accessing fields of the FILE structure directly, rendering them possibly non-portable to new implementations of stdio, or more likely, preventing enhancements to stdio that would cause those applications to break.

In the 64-bit environment, the FILE structure is opaque. The functions described here are provided as a means of obtaining the information that up to now has been retrieved directly from the FILE structure. Because they are based on the needs of existing applications (such as mh and emacs), they may be extended as other programs are ported. Although they may still be non-portable to other operating systems, they will be compatible from each Solaris release to the next. Interfaces that are more portable are under development.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>__fsetlocking() is Unsafe; all others are MT-Safe</td>
</tr>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
</tbody>
</table>

**SEE ALSO**

fgetc(3C), flockfile(3C), fputc(3C), fread(3C), fwrite(3C), stdio(3C), attributes(5)
fclose – close a stream

#include <stdio.h>

int fclose(FILE *stream);

The fclose() function causes the stream pointed to by stream to be flushed and the associated file to be closed. Any unwritten buffered data for the stream is written to the file; any unread buffered data is discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated.

The fclose() function marks for update the st_ctime and st_mtime fields of the underlying file if the stream is writable and if buffered data has not yet been written to the file. It will perform a close(2) operation on the file descriptor that is associated with the stream pointed to by stream.

After the call to fclose(), any use of stream causes undefined behavior.

The fclose() function is performed automatically for all open files upon calling exit(2).

Upon successful completion, fclose() returns 0. Otherwise, it returns EOF and sets errno to indicate the error.

The fclose() function will fail if:

- EAGAIN The O_NONBLOCK flag is set for the file descriptor underlying stream and the process would be delayed in the write operation.
- EBADF The file descriptor underlying stream is not valid.
- EFBIG An attempt was made to write a file that exceeds the maximum file size or the process’s file size limit; or the file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
- EINTR The fclose() function was interrupted by a signal.
- EIO The process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU and the process group of the process is orphaned.
- ENOSPC There was no free space remaining on the device containing the file.
- EPIPE An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal will also be sent to the process.

The fclose() function may fail if:

- ENXIO A request was made of a non-existent device, or the request was beyond the limits of the device.
fclose(3C)

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO close(2), exit(2), getrlimit(2), ulimit(2), fopen(3C), stdio(3C), attributes(5)
NAME
fdetach – detach a name from a STREAMS-based file descriptor

SYNOPSIS
#include <stropts.h>

int fdetach(const char *path);

DESCRIPTION
The fdetach() function detaches a STREAMS-based file from the file to which it was
attached by a previous call to fattach(3C). The path argument points to the
pathname of the attached STREAMS file. The process must have appropriate
privileges or be the owner of the file. A successful call to fdetach() causes all
pathnames that named the attached STREAMS file to again name the file to which
the STREAMS file was attached. All subsequent operations on path will operate on the
underlying file and not on the STREAMS file.

All open file descriptions established while the STREAMS file was attached to the file
referenced by path, will still refer to the STREAMS file after the fdetach() has taken
effect.

If there are no open file descriptors or other references to the STREAMS file, then a
successful call to fdetach() has the same effect as performing the last close(2) on
the attached file.

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

ERRORS
The fdetach() function will fail if:

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

EPERM The effective user ID is not the owner of path and the
process does not have appropriate privileges.

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

ENOENT A component of path does not name an existing file or
path is an empty string.

EINVAL The path argument names a file that is not currently
attached.

ENAMETOOLONG The size of a pathname exceeds PATH_MAX, or a
pathname component is longer than NAME_MAX while
_POSIX_NO_TRUNC is in effect.

ELOOP Too many symbolic links were encountered in
resolving path.

The fdetach() function may fail if:

ENAMETOOLONG Pathname resolution of a symbolic link produced an
intermediate result whose length exceeds PATH_MAX.

SEE ALSO fdetach(1M), close(2), fattach(3C), streamio(7I)
fdopen – associate a stream with a file descriptor

SYNOPSIS

```c
#include <stdio.h>

FILE *fdopen(int fildes, const char *mode);
```

DESCRIPTION

The `fdopen()` function associates a stream with a file descriptor `fildes`.

The `mode` argument is a character string having one of the following values:

- `r` or `rb` Open a file for reading.
- `w` or `wb` Open a file for writing.
- `a` or `ab` Open a file for writing at end of file.
- `r+` or `rb+` or `r+b` Open a file for update (reading and writing).
- `w+` or `wb+` or `w+b` Open a file for update (reading and writing).
- `a+` or `ab+` or `a+b` Open a file for update (reading and writing) at end of file.

The meaning of these flags is exactly as specified for the `fopen(3C)` function, except that modes beginning with `w` do not cause truncation of the file.

The mode of the stream must be allowed by the file access mode of the open file. The file position indicator associated with the new stream is set to the position indicated by the file offset associated with the file descriptor.

The `fdopen()` function preserves the offset maximum previously set for the open file description corresponding to `fildes`.

The error and end-of-file indicators for the stream are cleared. The `fdopen()` function may cause the `st_atime` field of the underlying file to be marked for update.

If `fildes` refers to a shared memory object, the result of the `fdopen()` function is unspecified.

RETURN VALUES

Upon successful completion, `fdopen()` returns a pointer to a stream. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

The `fdopen()` function may fail and not set `errno` if there are no free `stdio` streams.

ERRORS

The `fdopen()` function may fail if:

- `EBADF` The `fildes` argument is not a valid file descriptor.
- `EINVAL` The `mode` argument is not a valid mode.
- `EMFILE` The number of streams currently open in the calling process is either `FOPEN_MAX` or `STREAM_MAX`.

Basic Library Functions 115
The number of streams that a process can have open at one time is `STREAM_MAX`. If defined, it has the same value as `FOPEN_MAX`.

File descriptors are obtained from calls like `open(2)`, `dup(2)`, `creat(2)` or `pipe(2)`, which open files but do not return streams. Streams are necessary input for almost all of the Section 3S library routines.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

`SEE ALSO` `creat(2)`, `dup(2)`, `open(2)`, `pipe(2)`, `fclose(3C)`, `fopen(3C)`, `attributes(5)`
NAME
ferror, feof, clearerr, fileno – stream status inquiries

SYNOPSIS
#include <stdio.h>

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

DESCRIPTION
The ferror() function returns a non-zero value when an error has previously occurred reading from or writing to the named stream (see intro(3)). It returns 0 otherwise.

The feof() function returns a non-zero value when EOF has previously been detected reading the named input stream. It returns 0 otherwise.

The clearerr() function resets the error indicator and EOF indicator to 0 on the named stream.

The fileno() function returns the integer file descriptor associated with the named stream; see open(2).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
open(2), intro(3), fopen(3C), stdio(3C), attributes(5)
flush(3C)

NAME  fflush – flush a stream
SYNOPSIS  
#include <stdio.h>
int fflush(FILE *stream);

DESCRIPTION  If stream points to an output stream or an update stream in which the most recent operation was not input, fflush() causes any unwritten data for that stream to be written to the file, and the st_ctime and st_mtime fields of the underlying file are marked for update.

If stream is a null pointer, fflush() performs this flushing action on all streams for which the behavior is defined above. Additionally, an input stream or an update stream into which the most recent operation was input is also flushed if it is seekable and is not already at end-of-file. Flushing an input stream discards any buffered input and adjusts the file pointer such that the next input operation accesses the byte after the last one read. A stream is seekable if the underlying file is not a pipe, FIFO, socket, or TTY device. An input stream, seekable or non-seekable, can be flushed by explicitly calling fflush() with a non-null argument specifying that stream.

RETURN VALUES  Upon successful completion, fflush() returns 0. Otherwise, it returns EOF and sets errno to indicate the error.

ERRORS  The fflush() function will fail if:

EAGAIN  The O_NONBLOCK flag is set for the file descriptor underlying stream and the process would be delayed in the write operation.
EBADF  The file descriptor underlying stream is not valid.
EFBIG  An attempt was made to write a file that exceeds the maximum file size or the process's file size limit; or the file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
EINTR  The fflush() function was interrupted by a signal.
EIO  The process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned.
ENOSPC  There was no free space remaining on the device containing the file.
EPIPE  An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal will also be sent to the process.

The fflush() function may fail if:

ENXIO  A request was made of a non-existent device, or the request was beyond the limits of the device.
ATTRIBUTES See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO getrlimit(2), ulimit(2), attributes(5)
NAME
ffs – find first set bit

SYNOPSIS
#include <strings.h>

int ffs(const int i);

DESCRIPTION
The ffs() function finds the first bit set (beginning with the least significant bit) and returns the index of that bit. Bits are numbered starting at one (the least significant bit).

RETURN VALUES
The ffs() function returns the index of the first set bit. If i is 0, then ffs() returns 0.

ERRORS
No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5)
The `fgetc()` function obtains the next byte (if present) as an `unsigned char` converted to an `int`, from the input stream pointed to by `stream`, and advances the associated file position indicator for the stream (if defined).

The `fgetc()` function may mark the `st_atime` field of the file associated with `stream` for update. The `st_atime` field will be marked for update by the first successful execution of `fgetc()`, `fgets(3C)`, `fgetwc(3C)`, `fgetws(3C)`, `fread(3C)`, `fscanf(3C)`, `getc()`, `getchar()`, `gets(3C)` or `scanf(3C)` using `stream` that returns data not supplied by a prior call to `ungetc(3C)` or `ungetwc(3C)`.

The `getc()` routine is functionally identical to `fgetc()`, except that it is implemented as a macro. It runs faster than `fgetc()`, but it takes up more space per invocation and its name cannot be passed as an argument to a function call.

The `getchar()` routine is equivalent to `getc(stdin)`. It is implemented as a macro.

The `getc_unlocked()` and `getchar_unlocked()` routines are variants of `getc()` and `getchar()`, respectively, that do not lock the stream. It is the caller’s responsibility to acquire the stream lock before calling these routines and releasing the lock afterwards; see `flockfile(3C)` and `stdio(3C)`. These routines are implemented as macros.

The `getw()` function reads the next word from the `stream`. The size of a word is the size of an `int` and may vary from environment to environment. The `getw()` function presumes no special alignment in the file.

The `getw()` function may mark the `st_atime` field of the file associated with `stream` for update. The `st_atime` field will be marked for update by the first successful execution of `fgetc()`, `fgets(3C)`, `fread(3C)`, `getc()`, `getchar()`, `gets(3C)`, `fscanf(3C)` or `scanf(3C)` using `stream` that returns data not supplied by a prior call to `ungetc(3C)`.

### RETURN VALUES
Upon successful completion, `fgetc()`, `getc()`, `getc_unlocked()`, `getchar()`, `getchar_unlocked()`, and `getw()` return the next byte from the input stream pointed to by `stream`. If the stream is at end-of-file, the end-of-file indicator for the stream is set and these functions return `EOF`. If a read error occurs, the error indicator for the stream is set, `EOF` is returned, and `errno` is set to indicate the error.
The `fgetc()`, `getc()`, `getc_unlocked()`, `getchar()`, `getchar_unlocked()`, and `getw()` functions will fail if data needs to be read and:

- **EAGAIN** The O_NONBLOCK flag is set for the file descriptor underlying `stream` and the process would be delayed in the `fgetc()` operation.

- **EBADF** The file descriptor underlying `stream` is not a valid file descriptor open for reading.

- **EINVAL** The read operation was terminated due to the receipt of a signal, and no data was transferred.

- **EIO** A physical I/O error has occurred, or the process is in a background process group attempting to read from its controlling terminal, and either the process is ignoring or blocking the `SIGTTIN` signal or the process group is orphaned. This error may also be generated for implementation-dependent reasons.

- **EOVERFLOW** The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

The `fgetc()`, `getc()`, `getc_unlocked()`, `getchar()`, `getchar_unlocked()`, and `getw()` functions may fail if:

- **ENOMEM** Insufficient storage space is available.

- **ENXIO** A request was made of a non-existent device, or the request was outside the capabilities of the device.

** USAGE **

If the integer value returned by `fgetc()`, `getc()`, `getc_unlocked()`, `getchar()`, `getchar_unlocked()`, and `getw()` is stored into a variable of type `char` and then compared against the integer constant `EOF`, the comparison may never succeed, because sign-extension of a variable of type `char` on widening to integer is implementation-dependent.

The `ferror(3C)` or `feof(3C)` functions must be used to distinguish between an error condition and an end-of-file condition.

Functions exist for the `getc()`, `getc_unlocked()`, `getchar()`, and `getchar_unlocked()` macros. To get the function form, the macro name must be undefined (for example, `#undef getc`).

When the macro forms are used, `getc()` and `getc_unlocked()` evaluate the `stream` argument more than once. In particular, `getc(*f++)` does not work sensibly. The `fgetc()` function should be used instead when evaluating the `stream` argument has side effects.

Because of possible differences in word length and byte ordering, files written using `getw()` are machine-dependent, and may not be read using `getw()` on a different processor.
The `getw()` function is inherently byte stream-oriented and is not tenable in the context of either multibyte character streams or wide-character streams. Application programmers are recommended to use one of the character-based input functions instead.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See NOTES below.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`intro(3), fclose(3C), feof(3C), fgets(3C), fgetwc(3C), fgetws(3C), flockfile(3C), fopen(3C), fread(3C), fscanf(3C), gets(3C), putc(3C), scanf(3C), stdio(3C), ungetc(3C), ungetwc(3C), attributes(5)`

**NOTES**

The `fgetc()`, `getc()`, `getchar()`, and `getw()` routines are MT-Safe in multithreaded applications. The `getc_unlocked()` and `getchar_unlocked()` routines are unsafe in multithreaded applications.
fgetpos(3C)

NAME     fgetpos – get current file position information

SYNOPSIS #include <stdio.h>

   int fgetpos(FILE *stream, fpos_t *pos);

DESCRIPTION The fgetpos() function stores the current value of the file position indicator for the stream pointed to by stream in the object pointed to by pos. The value stored contains unspecified information usable by fsetpos(3C) for repositioning the stream to its position at the time of the call to fgetpos().

RETURN VALUES Upon successful completion, fgetpos() returns 0. Otherwise, it returns a non-zero value and sets errno to indicate the error.

ERRORS The fgetpos() function may fail if:

   EBADF       The file descriptor underlying stream is not valid.
   ESPIPE      The file descriptor underlying stream is associated with a pipe, a FIFO, or a socket.
   EOVERFLOW   The current value of the file position cannot be represented correctly in an object of type fpos_t.

USAGE The fgetpos() function has a transitional interface for 64-bit file offsets. See lff64(5).

SEE ALSO fopen(3C), fsetpos(3C), ftell(3C), rewind(3C), ungetc(3C), lff64(5)
fgetwc(3C)

NAME
fgetwc – get a wide-character code from a stream

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

wint_t fgetwc(FILE*stream);
```

DESCRIPTION
The `fgetwc()` function obtains the next character (if present) from the input stream pointed to by `stream`, converts that to the corresponding wide-character code and advances the associated file position indicator for the stream (if defined).

If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

The `fgetwc()` function may mark the `st_atime` field of the file associated with `stream` for update. The `st_atime` field will be marked for update by the first successful execution of `fgetwc()`, `fgetc(3C)`, `fgets(3C)`, `fgetws(3C)`, `fread(3C)`, `fscanf(3C)`, `getc(3C)`, `getchar(3C)`, `gets(3C)`, or `scanf(3C)` using `stream` that returns data not supplied by a prior call to `ungetc(3C)` or `ungetwc(3C)`.

RETURN VALUES
Upon successful completion the `fgetwc()` function returns the wide-character code of the character read from the input stream pointed to by `stream` converted to a type `wint_t`.

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 `WEOF` and sets `errno` to indicate the error.

ERRORS
The `fgetwc()` function will fail if data needs to be read and:

- **EAGAIN** The `O_NONBLOCK` flag is set for the file descriptor underlying `stream` and the process would be delayed in the `fgetwc()` operation.
- **EBADF** The file descriptor underlying `stream` is not a valid file descriptor open for reading.
- **EINTR** The read operation was terminated due to the receipt of a signal, and no data was transferred.
- **EIO** A physical I/O error has occurred, or the process is in a background process group attempting to read from its controlling terminal and either the process is ignoring or blocking the `SIGTTOINE` signal or the process group is orphaned.
- **EOVERFLOW** The file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding `stream`.

The `fgetwc()` function may fail if:
fgetwc(3C)

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.

The ferror(3C) or feof(3C) functions must be used to distinguish between an error condition and an end-of-file condition.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO ferror(3C), feof(3C), fgetc(3C), fgets(3C), fgetws(3C), fopen(3C), fread(3C), fscanf(3C), getc(3C), getchar(3C), gets(3C), scanf(3C), setlocale(3C), ungetc(3C), ungetwc(3C), attributes(5)
The `floating_to_decimal()` functions convert the floating-point value at `*px` into a decimal record at `*pd`, observing the modes specified in `*pm` and setting exceptions in `*ps`. If there are no IEEE exceptions, `*ps` will be zero.

If `*px` is zero, infinity, or NaN, then only `pd->sign` and `pd->fpclass` are set. Otherwise `pd->exponent` and `pd->ds` are also set so that

\[(\text{sig}) \times (\text{pd->ds}) \times 10^{\text{pd->exponent}}\]

is a correctly rounded approximation to `*px`, where `sig` is +1 or -1, depending upon whether `pd->sign` is 0 or -1. `pd->ds` has at least one and no more than `DECIMAL_STRING_LENGTH-1` significant digits because one character is used to terminate the string with a NULL.

`pd->ds` is correctly rounded according to the IEEE rounding modes in `pm->rd`. `*ps` has `fp_inexact` set if the result was inexact, and has `fp_overflow` set if the string result does not fit in `pd->ds` because of the limitation `DECIMAL_STRING_LENGTH`.

If `pm->df == floating_form`, then `pd->ds` always contains `pm->ndigits` significant digits. Thus if `*px == 12.34` and `pm->ndigits == 8`, then `pd->ds` will contain 12340000 and `pd->exponent` will contain -6.

If `pm->df == fixed_form` and `pm->ndigits >= 0`, then `pd->ds` always contains `pm->ndigits` after the point and as many digits as necessary before the point. Since the latter is not known in advance, the total number of digits required is returned in `pd->ndigits`; if that number >= `DECIMAL_STRING_LENGTH`, then `ds` is undefined. `pd->exponent` always gets `-pm->ndigits`. Thus if `*px == 12.34` and `pm->ndigits == 1`, then `pd->ds` gets 123, `pd->exponent` gets -1, and `pd->ndigits` gets 3.

If `pm->df == fixed_form` and `pm->ndigits < 0`, then `pd->ds` always contains `-pm->ndigits` trailing zeros; in other words, rounding occurs `-pm->ndigits` to the left of the decimal point, but the digits rounded away are retained as zeros. The total number of digits required is in `pd->ndigits`. `pd->exponent` always gets 0. Thus if `*px == 12.34` and `pm->ndigits == -1`, then `pd->ds` gets 10, `pd->exponent` gets 0, and `pd->ndigits` gets 2.

`pd->more` is not used.
econvert(3C), fconvert(3C), gconvert(3C), printf(3C), and sprintf(3C) all use double_to_decimal().

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

econvert(3C), fconvert(3C), gconvert(3C), printf(3C), sprintf(3C), attributes(5)
NAME  flock – apply or remove an advisory lock on an open file

SYNOPSIS  
/usr/ucb/cc/flag ... file ...
#include <sys/file.h>

int flock(fd, operation);
int fd, operation;

DESCRIPTION  flock() applies or removes an advisory lock on the file associated with the file descriptor fd. The compatibility version of flock() has been implemented on top of fcntl(2) locking. It does not provide complete binary compatibility.

Advisory locks allow cooperating processes to perform consistent operations on files, but do not guarantee exclusive access (that is, processes may still access files without using advisory locks, possibly resulting in inconsistencies).

The locking mechanism allows two types of locks: shared locks and exclusive locks. More than one process may hold a shared lock for a file at any given time, but multiple exclusive, or both shared and exclusive, locks may not exist simultaneously on a file.

A lock is applied by specifying an operation parameter LOCK_SH for a shared lock or LOCK_EX for an exclusive lock. The operation parameter may be ORed with LOCK_NB to make the operation non-blocking. To unlock an existing lock, the operation should be LOCK_UN.

Read permission is required on a file to obtain a shared lock, and write permission is required to obtain an exclusive lock. Locking a segment that is already locked by the calling process causes the old lock type to be removed and the new lock type to take effect.

Requesting a lock on an object that is already locked normally causes the caller to block until the lock may be acquired. If LOCK_NB is included in operation, then this will not happen; instead, the call will fail and the error EWOULDBLOCK will be returned.

RETURN VALUES  flock() returns:
0 on success.
−1 on failure and sets errno to indicate the error.

ERRORS  
EBADF The argument fd is an invalid descriptor.
EINVAL operation is not a valid argument.
EOPNOTSUPP The argument fd refers to an object other than a file.
EWOULDBLOCK The file is locked and the LOCK_NB option was specified.

SEE ALSO  lockd(1M), chmod(2), close(2), dup(2), exec(2), fcntl(2), fork(2), open(2), lockf(3C)
flock(3UCB)

NOTES  Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

Locks are on files, not file descriptors. That is, file descriptors duplicated through `dup(2)` or `fork(2)` do not result in multiple instances of a lock, but rather multiple references to a single lock. If a process holding a lock on a file forks and the child explicitly unlocks the file, the parent will lose its lock. Locks are not inherited by a child process.

Processes blocked awaiting a lock may be awakened by signals.

Mandatory locking may occur, depending on the mode bits of the file. See `chmod(2)`.

Locks obtained through the `flock()` mechanism under SunOS 4.1 were known only within the system on which they were placed. This is no longer true.
## NAME
flockfile, funlockfile, ftrylockfile – acquire and release stream lock

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

void flockfile(FILE *stream);
void funlockfile(FILE *stream);
int ftrylockfile(FILE *stream);
```

## DESCRIPTION
The `flockfile()` function acquires an internal lock of a stream `stream`. If the lock is already acquired by another thread, the thread calling `flockfile()` is suspended until it can acquire the lock. In the case that the stream lock is available, `flockfile()` not only acquires the lock, but keeps track of the number of times it is being called by the current thread. This implies that the stream lock can be acquired more than once by the same thread.

The `funlockfile()` function releases the lock being held by the current thread. In the case of recursive locking, this function must be called the same number of times `flockfile()` was called. After the number of `funlockfile()` calls is equal to the number of `flockfile()` calls, the stream lock is available for other threads to acquire.

The `ftrylockfile()` function acquires an internal lock of a stream `stream`, only if that object is available. In essence `ftrylockfile()` is a non-blocking version of `flockfile()`.

## RETURN VALUES
The `ftrylockfile()` function returns 0 on success and non-zero to indicate a lock cannot be acquired.

## EXAMPLES
**EXAMPLE 1** A sample program of `flockfile()`.

The following example prints everything out together, blocking other threads that might want to write to the same file between calls to `fprintf(3C)`:

```
FILE iop;
    flockfile(iop);
    fprintf(iop, "hello ");
    fprintf(iop, " world");
    fputc(iop, 'a');
    funlockfile(iop);
```

An unlocked interface is available in case performance is an issue. For example:

```
flockfile(iop);
    while (!feof(iop)) {
        *c++ = getc_unlocked(iop);
    }
    funlockfile(iop);
```
flockfile(3C)

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

intro(3), ferror(3C), fprintf(3C), getc(3C), putc(3C), stdio(3C), ungetc(3C), attributes(5), standards(5)

NOTES

The interfaces on this page are as specified in IEEE Std 1003.1c. See standards(5).
fmtmsg(3C)

NAME
fmtmsg – display a message on stderr or system console
SYNOPSIS
#include <fmtmsg.h>

int fmtmsg(long classification, const char *label, int severity, const char *
text, const char *action, const char *tag);

DESCRIPTION
The fmtmsg() function writes a formatted message to stderr, to the console, or to both, on a message’s classification component. It can be used instead of the traditional printf(3C) interface to display messages to stderr, and in conjunction with gettext(3C), provides a simple interface for producing language-independent applications.

A formatted message consists of up to five standard components (label, severity, text, action, and tag) as described below. The classification component is not part of the standard message displayed to the user, but rather defines the source of the message and directs the display of the formatted message.

classification
Contains identifiers from the following groups of major classifications and subclassifications. Any one identifier from a subclass may be used in combination by ORing the values together with a single identifier from a different subclass. Two or more identifiers from the same subclass should not be used together, with the exception of identifiers from the display subclass. (Both display subclass identifiers may be used so that messages can be displayed to both stderr and the system console).

- “Major classifications” identify the source of the condition. Identifiers are: MM_HARD (hardware), MM_SOFT (software), and MM_FIRM (firmware).
- “Message source subclassifications” identify the type of software in which the problem is spotted. Identifiers are: MM_APPL (application), MM_UTIL (utility), and MM_OPSYS (operating system).
- “Display subclassifications” indicate where the message is to be displayed. Identifiers are: MM_PRINT to display the message on the standard error stream, MM_CONSOLE to display the message on the system console. Neither, either, or both identifiers may be used.
- “Status subclassifications” indicate whether the application will recover from the condition. Identifiers are: MM_RECOVER (recoverable) and MM_NRECOV (non-recoverable).
- An additional identifier, MM_NULLMC, indicates that no classification component is supplied for the message.

label
Identifies the source of the message. The format of this component is two fields separated by a colon. The first field is up to 10 characters long; the second is up to 14 characters. Suggested usage is that label identifies the package in which the application resides.
as well as the program or application name. For example, the label
UX:cat indicates the UNIX System V package and the cat(1)
utility.

severity Indicates the seriousness of the condition. Identifiers for the
standard levels of severity are:
- MM_HALT indicates that the application has encountered a
  severe fault and is halting. Produces the print string HALT.
- MM_ERROR indicates that the application has detected a fault.
  Produces the print string ERROR.
- MM_WARNING indicates a condition out of the ordinary that
  might be a problem and should be watched. Produces the print
  string WARNING.
- MM_INFO provides information about a condition that is not in
  error. Produces the print string INFO.
- MM_NOSEV indicates that no severity level is supplied for the
  message.

Other severity levels may be added by using the addseverity() routine.

text Describes the condition that produced the message. The text string
is not limited to a specific size.

action Describes the first step to be taken in the error recovery process.
fmtmsg() precedes each action string with the prefix: TOFIX:

The action string is not limited to a specific size.

tag An identifier which references on-line documentation for the
message. Suggested usage is that tag includes the label and a
unique identifying number. A sample tag is UX:cat:146.

Environment Variables The MSGVERB and SEV_LEVEL environment variables control the behavior of
fmtmsg() as follows:

MSGVERB This variable determines which message components fmtmsg() selects when writing messages to stderr. Its value is a
colon-separated list of optional keywords and can be set as follows:

```
MSGVERB= [keyword[:keyword[: . .]]]
export MSGVERB
```

Valid keywords are: label, severity, text, action, and tag. If
MSGVERB contains a keyword for a component and the
component's value is not the component's null value, fmtmsg() includes that component in the message when writing the message
to stderr. If MSGVERB does not include a keyword for a message
component, that component is not included in the display of the
message. The keywords may appear in any order. If MSGVERB is
not defined, if its value is the null string, if its value is not of the correct format, or if it contains keywords other than the valid ones listed above, fmtmsg() selects all components.

The first time fmtmsg() is called, it examines MSGVERB to determine which message components are to be selected when generating a message to write to the standard error stream, stderr. The values accepted on the initial call are saved for future calls.

The MSGVERB environment variable affects only those components that are selected for display to the standard error stream. All message components are included in console messages.

SEV_LEVEL

This variable defines severity levels and associates print strings with them for use by fmtmsg(). The standard severity levels listed below cannot be modified. Additional severity levels can also be defined, redefined, and removed using addseverity() (see addseverity(3C)). If the same severity level is defined by both SEV_LEVEL and addseverity(), the definition by addseverity() takes precedence.

0  (no severity is used)
1   HALT
2   ERROR
3   WARNING
4   INFO

The SEV_LEVEL variable can be set as follows:

```
SEV_LEVEL= [description[:description[: . . .]]]
export SEV_LEVEL
```

where description is a comma-separated list containing three fields:

description=severity_keyword,level,printstring

The severity_keyword field is a character string that is used as the keyword on the -s severity option to the fmtmsg(1) utility. (This field is not used by the fmtmsg() function.)

The level field is a character string that evaluates to a positive integer (other than 0, 1, 2, 3, or 4, which are reserved for the standard severity levels). If the keyword severity_keyword is used, level is the severity value passed on to the fmtmsg() function.
The `printstring` field is the character string used by `fmtmsg()` in the standard message format whenever the severity value `level` is used.

If a `description` in the colon list is not a three-field comma list, or if the second field of a comma list does not evaluate to a positive integer, that `description` in the colon list is ignored.

The first time `fmtmsg()` is called, it examines the `SEV_LEVEL` environment variable, if defined, to determine whether the environment expands the levels of severity beyond the five standard levels and those defined using `addseverity()`.

The values accepted on the initial call are saved for future calls.

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.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Type</th>
<th>Null-Value</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>label</code></td>
<td><code>char*</code></td>
<td><code>(char*) NULL</code></td>
<td><code>MM_NULLLBL</code></td>
</tr>
<tr>
<td><code>severity</code></td>
<td><code>int</code></td>
<td><code>0</code></td>
<td><code>MM_NULLSEV</code></td>
</tr>
<tr>
<td><code>class</code></td>
<td><code>long</code></td>
<td><code>0L</code></td>
<td><code>MM_NULLMC</code></td>
</tr>
<tr>
<td><code>text</code></td>
<td><code>char*</code></td>
<td><code>(char*) NULL</code></td>
<td><code>MM_NULLTXT</code></td>
</tr>
<tr>
<td><code>action</code></td>
<td><code>char*</code></td>
<td><code>(char*) NULL</code></td>
<td><code>MM_NULLACT</code></td>
</tr>
<tr>
<td><code>tag</code></td>
<td><code>char*</code></td>
<td><code>(char*) NULL</code></td>
<td><code>MM_NULLTAG</code></td>
</tr>
</tbody>
</table>

Another means of systematically omitting a component is by omitting the component keyword(s) when defining the `MSGVERB` environment variable (see the `Environment Variables` section above).

**RETURN VALUES**

The `fmtmsg()` returns the following values:

- `MM_OK` The function succeeded.
- `MM_NOTOK` The function failed completely.
- `MM_NOMSG` The function was unable to generate a message on the standard error stream, but otherwise succeeded.
- `MM_NOCON` The function was unable to generate a console message, but otherwise succeeded.
The following example of `fmtmsg()`:

```c
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

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

When the environment variable `SEV_LEVEL` is set as follows:

```
SEV_LEVEL=note,5,NOTE
```

the following call to `fmtmsg()`

```c
fmtmsg(MM_UTIL | MM_PRINT, "UX:cat", 5, "invalid syntax", "refer to manual", "UX:cat:001")
```

produces

UX:cat: NOTE: invalid syntax
TO FIX: refer to manual UX:cat:001

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

fmtmsg(1), addseverity(3C), gettxt(3C), printf(3C), attributes(5)
The `fnmatch()` function matches patterns as described on the `fnmatch(5)` manual page. It checks the `string` argument to see if it matches the `pattern` argument.

The `flags` argument modifies the interpretation of `pattern` and `string`. It is the bitwise inclusive OR of zero or more of the following flags defined in the header `<fnmatch.h>`.

- **FNM_PATHNAME**: If set, a slash (\/) character in `string` will be explicitly matched by a slash in `pattern`; it will not be matched by either the asterisk (*) or question-mark (?) special characters, nor by a bracket ([ ] ) expression.
  
  If not set, the slash character is treated as an ordinary character.

- **FNM_NOESCAPE**: If not set, a backslash character (\) in `pattern` followed by any other character will match that second character in `string`. In particular, “\" will match a backslash in `string`.
  
  If set, a backslash character will be treated as an ordinary character.

- **FNM_PERIOD**: If set, a leading period in `string` will match a period in `pattern`; where the location of “leading” is indicated by the value of `FNM_PATHNAME`:
  - If `FNM_PATHNAME` is set, a period is “leading” if it is the first character in `string` or if it immediately follows a slash.
  - If `FNM_PATHNAME` is not set, a period is “leading” only if it is the first character of `string`.

  If not set, no special restrictions are placed on matching a period.

If `string` matches the pattern specified by `pattern`, then `fnmatch()` returns 0. If there is no match, `fnmatch()` returns `FNM_NOMATCH`, which is defined in the header `<fnmatch.h>`. If an error occurs, `fnmatch()` returns another non-zero value.

The `fnmatch()` function has two major uses. It could be used by an application or utility that needs to read a directory and apply a pattern against each entry. The `find(1)` utility is an example of this. It can also be used by the `pax(1)` utility to process its `pattern` operands, or by applications that need to match strings in a similar manner.
The name `fnmatch()` is intended to imply _filename_ match, rather than _pathname_ match. The default action of this function is to match filenames, rather than path names, since it gives no special significance to the slash character. With the `FNM_PATHNAME` flag, `fnmatch()` does match path names, but without tilde expansion, parameter expansion, or special treatment for period at the beginning of a filename.

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

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`find(1), pax(1), glob(3C), setlocale(3C), wordexp(3C), attributes(5), fnmatch(5)"
fopen(3C)

NAME  
fopen — open a stream

SYNOPSIS  
#include <stdio.h>

FILE *
  fopen(const char *filename, const char *mode);

DESCRIPTION  
The fopen() function opens the file whose pathname is the string pointed to by filename, and associates a stream with it.

The argument mode points to a string beginning with one of the following sequences:

- r or rb  Open file for reading.
- w or wb  Truncate to zero length or create file for writing.
- a or ab  Append; open or create file for writing at end-of-file.
- r+ or rb+ or r+b  Open file for update (reading and writing).
- w+ or wb+ or w+b  Truncate to zero length or create file for update.
- a+ or ab+ or a+b  Append; open or create file for update, writing at end-of-file.

The character b has no effect, but is allowed for ISO C standard conformance (see standards(5)). Opening a file with read mode (r as the first character in the mode argument) fails if the file does not exist or cannot be read.

Opening a file with append mode (a as the first character in the mode argument) causes all subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening calls to fseek(3C). If two separate processes open the same file for append, each process may write freely to the file without fear of destroying output being written by the other. The output from the two processes will be intermixed in the file in the order in which it is written.

When a file is opened with update mode (+ as the second or third character in the mode argument), both input and output may be performed on the associated stream. However, output must not be directly followed by input without an intervening call to fflush(3C) or to a file positioning function (fseek(3C), fsetpos(3C) or rewind(3C)), and input must not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

If mode is w, a, w+ or a+ and the file did not previously exist, upon successful completion, fopen() function will mark for update the st_atime, st_ctime and st_mtime fields of the file and the st_ctime and st_mtime fields of the parent directory.
If `mode` is `w` or `w+` and the file did previously exist, upon successful completion, `fopen()` will mark for update the `st_ctime` and `st_mtime` fields of the file. The `fopen()` function will allocate a file descriptor as `open(2)` does.

The largest value that can be represented correctly in an object of type `off_t` will be established as the offset maximum in the open file description.

### RETURN VALUES

Upon successful completion, `fopen()` returns a pointer to the object controlling the stream. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

The `fopen()` function may fail and not set `errno` if there are no free `stdio` streams.

### ERRORS

The `fopen()` function will fail if:

- **EACCES** Search permission is denied on a component of the path prefix, or the file exists and the permissions specified by `mode` are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.

- **EINTR** A signal was caught during the execution of `fopen()`.

- **EISDIR** The named file is a directory and `mode` requires write access.

- **ELOOP** Too many symbolic links were encountered in resolving `path`.

- **EMFILE** There are `OPEN_MAX` file descriptors currently open in the calling process.

- **ENAMETOOLONG** The length of the `filename` exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX`.

- **ENFILE** The maximum allowable number of files is currently open in the system.

- **ENOENT** A component of `filename` does not name an existing file or `filename` is an empty string.

- **ENOSPC** The directory or file system that would contain the new file cannot be expanded, the file does not exist, and it was to be created.

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

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

- **EOVERFLOW** The current value of the file position cannot be represented correctly in an object of type `fpos_t`. 

---

**Basic Library Functions** | 141
The named file resides on a read-only file system and mode requires write access.

The fopen() function may fail if:

- **EINVAL** The value of the mode argument is not valid.
- **EMFILE** The number of streams currently open in the calling process is either FOPEN_MAX or STREAM_MAX.
- **ENAMETOOLONG** Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
- **ENOMEM** Insufficient storage space is available.
- **ETXTBSY** The file is a pure procedure (shared text) file that is being executed and mode requires write access.

The number of streams that a process can have open at one time is STREAM_MAX. If defined, it has the same value as FOPEN_MAX.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

See also fclose(3C), fdopen(3C), fflush(3C), freopen(3C), fsetpos(3C), rewind(3C), attributes(5), lf64(5), standards(5)
fopen(3UCB)

NAME   fopen, freopen – open a stream

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

FILE *fopen (file, mode);
const char *file, *mode;

FILE *freopen(file, mode, iop);
const char *file, *mode;
register FILE *iop;
```

DESCRIPTION  
fopen() opens the file named by file and associates a stream with it. If the open succeeds, fopen() returns a pointer to be used to identify the stream in subsequent operations.

file points to a character string that contains the name of the file to be opened.

mode 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 file and associates the stream pointed to by iop with it. The mode 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 iop.

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(3C) or rewind(3C), and input may not be directly followed by output without an intervening fseek(3C) or rewind(3C). An input operation which encounters EOF will fail.

RETURN VALUES  
fopen() and freopen() return a NULL pointer on failure.

SEE ALSO  
open(2), fclose(3C), fopen(3C), freopen(3C), fseek(3C), malloc(3C), rewind(3C)
Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

In order to support the same number of open files that the system does, `fopen()` must allocate additional memory for data structures using `malloc(3C)` after 64 files have been opened. This confuses some programs which use their own memory allocators.

The interfaces of `fopen()` and `freopen()` differ from the Standard I/O Functions `fopen(3C)` and `freopen(3C)`. The Standard I/O Functions distinguish binary from text files with an additional use of 'b' as part of the `mode`. This enables portability of `fopen(3C)` and `freopen(3C)` beyond SunOS 4.X systems.
### NAME
fpgetround, fpsetround, fpgetmask, fpsetmask, fpgetsticky, fpsetsticky – IEEE floating-point environment control

### SYNOPSIS
```c
#include <ieeefp.h>

fp_rnd fpgetround(void);
fp_rnd fpsetround(fp_rnd rnd_dir);
fp_except fpgetmask(void);
fp_except fpsetmask(fp_except mask);
fp_except fpgetsticky(void);
fp_except fpsetsticky(fp_except sticky);
```

### DESCRIPTION
There are five floating-point exceptions:
- divide-by-zero,
- overflow,
- underflow,
- imprecise (inexact) result, and
- invalid operation.

When a floating-point exception occurs, the corresponding sticky bit is set (1), and if the mask bit is enabled (1), the trap takes place. These routines let the user change the behavior on occurrence of any of these exceptions, as well as change the rounding mode for floating-point operations.

The `mask` argument is formed by the logical OR operation of the following floating-point exception masks:

- `FP_X_INV /* invalid operation exception */`
- `FP_X_OFL /* overflow exception */`
- `FP_X_UFL /* underflow exception */`
- `FP_X_DZ /* divide-by-zero exception */`
- `FP_X_IMP /* imprecise (loss of precision) */`

The following floating-point rounding modes are passed to `fpsetround` and returned by `fpgetround()`.

- `FP_RN /* round to nearest representative number */`
- `FP_RP /* round to plus infinity */`
- `FP_RM /* round to minus infinity */`
- `FP_RZ /* round to zero (truncate) */`

The default environment is rounding mode set to nearest (`FP_RN`) and all traps disabled.

The `fpsetsticky()` function modifies all sticky flags. The `fpsetmask()` function changes all mask bits. The `fpsetmask()` function clears the sticky bit corresponding to any exception being enabled.

### RETURN VALUES
The `fpgetround()` function returns the current rounding mode.
The `fpsetround()` function sets the rounding mode and returns the previous rounding mode.

The `fpgetmask()` function returns the current exception masks.

The `fpsetmask()` function sets the exception masks and returns the previous setting.

The `fpgetsticky()` function returns the current exception sticky flags.

The `fpsetsticky()` function sets (clears) the exception sticky flags and returns the previous setting.

**USAGE**

The C programming language requires truncation (round to zero) for floating point to integral conversions. The current rounding mode has no effect on these conversions.

The sticky bit must be cleared to recover from the trap and proceed. If the sticky bit is not cleared before the next trap occurs, a wrong exception type may be signaled.

Individual bits may be examined using the constants defined in `<ieeefp.h>`.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`isnan(3C), attributes(5)`
fputc(3C)

NAME
fputc, putc, putc_unlocked, putchar, putchar_unlocked, putw – put a byte on a stream

SYNOPSIS
#include <stdio.h>

int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putc_unlocked(int c, FILE *stream);
int putchar(int c);
int putchar_unlocked(int c);
int putw(int w, FILE *stream);

DESCRIPTION
The fputc() function writes the byte specified by c (converted to an unsigned char) to the output stream pointed to by stream, at the position indicated by the associated file-position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the byte is appended to the output stream.

The st_ctime and st_mtime fields of the file will be marked for update between the successful execution of fputc() and the next successful completion of a call to fflush(3C) or fclose(3C) on the same stream or a call to exit(3C) or abort(3C).

The putc() routine behaves like fputc(), except that it is implemented as a macro. It runs faster than fputc(), but it takes up more space per invocation and its name cannot be passed as an argument to a function call.

The call putchar(c) is equivalent to putc(c, stdout). The putchar() routine is implemented as a macro.

The putc_unlocked() and putchar_unlocked() routines are variants of putc() and putchar(), respectively, that do not lock the stream. It is the caller’s responsibility to acquire the stream lock before calling these routines and releasing the lock afterwards; see flockfile(3C) and stdio(3C). These routines are implemented as macros.

The putw() function writes the word (that is, type int) w to the output stream (at the position at which the file offset, if defined, is pointing). The size of a word is the size of a type int and varies from machine to machine. The putw() function neither assumes nor causes special alignment in the file.

The st_ctime and st_mtime fields of the file will be marked for update between the successful execution of putw() and the next successful completion of a call to fflush(3C) or fclose(3C) on the same stream or a call to exit(3C) or abort(3C).

RETURN VALUES
Upon successful completion, fputc(), putc(), putc_unlocked(), putchar(), and putchar_unlocked() return the value that was written. Otherwise, these functions return EOF, the error indicator for the stream is set, and errno is set to indicate the error.
Upon successful completion, `putw()` returns 0. Otherwise, it returns a non-zero value, sets the error indicator for the associated `stream`, and sets `errno` to indicate the error.

An unsuccessful completion will occur, for example, if the file associated with `stream` is not open for writing or if the output file cannot grow.

The `fputc()`, `putc()`, `putc_unlocked()`, `putchar()`, `putchar_unlocked()`, and `putw()` functions will fail if either the `stream` is unbuffered or the `stream`'s buffer needs to be flushed, and:

- **EAGAIN** The `O_NONBLOCK` flag is set for the file descriptor underlying `stream` and the process would be delayed in the write operation.
- **EBADF** The file descriptor underlying `stream` is not a valid file descriptor open for writing.
- **EFBIG** An attempt was made to write to a file that exceeds the maximum file size or the process' file size limit.
- **EFBIG** The file is a regular file and an attempt was made to write at or beyond the offset maximum.
- **EINTR** The write operation was terminated due to the receipt of a signal, and no data was transferred.
- **EIO** A physical I/O error has occurred, or the process is a member of a background process group attempting to write to its controlling terminal, `TOSTOP` is set, the process is neither ignoring nor blocking `SIGTTOU` and the process group of the process is orphaned. This error may also be returned under implementation-dependent conditions.
- **ENOSPC** There was no free space remaining on the device containing the file.
- **EPIPE** An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A `SIGPIPE` signal will also be sent to the process.

The `fputc()`, `putc()`, `putc_unlocked()`, `putchar()`, `putchar_unlocked()`, and `putw()` functions may fail if:

- **ENOMEM** Insufficient storage space is available.
- **ENXIO** A request was made of a non-existent device, or the request was outside the capabilities of the device.

### USAGE

Functions exist for the `putc()`, `putc_unlocked()`, `putchar()`, and `putchar_unlocked()` macros. To get the function form, the macro name must be undefined (for example, `#undef putc`).
When the macro forms are used, `putc()` and `putc_unlocked()` evaluate the `stream` argument more than once. In particular, `putc(c, *f++)`; does not work sensibly. The `fputc()` function should be used instead when evaluating the `stream` argument has side effects.

Because of possible differences in word length and byte ordering, files written using `putw()` are implementation-dependent, and possibly cannot be read using `getw(3C)` by a different application or by the same application running in a different environment.

The `putw()` function is inherently byte stream oriented and is not tenable in the context of either multibyte character streams or wide-character streams. Application programmers are encouraged to use one of the character-based output functions instead.

**ATTRIBUTES**

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

```
<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See NOTES below.</td>
</tr>
</tbody>
</table>
```

**SEE ALSO**

getrlimit(2), ulimit(2), write(2), intro(3), abort(3C), exit(3C), fclose(3C), ferror(3C), fflush(3C), flockfile(3C), fopen(3UCB), printf(3C), putc(3C), puts(3C), setbuf(3C), stdio(3C), attributes(5)

**NOTES**

The `fputc()`, `putc()`, `putchar()`, and `putw()` routines are MT-Safe in multithreaded applications. The `putc_unlocked()` and `putchar_unlocked()` routines are unsafe in multithreaded applications.
fputwc(3C)

NAME
fputwc, putwc, putwchar – put wide-character code on a stream

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

wint_t fputwc(wchar_t wc, FILE* stream);
wint_t putwc(wchar_t wc, FILE* stream);

#include <wchar.h>
wint_t putwchar(wchar_t wc);

DESCRIPTION
The fputwc() function writes the character corresponding to the wide-character code wc to the output stream pointed to by stream, at the position indicated by the associated file-position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream. If an error occurs while writing the character, the shift state of the output file is left in an undefined state.

The st_ctime and st_mtime fields of the file will be marked for update between the successful execution of fputwc() and the next successful completion of a call to fflush(3C) or fclose(3C) on the same stream or a call to exit(2) or abort(3C).

The putwc() function is equivalent to fputwc(), except that it is implemented as a macro.

The call putwchar(wc) is equivalent to putwc(wc, stdout). The putwchar() routine is implemented as a macro.

RETURN VALUES
Upon successful completion, fputwc(), putwc(), and putwchar() return wc. Otherwise, they return WEOF, the error indicator for the stream is set, and errno is set to indicate the error.

ERRORS
The fputwc(), putwc(), and putwchar() functions will fail if either the stream is unbuffered or data in the stream's buffer needs to be written, and:

EAGAIN The O_NONBLOCK flag is set for the file descriptor underlying stream and the process would be delayed in the write operation.

EBADF The file descriptor underlying stream is not a valid file descriptor open for writing.

EFBIG An attempt was made to write to a file that exceeds the maximum file size or the process's file size limit; or the file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

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

EIO A physical I/O error has occurred, or the process is a member of a background process group attempting to write to its controlling
terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTTOU, and the process group of the process is orphaned.

ENOSPC There was no free space remaining on the device containing the file.

EPIPE An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal will also be sent to the process.

The fputwc(), putwc(), and putwchar() functions may fail if:

ENOMEM Insufficient storage space is available.

ENXIO A request was made of a non-existent device, or the request was outside the capabilities of the device.

EILSEQ The wide-character code wc does not correspond to a valid character.

**USAGE**

Functions exist for the putwc() and putwchar() macros. To get the function form, the macro name must be undefined (for example, #undef putc).

When the macro form is used, putwc() evaluates the stream argument more than once. In particular, putwc(wc, *f++) does not work sensibly. The fputwc() function should be used instead when evaluating the stream argument has side effects.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

exit(2), ulimit(2), abort(3C), fclose(3C), ferror(3C), fflush(3C), fopen(3C), setbuf(3C), attributes(5)
fputws(3C)

NAME
fputws — put wide character string on a stream

SYNOPSIS

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

int fputws(const wchar_t *s, FILE *stream);

DESCRIPTION
The fputws() function writes a character string corresponding to the
(null-terminated) wide character string pointed to by *s to the stream pointed to by
stream. No character corresponding to the terminating null wide-character code is
written, nor is a NEWLINE character appended.

The st_ctime and st_mtime fields of the file will be marked for update between the
successful execution of fputws() and the next successful completion of a call to
fflush(3C) or fclose(3C) on the same stream or a call to exit(2) or abort(3C).

RETURN VALUES
Upon successful completion, fputws() returns a non-negative value. Otherwise, it
returns −1, sets an error indicator for the stream, and sets errno to indicate the error.

ERRORS
Refer to fputwc(3C).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
exit(2), abort(3C), fclose(3C), fflush(3C), fopen(3C), fputwc(3C),
attributes(5)
The `fread()` function reads into the array pointed to by `ptr` up to `nitems` elements whose size is specified by `size` in bytes, from the stream pointed to by `stream`. For each object, `size` calls are made to the `fgetc()` function and the results stored, in the order read, in an array of `unsigned char` exactly overlaying the object. The file-position indicator for the stream (if defined) is advanced by the number of bytes successfully read. If an error occurs, the resulting value of the file-position indicator for the stream is unspecified. If a partial element is read, its value is unspecified.

The `fread()` function may mark the `st_atime` field of the file associated with `stream` for update. The `st_atime` field will be marked for update by the first successful execution of `fgetc()`, `fgets()`, `fgetwc()`, `fgetws()`, `fread()`, `fscanf()`, `getc()`, `getchar()`, `gets()`, or `scanf()` using `stream` that returns data not supplied by a prior call to `ungetc()` or `ungetwc()`.

Upon successful completion, `fread()` returns the number of elements successfully read, which is less than `nitems` only if a read error or end-of-file is encountered. If `size` or `nitems` is 0, `fread()` returns 0 and the contents of the array and the state of the stream remain unchanged. Otherwise, if a read error occurs, the error indicator for the stream is set and `errno` is set to indicate the error.

Errors

Refer to `fgetc()`. EXAMPLE 1 Reading from a Stream

The following example reads a single element from the `fp` stream into the array pointed to by `buf`.

```c
#include <stdio.h>
...
size_t bytes_read;
char buf[100];
FILE *fp;
...
bytes_read = fread(buf, sizeof(buf), 1, fp);
...
```

Usage

The `ferror()` or `feof()` functions must be used to distinguish between an error condition and end-of-file condition. See `ferror()`. Because of possible differences in element length and byte ordering, files written using `fwrite()` are application-dependent, and possibly cannot be read using `fread()` by a different application or by the same application on a different processor.
fread(3C)

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  read(2), fclose(3C), ferror(3C), fopen(3C), getc(3C), gets(3C), printf(3C), putc(3C), puts(3C), attributes(5)
NAME
freopen – open a stream

SYNOPSIS
#include <stdio.h>

FILE *freopen(const char *filename, const char *mode, FILE *stream);

DESCRIPTION
The freopen() function first attempts to flush the stream and close any file descriptor associated with stream. Failure to flush or close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared.

The freopen() function opens the file whose pathname is the string pointed to by filename and associates the stream pointed to by stream with it. The mode argument is used just as in fopen(3C).

The original stream is closed regardless of whether the subsequent open succeeds.

After a successful call to the freopen() function, the orientation of the stream is cleared and the associated mbstate_t object is set to describe an initial conversion state.

The largest value that can be represented correctly in an object of type off_t will be established as the offset maximum in the open file description.

RETURN VALUES
Upon successful completion, freopen() returns the value of stream. Otherwise, a null pointer is returned and errno is set to indicate the error.

ERRORS
The freopen() function will fail if:

EACCESS 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 There are OPEN_MAX file descriptors currently open in the calling process.

ENAMETOOLONG The length of the filename exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENFILE The maximum allowable number of files is currently open in the system.

ENOENT A component of filename does not name an existing file or filename is an empty string.
The `freopen()` function may fail if:

- **EINVAL** The value of the `mode` argument is not valid.
- **ENAMETOOLONG** Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.
- **ENOMEM** Insufficient storage space is available.
- **ENXIO** A request was made of a non-existent device, or the request was outside the capabilities of the device.
- **ETXTBSY** The file is a pure procedure (shared text) file that is being executed and `mode` requires write access.

**USAGE**

The `freopen()` function is typically used to attach the preopened `streams` associated with `stdin`, `stdout` and `stderr` to other files. By default `stderr` is unbuffered, but the use of `freopen()` will cause it to become buffered or line-buffered.

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

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`fclose(3C), fdopen(3C), fopen(3C), stdio(3C), attributes(5), lf64(5)`
frexp – extract mantissa and exponent from double precision number

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

double frexp(double num, int *exp);
```

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

**RETURN VALUES**
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 `$\ast exp$`.

If `num` is 0, both parts of the result are 0.

If `num` is NaN, NaN is returned and the value of `$\ast exp$` is unspecified.

If `num` is ±Inf, NaN is returned and the value of `$\ast exp$` is unspecified.

**USAGE**
An application wishing to check for error situations should set `errno` to 0 before calling `frexp()`. If `errno` is non-zero on return, or the return value is NaN, an error has occurred.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**
`isnan(3M), ldexp(3C), modf(3C), attributes(5)`
**NAME**

fseek, fseeko – reposition a file-position indicator in a stream

**SYNOPSIS**

```c
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);
int fseeko(FILE *stream, off_t offset, int whence);
```

**DESCRIPTION**

The `fseek()` function sets the file-position indicator for the stream pointed to by `stream`. The `fseeko()` function is identical to `fseek()` except for the type of `offset`.

The new position, measured in bytes from the beginning of the file, is obtained by adding `offset` to the position specified by `whence`, whose values are defined in `<stdio.h>` as follows:

- `SEEK_SET` Set position equal to `offset` bytes.
- `SEEK_CUR` Set position to current location plus `offset`.
- `SEEK_END` Set position to EOF plus `offset`.

If the stream is to be used with wide character input/output functions, `offset` must either be 0 or a value returned by an earlier call to `ftell(3C)` on the same stream and `whence` must be `SEEK_SET`.

A successful call to `fseek()` clears the end-of-file indicator for the stream and undoes any effects of `ungetc(3C)` and `ungetwc(3C)` on the same stream. After an `fseek()` call, the next operation on an update stream may be either input or output.

If the most recent operation, other than `ftell(3C)`, on a given stream is `fflush(3C)`, the file offset in the underlying open file description will be adjusted to reflect the location specified by `fseek()`.

The `fseek()` function allows the file-position indicator to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads of data in the gap will return bytes with the value 0 until data is actually written into the gap.

The value of the file offset returned by `fseek()` on devices which are incapable of seeking is undefined.

If the stream is writable and buffered data had not been written to the underlying file, `fseek()` will cause the unwritten data to be written to the file and mark the `st_ctime` and `st_mtime` fields of the file for update.

**RETURN VALUES**

The `fseek()` and `fseeko()` functions return 0 on success; otherwise, they returned -1 and set `errno` to indicate the error.

**ERRORS**

The `fseek()` and `fseeko()` functions will fail if, either the `stream` is unbuffered or the `stream`’s buffer needed to be flushed, and the call to `fseek()` or `fseeko()` causes an underlying `lseek(2)` or `write(2)` to be invoked:

- `EAGAIN` The O_NONBLOCK flag is set for the file descriptor and the process would be delayed in the write operation.
EBADF  The file descriptor underlying the stream file is not open for writing or the stream’s buffer needed to be flushed and the file is not open.

EFILE  An attempt was made to write a file that exceeds the maximum file size or the process’s file size limit, or the file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.

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

EINVAL  The whence argument is invalid. The resulting file-position indicator would be set to a negative value.

EIO  A physical I/O error has occurred; or the process is a member of a background process group attempting to perform a write(2) operation to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned.

ENOSPC  There was no free space remaining on the device containing the file.

EPIPE  The file descriptor underlying stream is associated with a pipe or FIFO.

EPIPE  An attempt was made to write to a pipe or FIFO that is not open for reading by any process. A SIGPIPE signal will also be sent to the process.

ENXIO  A request was made of a non-existent device, or the request was outside the capabilities of the device.

The fseek() function will fail if:

EOVERFLOW  The resulting file offset would be a value which cannot be represented correctly in an object of type long.

The fseeko() function will fail if:

EOVERFLOW  The resulting file offset would be a value which cannot be represented correctly in an object of type off_t.

Although on the UNIX system an offset returned by ftell() or ftello() (see ftell(3C)) is measured in bytes, and it is permissible to seek to positions relative to that offset, portability to non-UNIX systems requires that an offset be used by fseek() directly. Arithmetic may not meaningfully be performed on such an offset, which is not necessarily measured in bytes.

The fseeko() function has a transitional interface for 64-bit file offsets. See lfd64(5).
fseek(3C)

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
getrlimit(2), ulimit(2), fopen(3UCB), ftell(3C), rewind(3C), ungetc(3C), ungetwc(3C), attributes(5), lf64(5)
fsetpos() function sets the file position indicator for the stream pointed to by stream according to the value of the object pointed to by pos, which must be a value obtained from an earlier call to fgetpos(3C) on the same stream.

A successful call to fsetpos() function clears the end-of-file indicator for the stream and undoes any effects of ungetc(3C) on the same stream. After an fsetpos() call, the next operation on an update stream may be either input or output.

The fsetpos() function returns 0 if it succeeds; otherwise it returns a non-zero value and sets errno to indicate the error.

The fsetpos() function may fail if:
- EBADF The file descriptor underlying stream is not valid.
- ESPIPE The file descriptor underlying stream is associated with a pipe, a FIFO, or a socket.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO lseek(2), fgetpos(3C), fopen(3C), fseek(3C), ftell(3C), rewind(3C), ungetc(3C), attributes(5), lf64(5)
fsync(3C)

NAME    | fsync — synchronize changes to a file
SYNOPSIS| #include <unistd.h>
          | int fsync(int fildes);

DESCRIPTION
The fsync() function moves all modified data and attributes of the file descriptor fildes to a storage device. When fsync() returns, all in-memory modified copies of buffers associated with fildes have been written to the physical medium. The fsync() function is different from sync(), which schedules disk I/O for all files but returns before the I/O completes. The fsync() function forces all outstanding data operations to synchronized file integrity completion (see fcntl(3HEAD) definition of O_SYNC.)

The fsync() function forces all currently queued I/O operations associated with the file indicated by the file descriptor fildes to the synchronized I/O completion state. All I/O operations are completed as defined for synchronized I/O file integrity completion.

RETURN VALUES
Upon successful completion, 0 is returned. Otherwise, -1 is returned and errno is set to indicate the error. If the fsync() function fails, outstanding I/O operations are not guaranteed to have been completed.

ERRORS
The fsync() function will fail if:
EBADF    | The fildes argument is not a valid file descriptor.
EINTR    | A signal was caught during execution of the fsync() function.
EIO       | An I/O error occurred while reading from or writing to the file system.
ENOSPC    | There was no free space remaining on the device containing the file.
ETIMEDOUT | Remote connection timed out. This occurs when the file is on an NFS file system mounted with the soft option. See mount_nfs(1M).

In the event that any of the queued I/O operations fail, fsync() returns the error conditions defined for read(2) and write(2).

USAGE
The fsync() function should be used by applications that require that a file be in a known state. For example, an application that contains a simple transaction facility might use fsync() to ensure that all changes to a file or files caused by a given transaction were recorded on a storage medium.

The manner in which the data reach the physical medium depends on both implementation and hardware. The fsync() function returns when notified by the device driver that the write has taken place.
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  
mount_nfs(1M), read(2), sync(2), write(2), fcntl(3HEAD), fdatasync(3RT), attributes(5)
ftell(3C)

NAME   ftell, ftell – return a file offset in a stream

SYNOPSIS  #include <stdio.h>

long ftell(FILE *stream);
off_t ftello(FILE *stream);

DESCRIPTION  The ftell() function obtains the current value of the file-position indicator for the stream pointed to by stream. The ftello() function is identical to ftell() except for the return type.

RETURN VALUES  Upon successful completion, the ftell() and ftello() functions return the current value of the file-position indicator for the stream measured in bytes from the beginning of the file. Otherwise, they return −1 and sets errno to indicate the error.

ERRORS  The ftell() and ftello() functions will fail if:

EBADF  The file descriptor underlying stream is not an open file descriptor.
ESPIPE  The file descriptor underlying stream is associated with a pipe, a FIFO, or a socket.

The ftell() function will fail if:

EOVERFLOW  The current file offset cannot be represented correctly in an object of type long.

The ftello() function will fail if:

EOVERFLOW  The current file offset cannot be represented correctly in an object of type off_t.

USAGE  The ftello() function has a transitional interface for 64-bit file offsets. See l64(5).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  lseek(2), fopen(3C), fseek(3C), attributes(5), l64(5)
ftime – get date and time

#include <sys/timeb.h>

int ftime(struct timeb *tp);

The ftime() function sets the time and millitm members of the timeb structure pointed to by tp. The structure is defined in <sys/timeb.h> and contains the following members:

time_t time;
unsigned short millitm;
short timezone;
short dstflag;

The time and millitm members contain the seconds and milliseconds portions, respectively, of the current time in seconds since 00:00:00 UTC (Coordinated Universal Time), January 1, 1970.

The timezone member contains the local time zone. The dstflag member contains a flag that, if non-zero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

The contents of the timezone and dstflag members of tp after a call to ftime() are unspecified.

Upon successful completion, the ftime() function returns 0. Otherwise −1 is returned.

No errors are defined.

For portability to implementations conforming to earlier versions of this document, time(2) is preferred over this function.

The millisecond value usually has a granularity greater than one due to the resolution of the system clock. Depending on any granularity (particularly a granularity of one) renders code non-portable.

date(1), time(2), ctime(3C), gettimeofday(3C), timezone(4)
The `ftok()` function returns a key based on `path` and `id` that is usable in subsequent calls to `msgget(2)`, `semget(2)` and `shmget(2)`. The `path` argument must be the pathname of an existing file that the process is able to `stat(2)`.

The `ftok()` function will return the same key value for all paths that name the same file, when called with the same `id` value, and will return different key values when called with different `id` values.

If the file named by `path` is removed while still referred to by a key, a call to `ftok()` with the same `path` and `id` returns an error. If the same file is recreated, then a call to `ftok()` with the same `path` and `id` is likely to return a different key.

Only the low order 8-bits of `id` are significant. The behavior of `ftok()` is unspecified if these bits are 0.

Upon successful completion, `ftok()` returns a key. Otherwise, `ftok()` returns `(key_t)-1` and sets `errno` to indicate the error.

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.

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `{PATH_MAX}`.

For maximum portability, `id` should be a single-byte character.

Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If some standard is not adhered to, it will be possible for unrelated processes to
unintentionally interfere with each other’s operation. It is still possible to interfere intentionally. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

NOTES Since the `ftok()` function returns a value based on the id given and the file serial number of the file named by `path` in a type that is no longer large enough to hold all file serial numbers, it may return the same key for paths naming different files on large filesystems.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO `msgget(2), semget(2), shmget(2), stat(2), attributes(5)`
#include <ftw.h>

```c
int ftw(const char *path, int (*fn)(const char *, const struct stat *, int), int depth);

int nftw(const char *path, int (*fn)(const char *, const struct stat *, int, struct FTW *), int depth, int flags);
```

**DESCRIPTION**

The `ftw()` function recursively descends the directory hierarchy rooted in `path`. For each object in the hierarchy, `ftw()` calls the user-defined function `fn`, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a `stat` structure (see `stat(2)` containing information about the object, and an integer. Possible values of the integer, defined in the `<ftw.h>` header, are:

- `FTW_F`: The object is a file.
- `FTW_D`: The object is a directory.
- `FTW_DNR`: The object is a directory that cannot be read. Descendants of the directory are not processed.
- `FTW_NS`: The `stat()` function failed on the object because of lack of appropriate permission or the object is a symbolic link that points to a non-existent file. The `stat` buffer passed to `fn` is undefined.

The `ftw()` function visits a directory before visiting any of its descendants. The tree traversal continues until the tree is exhausted, an invocation of `fn` returns a non-zero value, or some error is detected within `ftw()` (such as an I/O error). If the tree is exhausted, `ftw()` returns 0. If `fn` returns a non-zero value, `ftw()` stops its tree traversal and returns whatever value was returned by `fn`.

The `nftw()` function is similar to `ftw()` except that it takes the additional argument `flags`, which is a bitwise-inclusive OR of zero or more of the following flags:

- `FTW_CHDIR`: If set, `nftw()` changes the current working directory to each directory as it reports files in that directory. If clear, `nftw()` does not change the current working directory.
- `FTW_DEPTH`: If set, `nftw()` reports all files in a directory before reporting the directory itself. If clear, `nftw()` reports any directory before reporting the files in that directory.
- `FTW_MOUNT`: If set, `nftw()` reports only files in the same file system as `path`. If clear, `nftw()` reports all files encountered during the walk.
- `FTW_PHYS`: If set, `nftw()` performs a physical walk and does not follow symbolic links.
If \texttt{FTW\_PHYS} is clear and \texttt{FTW\_DEPTH} is set, \texttt{nftw()} follows links instead of reporting them, but does not report any directory that would be a descendant of itself. If \texttt{FTW\_PHYS} is clear and \texttt{FTW\_DEPTH} is clear, \texttt{nftw()} follows links instead of reporting them, but does not report the contents of any directory that would be a descendant of itself.

At each file it encounters, \texttt{nftw()} calls the user-supplied function \texttt{fn} with four arguments:

- The first argument is the pathname of the object.
- The second argument is a pointer to the \texttt{stat} buffer containing information on the object.
- The third argument is an integer giving additional information. Its value is one of the following:
  - \texttt{FTW\_F} The object is a file.
  - \texttt{FTW\_D} The object is a directory.
  - \texttt{FTW\_DP} The object is a directory and subdirectories have been visited. (This condition only occurs if the \texttt{FTW\_DEPTH} flag is included in flags.)
  - \texttt{FTW\_SL} The object is a symbolic link. (This condition only occurs if the \texttt{FTW\_PHYS} flag is included in flags.)
  - \texttt{FTW\_SLN} The object is a symbolic link that points to a non-existent file. (This condition only occurs if the \texttt{FTW\_PHYS} flag is not included in flags.)
  - \texttt{FTW\_DNR} The object is a directory that cannot be read. The user-defined function \texttt{fn} will not be called for any of its descendants.
  - \texttt{FTW\_NS} The \texttt{stat()} function failed on the object because of lack of appropriate permission. The stat buffer passed to \texttt{fn} is undefined. Failure of \texttt{stat()} for any other reason is considered an error and \texttt{nftw()} returns \texttt{-1}.

- The fourth argument is a pointer to an \texttt{FTW} structure that contains the following members:
  ```
  int base;
  int level;
  ```
  The \texttt{base} member is the offset of the object’s filename in the pathname passed as the first argument to \texttt{fn()}. The value of \texttt{level} indicates the depth relative to the root of the walk, where the root level is 0.

Both \texttt{ftw()} and \texttt{nftw()} use one file descriptor for each level in the tree. The \texttt{depth} argument limits the number of file descriptors used. If \texttt{depth} is zero or negative, the effect is the same as if it were 1. It must not be greater than the number of file
descriptors currently available for use. The \texttt{ftw()} function runs faster if \textit{depth} is at least as large as the number of levels in the tree. When \texttt{ftw()} and \texttt{nftw()} return, they close any file descriptors they have opened; they do not close any file descriptors that might have been opened by \texttt{fn}.

**RETURN VALUES**

If the tree is exhausted, \texttt{ftw()} and \texttt{nftw()} return 0. If the function pointed to by \texttt{fn} returns a non-zero value, \texttt{ftw()} and \texttt{nftw()} stop their tree traversal and return whatever value was returned by the function pointed to by \texttt{fn}. If \texttt{ftw()} and \texttt{nftw()} detect an error, they return \texttt{-1} and set \texttt{errno} to indicate the error.

If \texttt{ftw()} and \texttt{nftw()} encounter an error other than \texttt{EACCES} (see \texttt{FTW_DNR} and \texttt{FTW_NS} above), they return \texttt{-1} and set \texttt{errno} to indicate the error. The external variable \texttt{errno} can contain any error value that is possible when a directory is opened or when one of the \texttt{stat} functions is executed on a directory or file.

**ERRORS**

The \texttt{ftw()} and \texttt{nftw()} functions will fail if:

- **ELOOP**: A loop exists in symbolic links encountered during resolution of the \texttt{path} argument.
- **ENAMETOOLONG**: The length of the path exceeds \{\texttt{PATH_MAX}\}, or a path name component is longer than \{\texttt{NAME_MAX}\}.
- **ENOENT**: A component of \texttt{path} does not name an existing file or \texttt{path} is an empty string.
- **ENOTDIR**: A component of \texttt{path} is not a directory.
- **EOVERFLOW**: A field in the \texttt{stat} structure cannot be represented correctly in the current programming environment for one or more files found in the file hierarchy.

The \texttt{ftw()} function will fail if:

- **EACCES**: Search permission is denied for any component of \texttt{path} or read permission is denied for \texttt{path}.

The \texttt{nftw()} function will fail if:

- **EACCES**: Search permission is denied for any component of \texttt{path} or read permission is denied for \texttt{path}, or \texttt{fn()} returns \texttt{-1} and does not reset \texttt{errno}.

The \texttt{nftw()} and \texttt{ftw()} functions may fail if:

- **ELOOP**: Too many symbolic links were encountered during resolution of the \texttt{path} argument.
- **ENAMETOOLONG**: Pathname resolution of a symbolic link produced an intermediate result whose length exceeds \{\texttt{PATH_MAX}\}.

The \texttt{ftw()} function may fail if:
EINVAL

The value of the `ndirs` argument is invalid.

The `nftw()` function may fail if:

EMFILE

There are `OPEN_MAX` file descriptors currently open in the calling process.

ENFILE

Too many files are currently open in the system.

If the function pointed to by `fn` encounters system errors, `errno` may be set accordingly.

EXAMPLES

EXAMPLE 1 Walk a directory structure using `ftw()`.

The following example walks the current directory structure, calling the `fn()` function for every directory entry, using at most 10 file descriptors:

```c
#include <ftw.h>
...
if (ftw(".", fn, 10) != 0) {
    perror("ftw"); exit(2);
}
```

EXAMPLE 2 Walk a directory structure using `nftw()`.

The following example walks the `/tmp` directory and its subdirectories, calling the `nftw()` function for every directory entry, to a maximum of 5 levels deep.

```c
#include <ftw.h>
...
int nftwfunc(const char *, const struct stat *, int, struct FTW *);
int nftwfunc(const char *filename, const struct stat *statptr,  
    int fileflags, struct FTW *pfwt)  
{
    return 0;
}
...
char *startpath = "/tmp";
int depth = 5;
int flags = FTW_CHDIR | FTW_DEPTH | FTW_MOUNT;
int ret;
ret = nftw(startpath, nftwfunc, depth, flags);
```

USAGE

Because `ftw()` is recursive, it can terminate with a memory fault when applied to very deep file structures.

The `ftw()` function uses `malloc(3C)` to allocate dynamic storage during its operation. If `ftw()` is forcibly terminated, such as by `longjmp(3C)` being executed by `fn` or an interrupt routine, `ftw()` will not have a chance to free that storage, so it remains permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred and arrange to have `fn` return a non-zero value at its next invocation.
The `ftw()` and `nftw()` functions have transitional interfaces for 64-bit file offsets. See `lf64(5)`.

The `ftw()` function is safe in multithreaded applications. The `nftw()` function is safe in multithreaded applications when the `FTW_CHDIR` flag is not set.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Safe with exceptions</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`stat(2), longjmp(3C), malloc(3C), attributes(5), lf64(5), standards(5)`
NAME
fwide – set stream orientation

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

int fwide(FILE *stream, int mode);

DESCRIPTION
The fwide() function determines the orientation of the stream pointed to by stream. If
mode is greater than 0, the function first attempts to make the stream wide-orientated.
If mode is less than 0, the function first attempts to make the stream byte-orientated.
Otherwise, mode is 0 and the function does not alter the orientation of the stream.

If the orientation of the stream has already been determined, fwide() does not
change it.

Because no return value is reserved to indicate an error, an application wishing to
check for error situations should set errno to 0, then call fwide(), then check errno
and if it is non-zero, assume an error has occurred.

RETURN VALUES
The fwide() function returns a value greater than 0 if, after the call, the stream has
wide-orientation, a value less than 0 if the stream has byte-orientation, or 0 if the
stream has no orientation.

ERRORS
The fwide() function may fail if:
EBADF  The stream argument is not a valid stream.

USAGE
A call to fwide() with mode set to 0 can be used to determine the current orientation
of a stream.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5)
fwprintf(3C)

NAME fwprintf, wprintf, swprintf – print formatted wide-character output

SYNOPSIS

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

int fwprintf(FILE *stream, const wchar_t *format, ...);

int wprintf(const wchar_t *format, ...);

int swprintf(wchar_t *s, size_t n, const wchar_t *format, ...);

DESCRIPTION

The fwprintf() function places output on the named output stream. The wprintf() function places output on the standard output stream stdout. The swprintf() function places output followed by the null wide-character in consecutive wide-characters starting at *s; no more than *n wide-characters are written, including a terminating null wide-character, which is always added (unless *n is zero).

Each of these functions converts, formats and prints its arguments under control of the format wide-character string. The format is composed of zero or more directives: ordinary wide-characters, which are simply copied to the output stream and conversion specifications, each of which results in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Conversions can be applied to the *n*th argument after the format in the argument list, rather than to the next unused argument. In this case, the conversion wide-character % (see below) is replaced by the sequence %n$, where *n* is a decimal integer in the range [1, NL_ARGMAX], giving the position of the argument in the argument list. This feature provides for the definition of format wide-character strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

In format wide-character strings containing the %n$ form of conversion specifications, numbered arguments in the argument list can be referenced from the format wide-character string as many times as required.

In format wide-character strings containing the % form of conversion specifications, each argument in the argument list is used exactly once.

All forms of the fwprintf() functions allow for the insertion of a language-dependent radix character in the output string, output as a wide-character value. The radix character is defined in the program’s locale (category LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.).

Each conversion specification is introduced by the % wide-character or by the wide-character sequence %n$, after which the following appear in sequence:

- Zero or more flags (in any order), which modify the meaning of the conversion specification.
An optional minimum field width. If the converted value has fewer wide-characters than the field width, it will be padded with spaces by default on the left; it will be padded on the right, if the left-adjustment flag (−), described below, is given to the field width. The field width takes the form of an asterisk (*), described below, or a decimal integer.

An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions; the number of digits to appear after the radix character for the e, E, and f conversions; the maximum number of significant digits for the g and G conversions; or the maximum number of wide-characters to be printed from a string in s conversions. The precision takes the form of a period (.) followed by either an asterisk (*), described below, or an optional decimal digit string, where a null digit string is treated as 0. If a precision appears with any other conversion wide-character, the behavior is undefined.

An optional l (ell) specifying that a following c conversion wide-character applies to a wint_t argument; an optional l specifying that a following s conversion wide-character applies to a wchar_t argument; an optional h specifying that a following d, i, o, u, x, and X conversion wide-character applies to a type short int or type unsigned short int argument (the argument will have been promoted according to the integral promotions, and its value will be converted to type short int or type unsigned short int before printing); an optional h specifying that a following n conversion wide-character applies to a pointer to a type shortint argument; an optional l specifying that a following d, i, o, u, x, and X conversion wide-character applies to a type longint or type unsigned longint argument; an optional l specifying that a following n conversion wide-character applies to a pointer to a type longint argument; or an optional L specifying that a following e, E, f, g, or G conversion wide-character applies to a type longdouble argument. If an h, l, or L appears with any other conversion wide-character, the behavior is undefined.

A conversion wide-character that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk (*). In this case an argument of type int supplies the field width or precision. Arguments specifying field width, or precision, or both must appear in that order before the argument, if any, to be converted. A negative field width is taken as a – flag followed by a positive field width. A negative precision is taken as if the precision were omitted. In format wide-character strings containing the %n$ form of a conversion specification, a field width or precision may be indicated by the sequence *m$, where m is a decimal integer in the range [1, NL_ARGMAX] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```c
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 Nth argument requires that all the leading arguments, from the first to the (N−1)th, are specified in the format wide-character string.

The flag wide-characters and their meanings are:

- The integer portion of the result of a decimal conversion (\%i, \%d, \%u, \%f, \%g, or \%G) will be formatted with thousands' grouping wide-characters. For other conversions the behavior is undefined. The non-monetary grouping wide-character is used.
- The result of the conversion will be left-justified within the field. The conversion will be right-justified if this flag is not specified.
+ The result of a signed conversion will always begin with a sign (+ or −). The conversion will begin with a sign only when a negative value is converted if this flag is not specified.
- The first wide-character of a signed conversion is not a sign or if a signed conversion results in no wide-characters, a space will be prefixed to the result. This means that if the space and + flags both appear, the space flag will be ignored.
- This flag specifies that the value is to be converted to an alternative form. For o conversion, it increases the precision (if necessary) to force the first digit of the result to be zero. For x or X conversions, a non-zero result will have 0x (or 0X) prefixed to it. For e, E, f, g, or G conversions, the result will always contain a radix character, even if no digits follow it. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversions, trailing zeros will not be removed from the result as they normally are. For other conversions, the behavior is undefined.
- 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. If the 0 and − flags both appear, the grouping wide-characters are inserted before zero padding. For other conversions, the behavior is undefined.

The conversion wide-characters and their meanings are:

d, i The int argument is converted to a signed decimal in the style [-] dddd. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.
The unsigned int argument is converted to unsigned octal format in the style \texttt{dddd}. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.

The unsigned int argument is converted to unsigned decimal format in the style \texttt{dddd}. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.

The unsigned int argument is converted to unsigned hexadecimal format in the style \texttt{dddd}; the letters abcdef are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.

Behaves the same as the \texttt{x} conversion wide-character except that letters ABCDEF are used instead of abcdef.

The double argument is converted to decimal notation in the style \texttt{[-]}\texttt{ddd.ddd}, where the number of digits after the radix character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly 0 and no \# flag is present, no radix character appears. If a radix character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

The \texttt{fwprintf()} family of functions may make available wide-character string representations for infinity and NaN.

The double argument is converted in the style \texttt{[\text{-}]d.dde \pm dd}, where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is 0 and no \# flag is present, no radix character appears. The value is rounded to the appropriate number of digits. The \texttt{e} conversion wide-character will produce a number with \texttt{E} instead of \texttt{e} introducing the exponent. The exponent always contains at least two digits. If the value is 0, the exponent is 0.

The \texttt{fwprintf()} family of functions may make available wide-character string representations for infinity and NaN.

The double argument is converted in the style \texttt{f} or \texttt{e} (or in the style \texttt{E} in the case of a \texttt{G} conversion wide-character), with the precision specifying the number of significant digits. If an explicit precision is 0, it is taken as 1. The style used depends on the value converted; style \texttt{e} (or \texttt{E}) will be used only if the exponent resulting from such a conversion is less than \texttt{-4} or
greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a radix character appears only if it is followed by a digit.

The `fwprintf()` family of functions may make available wide-character string representations for infinity and NaN.

If no `l` (ell) qualifier is present, the `int` argument is converted to a wide-character as if by calling the `btowc(3C)` function and the resulting wide-character is written. Otherwise the `wint_t` argument is converted to `wchar_t`, and written.

If no `l` (ell) qualifier is present, the argument must be a pointer to a character array containing a character sequence beginning in the initial shift state. Characters from the array are converted as if by repeated calls to the `mbtowc(3C)` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first character is converted, and written up to (but not including) the terminating null wide-character. If the precision is specified, no more than that many wide-characters are written. If the precision is not specified or is greater than the size of the array, the array must contain a null wide-character.

If an `l` (ell) qualifier is present, the argument must be a pointer to an array of type `wchar_t`. Wide characters from the array are written up to (but not including) a terminating null wide-character. If no precision is specified or is greater than the size of the array, the array must contain a null wide-character. If a precision is specified, no more than that many wide-characters are written.

The argument must be a pointer to `void`. The value of the pointer is converted to a sequence of printable wide-characters.

The argument must be a pointer to an integer into which is written the number of wide-characters written to the output so far by this call to one of the `fwprintf()` functions. No argument is converted.

Same as `lc`.

Same as `ls`.

Output a `%` wide-character; no argument is converted. The entire conversion specification must be `%%`.

If a conversion specification does not match one of the above forms, the behavior is undefined.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by `fwprintf()` and `wprintf()` are printed as if `fputwc(3C)` had been called.
fwprintf(3C)

The st_ctime and st_mtime fields of the file will be marked for update between the call to a successful execution of fwprintf() or wprintf() and the next successful completion of a call to fflush(3C) or fclose(3C) on the same stream or a call to exit(3C) or abort(3C).

RETURN VALUES

Upon successful completion, these functions return the number of wide-characters transmitted excluding the terminating null wide-character in the case of swprintf() or a negative value if an output error was encountered.

ERRORS

For the conditions under which fwprintf() and wprintf() will fail and may fail, refer to fputwc(3C).

In addition, all forms of fwprintf() may fail if:

EILSEQ A wide-character code that does not correspond to a valid character has been detected.

EINVAL There are insufficient arguments.

In addition, wprintf() and fwprintf() may fail if:

ENOMEM Insufficient storage space is available.

EXAMPLES

EXAMPLE 1 Print language-dependent date and time format.

To print the language-independent date and time format, the following statement could be used:

wprintf(format, weekday, month, day, hour, min);

For American usage, format could be a pointer to the wide-character string:

L\"%s, %s %d, %d:%.2d
\"producing the message:

Sunday, July 3, 10:02

whereas for German usage, format could be a pointer to the wide-character string:

L\"%1$s, %3$d. %2$s, %4$d:%5$.2d\"producing the message:

Sonntag, 3. Juli, 10:02

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
</tbody>
</table>

SEE ALSO

btowc(3C), fputwc(3C), fwscanf(3C), mbtowc(3C), setlocale(3C), attributes(5)
The `fwprintf()`, `wprintf()`, and `swprintf()` functions can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale.
fwrite(3C)

NAME    fwrite – binary output

SYNOPSIS #include <stdio.h>

    size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);

DESCRIPTION The fwrite() function writes, from the array pointed to by ptr, up to nitems elements whose size is specified by size, to the stream pointed to by stream. For each object, size calls are made to the fputc(3C) function, taking the values (in order) from an array of unsigned char exactly overlaying the object. The file-position indicator for the stream (if defined) is advanced by the number of bytes successfully written. If an error occurs, the resulting value of the file-position indicator for the stream is unspecified.

The st_ctime and st_mtime fields of the file will be marked for update between the successful execution of fwrite() and the next successful completion of a call to fflush(3C) or fclose(3C) on the same stream or a call to exit(2) or abort(3C).

RETURN VALUES The fwrite() function returns the number of elements successfully written, which might be less than nitems if a write error is encountered. If size or nitems is 0, fwrite() returns 0 and the state of the stream remains unchanged. Otherwise, if a write error occurs, the error indicator for the stream is set and errno is set to indicate the error.

ERRORS Refer to fputc(3C).

USAGE Because of possible differences in element length and byte ordering, files written using fwrite() are application-dependent, and possibly cannot be read using fread(3C) by a different application or by the same application on a different processor.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO write(2), fclose(3C), ferror(3C), fopen(3C), fread(3C), getc(3C), gets(3C), printf(3C), putc(3C), puts(3C), attributes(5)
fwscanf(3C)

NAME fwscanf, wscanf, swscanf, vfwscanf, vwscanf, vswscanf – convert formatted wide-character input

SYNOPSIS #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, ...);
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwscanf(FILE *stream, const wchar_t *format, va_list arg);
int vswcanf(const wchar_t *ws, const wchar_t *format, va_list arg);
int vswscanf(const wchar_t *format, va_list arg);

DESCRIPTION The fwscanf() function reads from the named input stream.

The wscanf() function reads from the standard input stream stdin.

The swscanf() function reads from the wide-character string s.

The vfwscanf(), vswcanf(), and vswscanf() functions are equivalent to the fwscanf(), swscanf(), and wscanf() functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by the <stdarg.h> header (see stdarg(3HEAD)). These functions do not invoke the va_end() macro. Applications using these functions should call va_end(ap) afterwards to clean up.

Each function reads wide-characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control wide-character string format described below, and a set of pointer arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Conversions can be applied to the nth 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 `fwscanf()` function in all its forms allows for detection of a language-dependent radix character in the input string, encoded as a wide-character value. The radix character is defined in the program’s locale (category `LC_NUMERIC`). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (\`).

The format is a wide-character string composed of zero or more directives. Each directive is composed of one of the following: one or more white-space wide-characters (space, tab, newline, vertical-tab or form-feed characters); an ordinary wide-character (neither \% nor a white-space character); or a conversion specification. Each conversion specification is introduced by a \% or the sequence \%n\$, after which the following appear in sequence:

- An optional assignment-suppressing character *.
- An optional non-zero decimal integer that specifies the maximum field width.
- An optional size modifier h, l(ell), or L indicating the size of the receiving object. The conversion wide-characters c, s, and l must be preceded by h (ell) if the corresponding argument is a pointer to `wchar_t` rather than a pointer to a character type. The conversion wide-characters d, i, and n must be preceded by h if the corresponding argument is a pointer to `short int` rather than a pointer to `int`, or by l (ell) if it is a pointer to `long int`. Similarly, the conversion wide-characters o, u, and x must be preceded by h if the corresponding argument is a pointer to `short int` rather than a pointer to `unsigned int`, or by l (ell) if it is a pointer to `unsigned long int`. The conversion wide-characters e, f, and g must be preceded by l (ell) if the corresponding argument is a pointer to `double` rather than a pointer to `float`, or by L if it is a pointer to `long double`. If an h, l (ell), or L appears with any other conversion wide-character, the behavior is undefined.
- A conversion wide-character that specifies the type of conversion to be applied. The valid conversion wide-characters are described below.

The `fwscanf()` functions execute each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space wide-characters is executed by reading input until no more valid input can be read, or up to the first wide-character which is not a white-space wide-character, which remains unread.

A directive that is an ordinary wide-character is executed as follows. The next wide-character is read from the input and compared with the wide-character that comprises the directive; if the comparison shows that they are not equivalent, the directive fails, and the differing and subsequent wide-characters remain unread.
A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion wide-character. A conversion specification is executed in the following steps:

Input white-space wide-characters (as specified by \texttt{isspace(3C)}) are skipped, unless the conversion specification includes a \texttt{[, c}, or \texttt{n} conversion character.

An item is read from the input, unless the conversion specification includes an \texttt{n} conversion wide-character. An input item is defined as the longest sequence of input wide-characters, not exceeding any specified field width, which is an initial subsequence of a matching sequence. The first wide-character, if any, after the input item remains unread. If the length of the input item is 0, the execution of the conversion specification 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 \texttt{%} conversion wide-character, the input item (or, in the case of a \texttt{%n} conversion specification, the count of input wide-characters) is converted to a type appropriate to the conversion wide-character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by a \texttt{*}, the result of the conversion is placed in the object pointed to by the first argument following the \textit{format} argument that has not already received a conversion result if the conversion specification is introduced by \texttt{%}, or in the \textit{n}th argument if introduced by the wide-character sequence \texttt{%n$}. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion wide-characters are valid:

- **d**: Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of \texttt{wcstol(3C)} with the value 10 for the \textit{base} argument. In the absence of a size modifier, the corresponding argument must be a pointer to \texttt{int}.

- **i**: Matches an optionally signed integer, whose format is the same as expected for the subject sequence of \texttt{wcstol(3C)} with 0 for the \textit{base} argument. In the absence of a size modifier, the corresponding argument must be a pointer to \texttt{int}.

- **o**: Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of \texttt{wcstoul(3C)} with the value 8 for the \textit{base} argument. In the absence of a size modifier, the corresponding argument must be a pointer to \texttt{unsigned int}.

- **u**: Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of \texttt{wcstoul(3C)} with the value 10 for the \textit{base} argument. In the absence of a size modifier, the corresponding argument must be a pointer to \texttt{unsigned int}.
Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of `wcstoul(3C)` with the value 16 for the `base` argument. In the absence of a size modifier, the corresponding argument must be a pointer to `unsigned int`.

Matches an optionally signed floating-point number, whose format is the same as expected for the subject sequence of `wcstod(3C)`. In the absence of a size modifier, the corresponding argument must be a pointer to `float`.

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 recognize them as input.

Matches a sequence of non white-space wide-characters. If no `ell` qualifier is present, characters from the input field are converted as if by repeated calls to the `wcrtomb(3C)` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence and the terminating null character, which will be added automatically.

Otherwise, the corresponding argument must be a pointer to an array of `wchar_t` large enough to accept the sequence and the terminating null wide-character, which will be added automatically.

Matches a non-empty sequence of wide-characters from a set of expected wide-characters (the `scanset`). If no `ell` qualifier is present, wide-characters from the input field are converted as if by repeated calls to the `wcrtomb(3C)` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence and the terminating null wide-character, which will be added automatically.

If an `ell` qualifier is present, the corresponding argument must be a pointer to an array of `wchar_t` large enough to accept the sequence and the terminating null wide-character, which will be added automatically.

The conversion specification includes all subsequent widw characters in the `format` string up to and including the matching right square bracket (`]`). The wide-characters between the square brackets (the `scanlist`) comprise the `scanset`, unless the wide-character after the left square bracket is a circumflex (`^`), in which case the `scanset` contains all wide-characters that do not appear in the `scanlist` between the circumflex and the right square bracket. If the conversion specification begins with `[` or `[^`], the right square bracket is included in the `scanlist` and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise the first right square bracket is the one that ends the conversion specification.
fwscanf(3C)

specification. If a minus-sign (−) is in the scanlist and is not the first
wide-character, nor the second where the first wide-character is a ^, nor the
last wide-character, it indicates a range of characters to be matched.

\textbf{c}

Matches a sequence of wide-characters of the number specified by the field
width (1 if no field width is present in the conversion specification). If no 1
(ell) qualifier is present, wide-characters from the input field are converted
as if by repeated calls to the \texttt{wcrtomb()} function, with the conversion
state described by an \texttt{mbstate_t} object initialized to zero before the first
wide-character is converted. The corresponding argument must be a
pointer to a character array large enough to accept the sequence. No null
character is added.

Otherwise, the corresponding argument must be a pointer to an array of
\texttt{wchar_t} large enough to accept the sequence. No null wide-character is
added.

\textbf{p}

Matches the set of sequences that is the same as the set of sequences that is
produced by the \%p conversion of the corresponding \texttt{fwprintf(3C)}
functions. The corresponding argument must be a pointer to a pointer to
\texttt{void}. If the input item is a value converted earlier during the same
program execution, the pointer that results will compare equal to that
value; otherwise the behavior of the \%p conversion is undefined.

\textbf{n}

No input is consumed. The corresponding argument must be a pointer to
the integer into which is to be written the number of wide-characters read
from the input so far by this call to the \texttt{fwscanf()} functions. Execution of
a \%n conversion specification does not increment the assignment count
returned at the completion of execution of the function.

\textbf{C}

Same as \texttt{lc}.

\textbf{S}

Same as \texttt{ls}.

\textbf{\%}

Matches a single \%; no conversion or assignment occurs. The complete
conversion specification must be \%%.

If a conversion specification is invalid, the behavior is undefined.

The conversion characters E, G, and X are also valid and behave the same as,
respectively, e, g, and x.

If end-of-file is encountered during input, conversion is terminated. If end-of-file
occurs before any wide-characters matching the current conversion specification
(except for \%n) have been read (other than leading white-space, where permitted),
exection of the current conversion specification terminates with an input failure.
Otherwise, unless execution of the current conversion specification is terminated with
a matching failure, execution of the following conversion specification (if any) is
terminated with an input failure.
Reaching the end of the string in `swscanf()` is equivalent to encountering end-of-file for `fwscanf()`.

If conversion terminates on a conflicting input, the offending input is left unread in the input. Any trailing white space (including newline) is left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the `%n` conversion specification.

The `fwscanf()` and `wscanf()` functions may mark the `st_atime` field of the file associated with `stream` for update. The `st_atime` field will be marked for update by the first successful execution of `fgetc(3C)`, `fgetwc(3C)`, `fgets(3C)`, `fgetws(3C)`, `fread(3C)`, `getc(3C)`, `getwc(3C)`, `getwchar(3C)`, `fgets(3C)`, `fscanf(3C)` or `fwscanf()` using `stream` that returns data not supplied by a prior call to `ungetc(3C)`.

**RETURN VALUES**

Upon successful completion, these functions return the number of successfully matched and assigned input items; this number can be 0 in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF is returned. If a read error occurs the error indicator for the stream is set, EOF is returned, and `errno` is set to indicate the error.

**ERRORS**

For the conditions under which the `fwscanf()` functions will fail and may fail, refer to `fgetwc(3C)`.

In addition, `fwscanf()` may fail if:

- **EILSEQ** Input byte sequence does not form a valid character.
- **EINVAL** There are insufficient arguments.

**USAGE**

In format strings containing the `%` form of conversion specifications, each argument in the argument list is used exactly once.

**EXAMPLES**

**EXAMPLE 1** `wscanf()` example

The call:

```c
int i, n; float x; char name[50];

n = wscanf(L"%d%f%s", &i, &x, name);
```

with the input line:

```
25 5.432E-1 Hamster
```

will assign to `n` the value 3, to `i` the value 25, to `x` the value 5.432, and `name` will contain the string Hamster.

The call:

```c
int i; float x; char name[50];

(void) wscanf(L"%2d%f%-d %>0123456789", &i, &x, name);
```

with input:

```
25 5.432E-1 Hamster
```
fscanf(3C)

EXAMPLE 1 wscanf() example  (Continued)

56789 0123 56a72
will assign 56 to i, 789.0 to x, skip 0123, and place the string 56\0 in name. The next call
to getchar(3C) will return the character a.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

fgetc(3C), fgets(3C), fgetwc(3C), fgetws(3C), fread(3C), fscanf(3C),
fwprintf(3C), getc(3C), getchar(3C), gets(3C), getwc(3C), getwchar(3C),
setlocale(3C), wcrtomb(3C), wcstod(3C), wcstol(3C), wcstoul(3C),
attributes(5), standards(5)
getcpuid, gethomelgroup – obtain information on scheduling decisions

NAME
getcpuid, gethomelgroup – obtain information on scheduling decisions

SYNOPSIS
#include <sys/processor.h>

processorid_t getcpuid(void);
lgrpid_t gethomelgroup(void);

DESCRIPTION
The getcpuid() function returns the processor ID on which the calling thread is currently executing.

The gethomelgroup() function returns the home latency group ID of the calling thread.

RETURN VALUES
See DESCRIPTION.

ERRORS
No errors are defined.

USAGE
Both the current CPU and the home latency group are subject to change at any time, so the value returned by these functions might already be incorrect upon completion of the call.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Stable</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
psradm(1M), psrinfo(1M), psrset(1M), p_online(2), processor_bind(2), processor_info(2), pset_assign(2), pset_bind(2), pset_info(2), meminfo(2), sysconf(3C), attributes(5)
getcwd(3C)

NAME
getcwd – get pathname of current working directory

SYNOPSIS
#include <unistd.h>

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

DESCRIPTION
The getcwd() function places an absolute pathname of the current working directory
in the array pointed to by buf, and returns buf. The size argument is the size in bytes of
the character array pointed to by buf and must be at least one greater than the length
of the pathname to be returned.

If buf is not a null pointer, the pathname is stored in the space pointed to by buf.

If buf is a null pointer, getcwd() obtains size bytes of space using malloc(3C). The
pointer returned by getcwd() can be used as the argument in a subsequent call to
free().

RETURN VALUES
Upon successful completion, getcwd() returns the buf argument. Otherwise, the
function returns a null pointer and sets errno to indicate the error.

ERRORS
The getcwd() function will fail if:

EINVAL
The size argument is equal to 0.

ERANGE
The size argument is greater than 0 and less than the length of the
pathname plus 1.

The getcwd() function may fail if:

EACCES
A parent directory cannot be read to get its name.

ENOMEM
Insufficient storage space is available.

USAGE
Applications should exercise care when using chdir(2) in conjunction with
getcwd(). The current working directory is global to all threads within a process. If
more than one thread calls chdir() to change the working directory, a subsequent
call to getcwd() could produce unexpected results.

EXAMPLES

EXAMPLE 1 Printing the current working directory

The following example prints the current working directory.

#include <unistd.h>
#include <stdio.h>

main( )
{
    char *cwd;
    if ((cwd = getcwd(NULL, 64)) == NULL) {
        perror("pwd");
        exit(2);
    }
    (void)printf("%s\n", cwd);
    return(0);
}
ATTRIBUTES  See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  chdir(2), malloc(3C), attributes(5)
getdate() function converts user-definable date and/or time specifications pointed to by `string` to a `tm` structure. The `tm` structure is defined in the `<time.h>` header.

User-supplied templates are used to parse and interpret the input string. The templates are text files created by the user and identified via the environment variable `DATEMSK`. Each line in the template represents an acceptable date and/or time specification using conversion specifications similar to those used by `strftime(3C)` and `strptime(3C)`. Dates before 1902 and after 2037 are illegal. The first line in the template that matches the input specification is used for interpretation and conversion into the internal time format.

The following conversion specifications are supported:

- `%%` Same as `%`.
- `%a` Locale’s abbreviated weekday name.
- `%A` Locale’s full weekday name.
- `%b` Locale’s abbreviated month name.
- `%B` Locale’s full month name.
- `%c` Locale’s appropriate date and time representation.
- `%C` Century number (the year divided by 100 and truncated to an integer as a decimal number [1,99]); single digits are preceded by 0; see `standards(5)`.
  - If used without the `%y` specifier, this format specifier will assume the current year offset in whichever century is specified. The only valid years are between 1902-2037.
- `%d` Day of month [01,31]; leading zero is permitted but not required.
- `%D` Date as `%m/%d/%y`.
- `%e` Same as `%d`.
- `%h` Locale’s abbreviated month name.
- `%H` Hour (24-hour clock) [0,23]; leading zero is permitted but not required.
- `%I` Hour (12-hour clock) [1,12]; leading zero is permitted but not required.
- `%j` Day number of the year [1,366]; leading zeros are permitted but not required.
- `%m` Month number [1,12]; leading zero is permitted but not required.
Minute [0,59]; leading zero is permitted but not required.
%n Any white space.
%p Locale’s equivalent of either a.m. or p.m.
%r Appropriate time representation in the 12-hour clock format with %p.
%R Time as %H:%M.
%S Seconds [0,61]; leading zero is permitted but not required. The range of values is [00,61] rather than [00,59] to allow for the occasional leap second and even more occasional double leap second.
%t Any white space.
%T Time as %H:%M:%S.
%U Week number of the year as a decimal number [0,53], with Sunday as the first day of the week; leading zero is permitted but not required.
%W Week number of the year as a decimal number [0,53], with Monday as the first day of the week; leading zero is permitted but not required.
%x Locale’s appropriate date representation.
%x Locale’s appropriate time representation.
%y Year within century. When a century is not otherwise specified, values in the range 69-99 refer to years in the twentieth century (1969 to 1999 inclusive); values in the range 00-68 refer to years in the twenty-first century (2000 to 2068 inclusive).
%Y Year, including the century (for example, 1993).
%Z Time zone name or no characters if no time zone exists.

Some conversion specifications can be modified by the E and C modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified specification. If the alternative format or specification does not exist in the current locale, the behavior be as if the unmodified conversion specification were used.

%Ec Locale’s alternative appropriate date and time representation.
%EC Name of the base year (period) in the locale’s alternative representation.
%Ex Locale’s alternative date representation.
%EX Locale’s alternative time representation.
%Ey Offset from %Ec (year only) in the locale’s alternative representation.
%EY Full alternative year representation.
Day of the month using the locale’s alternative numeric symbols; leading zeros are permitted but not required.

Same as %Od.

Hour (24-hour clock) using the locale’s alternative numeric symbols.

Hour (12-hour clock) using the locale’s alternative numeric symbols.

Month using the locale’s alternative numeric symbols.

Minutes using the locale’s alternative numeric symbols.

Seconds using the locale’s alternative numeric symbols.

Week number of the year (Sunday as the first day of the week) using the locale’s alternative numeric symbols.

Number of the weekday (Sunday=0) using the locale’s alternative numeric symbols.

Week number of the year (Monday as the first day of the week) using the locale’s alternative numeric symbols.

Year (offset from %C) in the locale’s alternative representation and using the locale’s alternative numeric symbols.

The following rules are applied for converting the input specification into the internal format:

- If only the weekday is given, today is assumed if the given day is equal to the current day and next week if it is less.
- If only the month is given, the current month is assumed if the given month is equal to the current month and next year if it is less and no year is given. (The first day of month is assumed if no day is given.)
- If only the year is given, the values of the tm_mon, tm_mday, tm_yday, tm_wday, and tm_isdst members of the returned tm structure are not specified.
- If the century is given, but the year within the century is not given, the current year within the century is assumed.
- If no hour, minute, and second are given, the current hour, minute, and second are assumed.
- If no date is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less.

A conversion specification that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the conversion specification, the specification fails, and the differing and subsequent characters remain unscanned.
A series of conversion specifications composed of %n, %t, white space characters, or any combination is executed by scanning up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

Any other conversion specification is executed by scanning characters until a character matching the next conversion specification is scanned, or until no more characters can be scanned. These characters, except the one matching the next conversion specification, are then compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate \texttt{tm} structure members are set to values corresponding to the locale information. If no match is found, \texttt{getdate()} fails and no more characters are scanned.

The month names, weekday names, era names, and alternative numeric symbols can consist of any combination of upper and lower case letters. The user can request that the input date or time specification be in a specific language by setting the \texttt{LC_TIME} category using \texttt{setlocale(3C)}.

**RETURN VALUES**

If successful, \texttt{getdate()} returns a pointer to a \texttt{tm} structure; otherwise, it returns \texttt{NULL} and sets the global variable \texttt{getdate_err} to indicate the error. Subsequent calls to \texttt{getdate()} alter the contents of \texttt{getdate_err}.

The following is a complete list of the \texttt{getdate_err} settings and their meanings:

1. The \texttt{DATEMSK} environment variable is null or undefined.
2. The template file cannot be opened for reading.
3. Failed to get file status information.
4. The template file is not a regular file.
5. An error is encountered while reading the template file.
6. The \texttt{malloc()} function failed (not enough memory is available).
7. There is no line in the template that matches the input.
8. The input specification is invalid (for example, February 31).

**USAGE**

The \texttt{getdate()} function makes explicit use of macros described on the \texttt{ctype(3C)} manual page.

**EXAMPLES**

**EXAMPLE 1** Examples of the \texttt{getdate()} function.

The following example shows the possible contents of a template:

\begin{verbatim}
%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
\end{verbatim}
EXAMPLE 1 Examples of the `getdate()` function. (Continued)

run job at %I %p, %B %dnd
%A %d. %B %Y %H.%M Uhr

The following are examples of valid input specifications for the above template:

```
getdate("10/1/87 4 PM")
getdate("Friday")
getdate("Friday September 19 1987, 10:30:30")
getdate("at monday the 1st of december in 1986")
getdate("run job at 3 PM, december 2nd")
```

If the `LANG` environment variable is set to de (German), the following is valid:

```
getdate("freitag den 10. oktober 1986 10.30 Uhr")
```

Local time and date specification are also supported. The following examples show how local date and time specification can be defined in the template.

<table>
<thead>
<tr>
<th>Invocation Line in Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>getdate(&quot;11/27/86&quot;) %m/%d/%y</td>
</tr>
<tr>
<td>getdate(&quot;27.11.86&quot;) %d.%m.%y</td>
</tr>
<tr>
<td>getdate(&quot;86-11-27&quot;) %y-%m-%d</td>
</tr>
<tr>
<td>getdate(&quot;Friday 12:00:00&quot;) %A %H:%M:%S</td>
</tr>
</tbody>
</table>

The following examples illustrate the Internal Format Conversion rules. Assume that the current date is Mon Sep 22 12:19:47 EDT 1986 and the `LANG` environment variable is not set.

<table>
<thead>
<tr>
<th>Input</th>
<th>Template Line</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon</td>
<td>%a</td>
<td>Mon Sep 22 12:19:48 EDT 1986</td>
</tr>
<tr>
<td>Sun</td>
<td>%a</td>
<td>Sun Sep 28 12:19:49 EDT 1986</td>
</tr>
<tr>
<td>Fri</td>
<td>%a</td>
<td>Fri Sep 26 12:19:49 EDT 1986</td>
</tr>
<tr>
<td>September</td>
<td>%B</td>
<td>Mon Sep 1 12:19:49 EDT 1986</td>
</tr>
<tr>
<td>January</td>
<td>%B</td>
<td>Thu Jan 1 12:19:49 EST 1987</td>
</tr>
<tr>
<td>December</td>
<td>%B</td>
<td>Mon Dec 1 12:19:49 EDT 1986</td>
</tr>
<tr>
<td>Sep Mon</td>
<td>%b %a</td>
<td>Mon Sep 1 12:19:50 EDT 1986</td>
</tr>
<tr>
<td>Jan Fri</td>
<td>%b %a</td>
<td>Fri Jan 2 12:19:50 EST 1987</td>
</tr>
</tbody>
</table>
ATTRIBUTES
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO
ctype(3C), mktime(3C), setlocale(3C), strftime(3C), strptime(3C), attributes(5), environ(5), standards(5)
### NAME
getdtablesize – get the file descriptor table size

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

int getdtablesize(void);
```

### DESCRIPTION
The `getdtablesize()` function is equivalent to `getrlimit(2)` with the `RLIMIT_NOFILE` option.

### RETURN VALUES
The `getdtablesize()` function returns the current soft limit as if obtained from a call to `getrlimit()` with the `RLIMIT_NOFILE` option.

### ERRORS
No errors are defined.

### USAGE
There is no direct relationship between the value returned by `getdtablesize()` and `OPEN_MAX` defined in `<limits.h>`.

Each process has a file descriptor table which is guaranteed to have at least 20 slots. The entries in the descriptor table are numbered with small integers starting at 0. The `getdtablesize()` function returns the current maximum size of this table by calling the `getrlimit()` function.

### SEE ALSO
`close(2), getrlimit(2), open(2), setrlimit(2), select(3C)`
#include <stdlib.h>

char *getenv(const char *name);

The `getenv()` function searches the environment list (see `environ(5)`) for a string of the form `name=value` and, if the string is present, returns a pointer to the `value` in the current environment.

If successful, `getenv()` returns a pointer to the `value` in the current environment; otherwise, it returns a null pointer.

The `getenv()` function can be safely called from a multithreaded application. Care must be exercised when using both `getenv()` and `putenv(3C)` in a multithreaded application. These functions examine and modify the environment list, which is shared by all threads in an application. The system prevents the list from being accessed simultaneously by two different threads. It does not, however, prevent two threads from successively accessing the environment list using `getenv()` or `putenv(3C)`.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

See also `exec(2), putenv(3C), attributes(5), environ(5)`
getexecname(3C)

NAME    getexecname – return pathname of executable

SYNOPSIS #include <stdlib.h>

        const char *getexecname(void);

DESCRIPTION The getexecname() function returns the pathname (the first argument of one of the exec family of functions; see exec(2)) of the executable that started the process.

Normally this is an absolute pathname, as the majority of commands are executed by the shells that append the command name to the user’s PATH components. If this is not an absolute path, the output of getcwd(3C) can be prepended to it to create an absolute path, unless the process or one of its ancestors has changed its root directory or current working directory since the last successful call to one of the exec family of functions.

RETURN VALUES If successful, getexecname() returns a pointer to the executables pathname; otherwise, it returns 0.

USAGE The getexecname() function obtains the executable pathname from the AT_SUN_EXECNAME aux vector. These vectors are made available to dynamically linked processes only.

A successful call to one of the exec family of functions will always have AT_SUN_EXECNAME in the aux vector. The associated pathname is guaranteed to be less than or equal to PATH_MAX, not counting the trailing null byte that is always present.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO exec(2), getcwd(3C), attributes(5)
These functions are used to obtain entries describing user groups. Entries can come from any of the sources for group specified in the /etc/nsswitch.conf file (see nsswitch.conf(4)).

The `getgrnam()` function searches the group database for an entry with the group name specified by the character string parameter `name`.

The `getgrgid()` function searches the group database for an entry with the (numeric) group id specified by `gid`.

The `setgrent()`, `getgrent()`, and `endgrent()` functions are used to enumerate group entries from the database.

The `setgrent()` function effectively rewinds the group database to allow repeated searches. It sets (or resets) the enumeration to the beginning of the set of group entries. This function should be called before the first call to `getgrent()`.
The `getgrent()` function returns a pointer to a structure containing the broken-out fields of an entry in the group database. When first called, `getgrent()` returns a pointer to a group structure containing the next group structure in the group database. Successive calls may be used to search the entire database.

The `endgrent()` function may be called to close the group database and deallocate resources when processing is complete. It is permissible, though possibly less efficient, for the process to call more group functions after calling `endgrent()`.

The `fgetgrent()` function, unlike the other functions above, does not use `nsswitch.conf`. It reads and parses the next line from the stream, which is assumed to have the format of the group file (see `group(4)`).

The `getgrnam()`, `getgrgid()`, `getgrent()`, and `fgetgrent()` functions use static storage that is reused in each call, making them unsafe for multithreaded applications.

The parallel functions `getgrnam_r()`, `getgrgid_r()`, `getgrent_r()`, and `fgetgrent_r()` provide reentrant interfaces for these operations.

Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the `_r` suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results, and are safe for use in both single-threaded and multithreaded applications.

Each reentrant interface takes the same arguments as its non-reentrant counterpart, as well as the following additional parameters. The `grp` argument must be a pointer to a `struct group` structure allocated by the caller. On successful completion, the function returns the group entry in this structure. Storage referenced by the group structure is allocated from the memory provided with the `buffer` argument, which is `bufsize` characters in size. The maximum size needed for this buffer can be determined with the `_SC_GETGR_R_SIZE_MAX` `sysconf(3C)` parameter. The POSIX versions place a pointer to the modified `grp` structure in the `result` parameter, instead of returning a pointer to this structure.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. `setgrent()` may be used in a multithreaded application but resets the enumeration position for all threads. If multiple threads interleave calls to `getgrent_r()`, the threads will enumerate disjoint subsets of the group database. Like their non-reentrant counterparts, `getgrnam_r()` and `getgrgid_r()` leave the enumeration position in an indeterminate state.

**RETURN VALUES**

Group entries are represented by the `struct group` structure defined in `<grp.h>`:

```c
struct group {
    char *gr_name;    /* the name of the group */
    char *gr_passwd;  /* the encrypted group password */
    gid_t gr_gid;     /* the numerical group ID */
    char **gr_mem;    /* vector of pointers to member names */
};
```

---

Getgrnam(3C)
The `getgrnam()`, `getgrnam_r()`, `getgrgid()`, and `getgrgid_r()` functions each return a pointer to a `struct group` if they successfully locate the requested entry; otherwise they return `NULL`. The POSIX functions `getgrnam_r()` and `getgrgid_r()` return 0 upon success or the error number in case of failure.

The `getgrent()`, `getgrent_r()`, `fgetgrent()`, and `fgetgrent_r()` functions each return a pointer to a `struct group` if they successfully enumerate an entry; otherwise they return `NULL`, indicating the end of the enumeration.

The `getgrnam()`, `getgrgid()`, `getgrent()`, and `fgetgrent()` functions use static storage, so returned data must be copied before a subsequent call to any of these functions if the data is to be saved.

When the pointer returned by the reentrant functions `getgrnam_r()`, `getgrgid_r()`, `getgrent_r()`, and `fgetgrent_r()` is non-null, it is always equal to the `grp` pointer that was supplied by the caller.

**ERRORS**

The `getgrnam()`, `getgrgid()`, `getgrent()`, `fgetgrent()`, and `fgetgrent_r()` functions may fail if:

- **EINTR** A signal was caught during the operation.
- **EIO** An I/O error has occurred.
- **EMFILE** There are `OPEN_MAX` file descriptors currently open in the calling process.
- **ENFILE** The maximum allowable number of files is currently open in the system.
- **ERANGE** The group file contains a line that exceeds 512 bytes.

The `getgrnam_r()`, `getgrgid_r()`, and `getgrent_r()` functions may fail if:

- **ERANGE** Insufficient storage was supplied by `buffer` and `bufsize` to contain the data to be referenced by the resulting `group` structure.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See “Reentrant Interfaces” in DESCRIPTION.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`Intro(3)`, `getpwnam(3C)`, `group(4)`, `nsswitch.conf(4)`, `passwd(4)`, `attributes(5)`, `standards(5)`

**NOTES**

When compiling multithreaded programs, see `Intro(3)`, `Notes On Multithreaded Applications`. 
Programs that use the interfaces described in this manual page cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time.

Use of the enumeration interfaces `getgrent()` and `getgrent_r()` is discouraged; enumeration is supported for the group file, NIS, and NIS+, but in general is not efficient and may not be supported for all database sources. The semantics of enumeration are discussed further in `nsswitch.conf(4)`.

Previous releases allowed the use of “+” and “−” entries in `/etc/group` to selectively include and exclude entries from NIS. The primary usage of these entries is superseded by the name service switch, so the “+/−” form may not be supported in future releases.

If required, the “+/−” functionality can still be obtained for NIS by specifying `compat` as the source for `group`.

If the “+/−” functionality is required in conjunction with NIS+, specify both `compat` as the source for `group` and `nisplus` as the source for the pseudo-database `group_compat`. See `group(4)`, and `nsswitch.conf(4)` for details.

Solaris 2.4 and earlier releases provided definitions of the `getgrnam_r()` and `getgrgid_r()` functions as specified in POSIX.1c Draft 6. The final POSIX.1c standard changed the interface for these functions. Support for the Draft 6 interface is provided for compatibility only and may not be supported in future releases. New applications and libraries should use the POSIX standard interface.

For POSIX.1c-compliant applications, the `_POSIX_PTHREAD_SEMANTICS` and `_REENTRANT` flags are automatically turned on by defining the `_POSIX_C_SOURCE` flag with a value >= 199506L.
NAME
gethostid – get an identifier for the current host

SYNOPSIS
#include <unistd.h>

long gethostid(void);

DESCRIPTION
The gethostid() function returns the 32-bit identifier for the current host. This identifier is taken from the CPU board’s ID PROM. It is not guaranteed to be unique.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
hostid(1), sysinfo(2), attributes(5)
The `gethostname()` function returns the standard host name for the current processor, as previously set by `sethostname()`. The `namelen` argument specifies the size of the array pointed to by `name`. The returned name is null-terminated unless insufficient space is provided.

The `sethostname()` function sets the name of the host machine to be `name`, which has length `namelen`. This call is restricted to the superuser and is normally used only when the system is bootstrapped.

Host names are limited to `MAXHOSTNAMELEN` characters, currently 256, defined in the `<netdb.h>` header.

Upon successful completion, `gethostname()` and `sethostname()` return 0. Otherwise, they return −1 and set `errno` to indicate the error.

The `gethostname()` and `sethostname()` functions will fail if:

- `EFAULT` The `name` or `namelen` argument gave an invalid address.

The `sethostname()` function will fail if:

- `EPERM` The caller was not the superuser.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

See also `sysinfo(2), uname(2), gethostid(3C), attributes(5)`
NAME | gethrtime, gethrvtime – get high resolution time  
SYNOPSIS | ```c
#include <sys/time.h>

hrtime_t gethrtime(void);
hrtime_t gethrvtime(void);
```
DESCRIPTION | The `gethrtime()` function returns the current high-resolution real time. Time is expressed as nanoseconds since some arbitrary time in the past; it is not correlated in any way to the time of day, and thus is not subject to resetting or drifting by way of `adjtime(2)` or `settimeofday(3C)`. The hi-res timer is ideally suited to performance measurement tasks, where cheap, accurate interval timing is required.

The `gethrvtime()` function returns the current high-resolution LWP virtual time, expressed as total nanoseconds of execution time. This function requires that micro state accounting be enabled with the `ptime` utility (see `proc(1)`).

The `gethrtime()` and `gethrvtime()` functions both return an `hrtime_t`, which is a 64-bit (long long) signed integer.

EXAMPLES | The following code fragment measures the average cost of `getpid(2)`:  
```c
hrtime_t start, end;
int i, iters = 100;

start = gethrtime();
for (i = 0; i < iters; i++)
    getpid();
end = gethrtime();

printf("Avg getpid() time = %lld nsec\n", (end - start) / iters);
```
ATTRIBUTES | See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>
SEE ALSO | `proc(1)`, `adjtime(2)`, `gettimeofday(3C)`, `settimeofday(3C)`, attributes(5)
NOTES | Although the units of hi-res time are always the same (nanoseconds), the actual resolution is hardware dependent. Hi-res time is guaranteed to be monotonic (it won’t go backward, it won’t periodically wrap) and linear (it won’t occasionally speed up or slow down for adjustment, like the time of day can), but not necessarily unique: two sufficiently proximate calls may return the same value.
getloadavg(3C)

NAME  getloadavg – get system load averages

SYNOPSIS

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

int getloadavg(double loadavg[], int nelem);
```

DESCRIPTION

The `getloadavg()` function returns the number of processes in the system run queue averaged over various periods of time. Up to `nelem` samples are retrieved and assigned to successive elements of `loadavg[]`. The system imposes a maximum of 3 samples, representing averages over the last 1, 5, and 15 minutes, respectively. The `LOADAVG_1MIN`, `LOADAVG_5MIN`, and `LOADAVG_15MIN` indices, defined in `<sys/loadavg.h>`, can be used to extract the data from the appropriate element of the `loadavg[]` array.

RETURN VALUES

Upon successful completion, the number of samples actually retrieved is returned. If the load average was unobtainable, −1 is returned and `errno` is set to indicate the error.

ERRORS

The `getloadavg()` function will fail if:

- `EINVAL` The number of elements specified is less than 0.

ATTRIBUTES

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

```
ATTRIBUTE TYPE       ATTRIBUTE VALUE

MT-Level             Async-Signal-Safe
```

SEE ALSO

`uptime(1)`, `w(1)`, `kstat(3KSTAT)`, `standards(5)`
NAME | getlogin, getlogin_r – get login name
SYNOPSIS | 
```c
#include <unistd.h>

char *getlogin(void);

char *getlogin_r(char *name, int namelen);
```
POSIX | 
```shell
cc [ flagi ] file... -D_POSIX_PTHREAD_SEMANTICS [ library... ]

int getlogin_r(char *name, size_t namesize);
```
DESCRIPTION | The getlogin() function returns a pointer to the login name as found in /var/adm/utmpx. It may be used in conjunction with getpwnam(3C) to locate the correct password file entry when the same user ID is shared by several login names.

If getlogin() is called within a process that is not attached to a terminal, it returns a null pointer. The correct procedure for determining the login name is to call cuserid(3C), or to call getlogin() and if it fails to call getpwuid(3C).

The getlogin_r() function has the same functionality as getlogin() except that the caller must supply a buffer name with length namelen to store the result. The name buffer must be at least _POSIX_LOGIN_NAME_MAX bytes in size (defined in `<limits.h>`). The POSIX version (see standards(5)) of getlogin_r() takes a namesize parameter of type size_t.

RETURN VALUES | Upon successful completion, getlogin() returns a pointer to the login name or a null pointer if the user’s login name cannot be found. Otherwise it returns a null pointer and sets errno to indicate the error.

The POSIX getlogin_r() returns 0 if successful, or the error number upon failure.

ERRORS | The getlogin() function may fail if:

- **EMFILE** There are OPEN_MAX file descriptors currently open in the calling process.
- **ENFILE** The maximum allowable number of files is currently open in the system.
- **ENXIO** The calling process has no controlling terminal.

The getlogin_r() function will fail if:

- **ERANGE** The size of the buffer is smaller than the result to be returned.
- **EINVAL** And entry for the current user was not found in the /var/adm/utmpx file.

USAGE | The return value may point to static data whose content is overwritten by each call.
getlogin(3C)

Three names associated with the current process can be determined:
getpwuid(geteuid()) returns the name associated with the effective user ID of the
process; getlogin() returns the name associated with the current login activity; and
getpwuid(getuid()) returns the name associated with the real user ID of the
process.

FILES /var/adm/utmpx user access and administration information

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See NOTES below.</td>
</tr>
</tbody>
</table>

SEE ALSO geteuid(2), getuid(2), cuserid(3C), getgrnam(3C), getpwnam(3C),
getpwuid(3C), utmpx(4), attributes(5), standards(5)

NOTES When compiling multithreaded programs, see Intro(3),
Notes On Multithreaded Applications.

The getlogin() function is unsafe in multithreaded applications. The
getlogin_r() function should be used instead.

Solaris 2.4 and earlier releases provided a getlogin_r() as specified in POSIX.1c
Draft 6. The final POSIX.1c standard changed the interface as described above.
Support for the Draft 6 interface is provided for compatibility only and may not be
supported in future releases. New applications and libraries should use the POSIX
standard interface.
getmntent(3C)

NAME
getmntent, getmntany, getextmntent, hasmntopt, putmntent, resetmnttab – get
mounted device information

SYNOPSIS
#include <stdio.h>
#include <sys/mnttab.h>

int getmntent(FILE *fp, struct mnttab *mp);
int getmntany(FILE *fp, struct mnttab *mp, struct mnttab *mpref);
int getextmntent(FILE *fp, struct extmnttab *mp, int len);
char *hasmntopt(struct mnttab *mnt, char *opt);
int putmntent(FILE *iop, struct mnttab *mp);
void resetmnttab(FILE *fp);

getmntent() and
getmntany()

The getmntent() and getmntany() functions each fill in the structure pointed to
by mp with the broken-out fields of a line in the mnttab file. Each line read from the
file contains a mnttab structure, which is defined in the <sys/mnttab.h> header.
The structure contains the following members, which correspond to the broken-out
fields from a line in /etc/mnttab (see mnttab(4)).

char *mnt_special; /* name of mounted resource */
char *mnt_mountp; /* mount point */
char *mnt_fstype; /* type of file system mounted */
char *mnt_mntopts; /* options for this mount */
char *mnt_time; /* time file system mounted */

Each getmntent() call causes a new line to be read from the mnttab file. Successive
calls can be used to search the entire list. The getmntany() function searches the file
referenced by fp until a match is found between a line in the file and mpref. A match
occurs if all non-null entries in mpref match the corresponding fields in the file. Note
that these functions do not open, close, or rewind the file.

getextmntent()

The getextmntent() function is an extended version of the getmntent() function
that returns, in addition to the information that getmntent() returns, the major and
minor number of the mounted resource to which the line in mnttab corresponds. The
getextmntent() function also fills in the extmntent structure defined in the
<sys/mnttab.h> header. For getextmntent() to function properly, it must be
notified when the mnttab file has been reopened or rewound since a previous
gextmntent() call. This notification is accomplished by calling resetmnttab().
Otherwise, it behaves exactly as getmntent() described above.

The data pointed to by the mnttab structure members are stored in a static area and
must be copied to be saved between successive calls.

hasmntopt()

The hasmntopt() function scans the mnt_mntopts member of the mnttab structure
mnt for a substring that matches opt. It returns the address of the substring if a match
is found; otherwise it returns 0. Substrings are delimited by commas and the end of
the mnt_mntopts string.

Basic Library Functions 211
**putmntent(3C)**

The `putmntent()` function is obsolete and no longer has any effect. Entries appear in `mnttab` as a side effect of a `mount(2)` call. The function name is still defined for transition purposes.

**resetmnttab()**

The `resetmnttab()` function notifies `getextmntent()` to reload from the kernel the device information that corresponds to the new snapshot of the `mnttab` information (see `mnttab(4)`). Subsequent `getextmntent()` calls then return correct `extmnttab` information. This function should be called whenever the `mnttab` file is either rewound or closed and reopened before any calls are made to `getextmntent()`.

**getmntent() and getmntany()**

If the next entry is successfully read by `getmntent()` or a match is found with `getmntany()`, 0 is returned. If an EOF is encountered on reading, these functions return -1. If an error is encountered, a value greater than 0 is returned. The following error values are defined in `<sys/mnttab.h>`:

- MNT_TOOLONG: A line in the file exceeded the internal buffer size of `MNT_LINE_MAX`.
- MNT_TOOMANY: A line in the file contains too many fields.
- MNT_TOOFEW: A line in the file contains too few fields.

**hasmntopt()**

Upon successful completion, `hasmntopt()` returns the address of the substring if a match is found. Otherwise, it returns 0.

**putmntent()**

The `putmntent()` is obsolete and always returns -1.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`mount(2)`, `mnttab(4)`, attributes(5)
These functions are used to test membership in and enumerate members of "netgroup" network groups defined in a system database. Netgroups are sets of (machine, user, domain) triples (see netgroup(4)).

These functions consult the source specified for netgroup in the /etc/nsswitch.conf file (see nsswitch.conf(4)).

The function innetgr() returns 1 if there is a netgroup netgroup that contains the specified machine, user, domain triple as a member; otherwise it returns 0. Any of the supplied pointers machine, user, and domain may be NULL, signifying a "wild card" that matches all values in that position of the triple.

The innetgr() function is safe for use in single-threaded and multithreaded applications.

The functions setnetgrent(), getnetgrent(), and endnetgrent() are used to enumerate the members of a given network group.

The function setnetgrent() establishes the network group specified in the parameter netgroup as the current group whose members are to be enumerated.

Successive calls to the function getnetgrent() will enumerate the members of the group established by calling setnetgrent(); each call returns 1 if it succeeds in obtaining another member of the network group, or 0 if there are no further members of the group.

When calling either getnetgrent() or getnetgrent_r(), addresses of the three character pointers are used as arguments, for example:

```c
char *mp, *up, *dp;
getnetgrent(&mp, &up, &dp);
```
Upon successful return from `getnetgrent()`, the pointer `mp` points to a string containing the name of the machine part of the member triple, `up` points to a string containing the user name and `dp` points to a string containing the domain name. If the pointer returned for `mp`, `up`, or `dp` is `NULL`, it signifies that the element of the netgroup contains wild card specifier in that position of the triple.

The pointers returned by `getnetgrent()` point into a buffer allocated by `setnetgrent()` that is reused by each call. This space is released when an `endnetgrent()` call is made, and should not be released by the caller. This implementation is not safe for use in multi-threaded applications.

The function `getnetgrent_r()` is similar to `getnetgrent()` function, but it uses a buffer supplied by the caller for the space needed to store the results. The parameter `buffer` should be a pointer to a buffer allocated by the caller and the length of this buffer should be specified by the parameter `buflen`. The buffer must be large enough to hold the data associated with the triple. The `getnetgrent_r()` function is safe for use both in single-threaded and multi-threaded applications.

The function `endnetgrent()` frees the space allocated by the previous `setnetgrent()` call. The equivalent of an `endnetgrent()` implicitly performed whenever a `setnetgrent()` call is made to a new network group.

Note that while `setnetgrent()` and `endnetgrent()` are safe for use in multi-threaded applications, the effect of each is process-wide. Calling `setnetgrent()` resets the enumeration position for all threads. If multiple threads interleave calls to `getnetgrent_r()` each will enumerate a disjoint subset of the netgroup. Thus the effective use of these functions in multi-threaded applications may require coordination by the caller.

**ERRORS**

The function `getnetgrent_r()` will return 0 and set `errno` to `ERANGE` if the length of the buffer supplied by caller is not large enough to store the result. See Intro(2) for the proper usage and interpretation of `errno` in multi-threaded applications.

The functions `setnetgrent()` and `endnetgrent()` return 0 upon success.

**FILES**

`/etc/nsswitch.conf`

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See DESCRIPTION section.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

Intro(2), Intro(3), netgroup(4), nsswitch.conf(4), attributes(5)

**WARNINGS**

The function `getnetgrent_r()` is included in this release on an uncommitted basis only, and is subject to change or removal in future minor releases.
Only the Network Information Services, NIS and NIS+, are supported as sources for the netgroup database.

Programs that use the interfaces described in this manual page cannot be linked statically since the implementations of these functions employ dynamic loading and linking of shared objects at run time.

When compiling multi-threaded applications, see Intro(3), Notes On Multithread Applications, for information about the use of the _REENTRANT flag.
The getopt() function returns the next option letter in *argv* that matches a letter in *optstring*. It supports all the rules of the command syntax standard (see `intro(1)`). 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.

The *optstring* argument 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. The optarg argument is set to point to the start of the option argument on return from getopt().

The getopt() function 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(). When all options have been processed (that is, up to the first non-option argument), getopt() returns −1. The special option “−−” (two hyphens) may be used to delimit the end of the options; when it is encountered, −1 is returned and “−−” is skipped. This is useful in delimiting non-option arguments that begin with “−” (hyphen).

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 (‘:’).

The getopt() function outputs an error message to standard error and returns a question mark (‘?’) when it encounters an option letter not included in *optstring* or no argument after an option that expects one. This error message can be disabled by setting opterr to 0. The value of the character that caused the error is in optopt.

Otherwise, getopt() returns −1 when all command line options are parsed.

No errors are defined.
EXAMPLE 1 Parsing Command Line Options

The following code fragment shows how you might process the arguments for a utility that can take the mutually-exclusive options a and b and the options f and o, both of which require arguments:

```c
#include <unistd.h>

int
main(int argc, char *argv[])
{
    int c;
    int bflg, aflg, errflg;
    char *ifile;
    char *ofile;
    extern char *optarg;
    extern int optind, optopt;
    ...
    while ((c = getopt(argc, argv, ':abf:o:')) != -1) {
        switch(c) {
        case 'a':
            if (bflg)
                errflg++;
            else
                aflg++;
            break;
        case 'b':
            if (aflg)
                errflg++;
            else {
                bflg++;
                bproc();
            }
            break;
        case 'f':
            ifile = optarg;
            break;
        case 'o':
            ofile = optarg;
            break;
        case ':': /* -f or -o without operand */
            fprintf(stderr,
                "Option -%c requires an operand\n", optopt);
            errflg++;
            break;
        case '?':
            fprintf(stderr,
                "Unrecognized option: -%c\n", optopt);
            errflg++;
        }
    }
    if (errflg) {
        fprintf(stderr, "usage:...");
        exit(2);
    }
    for ( ; optind < argc; optind++) {
        if (access(argv[optind], R_OK)) {
            ...
        }
```

Basic Library Functions 217
EXAMPLE 1 Parsing Command Line Options  (Continued)

This code accepts any of the following as equivalent:

```
  cmd -ao arg path path
  cmd -a -o arg path path
  cmd -o arg -a path path
  cmd -a -o arg -- path path
  cmd -a -o arg path path
```

EXAMPLE 2 Check Options and Arguments.

The following example parses a set of command line options and prints messages to standard output for each option and argument that it encounters.

```c
#include <unistd.h>
#include <stdio.h>
...
int c;
char *filename;
extern char *optarg;
extern int optind, optopt, opterr;
...
while ((c = getopt(argc, argv, "a:b:f:")) != -1) {
  switch(c) {
  case 'a':
    printf("a is set\n");
    break;
  case 'b':
    printf("b is set\n");
    break;
  case 'f':
    filename = optarg;
    printf("filename is \%s\n", filename);
    break;
  case ':':
    printf("-%c without filename\n", optopt);
    break;
  case '?':
    printf("unknown arg \%c\n", optopt);
    break;
  }
}
```

EXAMPLE 3 Select Options from the Command Line.

The following example selects the type of database routines the user wants to use based on the Options argument.

```c
#include <unistd.h>
#include <string.h>
...
```
EXAMPLE 3 Select Options from the Command Line.  (Continued)

```
char *Options = "hdbtl";
...
int dbtype, i;
char c;
char *st;
...
dbtype = 0;
while ((c = getopt(argc, argv, Options)) != -1) {
    if ((st = strchr(Options, c)) != NULL) {
        dbtype = st - Options;
        break;
    }
}
```

See environ(5) for descriptions of the following environment variables that affect the execution of getopt(): LANG, LC_ALL, and LC_MESSAGES.

**LC_CTYPE** Determine the locale for the interpretation of sequences of bytes as characters in optstring.

**USAGE**

The getopt() function does not fully check for mandatory arguments; that is, given an option string a:b and the input -a -b, getopt() assumes that -b is the mandatory argument to the -a option and not that -a is missing a mandatory argument.

It is a violation of the command syntax standard (see intro(1)) for options with arguments to be grouped with other options, as in cmd -abo filename, where a and b are options, o is an option that requires an argument, and filename is the argument to o. Although this syntax is permitted in the current implementation, it should not be used because it may not be supported in future releases. The correct syntax to use is:

```
    cmd -ab -o filename
```

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Unsafe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

intro(1), getopt(1), getopts(1), getsubopt(3C), gettext(3C), setlocale(3C), attributes(5), environ(5), standards(5)
NAME | getpagesize - get system page size
SYNOPSIS | #include <unistd.h>
int getpagesize(void);
DESCRIPTION | The getpagesize() function returns the number of bytes in a page. Page
granularity is the granularity of many of the memory management calls.
The page size is a system page size and need not be the same as the underlying
hardware page size.
The getpagesize() function is equivalent to sysconf(_SC_PAGE_SIZE) and
sysconf(_SC_PAGESIZE). See sysconf(3C).
RETURN VALUES | The getpagesize() function returns the current page size.
ERRORS | No errors are defined.
USAGE | The value returned by getpagesize() need not be the minimum value that
malloc(3C) can allocate. Moreover, the application cannot assume that an object of
this size can be allocated with malloc().
ATTRIBUTES | See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO | pagesize(1), brk(2), getrlimit(2), mmap(2), mprotect(2), munmap(2),
malloc(3C), msync(3C), sysconf(3C), attributes(5)
The `getpagesize()` function returns either the number of different page sizes supported by the system or the actual sizes themselves. When called with `nelem` as 0 and `pagesize` as `NULL`, `getpagesize()` returns the number of supported page sizes. Otherwise, up to `nelem` page sizes are retrieved and assigned to successive elements of `pagesize[]`. The return value is the number of page sizes retrieved and set in `pagesize[]`.

Upon successful completion, the number of pagesizes supported or actually retrieved is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

The `getpagesize()` function will fail if:

- `EINVAL` The `nelem` argument is less than 0 or `pagesize` is `NULL` but `nelem` is non-zero.

The `getpagesize()` function returns all the page sizes for which the hardware and system software provide support for the `memcntl(2)` command `MC_HATMAPSIZE`. However, not all processors support all page sizes and/or combinations of page sizes with equal efficiency. Applications programmers should take this into consideration when using `getpagesize()`.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

See also `memcntl(2)`, `mmap(2)`, `getpagesize(3C)`, `attributes(5)`
getpass(3C)

NAME
getpass, getpassphrase – read a string of characters without echo

Default
#include <stdlib.h>
char *getpass(const char *prompt);
char *getpassphrase(const char *prompt);

XPG4, SUS, SUSv2
#include <unistd.h>
char *getpass(const char *prompt);

DESCRIPTION
The getpass() function opens the process’s controlling terminal, writes to that
device the null-terminated string prompt, disables echoing, reads a string of characters
up to the next newline character or EOF, restores the terminal state and closes the
terminal.

The getpassphrase() function is identical to getpass(), except that it reads and
returns a string of up to 256 characters in length.

RETURN VALUES
Upon successful completion, getpass() returns a pointer to a null-terminated string
of at most PASS_MAX bytes that were read from the terminal device. If an error is
encountered, the terminal state is restored and a null pointer is returned.

ERRORS
The getpass() and getpassphrase() functions may fail if:

EINTR The function was interrupted by a signal.

EIO The process is a member of a background process attempting to
read from its controlling terminal, the process is ignoring or
blocking the SIGTTIN signal or the process group is orphaned.

EMFILE OPEN_MAX file descriptors are currently open in the calling
process.

ENFILE The maximum allowable number of files is currently open in the
system.

ENXIO The process does not have a controlling terminal.

USAGE
The return value points to static data whose content may be overwritten by each call.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Unsafe</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5), standards(5)
getpriority(3C)

NAME
getpriority, setpriority – get or set process scheduling priority

SYNOPSIS
#include <sys/resource.h>

int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int priority);

DESCRIPTION
The getpriority() function obtains the current scheduling priority of a process, process group, or user. The setpriority() function sets the scheduling priority of a process, process group, or user.

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, PRIO_USER, PRIO_GROUP, PRIO_SESSION, PRIO_LWP, PRIO_LWP, or PRIO_PROJECT, indicating that the who argument is to be interpreted as a process ID, a process group ID, a user ID, a group ID, a session ID, an lwp ID, a task ID, or a project ID, respectively. A 0 value for the who argument specifies the current process, process group, or user. A 0 value for the who argument is treated as valid group ID, session ID, lwp ID, task ID, or project ID. A P_MYID value for the who argument can be used to specify the current group, session, lwp, task, or project, respectively.

If more than one process is specified, getpriority() returns the highest priority (lowest numerical value) pertaining to any of the specified processes, and setpriority() sets the priorities of all of the specified processes to the specified value.

The default priority is 0; negative priorities cause more favorable scheduling. While the range of valid priority values is \([-20, 20]\], implementations may enforce more restrictive limits. If the value specified to setpriority() is less than the system’s lowest supported priority value, the system’s lowest supported value is used. If it is greater than the system’s highest supported value, the system’s highest supported value is used.

Only a process with appropriate privileges can raise its priority (that is, assign a lower numerical priority value).

RETURN VALUES
Upon successful completion, getpriority() returns an integer in the range from \(-20\) to \(20\). 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 recognized, or the value of the who argument is not a valid process ID, process group ID, user ID, group ID, session ID, lwp ID, task ID, or project ID.
In addition, `setpriority()` may fail if:

- **EPERM**: A process was located, but neither the real nor effective user ID of the executing process is the privileged user or match the effective user ID of the process whose 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.

**USAGE**

The effect of changing the scheduling priority can vary depending on the process-scheduling algorithm in effect.

Because `getpriority()` can return −1 on successful completion, it is necessary to set `errno` to 0 prior to a call to `getpriority()`. If `getpriority()` returns −1, then `errno` can be checked to see if an error occurred or if the value is a legitimate priority.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`nice(1)`, `renice(1)`, `fork(2)`, `attributes(5)`
getpw(3C)

NAME
getpw – get passwd entry from UID

SYNOPSIS
#include <stdlib.h>

int getpw(uid_t uid, char *buf);

DESCRIPTION
The getpw() function searches the user data base for a user id number that equals
uid, copies the line of the password file in which uid was found into the array pointed
to by buf, and returns 0. getpw() returns non-zero if uid cannot be found.

USAGE
This function is included only for compatibility with prior systems and should not be
used; the functions described on the getpwnam(3C) manual page should be used
instead.

If the /etc/passwd and the /etc/group files have a plus sign (+) for the NIS entry,
then getpwent() and getgrent() will not return NULL when the end of file is
reached. See getpwnam(3C).

RETURN VALUES
The getpw() function returns non-zero on error.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
getpwnam(3C), passwd(4), attributes(5)
getpwnam(3C)

NAME  getpwnam, getpwnam_r, getpwent, getpwent_r, getpwuid, getpwuid_r, setpwent,
      endpwent, fgetpwent, fgetpwent_r – get password entry

SYNOPSIS  #include <pwd.h>

        struct passwd *getpwnam(const char *name);
        struct passwd *getpwnam_r(const char *name, struct passwd *pwd,
                                   char *buffer, int buflen);
        struct passwd *getpwent(void);
        struct passwd *getpwent_r(struct passwd *pwd, char *buffer, int buflen);
        struct passwd *getpwuid(uid_t uid);
        struct passwd *getpwuid_r(uid_t uid, struct passwd *pwd, char
                                   *buffer, int buflen);
        void setpwent(void);
        void endpwent(void);
        struct passwd *fgetpwent(FILE *f);
        struct passwd *fgetpwent_r(FILE *f, struct passwd *pwd, char
                                   *buffer, int buflen);

DESCRIPTION  These functions are used to obtain password entries. Entries can come from any of the
              sources for passwd specified in the /etc/nsswitch.conf file (see
              nsswitch.conf(4)).

              The getpwnam() function searches for a password entry with the login name
              specified by the character string parameter name.

              The getpwuid() function searches for a password entry with the (numeric) user ID
              specified by the parameter uid.

              The setpwent(), getpwent(), and endpwent() functions are used to enumerate
              password entries from the database. setpwent() sets (or resets) the enumeration to
              the beginning of the set of password entries. This function should be called before
              the first call to getpwent(). Calls to getpwnam() and getpwuid() leave the
              enumeration position in an indeterminate state. Successive calls to getpwent() return
              either successive entries or NULL, indicating the end of the enumeration.
The `endpwent()` function may be called to indicate that the caller expects to do no further password retrieval operations; the system may then close the password file, deallocate resources it was using, and so forth. It is still allowed, but possibly less efficient, for the process to call more password functions after calling `endpwent()`.

The `fgetpwent()` function, unlike the other functions above, does not use `nsswitch.conf`; it reads and parses the next line from the stream `f`, which is assumed to have the format of the `passwd` file. See `passwd(4)`.

The functions `getpwnam()`, `getpwuid()`, `getpwent()`, and `fgetpwent()` use static storage that is reused in each call, making these routines unsafe for use in multithreaded applications.

The parallel functions `getpwnam_r()`, `getpwuid_r()`, `getpwent_r()`, and `fgetpwent_r()` provide reentrant interfaces for these operations.

Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the "_r" suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results, and are safe for use in both single-threaded and multithreaded applications.

Each reentrant interface takes the same parameters as its non-reentrant counterpart, as well as the following additional parameters. The parameter `pwd` must be a pointer to a `struct passwd` structure allocated by the caller. On successful completion, the function returns the password entry in this structure. The parameter `buffer` is a pointer to a buffer supplied by the caller, used as storage space for the password data. All of the pointed to the returned `struct passwd` point to data stored within this buffer; see RETURN VALUES. The buffer must be large enough to hold all the data associated with the password entry. The parameter `buflen` (or `bufsize` for the POSIX versions; see `standards(5)`) should give the size in bytes of `buffer`. The POSIX versions place a pointer to the modified `pwd` structure in the `result` parameter, instead of returning a pointer to this structure.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. The `setpwent()` function may be used in a multithreaded application but resets the enumeration position for all threads. If multiple threads interleave calls to `getpwent_r()`, the threads will enumerate disjoint subsets of the password database.

Like their non-reentrant counterparts, `getpwnam_r()` and `getpwuid_r()` leave the enumeration position in an indeterminate state.

### RETURN VALUES
Password entries are represented by the `struct passwd` structure defined in `<pwd.h>`:

```c
struct passwd {
    char *pw_name;  /* user's login name */
    char *pw_passwd; /* no longer used */
    uid_t pw_uid;   /* user's uid */
    gid_t pw_gid;   /* user's gid */
    char *pw_age;   /* not used */
};
```

Basic Library Functions  227
The `pw_passwd` member should not be used as the encrypted password for the user; use `getspnam()` or `getspnam_r()` instead. See `getspnam(3C)`. The `getpwnam()`, `getpwnam_r()`, `getpwuid()`, and `getpwuid_r()` functions each return a pointer to a `struct passwd` if they successfully locate the requested entry; otherwise they return `NULL`. Upon successful completion (including the case when the requested entry is not found), the POSIX functions `getpwnam_r()` and `getpwuid_r()` return 0. Otherwise, an error number is returned to indicate the error. The `getpwent()`, `getpwent_r()`, `fgetpwent()`, and `fgetpwent_r()` functions each return a pointer to a `struct passwd` if they successfully enumerate an entry; otherwise they return `NULL`, indicating the end of the enumeration. The `getpwnam()`, `getpwuid()`, `getpwent()`, and `fgetpwent()` functions use static storage, so returned data must be copied before a subsequent call to any of these functions if the data is to be saved. When the pointer returned by the reentrant functions `getpwnam_r()`, `getpwuid_r()`, `getpwent_r()`, and `fgetpwent_r()` is non-null, it is always equal to the `pwd` pointer that was supplied by the caller. **ERRORS** The reentrant functions `getpwnam_r()`, `getpwuid_r()`, `getpwent_r()`, and `fgetpwent_r()` will return `NULL` and set `errno` to `ERANGE` (or in the case of POSIX functions `getpwnam_r()` and `getpwuid_r()` return the `ERANGE` error) if the length of the buffer supplied by caller is not large enough to store the result. See `Intro(2)` for the proper usage and interpretation of `errno` in multithreaded applications. **USAGE** Applications that use the interfaces described on this manual page cannot be linked statically, since the implementations of these functions employ dynamic loading and linking of shared objects at run time. **ATTRIBUTES** See `attributes(5)` for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See “Reentrant Interfaces” in DESCRIPTION.</td>
</tr>
</tbody>
</table>

**SEE ALSO** `nispasswd(1), passwd(1), yppasswd(1), Intro(2), Intro(3), cuserid(3C), getgrnam(3C), getlogin(3C), getspnam(3C), nsswitch.conf(4), passwd(4), shadow(4), attributes(5), standards(5)`

**NOTES** When compiling multithreaded programs, see `Intro(3), Notes On Multithreaded Applications.`
Use of the enumeration interfaces \texttt{getpwent()} and \texttt{getpwent\_r()} is discouraged; enumeration is supported for the \texttt{passwd} file, NIS, and NIS+, but in general is not efficient and may not be supported for all database sources. The semantics of enumeration are discussed further in \texttt{nsswitch.conf(4)}.

Previous releases allowed the use of ‘+’ and ‘-’ entries in \texttt{/etc/passwd} to selectively include and exclude NIS entries. The primary usage of these ‘+/-’ entries is superseded by the name service switch, so the ‘+/-’ form may not be supported in future releases.

If required, the ‘+/-’ functionality can still be obtained for NIS by specifying \texttt{compat} as the source for \texttt{passwd}.

If the ‘+/-’ functionality is required in conjunction with NIS+, specify both \texttt{compat} as the source for \texttt{passwd} and \texttt{nisplus} as the source for the pseudo-database \texttt{passwd\_compat}. See \texttt{passwd(4)}, \texttt{shadow(4)}, and \texttt{nsswitch.conf(4)} for details.

If the ‘+/-’ is used, both \texttt{/etc/shadow} and \texttt{/etc/passwd} should have the same ‘+’ and ‘-’ entries to ensure consistency between the password and shadow databases.

If a password entry from any of the sources contains an empty \texttt{uid} or \texttt{gid} field, that entry will be ignored by the files, NIS, and NIS+ name service switch backends. This will cause the user to appear unknown to the system.

If a password entry contains an empty \texttt{gecos}, \texttt{home directory}, or \texttt{shell} field, \texttt{getpwnam()} and \texttt{getpwnam\_r()} return a pointer to a null string in the respective field of the \texttt{passwd} structure.

If the shell field is empty, \texttt{login(1)} automatically assigns the default shell. See \texttt{login(1)}.

Solaris 2.4 and earlier releases provided definitions of the \texttt{getpwnam\_r()} and \texttt{getpwuid\_r()} functions as specified in POSIX.1c Draft 6. The final POSIX.1c standard changed the interface for these functions. Support for the Draft 6 interface is provided for compatibility only and may not be supported in future releases. New applications and libraries should use the POSIX standard interface.

For POSIX.1c-compliant applications, the \texttt{POSIX\_PTHREAD\_SEMANTICS} and \texttt{REENTRANT} flags are automatically turned on by defining the \texttt{POSIX\_C\_SOURCE} flag with a value \( \geq 199506L \).
getrusage(3C)

NAME
getrusage – get information about resource utilization

SYNOPSIS
#include <sys/resource.h>

int getrusage(int who, struct rusage *r_usage);

DESCRIPTION
The getrusage() function provides measures of the resources used by the current
process or its terminated and waited-for child processes. If the value of the who
argument is RUSAGE_SELF, information is returned about resources used by the
current process. If the value of the who argument is RUSAGE_CHILDREN, information
is returned about resources used by the terminated and waited-for children of the
current process. If the child is never waited for (for instance, if the parent has
SA_NOCLEWDWAIT set or sets SIGCHLD to SIG_IGN), the resource information for the
child process is discarded and not included in the resource information provided by
getrusage().

The r_usage argument is a pointer to an object of type struct rusage in which the
returned information is stored. The members of rusage are as follows:

```
struct timeval ru_utime; /* user time used */
struct timeval ru_stime; /* system time used */
long ru_maxrss; /* maximum resident set size */
long ru_idrss; /* integral resident set size */
long ru_minflt; /* page faults not requiring physical I/O */
long ru_majflt; /* page faults requiring physical I/O */
long ru_nswap; /* swaps */
long ru_inblock; /* block input operations */
long ru_oublock; /* block output operations */
long ru_msgsnd; /* messages sent */
long ru_msgrcv; /* messages received */
long ru_nsignals; /* signals received */
long ru_nvcsw; /* voluntary context switches */
long ru_nivcsw; /* involuntary context switches */
```

The structure members are interpreted as follows:

ru_utime The total amount of time spent executing in user mode. Time is
given in seconds and microseconds.

ru_stime The total amount of time spent executing in system mode. Time is
given in seconds and microseconds.

ru_maxrss The maximum resident set size. Size is given in pages (the size of a
page, in bytes, is given by the getpagesize(3C) function). See
the NOTES section of this page.

ru_idrss An “integral” value indicating the amount of memory in use by a
process while the process is running. This value is the sum of the
resident set sizes of the process running when a clock tick occurs.
The value is given in pages times clock ticks. It does not take
sharing into account. See the NOTES section of this page.

ru_minflt The number of page faults serviced which did not require any
physical I/O activity. See the NOTES section of this page.
ru_majflt The number of page faults serviced which required physical I/O activity. This could include page ahead operations by the kernel. See the NOTES section of this page.

ru_nswap The number of times a process was swapped out of main memory.

ru_inblock The number of times the file system had to perform input in servicing a read(2) request.

ru_oublock The number of times the file system had to perform output in servicing a write(2) request.

ru_msgsnd The number of messages sent over sockets.

ru_msgrcv The number of messages received from sockets.

ru_nsignals The number of signals delivered.

ru_nvcsw The number of times a context switch resulted due to a process voluntarily giving up the processor before its time slice was completed (usually to await availability of a resource).

ru_nivcsw The number of times a context switch resulted due to a higher priority process becoming runnable or because the current process exceeded its time slice.

RETURN VALUES Upon successful completion, getrusage() returns 0. Otherwise, −1 is returned and errno is set to indicate the error.

ERRORS The getrusage() function will fail if:

EFAULT The address specified by the r_usage argument is not in a valid portion of the process’ address space.

EINVAL The who parameter is not a valid value.

SEE ALSO sar(1M), read(2), times(2), wait(2), write(2), getpagesize(3C), gettimeofday(3C)

NOTES Only the timeval member of struct rusage are supported in this implementation.

The numbers ru_inblock and ru_oublock account only for real I/O, and are approximate measures at best. Data supplied by the cache mechanism is charged only to the first process to read and the last process to write the data.

The way resident set size is calculated is an approximation, and could misrepresent the true resident set size.
Page faults can be generated from a variety of sources and for a variety of reasons. The customary cause for a page fault is a direct reference by the program to a page which is not in memory. Now, however, the kernel can generate page faults on behalf of the user, for example, servicing \texttt{read(2)} and \texttt{write(2)} functions. Also, a page fault can be caused by an absent hardware translation to a page, even though the page is in physical memory.

In addition to hardware detected page faults, the kernel may cause pseudo page faults in order to perform some housekeeping. For example, the kernel may generate page faults, even if the pages exist in physical memory, in order to lock down pages involved in a raw I/O request.

By definition, major page faults require physical I/O, while minor page faults do not require physical I/O. For example, reclaiming the page from the free list would avoid I/O and generate a minor page fault. More commonly, minor page faults occur during process startup as references to pages which are already in memory. For example, if an address space faults on some “hot” executable or shared library, this results in a minor page fault for the address space. Also, any one doing a \texttt{read(2)} or \texttt{write(2)} to something that is in the page cache will get a minor page fault(s) as well.

There is no way to obtain information about a child process which has not yet terminated.
NAME
gets, fgets – get a string from a stream

SYNOPSIS
#include <stdio.h>

char *gets(char *s);
char *fgets(char *s, int n, FILE *stream);

DESCRIPTION
The gets() function reads bytes from the standard input stream (see intro(3)), stdin, into the array pointed to by s, until a newline character is read or an end-of-file condition is encountered. The newline character is discarded and the string is terminated with a null byte.

If the length of an input line exceeds the size of s, indeterminate behavior may result. For this reason, it is strongly recommended that gets() be avoided in favor of fgets().

The fgets() function reads bytes from the stream into the array pointed to by s, until n−1 bytes are read, or a newline character is read and transferred to s, or an end-of-file condition is encountered. The string is then terminated with a null byte.

The fgets() function may mark the st_atime field of the file associated with stream for update. The st_atime field will be marked for update by the first successful execution of fgetc(3C), fgets(1), fgetwc(3C), fgetws(3C), fread(3C), fscanf(3C), getc(3C), getchar(3C), gets(1), or scanf(3C) using stream that returns data not supplied by a prior call to ungetc(3C) or ungetwc(3C).

RETURN VALUES
If end-of-file is encountered and no bytes have been read, no bytes are transferred to s and a null pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a null pointer is returned and the error indicator for the stream is set. If end-of-file is encountered, the EOF indicator for the stream is set. Otherwise s is returned.

ERRORS
Refer to fgetc(3C).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
lseek(2), read(2), ferror(3C), fgetc(3C), fgetwc(3C), fopen(3C), fread(3C), getchar(3C), scanf(3C), stdio(3C), ungetc(3C), ungetwc(3C), attributes(5)
These functions are used to obtain shadow password entries. An entry may come from any of the sources for shadow specified in the /etc/nsswitch.conf file (see nsswitch.conf(4)).

The getspnam() function searches for a shadow password entry with the login name specified by the character string argument name.

The setspent(), getspent(), and endspent() functions are used to enumerate shadow password entries from the database.

The setspent() function sets (or resets) the enumeration to the beginning of the set of shadow password entries. This function should be called before the first call to getspent(). Calls to getspnam() leave the enumeration position in an indeterminate state.

Successive calls to getspent() return either successive entries or NULL, indicating the end of the enumeration.

The endspent() function may be called to indicate that the caller expects to do no further shadow password retrieval operations; the system may then close the shadow password file, deallocate resources it was using, and so forth. It is still allowed, but possibly less efficient, for the process to call more shadow password functions after calling endspent().

The fgetspent() function, unlike the other functions above, does not use nsswitch.conf; it reads and parses the next line from the stream fp, which is assumed to have the format of the shadow file (see shadow(4)).

The getspnam(), getspent(), and fgetspent() functions use static storage that is re-used in each call, making these routines unsafe for use in multithreaded applications.
The `getspnam_r()`, `getspent_r()`, and `fgetspent_r()` functions provide reentrant interfaces for these operations.

Each reentrant interface performs the same operation as its non-reentrant counterpart, named by removing the `_r` suffix. The reentrant interfaces, however, use buffers supplied by the caller to store returned results, and are safe for use in both single-threaded and multithreaded applications.

Each reentrant interface takes the same argument as its non-reentrant counterpart, as well as the following additional arguments. The `result` argument must be a pointer to a `struct spwd` structure allocated by the caller. On successful completion, the function returns the shadow password entry in this structure. The `buffer` argument must be a pointer to a buffer supplied by the caller. This buffer is used as storage space for the shadow password data. All of the pointers within the returned `struct spwd` point to data stored within this buffer (see RETURN VALUES). The buffer must be large enough to hold all of the data associated with the shadow password entry. The `buflen` argument should give the size in bytes of the buffer indicated by `buffer`.

For enumeration in multithreaded applications, the position within the enumeration is a process-wide property shared by all threads. The `setspent()` function may be used in a multithreaded application but resets the enumeration position for all threads. If multiple threads interleave calls to `getspent_r()`, the threads will enumerate disjoint subsets of the shadow password database.

Like its non-reentrant counterpart, `getspnam_r()` leaves the enumeration position in an indeterminate state.

### RETURN VALUES

Password entries are represented by the `struct spwd` structure defined in `<shadow.h>`:

```c
struct spwd{
    char *sp_namp; /* login name */
    char *sp_pwdp; /* encrypted passwd */
    long sp_lstchg; /* date of last change */
    long sp_min; /* min days to passwd change */
    long sp_max; /* max days to passwd change */
    long sp_warn; /* warning period */
    long sp_inact; /* max days inactive */
    long sp_expire; /* account expiry date */
    unsigned long sp_flag; /* not used */
};
```

See shadow(4) for more information on the interpretation of this data.

The `getspnam()` and `getspnam_r()` functions each return a pointer to a `struct spwd` if they successfully locate the requested entry; otherwise they return `NULL`.

The `getspent()`, `getspent_r()`, `fgetspent()`, and `fgetspent()` functions each return a pointer to a `struct spwd` if they successfully enumerate an entry; otherwise they return `NULL`, indicating the end of the enumeration.
getspnam(3C)

The getspnam(), getspent(), and fgetspent() functions use static storage, so returned data must be copied before a subsequent call to any of these functions if the data is to be saved.

When the pointer returned by the reentrant functions getspnam_r(), getspent_r(), and fgetspent_r() is non-null, it is always equal to the result pointer that was supplied by the caller.

ERRORS

The reentrant functions getspnam_r(), getspent_r(), and fgetspent_r() will return NULL and set errno to ERANGE if the length of the buffer supplied by caller is not large enough to store the result. See intro(2) for the proper usage and interpretation of errno in multithreaded applications.

USAGE

Applications that use the interfaces described on this manual page cannot be linked statically, since the implementations of these functions employ dynamic loading and linking of shared objects at run time.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See “Reentrant Interfaces” in DESCRIPTION.</td>
</tr>
</tbody>
</table>

SEE ALSO

nispasswd(1), passwd(1), yppasswd(1), intro(3), getlogin(3C), getpwnam(3C), nsswitch.conf(4), passwd(4), shadow(4), attributes(5)

WARNINGS

The reentrant interfaces getspnam_r(), getspent_r(), and fgetspent_r() are included in this release on an uncommitted basis only, and are subject to change or removal in future minor releases.

NOTES

When compiling multithreaded applications, see intro(3), Notes On Multithreaded Applications, for information about the use of the _REENTRANT flag.

Use of the enumeration interfaces getspent() and getspent_r() is not recommended; enumeration is supported for the shadow file, NIS, and NIS+, but in general is not efficient and may not be supported for all database sources. The semantics of enumeration are discussed further in nsswitch.conf(4).

Access to shadow password information may be restricted in a manner depending on the database source being used. Access to the /etc/shadow file is generally restricted to processes running as the super-user (root). Other database sources may impose stronger or less stringent restrictions.

When NIS is used as the database source, the information for the shadow password entries is obtained from the “passwd.byname” map. This map stores only the information for the sp_namp and sp_pwdp fields of the struct spwd structure. Shadow password entries obtained from NIS will contain the value -1 in the remainder of the fields.
When NIS+ is used as the database source, and the caller lacks the permission needed to retrieve the encrypted password from the NIS+ “passwd.org_dir” table, the NIS+ service returns the string “*NP*” instead of the actual encrypted password string. The functions described on this page will then return the string “*NP*” to the caller as the value of the member sp_pwdp in the returned shadow password structure.
getsubopt(3C)

NAME      getsubopt – parse suboptions from a string

SYNOPSIS  
#include <stdlib.h>

int getsubopt(char **optionp, char * const *tokens, char **valuep);

DESCRIPTION The getsubopt() function parses suboptions in a flag argument that was initially parsed by getopt(3C). The 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; if present in the option input string, they are changed to null characters. White spaces within tokens or token-value pairs must be protected from the shell by quotes.

The syntax described above is used in the following example by the mount(1M), utility, which allows the user to specify mount parameters with the -o option as follows:

mount -o rw,hard,bg,wsize=1024 speed:/usr /usr

In this example there are four suboptions: rw, hard, bg, and wsize, the last of which has an associated value of 1024.

The getsubopt() function 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 pointed to by 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, a non-null value for valuep indicates that the suboption that was processed included a value. The calling program may use this information to determine if the presence or absence of a value for this suboption is an error.

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.

RETURN VALUES The getsubopt() function 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.
EXAMPLE 1 Example of getsubopt() function.

The following example demonstrates the processing of options to the mount(1M) utility using getsubopt().

```c
#include <stdlib.h>

char *myopts[] = {
#define READONLY 0
    "ro",
#define READWRITE 1
    "rw",
#define WRITESIZE 2
    "wsize",
#define READSIZE 3
    "rsize",
    NULL};

main(argc, argv)
    int argc;
    char **argv;
{
    int sc, c, errflag;
    char *options, *value;
    extern char *optarg;
    extern int optind;

    while((c = getopt(argc, argv, "abf:o:")) != -1) {
        switch (c) {
        case 'a': /* process a option */
            break;
        case 'b': /* process b option */
            break;
        case 'f':
            ofile = optarg;
            break;
        case '?':
            errflag++;
            break;
        case 'o':
            options = optarg;
            while (*options != '\0') {
                switch(getsubopt(options, myopts, &value)) {
                case READONLY : /* process ro option */
                    break;
                case READWRITE : /* process rw option */
                    break;
                case WRITESIZE : /* process wsize option */
                    if (value != NULL) {
                        error_no_arg();
                        errflag++;
                    } else
                        write_size = atoi(value);
                    break;
                case READSIZE : /* process rsize option */
```
EXAMPLE 1 Example of getsubopt() function.  

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 */

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

mount(1M), getopt(3C), attributes(5)
gettext(3C)

NAME

gettext, dgettext, ngettext, dngettext, dcgettext, dcngettext, textdomain, bindtextdomain, bind_textdomain_codeset – message handling functions

Solaris and GNU-compatible

#include <libintl.h>
char *gettext(const char *msgid);
char *dgettext(const char *domainname, const char *msgid);
char *textdomain(const char *domainname);
char *bindtextdomain(const char *domainname, const char *dirname);

#include <locale.h>
char *dcgettext(const char *domainname, const char *msgid, int category);

GNU-compatible

#include <libintl.h>
char *ngettext(const char *msgid1, const char *msgid2, unsigned long int n);
char *dngettext(const char *domainname, const char *msgid1, const char *msgid2, unsigned long int n);
char *bind_textdomain_codeset(const char *domainname, const char *codeset);

#include <locale.h>
char *dcngettext(const char *domainname, const char *msgid1, const char *msgid2, unsigned long int n, int category);

DESCRIPTION

The gettext(), dgettext() and dcgettext() functions attempt to retrieve a target string based on the specified msgid argument within the context of a specific domain and the current locale. The length of strings returned by gettext(), dgettext() and dcgettext() is undetermined until the function is called. The msgid argument is a null-terminated string.

The ngettext(), dngettext() and dcngettext() functions are equivalent to gettext(), dgettext() and dcgettext(), respectively, except for the handling of plural forms. These functions work only with GNU-compatible message catalogues.

The ngettext(), dngettext() and dcngettext() functions search for the message string using the msgid1 argument as the key and the n argument to determine the plural form. If no message catalogues are found, msgid1 is returned if n == 1, otherwise msgid2 is returned.

The NLSPATH environment variable (see environ(5)) is searched first for the location of the LC_MESSAGES catalogue. The setting of the LC_MESSAGES category of the current locale determines the locale used by gettext() and dgettext() for string retrieval. The category argument determines the locale used by dcgettext(). If NLSPATH is not defined and the current locale is "C", gettext(), dgettext() and
gettext(3C)

dgettext() simply return the message string that was passed. In a locale other
than "C", if NLSPATH is not defined or if a message catalogue is not found in any of the
components specified by NLSPATH, the routines search for the message catalogue
using the scheme described in the following paragraph.

The LANGUAGE environment variable is examined to determine the GNU-compatible
message catalogues to be used. The value of LANGUAGE is a list of locale names
separated by a colon (':') character. If LANGUAGE is defined, each locale name is tried in
the specified order and if a GNU-compatible message catalogue is found, the message
is returned. If a GNU-compatible message catalogue is found but failed to find a
corresponding msgid, the msgid string is return. If LANGUAGE is not defined or if a
Solaris message catalogue is found or no GNU-compatible message catalogue is found
in processing LANGUAGE, the pathname used to locate the message catalogue is
dirname/locale/category/domainname.mo, where dirname is the directory specified by
bindtextdomain(), locale is a locale name, and category is either LC_MESSAGES if
gettext(), dgettext(), ngettext() or dgettext() is called, or LC_XXX
where the name is the same as the locale category name specified by the category
argument to dcgettext() or dcngettext().

For gettext() and ngettext(), the domain used is set by the last valid call to
textdomain(). If a valid call to textdomain() has not been made, the default
domain (called messages) is used.

For dgettext(), dcgettext(), dgettext(), and dcgettext(), the domain
used is specified by the domainname argument. The domainname argument is equivalent
in syntax and meaning to the domainname argument to textdomain(), except that
the selection of the domain is valid only for the duration of the dgettext(),
dcgettext(), dgettext(), or dcgettext() function call.

The textdomain() function sets or queries the name of the current domain of the
active LC_MESSAGES locale category. The domainname argument is a null-terminated
string that can contain only the characters allowed in legal filenames.

The domainname argument is the unique name of a domain on the system. If there are
multiple versions of the same domain on one system, namespace collisions can be
avoided by using bindtextdomain(). If textdomain() is not called, a default
domain is selected. The setting of domain made by the last valid call to
textdomain() remains valid across subsequent calls to setlocale(3C), and
gettext().

The domainname argument is applied to the currently active LC_MESSAGES locale.

The current setting of the domain can be queried without affecting the current state of
the domain by calling textdomain() with domainname set to the null pointer. Calling
textdomain() with a domainname argument of a null string sets the domain to the
default domain (messages).

The bindtextdomain() function binds the path predicate for a message domain
domainname to the value contained in dirname. If domainname is a non-empty string and
has not been bound previously, bindtextdomain() binds domainname with dirname.
If `domainname` is a non-empty string and has been bound previously, `bindtextdomain()` replaces the old binding with `dirname`. The `dirname` argument can be an absolute or relative pathname being resolved when `gettext()`, `dgettext()`, or `dcgettext()` are called. If `domainname` is a null pointer or an empty string, `bindtextdomain()` returns NULL. User defined domain names cannot begin with the string `SYS_`. Domain names beginning with this string are reserved for system use.

The `bind_textdomain_codeset()` function can be used to specify the output codeset for message catalogues for domain `domainname`. The `codeset` argument must be a valid codeset name that can be used for the `iconv_open(3C)` function, or a null pointer. If the `codeset` argument is the null pointer, `bind_textdomain_codeset()` returns the currently selected codeset for the domain with the name `domainname`. It returns a null pointer if a codeset has not yet been selected. The `bind_textdomain_codeset()` function can be used multiple times. If used multiple times with the same `domainname` argument, the later call overrides the settings made by the earlier one. The `bind_textdomain_codeset()` function returns a pointer to a string containing the name of the selected codeset. The string is allocated internally in the function and must not be changed by the user.

The `gettext()`, `dgettext()`, and `dcgettext()` functions return the message string if the search succeeds. Otherwise they return the `msgid` string.

The `ngettext()`, `dngettext()`, and `dcngettext()` functions return the message string if the search succeeds. If the search fails, `msgid1` is returned if `n == 1`. Otherwise `msgid2` is returned.

The individual bytes of the string returned by `gettext()`, `dgettext()`, `dcgettext()`, `ngettext()`, `dngettext()`, or `dcngettext()` can contain any value other than NULL. If `msgid` is a null pointer, the return value is undefined. The string returned must not be modified by the program and can be invalidated by a subsequent call to `bind_textdomain_codeset()` or `setlocale(3C)`. If the `domainname` argument to `dgettext()`, `dcgettext()`, `ngettext()`, `dngettext()`, or `dcngettext()` is a null pointer, the the domain currently bound by `textdomain()` is used.

The normal return value from `textdomain()` is a pointer to a string containing the current setting of the domain. If `domainname` is a null pointer, `textdomain()` returns a pointer to the string containing the current domain. If `textdomain()` was not previously called and `domainname` is a null string, the name of the default domain is returned. The name of the default domain is `messages`. If `textdomain()` fails, a null pointer is returned.

The return value from `bindtextdomain()` is a null-terminated string containing `dirname` or the directory binding associated with `domainname` if `dirname` is NULL. If no binding is found, the default return value is `/usr/lib/locale`. If `domainname` is a null pointer or an empty string, `bindtextdomain()` takes no action and returns a null pointer. The string returned must not be modified by the caller. If `bindtextdomain()` fails, a null pointer is returned.

**RETURN VALUES**

The `gettext()`, `dgettext()`, and `dcgettext()` functions return the message string if the search succeeds. Otherwise they return the `msgid` string.

The `ngettext()`, `dngettext()`, and `dcngettext()` functions return the message string if the search succeeds. If the search fails, `msgid1` is returned if `n == 1`. Otherwise `msgid2` is returned.

The individual bytes of the string returned by `gettext()`, `dgettext()`, `dcgettext()`, `ngettext()`, `dngettext()`, or `dcngettext()` can contain any value other than NULL. If `msgid` is a null pointer, the return value is undefined. The string returned must not be modified by the program and can be invalidated by a subsequent call to `bind_textdomain_codeset()` or `setlocale(3C)`. If the `domainname` argument to `dgettext()`, `dcgettext()`, `ngettext()`, `dngettext()`, or `dcngettext()` is a null pointer, the the domain currently bound by `textdomain()` is used.

The normal return value from `textdomain()` is a pointer to a string containing the current setting of the domain. If `domainname` is a null pointer, `textdomain()` returns a pointer to the string containing the current domain. If `textdomain()` was not previously called and `domainname` is a null string, the name of the default domain is returned. The name of the default domain is `messages`. If `textdomain()` fails, a null pointer is returned.

The return value from `bindtextdomain()` is a null-terminated string containing `dirname` or the directory binding associated with `domainname` if `dirname` is NULL. If no binding is found, the default return value is `/usr/lib/locale`. If `domainname` is a null pointer or an empty string, `bindtextdomain()` takes no action and returns a null pointer. The string returned must not be modified by the caller. If `bindtextdomain()` fails, a null pointer is returned.

**gettext(3C)**
These functions impose no limit on message length. However, a text domainname is limited to TEXTDOMAINMAX (256) bytes.

The gettext(), dgettext(), dcgettext(), ngettext(), dngettext(), dcgettext(), textdomain(), and bindtextdomain() functions can be used safely in multithreaded applications, as long as setlocale(3C) is not being called to change the locale.

The gettext(), dgettext(), dcgettext(), textdomain(), and bindtextdomain() functions work with both Solaris message catalogues and GNU-compatible message catalogues. The ngettext(), dngettext(), dcgettext(), and bind_textdomain_codeset() functions work only with GNU-compatible message catalogues. See msgfmt(1) for information about Solaris message catalogues and GNU-compatible message catalogues.

FILES
/usr/lib/locale
default path predicate for message domain files
/usr/lib/locale/locale/LC_MESSAGES/domainname.mo
system default location for file containing messages for language locale and domainname
/usr/lib/locale/locale/LC_XXX/domainname.mo
system default location for file containing messages for language locale and domainname for dcgettext() calls where LC_XXX is LC_CTYPE, LC_NUMERIC, LC_TIME, LC_COLLATE, LC_MONETARY, or LC_MESSAGES
dirname/locale/LC_MESSAGES/domainname.mo
location for file containing messages for domain domainname and path predicate dirname after a successful call to bindtextdomain()
dirname/locale/LC_XXX/domainname.mo
location for files containing messages for domain domainname, language locale, and path predicate dirname after a successful call to bindtextdomain() for dcgettext() calls where LC_XXX is one of LC_CTYPE, LC_NUMERIC, LC_TIME, LC_COLLATE, LC_MONETARY, or LC_MESSAGES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe with exceptions</td>
</tr>
</tbody>
</table>

SEE ALSO msgfmt(1), xgettext(1), iconv_open(3C), setlocale(3C), attributes(5), environ(5)
gettimeofday, settimeofday – get or set the date and time

#include <sys/time.h>

int gettimeofday(struct timeval *tp, void *);
int settimeofday(struct timeval *tp, void *);

The `gettimeofday()` function gets and the `settimeofday()` function sets the system’s notion of the current time. The current time is expressed in elapsed seconds and microseconds since 00:00 Universal Coordinated Time, January 1, 1970. The resolution of the system clock is hardware dependent; the time may be updated continuously or in clock ticks.

The `tp` argument points to a `timeval` structure, which includes the following members:

long tv_sec; /* seconds since Jan. 1, 1970 */
long tv_usec; /* and microseconds */

If `tp` is a null pointer, the current time information is not returned or set.

The `TZ` environment variable holds time zone information. See `TIMEZONE(4)`.

The second argument to `gettimeofday()` and `settimeofday()` is ignored.

Only the super-user may set the time of day.

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

The `gettimeofday()` function will fail if:

- **EINVAL** The structure pointed to by `tp` specifies an invalid time.
- **EPERM** A user other than the privileged user attempted to set the time or time zone.

Additionally, the `gettimeofday()` function will fail for 32-bit interfaces if:

- **EOVERFLOW** The system time has progressed beyond 2038, thus the size of the `tv_sec` member of the `timeval` structure pointed to by `tp` is insufficient to hold the current time in seconds.

If the `tv_usec` member of `tp` is > 500000, `settimeofday()` rounds the seconds upward. If the time needs to be set with better than one second accuracy, call `settimeofday()` for the seconds and then `adjtime(2)` for finer accuracy.

See `attributes(5)` for descriptions of the following attributes:
### gettimeofday(3C)

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

#### SEE ALSO

adjtime(2), ctime(3C), TIMEZONE(4), attributes(5)
NAME
gmtimeofday, setmtimeofday – get or set the date and time

SYNOPSIS
/usr/ucb/cc [ flag ... ] file ...
#include <sys/time.h>

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

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

DESCRIPTION
The system’s notion of the current Greenwich time is obtained with the
gmtimeofday() call, and set with the setmtimeofday() 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 clock ticks.

long tv_sec; /* seconds since Jan. 1, 1970 */
long tv_usec; /* and microseconds */

tp points to a timeval structure, which includes the following members:

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 VALUES
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 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.

SEE ALSO
adjtime(2), ctime(3C), gettimeofday(3C), TIMEZONE(4)

NOTES
Use of these interfaces should be restricted to only applications written on BSD
platforms. Use of these interfaces with any of the system libraries or in multi-thread
applications is unsupported.

tzp is ignored in SunOS 5.X releases.
tv_usec is always 0.
gettxt(3C)

**NAME**
gettxt – retrieve a text string

**SYNOPSIS**
```c
#include <nl_types.h>
char *gettxt(const char *msgid, const char *dflt_str);
```

**DESCRIPTION**
The `gettxt()` function retrieves a text string from a message file. The arguments to the function are a message identification `msgid` and a default string `dflt_str` to be used if the retrieval fails.

The text strings are in files created by the `mkmsgs` utility (see `mkmsgs(1)`) and installed in directories in `/usr/lib/locale/locale/LC_MESSAGES`.

The directory `locale` can be viewed as the language in which the text strings are written. The user can request that messages be displayed in a specific language by setting the environment variable `LC_MESSAGES`. If `LC_MESSAGES` is not set, the environment variable `LANG` will be used. If `LANG` is not set, the files containing the strings are in `/usr/lib/locale/C/LC_MESSAGES/*`.

The user can also change the language in which the messages are displayed by invoking the `setlocale(3C)` function with the appropriate arguments.

If `gettxt()` fails to retrieve a message in a specific language it will try to retrieve the same message in U.S. English. On failure, the processing depends on what the second argument `dflt_str` points to. A pointer to the second argument is returned if the second argument is not the null string. If `dflt_str` points to the null string, a pointer to the U.S. English text string "Message not found!!\n" is returned.

The following depicts the acceptable syntax of `msgid` for a call to `gettxt()`.

```c
msgid = <msgfilename>:<msgnumber>
```

The first field is used to indicate the file that contains the text strings and must be limited to 14 characters. These characters must be selected from the set of all character values excluding \0 (null) and the ASCII code for / (slash) and : (colon). The names of message files must be the same as the names of files created by `mkmsgs` and installed in `/usr/lib/locale/locale/LC_MESSAGES/*`. The numeric field indicates the sequence number of the string in the file. The strings are numbered from 1 to `n` where `n` is the number of strings in the file.

**RETURN VALUES**
Upon failure to pass either the correct `msgid` or a valid message number to `gettxt()`, a pointer to the text string "Message not found!!\n" is returned.

**USAGE**
It is recommended that `gettext(3C)` be used in place of this function.

**EXAMPLES**

**EXAMPLE 1** Example of `gettxt()` function.

In the following example,
EXAMPLE 1 Example of gettext() function. (Continued)

```
gettext("UX:10", "hello world\n")
gettext("UX:10", "")
```

UX is the name of the file that contains the messages and 10 is the message number.

FILES

- `/usr/lib/locale/C/LC_MESSAGES/*`
  - contains default message files created by `mkmsgs`
- `/usr/lib/locale/locale/LC_MESSAGES/*`
  - contains message files for different languages created by `mkmsgs`

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe with exceptions</td>
</tr>
</tbody>
</table>

SEE ALSO

`exstr(1)`, `mkmsgs(1)`, `srtxt(1)`, `gettext(3C)`, `fmtmsg(3C)`, `setlocale(3C)`, `attributes(5)`, `environ(5)`
NAME
getusershell, setusershell, endusershell – get legal user shells

SYNOPSIS
char *getusershell()

void setusershell()

void endusershell()

DESCRIPTION
The getusershell() function returns a pointer to a legal user shell as defined by
the system manager in the file /etc/shells. If /etc/shells does not exist, the
following locations of the standard system shells are used in its place:

/bin/bash /bin/csh
/bin/jsh /bin/ksh
/bin/pfcsa /bin/pfsh
/bin/tcsh /bin/zsh
/sbin/jsh /sbin/sh
/usr/bin/bash /usr/bin/csh
/usr/bin/jsh /usr/bin/ksh
/usr/bin/pfcsa /usr/bin/pfsh
/usr/bin/tcsh /usr/bin/zsh
/usr/xpg4/bin/sh

The getusershell() function opens the file /etc/shells, if it exists, and returns
the next entry in the list of shells.

The setusershell() function rewinds the file or the list.

The endusershell() function closes the file, frees any memory used by
getusershell() and setusershell(), and rewinds the file /etc/shells.

RETURN VALUES
The getusershell() function returns a null pointer on EOF.

BUGS
All information is contained in memory that may be freed with a call to
endusershell(), so it must be copied if it is to be saved.
These functions provide access to the user accounting database, utmp. Entries in the database are described by the definitions and data structures in `<utmp.h>`.

The utmp structure contains the following members:

- `char ut_user[8]; /* user login name */`
- `char ut_id[4]; /* /sbin/inittab id (usually line #) */`
- `char ut_line[12]; /* device name (console, lnxn) */`
- `short ut_pid; /* process id */`
- `short ut_type; /* type of entry */`
- `struct exit_status ut_exit; /* exit status of a process */`
  - `short e_termination; /* termination status */`
  - `short e_exit; /* exit status */`
- `time_t ut_time; /* time entry was made */`

The structure exit_status includes the following members:

- `short e_termination; /* termination status */`
- `short e_exit; /* exit status */`

**getutent()**

The getutent() function reads in the next entry from a utmp database. If the database is not already open, it opens it. If it reaches the end of the database, it fails.

**getutid()**

The getutid() function searches forward from the current point in the utmp database until it finds an entry with a ut_type matching id->ut_type if the type specified is RUN_LVL, BOOT_TIME, OLD_TIME, or NEW_TIME. If the type specified in id is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, or DEAD_PROCESS, then getutid() will return a pointer to the first entry whose type is one of these four and whose ut_id member matches id->ut_id. If the end of database is reached without a match, it fails.

**getutline()**

The getutline() function searches forward from the current point in the utmp database until it finds an entry of the type LOGIN_PROCESS or ut_line string matching the line->ut_line string. If the end of database is reached without a match, it fails.
pututline()  The `pututline()` function writes the supplied `utmp` structure into the `utmp` database. It uses `getutid()` to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of `pututline()` will have searched for the proper entry using one of the these functions. If so, `pututline()` will not search. If `pututline()` does not find a matching slot for the new entry, it will add a new entry to the end of the database. It returns a pointer to the `utmp` structure. When called by a non-root user, `pututline()` invokes a `setuid()` root program to verify and write the entry, since the `utmp` database is normally writable only by root. In this event, the `ut_name` member must correspond to the actual user name associated with the process; the `ut_type` member must be either `USER_PROCESS` or `DEAD_PROCESS`; and the `ut_line` member must be a device special file and be writable by the user.

setutent()  The `setutent()` function resets the input stream to the beginning. This reset should be done before each search for a new entry if it is desired that the entire database be examined.

dendutent()  The `endutent()` function closes the currently open database.

utmpname()  The `utmpname()` function allows the user to change the name of the database file examined to another file. If the file does not exist, this will not be apparent until the first attempt to reference the file is made. The `utmpname()` function does not open the file but closes the old file if it is currently open and saves the new file name.

RETURN VALUES  A null pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write. If the file name given is longer than 79 characters, `utmpname()` returns 0. Otherwise, it returns 1.

USAGE  These functions use buffered standard I/O for input, but `pututline()` uses an unbuffered non-standard write to avoid race conditions between processes trying to modify the `utmp` and `wtmp` databases.

Applications should not access the `utmp` and `wtmp` databases directly, but should use these functions to ensure that these databases are maintained consistently. Using these functions, however, may cause applications to fail if user accounting data cannot be represented properly in the `utmp` structure (for example, on a system where PIDs can exceed 32767). Use the functions described on the `getutxent(3C)` manual page instead.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Unsafe</td>
</tr>
</tbody>
</table>

SEE ALSO  `getutxent(3C)`, `ttyslot(3C)`, `utmpx(4)`, `attributes(5)`
The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. On each call to either `getutid()` or `getutline()`, the function examines the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason, to use `getutline()` to search for multiple occurrences, it would be necessary to zero out the static area after each success, or `getutline()` would just return the same structure over and over again. There is one exception to the rule about emptying the structure before further reads are done. The implicit read done by `pututline()` (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the `getutent()`, `getutid()` or `getutline()` functions, if the user has just modified those contents and passed the pointer back to `pututline()`.
getutxent(3C)

NAME
getutxent, getutxid, getutxline, pututxline, setutxent, endutxent, utmpxname,
getutmp, getutmpx, updwtmp, updwtmpx – user accounting database functions

SYNOPSIS
#include <utmpx.h>

struct utmpx *getutxent(void);
struct utmpx *getutxid(const struct utmpx *id);
struct utmpx *getutxline(const struct utmpx *line);
struct utmpx *pututxline(const struct utmpx *utmpx);
void setutxent(void);
void endutxent(void);
int utmpxname(const char *file);
void getutmp(struct utmpx *utmpx, struct utmp *utmp);
void getutmpx(struct utmp *utmp, struct utmpx *utmpx);
void updwtmp(char *wfile, struct utmp *utmp);
void updwtmpx(char *wfilex, struct utmpx *utmpx);

DESCRIPTION
These functions provide access to the user accounting database, utmpx (see utmpx(4)).
Entries in the database are described by the definitions and data structures in
<utmpx.h>.

The utmpx structure contains the following members:

char ut_user[32]; /* user login name */
char ut_id[4]; /* /etc/inittab id (usually line #) */
char ut_line[32]; /* device name (console, lnxx) */
pid_t ut_pid; /* process id */
short ut_type; /* type of entry */
struct exit_status ut_exit; /* exit status of a process */
/* marked as DEAD_PROCESS */
struct timeval ut_tv; /* time entry was made */
int ut_session; /* session ID, used for windowing */
short ut_syslen; /* significant length of ut_host */
/* including terminating null */
char ut_host[257]; /* host name, if remote */

The exit_status structure includes the following members:

short e_termination; /* termination status */
short e_exit; /* exit status */

getutxent()  The getutxent() function reads in the next entry from a utmpx database. If the
database is not already open, it opens it. If it reaches the end of the database, it fails.

getutxid()  The getutxid() function searches forward from the current point in the utmpx
database until it finds an entry with a ut_type matching id->ut_type, if the type
specified is RUN_LVL, BOOT_TIME, OLD_TIME, or NEW_TIME. If the type specified in
id is INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, or DEAD_PROCESS, then
getutxid() will return a pointer to the first entry whose type is one of these four
and whose ut_id member matches id->ut_id. If the end of database is reached
without a match, it fails.

getutxline() The getutxline() function searches forward from the current point in the utmpx
database until it finds an entry of the type LOGIN_PROCESS or USER_PROCESS which
also has a ut_line string matching the line->ut_line string. If the end of the database
is reached without a match, it fails.

pututxline() The pututxline() function writes the supplied utmpx structure into the utmpx
database. It uses getutxid() to search forward for the proper place if it finds that it
is not already at the proper place. It is expected that normally the user of
pututxline() will have searched for the proper entry using one of the getutx() routines. So if, pututxline() will not search. If pututxline() does not find a
matching slot for the new entry, it will add a new entry to the end of the database. It
returns a pointer to the utmpx structure. When called by a non-root user,
pututxline() invokes a setuid() root program to verify and write the entry, since
the utmpx database is normally writable only by root. In this event, the ut_name
member must correspond to the actual user name associated with the process; the
ut_type member must be either USER_PROCESS or DEAD_PROCESS; and the
ut_line member must be a device special file and be writable by the user.

setutxent() The setutxent() function resets the input stream to the beginning. This should be
done before each search for a new entry if it is desired that the entire database be
examined.

endutxent() The endutxent() function closes the currently open database.

utmpxname() The utmpxname() function allows the user to change the name of the database file
examined from /var/adm/utmpx to any other file, most often /var/adm/wtmpx. If
the file does not exist, this will not be apparent until the first attempt to reference the
file is made. The utmpxname() function does not open the file, but closes the old file
if it is currently open and saves the new file name. The new file name must end with
the “x” character to allow the name of the corresponding utmp file to be easily
obtainable.; otherwise, an error value of 0 is returned. The function returns 1 on
success.

getutmp() The getutmp() function copies the information stored in the members of the utmpx
structure to the corresponding members of the utmp structure. If the information in
any member of utmpx does not fit in the corresponding utmp member, the data is
silently truncated. (See getutent(3C) for utmp structure)

getutmpx() The getutmpx() function copies the information stored in the members of the utmp
structure to the corresponding members of the utmpx structure. (See getutent(3C)
for utmp structure)

updwtmp() The updwtmp() function can be used in two ways.
getutxent(3C)

If \textit{wfile} is /var/adm/wtmp, the utmp format record supplied by the caller is converted to a utmpx format record and the /var/adm/wtmpx file is updated (because the /var/adm/wtmp file no longer exists, operations on wtmp are converted to operations on wtmpx by the library functions.

If \textit{wfile} is a file other than /var/adm/wtmp, it is assumed to be an old file in utmp format and is updated directly with the utmp format record supplied by the caller.

\textbf{updwtmpx()}

The \texttt{updwtmpx()} function writes the contents of the utmpx structure pointed to by \textit{utmpx} to the database.

\textbf{utmpx structure}

The values of the e\_termination and e\_exit members of the ut\_exit structure are valid only for records of type DEAD\_PROCESS. For utmpx entries created by \texttt{init(1M)}, these values are set according to the result of the \texttt{wait()} call that \texttt{init} performs on the process when the process exits. See the \texttt{wait(2)} manual page for the values \texttt{init} uses. Applications creating \texttt{utmpx} entries can set ut\_exit values using the following code example:

\begin{verbatim}
  u->ut_exit.e_termination = WTERMSIG(process->p_exit)
  u->ut_exit.e_exit = WEXITSTATUS(process->p_exit)
\end{verbatim}

See \texttt{wstat(3XFN)} for descriptions of the WTERMSIG and WEXITSTATUS macros.

The ut\_session member is not acted upon by the operating system. It is used by applications interested in creating utmpx entries.

For records of type USER\_PROCESS, the nonuser() and nonuserx() macros use the value of the ut\_exit.e\_exit member to mark utmpx entries as real logins (as opposed to multiple xterms started by the same user on a window system). This allows the system utilities that display users to obtain an accurate indication of the number of actual users, while still permitting each pty to have a utmpx record (as most applications expect). The NONROOT\_USER macro defines the value that login places in the ut\_exit.e\_exit member.

\textbf{RETURN VALUES}

Upon successful completion, getutxent(), getutxid(), and getutxline() each return a pointer to a utmpx structure containing a copy of the requested entry in the user accounting database. Otherwise a null pointer is returned.

The return value may point to a static area which is overwritten by a subsequent call to getutxid() or getutxline().

Upon successful completion, pututxline() returns a pointer to a utmpx structure containing a copy of the entry added to the user accounting database. Otherwise a null pointer is returned.

The \texttt{endutxent()} and \texttt{setutxent()} functions return no value.

A null pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.
These functions use buffered standard I/O for input, but pututxline() uses an unbuffered write to avoid race conditions between processes trying to modify the utmpx and wtmpx files.

Applications should not access the utmpx and wtmpx databases directly, but should use these functions to ensure that these databases are maintained consistently.

FILES
/var/adm/utmpx user access and accounting information
/var/adm/wtmpx history of user access and accounting information

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Unsafe</td>
</tr>
</tbody>
</table>

SEE ALSO
wait(2), getutent(3C), ttyslot(3C), utmpx(4), attributes(5), wstat(3XFN)

NOTES
The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made. On each call to either getutxid() or getutxline(), the routine examines the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason, to use getutxline() to search for multiple occurrences it would be necessary to zero out the static after each success, or getutxline() would just return the same structure over and over again. There is one exception to the rule about emptying the structure before further reads are done. The implicit read done by pututxline() (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the getutxent(), getutxid(), or getutxline() routines, if the user has just modified those contents and passed the pointer back to pututxline().
include <stdio.h>
#include <sys/vfstab.h>

int getvfsent(FILE *fp, struct vfstab *vp);
int getvfsfile(FILE *fp, struct vfstab *vp, char *file);
int getvfsspec(FILE *, struct vfstab *vp, char *spec);
int getvfsany(FILE *, struct vfstab *vp, struct vfstab *vref);

The getvfsent(), getvfsfile(), getvfsspec(), and getvfsany() functions each fill in the structure pointed to by vp with the broken-out fields of a line in the /etc/vfstab file. Each line in the file contains a vfstab structure, declared in the <sys/vfstab.h> header, whose following members are described on the vfstab(4) manual page:

char *vfs_special;
char *vfs_fsckdev;
char *vfs_mountp;
char *vfs_fstype;
char *vfs_fsckpass;
char *vfs_automnt;
char *vfs_mntopts;

The getvfsent() function returns a pointer to the next vfstab structure in the file; so successive calls can be used to search the entire file.

The getvfsfile() function searches the file referenced by fp until a mount point matching file is found and fills vp with the fields from the line in the file.

The getvfsspec() function searches the file referenced by fp until a special device matching spec is found and fills vp with the fields from the line in the file. The spec argument will try to match on device type (block or character special) and major and minor device numbers. If it cannot match in this manner, then it compares the strings.

The getvfsany() function searches the file referenced by fp until a match is found between a line in the file and vref. A match occurs if all non-null entries in vref match the corresponding fields in the file.

Note that these functions do not open, close, or rewind the file.

RETURN VALUES

If the next entry is successfully read by getvfsent() or a match is found with getvfsfile(), getvfsspec(), or getvfsany(), 0 is returned. If an end-of-file is encountered on reading, these functions return −1. If an error is encountered, a value greater than 0 is returned. The possible error values are:

VFS_TOOFEW A line in the file contains too few fields.
VFS_TOOMANY A line in the file contains too many fields.
VFS_TOOLONG A line in the file exceeded the internal buffer size of VFS_LINE_MAX.
ATTRIBUTES See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO vfstab(4), attributes(5)

NOTES The members of the vfstab structure point to information contained in a static area, so it must be copied if it is to be saved.
getwc(3C)

NAME  
getwc — get wide character from a stream

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

wint_t getwc(FILE *stream);

DESCRIPTION  
The getwc() function is equivalent to fgetwc(3C), except that if it is implemented as
a macro it may evaluate stream more than once, so the argument should never be an
expression with side effects.

RETURN VALUES  
Refer to fgetwc(3C).

ERRORS  
Refer to fgetwc(3C).

USAGE  
This interface is provided to align with some current implementations and with
possible future ISO standards.

Because it may be implemented as a macro, getwc() may treat incorrectly a stream
argument with side effects. In particular, getwc(*f ++) may not work as expected.
Therefore, use of this function is not recommended; fgetwc(3C) should be used
instead.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  
fgetwc(3C), attributes(5)
getwchar – get wide character from stdin stream

**SYNOPSIS**

```c
#include <wchar.h>

wint_t getwchar(void);
```

**DESCRIPTION**
The `getwchar()` function is equivalent to `getwc(stdin)`.

Refer to `fgetwc(3C)`.

**RETURN VALUES**
Refer to `fgetwc(3C)`.

**ERRORS**
Refer to `fgetwc(3C)`.

**USAGE**
If the `wint_t` value returned by `getwchar()` is stored into a variable of type `wchar_t` and then compared against the `wint_t` macro `WEOF`, the comparison may never succeed because `wchar_t` is defined as unsigned.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**
`fgetwc(3C), getwc(3C), attributes(5)`
getwd(3C)

NAME  getwd – get current working directory pathname

SYNOPSIS  #include <unistd.h>

    char *getwd(char *path_name);

DESCRIPTION  The getwd() function determines an absolute pathname of the current working
directory of the calling process, and copies that pathname into the array pointed to by
the path_name argument.

    If the length of the pathname of the current working directory is greater than
    (PATH_MAX + 1) including the null byte, getwd() fails and returns a null pointer.

RETURN VALUES  Upon successful completion, a pointer to the string containing the absolute pathname
of the current working directory is returned. Otherwise, getwd() returns a null
pointer and the contents of the array pointed to by path_name are undefined.

ERRORS  No errors are defined.

USAGE  For portability to implementations conforming to versions of the X/Open Portability
Guide prior to SUS, getcwd(3C) is preferred over this function.

SEE ALSO  getcwd(3C), standards(5)
NAME  | getwidth – get codeset information
SYNOPSIS  | #include <euc.h>
          | #include <getwidth.h>
          | void getwidth(eucwidth_t *ptr);
DESCRIPTION  | The getwidth() function reads the character class table for the current locale to get
              | information on the supplementary codesets. getwidth() sets this information into
              | the struct eucwidth_t. This struct is defined in <euc.h> and has the following
              | members:
              | short int _eucw1, _eucw2, _eucw3;
              | short int _scrw1, _scrw2, _scrw3;
              | short int _pcw;
              | char _multibyte;

Codeset width values for supplementary codesets 1, 2, and 3 are set in _eucw1,
_eucw2, and _eucw3, respectively. Screen width values for supplementary codesets 1,
2, and 3 are set in _scrw1, _scrw2, and _scrw3, respectively.

The width of Extended Unix Code (EUC) Process Code is set in _pcw. The
_multibyte entry is set to 1 if multibyte characters are used, and set to 0 if only
single-byte characters are used.

ATTRIBUTES  | See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
</tbody>
</table>

SEE ALSO  | euclen(3C), setlocale(3C), attributes(5)

NOTES  | The getwidth() function can be used safely in a multithreaded application, as long
       | as setlocale(3C) is not being called to change the locale.
       | The getwidth() function will only work with EUC locales.
getws(3C)

NAME
getws, fgetws – get a wide-character string from a stream

SYNOPSIS
#include <stdio.h>
#include <widec.h>

wchar_t *getws(wchar_t *ws);
#include <stdio.h>
#include <wchar.h>

wchar_t *fgetws(wchar_t *ws, int n, FILE *stream);

DESCRIPTION
The getws() function reads a string of characters from the standard input stream, stdin, converts these characters to the corresponding wide-character codes, and writes them to the array pointed to by ws, until a newline character is read, converted and transferred to ws or an end-of-file condition is encountered. The wide-character string, ws, is then terminated with a null wide-character code.

The fgetws() function reads characters from the stream, converts them to the corresponding wide-character codes, and places them in the wchar_t array pointed to by ws until n−1 characters are read, or until a newline character is read, converted and transferred to ws, or an end-of-file condition is encountered. The wide-character string, ws, is then terminated with a null wide-character code.

If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

The fgetws() function may mark the st_atime field of the file associated with stream for update. The st_atime field will be marked for update by the first successful execution of fgetc(3C), fgets(3C), fgetwc(3C), fgetws(), fread(3C), fscanf(3C), getc(3C), getchar(3C), gets(3C), or scanf(3C) using stream that returns data not supplied by a prior call to scanf(3C) or scanf(3C).

RETURN VALUES
Upon successful completion, getws() and fgetws() 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.

ERRORS
See fgetwc(3C) for the conditions that will cause fgetws() to fail.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
ferror(3C), fgetwc(3C), fread(3C), getwc(3C), putwc(3C), scanf(3C),
attributes(5)
glob(3C)

NAME
glob, globfree – generate path names 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 path name generator.

The globfree() function frees any memory allocated by glob() associated with pglob.

pattern Argument
The argument pattern is a pointer to a path name pattern to be expanded. The glob() function matches all accessible path names against this pattern and develops a list of all path names that match. In order to have access to a path name, glob() requires search permission on every component of a path except the last, and read permission on each directory of any filename component of pattern that contains any of the following special characters:

* ? [ ]

pglob Argument
The structure type glob_t is defined in the header <glob.h> and includes at least the following members:

size_t gl_pathc; /* count of paths matched by pattern */
char **gl_pathv; /* pointer to list of matched path names */
size_t gl_offs; /* slots to reserve at beginning of gl_pathv */

The glob() function stores the number of matched path names into pglob->gl_pathc and a pointer to a list of pointers to path names into pglob->gl_pathv. The path names are in sort order as defined by the current setting of the LC_COLLATE category. The first pointer after the last path name is a NULL pointer. If the pattern does not match any path names, the returned number of matched paths is set to 0, and the contents of pglob->gl_pathv are implementation-dependent.

It is the caller's responsibility to create the structure pointed to by pglob. The glob() function allocates other space as needed, including the memory pointed to by gl_pathv. The globfree() function frees any space associated with pglob from a previous call to glob().

flags Argument
The flags argument is used to control the behavior of glob(). The value of flags is a bitwise inclusive OR of zero or more of the following constants, which are defined in the header <glob.h>:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLOB_APPEND</td>
<td>Append path names generated to the ones from a previous call to glob().</td>
</tr>
<tr>
<td>GLOB_DOOFFS</td>
<td>Make use of pglob-&gt;gl_offs. If this flag is set, pglob-&gt;gl_offs is used to specify how many NULL pointers to add to the beginning of pglob-&gt;gl_pathv. In other words, pglob-&gt;gl_pathv will point</td>
</tr>
</tbody>
</table>
glob(3C)

to *pglob->g1_offs* NULL pointers, followed by *pglob->g1_pathc* path name pointers, followed by a NULL pointer.

GLOB_ERR

Causes *glob()* to return when it encounters a directory that it cannot open or read. Ordinarily, *glob()* continues to find matches.

GLOB_MARK

Each path name that is a directory that matches *pattern* has a slash appended.

GLOB_NOCHECK

If *pattern* does not match any path name, then *glob()* returns a list consisting of only *pattern*, and the number of matched path names is 1.

GLOB_NOESCAPE

Disable backslash escaping.

GLOB_NOSORT

Ordinarily, *glob()* sorts the matching path names according to the current setting of the LC_COLLATE category. When this flag is used the order of path names returned is unspecified.

The GLOB_APPEND flag can be used to append a new set of path names to those found in a previous call to *glob()*.

The following rules apply when two or more calls to *glob()* are made with the same value of *pglob* and without intervening calls to *globfree()*:

1. The first such call must not set GLOB_APPEND. All subsequent calls must set it.
2. All the calls must set GLOB_DOOFFS, or all must not set it.
3. After the second call, *pglob->g1_pathv* points to a list containing the following:
   a. Zero or more NULL pointers, as specified by GLOB_DOOFFS and *pglob->g1_offs*.
   b. Pointers to the path names that were in the *pglob->g1_pathv* list before the call, in the same order as before.
   c. Pointers to the new path names generated by the second call, in the specified order.
4. The count returned in *pglob->g1_pathc* will be the total number of path names from the two calls.
5. The application can change any of the fields after a call to *glob()*.
   If it does, it must reset them to the original value before a subsequent call, using the same *pglob* value, to *globfree()* or *glob()* with the GLOB_APPEND flag.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is not a NULL pointer, *glob()* calls (*errfunc*) with two arguments:

1. The *epath* argument is a pointer to the path that failed.
2. The *errno* argument is the value of *errno* from the failure, as set by the *opendir*(3C), *readdir*(3C) or *stat*(2) functions. (Other values may be used to report other errors not explicitly documented for those functions.)
The following constants are defined as error return values for `glob()`:

- **GLOB_ABORTED**: The scan was stopped because `GLOB_ERR` was set or `(*errfunc)` returned non-zero.
- **GLOB_NOMATCH**: The pattern does not match any existing path name, and `GLOB_NOCHECK` was not set in `flags`.
- **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 0, the error is ignored.

### RETURN VALUES

The following values are returned by `glob()`:

- **0**: Successful completion. The argument `pglob->gl_pathc` returns the number of matched path names and the argument `pglob->gl_pathv` contains a pointer to a null-terminated list of matched and sorted path names. However, if `pglob->gl_pathc` is 0, the content of `pglob->gl_pathv` is undefined.

- **non-zero**: An error has occurred. Non-zero constants are defined in `<glob.h>`. The arguments `pglob->gl_pathc` and `pglob->gl_pathv` are still set as defined above.

The `globfree()` function returns no value.

### USAGE

This function is not provided for the purpose of enabling utilities to perform path name expansion on their arguments, as this operation is performed by the shell, and utilities are explicitly not expected to redo this. Instead, it is provided for applications that need to do path name expansion on strings obtained from other sources, such as a pattern typed by a user or read from a file.

If a utility needs to see if a path name matches a given pattern, it can use `fnmatch(3C)`.

Note that `gl_pathc` and `gl_pathv` have meaning even if `glob()` fails. This allows `glob()` to report partial results in the event of an error. However, if `gl_pathc` is 0, `gl_pathv` is unspecified even if `glob()` did not return an error.

The `GLOB_NOCHECK` option could be used when an application wants to expand a path name if wildcards are specified, but wants to treat the pattern as just a string otherwise.

The new path names generated by a subsequent call with `GLOB_APPEND` are not sorted together with the previous path names. This mirrors the way that the shell handles path name expansion when multiple expansions are done on a command line.
Applications that need tilde and parameter expansion should use the `wordexp(3C)` function.

**EXAMPLES**

**EXAMPLE 1** Example of `glob_doofs` function.

One use of the `GLOB_DOOFFS` flag is by applications that build an argument list for use with the `execv()`, `execve()`, or `execvp()` functions (see `exec(2)`). Suppose, for example, that an application wants to do the equivalent of:

```
ls -l *.c
```

but for some reason:

```
system("ls -l *.c")
```

is not acceptable. The application could obtain approximately the same result using the sequence:

```
globbuf.gl_offs = 2;
glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);
globbuf.gl_pathv[0] = "ls";
globbuf.gl_pathv[1] = "-l";
execvp ("ls", &globbuf.gl_pathv[0]);
```

Using the same example:

```
ls -l *.c *.h
```

could be approximately simulated using `GLOB_APPEND` as follows:

```
globbuf.gl_offs = 2;
glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);
glob ("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf); ...
```

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`execv(2)`, `stat(2)`, `fnmatch(3C)`, `opendir(3C)`, `readdir(3C)`, `wordexp(3C)`, `attributes(5)`
NAME
grantpt – grant access to the slave pseudo-terminal device

SYNOPSIS
#include <stdlib.h>

int grantpt(int fildes);

DESCRIPTION
The grantpt() function changes the mode and ownership of the slave
pseudo-terminal device associated with its master pseudo-terminal counter part. fildes
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 UID of the calling process and the group ID is
set to a reserved group. The permission mode of the slave pseudo-terminal is set to
readable and writable by the owner and writable by the group.

RETURN VALUES
Upon successful completion, grantpt() returns 0. Otherwise, it returns -1 and sets
errno to indicate the error.

ERRORS
The grantpt() function may fail if:
EBADF The fildes argument is not a valid open file descriptor.
EINVAL The fildes argument is not associated with a master
pseudo-terminal device.
EACCES The corresponding slave pseudo-terminal device could not be
accessed.

USAGE
The grantpt() function will fail if it is unable to successfully invoke the setuid root
program. It may also fail if the application has installed a signal handler to catch
SIGCHLD signals.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
open(2), setuid(2), ptsname(3C), unlockpt(3C), attributes(5)
STREAMS Programming Guide
The `hsearch()` function is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. The comparison function used by `hsearch()` is `strcmp()` (see `string(3C)`). The `item` argument is a structure of type `ENTRY` (defined in the `<search.h>` header) containing two pointers: `item.key` points to the comparison key, and `item.data` points to any other data to be associated with that key. (Pointers to types other than void should be cast to pointer-to-void.) The `action` argument is a member of an enumeration type `ACTION` (defined in `<search.h>`) indicating the disposition of the entry if it cannot be found in the table. `ENTER` indicates that the item should be inserted in the table at an appropriate point. Given a duplicate of an existing item, the new item is not entered and `hsearch()` returns a pointer to the existing item. `FIND` indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a null pointer.

The `hcreate()` function allocates sufficient space for the table, and must be called before `hsearch()` is used. The `nslots` argument is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

The `hdestroy()` function destroys the search table, and may be followed by another call to `hcreate()`.

The `hsearch()` function returns a null pointer if either the action is `FIND` and the item could not be found or the action is `ENTER` and the table is full.

The `hcreate()` function returns 0 if it cannot allocate sufficient space for the table.

**EXAMPLE 1**

Example to read in strings.

The following example will read in strings followed by two numbers and store them in a hash table, discarding duplicates. It will then read in strings and find the matching entry in the hash table and print it.

```c
#include <stdio.h>
#include <search.h>
#include <string.h>
#include <stdlib.h>

struct info { /* this is the info stored in table */
```
EXAMPLE 1 Example to read in strings.  

```c
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;
}
```

ATTRIBUTES  See attributes(5) for descriptions of the following attributes:
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 code set, in the array specified by inbuf, into a sequence of corresponding characters in another code set, in the array specified by outbuf. The code sets are those specified in the iconv_open() call that returned the conversion descriptor, cd. The inbuf argument points to a variable that points to the first character in the input buffer and inbytesleft indicates the number of bytes to the end of the buffer to be converted. The outbuf argument points to a variable that points to the first available byte in the output buffer and outbytesleft indicates the number of the available bytes to the end of the buffer.

For state-dependent encodings, the conversion descriptor cd is placed into its initial shift state by a call for which inbuf is a null pointer, or for which inbuf points to a null pointer. When iconv() is called in this way, and if outbuf is not a null pointer or a pointer to a null pointer, and outbytesleft points to a positive value, iconv() will place, into the output buffer, the byte sequence to change the output buffer to its initial shift state. If the output buffer is not large enough to hold the entire reset sequence, iconv() will fail and set errno to E2BIG. Subsequent calls with inbuf as other than a null pointer or a pointer to a null pointer cause the conversion to take place from the current state of the conversion descriptor.

If a sequence of input bytes does not form a valid character in the specified code set, conversion stops after the previous successfully converted character. If the input buffer ends with an incomplete character or shift sequence, conversion stops after the previous successfully converted bytes. If the output buffer is not large enough to hold the entire converted input, conversion stops just prior to the input bytes that would cause the output buffer to overflow. The variable pointed to by inbuf is updated to point to the byte following the last byte successfully used in the conversion. The value pointed to by inbytesleft is decremented to reflect the number of bytes still not converted in the input buffer. The variable pointed to by outbuf is updated to point to the byte following the last byte of converted output data. The value pointed to by outbytesleft is decremented to reflect the number of bytes still available in the output buffer. For state-dependent encodings, the conversion descriptor is updated to reflect the shift state in effect at the end of the last successfully converted byte sequence.

If iconv() encounters a character in the input buffer that is legal, but for which an identical character does not exist in the target code set, iconv() performs an implementation-defined conversion on this character.

RETURN VALUES
The iconv() function updates the variables pointed to by the arguments to reflect the extent of the conversion and returns the number of non-identical conversions performed. If the entire string in the input buffer is converted, the value pointed to by...
inbytesleft will be 0. If the input conversion is stopped due to any conditions mentioned above, the value pointed to by inbytesleft will be non-zero and errno is set to indicate the condition. If an error occurs iconv() returns (size_t) -1 and sets errno to indicate the error.

ERRORS

The iconv() function will fail if:

EILSEQ     Input conversion stopped due to an input byte that does not belong to the input code set.

E2BIG      Input conversion stopped due to lack of space in the output buffer.

EINVAL     Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer.

The iconv() function may fail if:

EBADF      The cd argument is not a valid open conversion descriptor.

EXAMPLES

EXAMPLE 1 Using the iconv() Functions

The following example uses the iconv() functions:

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <iconv.h>
#include <stdlib.h>

/*
 * For state-dependent encodings, changes the state of the conversion
 * descriptor to initial shift state. Also, outputs the byte sequence
 * to change the state to initial state.
 * This code is assuming the iconv call for initializing the state
 * won't fail due to lack of space in the output buffer.
 */
#define INIT_SHIFT_STATE(cd, fptr, ileft, tptr, oleft)  
    {  
      fptr = NULL;  
      ileft = 0;  
      tptr = to;  
      oleft = BUFSIZ;  
      (void) iconv(cd, &fptr, &ileft, &tptr, &oleft);  
      (void) fwrite(to, 1, BUFSIZ - oleft, stdout);  
    }

int main(int argc, char **argv)
{
  iconv_t cd;
  char   *from[BUFSIZ], *to[BUFSIZ];
  char   *from_code, *to_code;
  char   *tptr;
  const char *fptr;
  size_t  ileft, oleft, num, ret;
```
EXAMPLE 1 Using the iconv() Functions  (Continued)

if (argc != 3) {
    (void) fprintf(stderr, "Usage: %s from_codeset to_codeset\n", argv[0]);
    return (1);
}

from_code = argv[1];
to_code = argv[2];

cd = iconv_open((const char *)to_code, (const char *)from_code);
if (cd == (iconv_t)-1) {
    /*
        iconv_open failed
    */
    (void) fprintf(stderr, "iconv_open(%s, %s) failed\n", to_code, from_code);
    return (1);
}

ileft = 0;
while ((ileft += (num = fread(from + ileft, 1, BUFSIZ - ileft, stdin))) > 0) {
    if (num == 0) {
        /*
            Input buffer still contains incomplete character
            or sequence. However, no more input character.
        */

        /*
            Initializes the conversion descriptor and outputs
            the sequence to change the state to initial state.
        */
        INIT_SHIFT_STATE(cd, fptr, ileft, tptr, oleft);
        (void) iconv_close(cd);
        (void) fprintf(stderr, "Conversion error\n");
        return (1);
    }

    fptr = from;
    for (;;) {
        tptr = to;
        oleft = BUFSIZ;
        ret = iconv(cd, &fptr, &ileft, &tptr, &oleft);
        if (ret != (size_t)-1) {
            /*
                iconv succeeded
            */

            /*
                Outputs converted characters
            */
        } else {
            /*
                iconv failed
            */

            /*
                Outputs error message
            */
        }
    }
}
EXAMPLE 1 Using the iconv() Functions (Continued)

    (void) fwrite(to, 1, BUFSIZ - oleft, stdout);
    break;
}

/*
 * iconv failed
*/
if (errno == EINVAL) {
    /*
     * Incomplete character or shift sequence
     */
    /*
     * Outputs converted characters
     */
    (void) fwrite(to, 1, BUFSIZ - oleft, stdout);
    /*
     * Copies remaining characters in input buffer
     * to the top of the input buffer.
     */
    (void) memmove(from, fptr, ileft);
    /*
     * Tries to fill input buffer from stdin
     */
    break;
} else if (errno == E2BIG) {
    /*
     * Lack of space in output buffer
     */
    /*
     * Outputs converted characters
     */
    (void) fwrite(to, 1, BUFSIZ - oleft, stdout);
    /*
     * Tries to convert remaining characters in
     * input buffer with emptied output buffer
     */
    continue;
} else if (errno == EILSEQ) {
    /*
     * Illegal character or shift sequence
     */
    /*
     * Outputs converted characters
     */
    (void) fwrite(to, 1, BUFSIZ - oleft, stdout);
    /*
     * Initializes the conversion descriptor and
     * outputs the sequence to change the state to
     * initial state.
     */
    INIT_SHIFT_STATE(cd, fptr, ileft, tptr, oleft);
EXAMPLE 1 Using the `iconv()` Functions (Continued)

```c
(void) iconv_close(cd);

(void) fprintf(stderr,
   "Illegal character or sequence\n");
return (1);
} else if (errno == EBADF) {
   /*
   * Invalid conversion descriptor.
   * Actually, this shouldn’t happen here.
   */
   (void) fprintf(stderr, "Conversion error\n");
   return (1);
} else {
   /*
   * This errno is not defined
   */
   (void) fprintf(stderr, "iconv error\n");
   return (1);
}

/*
* Initializes the conversion descriptor and outputs
* the sequence to change the state to initial state.
*/
INIT_SHIFT_STATE(cd, fptr, ileft, tptr, oleft);

(void) iconv_close(cd);
return (0);
```

FILES
/usr/lib/iconv/*.so  conversion modules
/usr/lib/iconv/sparcv9/*.so  conversion modules
/usr/lib/iconv/geniconvtbl/binarytables  conversion binary tables

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
`geniconvtbl(1)`, `iconv(1)`, `iconv_close(3C)`, `iconv_open(3C)`, `geniconvtbl(4)`, `attributes(5)`, `iconv(5)`, `iconv_unicode(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(3C) function.

If a file descriptor is used to implement the type iconv_t, that file descriptor will be
closed.

For examples using the iconv_close() function, see iconv(3C).

RETURN VALUES
Upon successful completion, iconv_close() returns 0; otherwise, it returns −1 and
sets errno to indicate the error.

ERRORS
The iconv_close() function may fail if:
EBADF The conversion descriptor is invalid.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO iconv(3C), iconv_open(3C), attributes(5)
include <iconv.h>

iconv_open(const char *fromcode, const char *tocode);

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.

The iconv_open() function supports the alias of the encoding name specified in tocode and fromcode. The alias table of the encoding name is described in the file /usr/lib/iconv/alias. See alias(4).

A conversion descriptor remains valid in a process until that process closes it.

For examples using the iconv_open() function, see iconv(3C).

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.

The iconv_open() function may fail if:

- EMFILE {OPEN_MAX} files descriptors are currently open in the calling process.
- ENFILE Too many files are currently open in the system.
- ENOMEM Insufficient storage space is available.
- EINVAL The conversion specified by fromcode and tocode is not supported by the implementation.

FILES /usr/lib/iconv/alias alias table file of the encoding name

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO exec(2), iconv(3C), iconv_close(3C), malloc(3C), alias(4), attributes(5)
iconv_open(3C)

NOTES The `iconv_open()` function uses `malloc(3C)` to allocate space for internal buffer areas. `iconv_open()` may fail if there is insufficient storage space to accommodate these buffers.

Portable applications must assume that conversion descriptors are not valid after a call to one of the `exec` functions (see `exec(2)`).
# include <strings.h>

char *index(const char *s, int c);
char *rindex(const char *s, int c);

The index() and rindex() functions operate on null-terminated strings.

The index() function returns a pointer to the first occurrence of character c in string s.

The rindex() function returns a pointer to the last occurrence of character c in string s.

Both index() and rindex() return a null pointer if c does not occur in the string.

The null character terminating a string is considered to be part of the string.

On most modern computer systems, you can not use a null pointer to indicate a null string. A null pointer is an error and results in an abort of the program. If you wish to indicate a null string, you must use a pointer that points to an explicit null string. On some machines and with some implementations of the C programming language, a null pointer, if dereferenced, would yield a null string. Though often used, this practice is not always portable. Programmers using a null pointer to represent an empty string should be aware of this portability issue. Even on machines where dereferencing a null pointer does not cause an abort of the program, it does not necessarily yield a null string.

SEE ALSO bstring(3C), malloc(3C), string(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
The initgroups() function reads the group database to get the group membership
for the user specified by name, and initializes the supplementary group access list of
the calling process (see getgrnam(3C) and getgroups(2)). The basegid group ID is
also included in the supplementary group access list. This is typically the real group
ID from the user database.

While scanning the group database, if the number of groups, including the basegid
entry, exceeds NGROUPS_MAX, subsequent group entries are ignored.

RETURN VALUES
Upon successful completion, 0 is returned. Otherwise, −1 is returned and errno is set
to indicate the error.

ERRORS
The initgroups() function will fail and not change the supplementary group access
list if:

EPERM The effective user ID is not super-user.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
getgroups(2), getgrnam(3C), attributes(5)
insque(3C)

NAME
insque, remque – insert/remove element from a queue

SYNOPSIS
#include <search.h>

void insque(struct qelem *elem, struct qelem *pred);
void remque(struct qelem *elem);

DESCRIPTION
The insque() and remque() functions manipulate queues built from doubly linked
lists. Each element in the queue must be in the following form:

struct qelem {
  struct qelem *q_forw;
  struct qelem *q_back;
  char q_data[ ];
};

The insque() function inserts elem in a queue immediately after pred. The remque() function removes an entry elem from a queue.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Unsafe</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5)
isaexec – invoke isa-specific executable

#include <unistd.h>

int isaexec(const char *path, char *const argv[], char *const envp[]);

The isaexec() function takes the path specified as path and breaks it into directory and file name components. It enquires from the running system the list of supported instruction set architectures; see isalist(5). The function traverses the list for an executable file in named subdirectories of the original directory. When such a file is located, execve() is invoked with argv[ ] and envp[ ]. See exec(2).

If no file is located, isaexec() returns ENOENT. Other return values are the same as for execve().

EXAMPLES

EXAMPLE 1 Example of isaexec() function.

On a system whose isalist is
sparcv7 sparc

the program

int
main(int argc, char *argv[], char *envp[])
{
    return (isaexec("/bin/thing", argv, envp));
}

will look first for an executable file named /bin/sparcv7/thing, then for an executable file named bin/sparcv7/thing. It will invoke execve() on the first executable file it finds named thing.

On that same system, a program called /u/bin/tofu can cause either
/u/bin/sparcv7/tofu or /u/bin/sparcv/tofu to be invoked using the following code:

int
main(int argc, char *argv[], char *envp[])
{
    return (isaexec(getexecname(), argv, envp));
}

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
<tr>
<td>Interface Stability</td>
<td>Stable</td>
</tr>
</tbody>
</table>

SEE ALSO

exec(2), getexecname(3C), attributes(5), isalist(5)
isastream() function determines if a file descriptor represents a STREAMS file. The fd argument refers to an open file descriptor. Upon successful completion, isastream() returns 1 if fd represents a STREAMS file, and 0 if it does not. Otherwise, -1 is return and errno is set to indicate the error.

The isastream() function will fail if:
EBADF The fd argument is not a valid file descriptor.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5), streamio(7I)

STREAMS Programming Guide
isatty(3C)

NAME  isatty – test for a terminal device

SYNOPSIS  
#include <unistd.h>

int isatty(int fildes);

DESCRIPTION  The isatty() function tests whether fildes, an open file descriptor, is associated with a terminal device.

RETURN VALUES  The isatty() function returns 1 if fildes is associated with a terminal; otherwise it returns 0 and may set errno to indicate the error.

ERRORS  The isatty() function may fail if:

EBADF  The fildes argument is not a valid open file descriptor.

ENOTTY  The fildes argument is not associated with a terminal.

USAGE  The isatty() function does not necessarily indicate that a human being is available for interaction via fildes. It is quite possible that non-terminal devices are connected to the communications line.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  ttynename(3C), attributes(5)
The \texttt{isnan()} function is identical to the \texttt{isnan\_d()} function.

The \texttt{isnan\_f()} function is implemented as a macro included in the \texttt{<ieeefp.h>} header.

The \texttt{fpclass()} function returns one of the following classes to which \textit{dsrc} belongs:

- \texttt{FP\_SNAN} signaling NaN
- \texttt{FP\_QNAN} quiet NaN
- \texttt{FP\_NINF} negative infinity
- \texttt{FP\_PINF} positive infinity
- \texttt{FP\_NDENORM} negative denormalized non-zero
- \texttt{FP\_PDENORM} positive denormalized non-zero
- \texttt{FP\_NZERO} negative zero
- \texttt{FP\_PZERO} positive zero
- \texttt{FP\_NNORM} negative normalized non-zero
- \texttt{FP\_PNORM} positive normalized non-zero

None of these routines generates an exception, even for signaling NaNs.

The \texttt{isnan()}\texttt{, isnand()}, and \texttt{isnan\_f()} function return \texttt{TRUE (1)} if the argument \textit{dsrc} or \textit{fsrc} is a NaN; otherwise they return \texttt{FALSE (0)}.

The \texttt{finite()} function returns \texttt{TRUE (1)} if the argument \textit{dsrc} is neither infinity nor NaN; otherwise it returns \texttt{FALSE (0)}.

The \texttt{unordered()} function returns \texttt{TRUE (1)} if one of its two arguments is unordered with respect to the other argument. This is equivalent to reporting whether either argument is NaN. If neither argument is NaN, \texttt{FALSE (0)} is returned.
isnan(3C)

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  fpgetround(3C), attributes(5)
NAME
iswalpha, iswupper, iswlower, iswdigit, iswxdigit, iswalnum, iswspace, iswpunct,
iswprint, iswcntrl, iswasci, iswgraph, isphonogram, isideogram, isenglish, isnumber,
ispecial – wide-character code classification functions

SYNOPSIS
#include <wchar.h>

int iswalpha(wint_t wc);

DESCRIPTION
These functions test whether wc is a wide-character code representing a character of a
particular class defined in the LC_CTYPE category of the current locale.

In all cases, wc is a wint_t, the value of which must be a wide-character code
corresponding to a valid character in the current locale or must equal the value of the
macro WEOF. If the argument has any other values, the behavior is undefined.

iswalpha(wc) Tests whether wc is a wide-character code representing
a character of class "alpha" in the program’s current
locale.

iswupper(wc) Tests whether wc is a wide-character code representing
a character of class "upper" in the program’s current
locale.

iswlower(wc) Tests whether wc is a wide-character code representing
a character of class "lower" in the program’s current
locale.

iswdigit(wc) Tests whether wc is a wide-character code representing
a character of class "digit" in the program’s current
locale.

iswxdigit(wc) Tests whether wc is a wide-character code representing
a character of class "xdigit" in the program’s current
locale.

iswalnum(wc) Tests whether wc is a wide-character code representing
a character of class "alpha" or "digit" in the program’s current
locale.

iswspace(wc) Tests whether wc is a wide-character code representing
a character of class "space" in the program’s current
locale.

iswpunct(wc) Tests whether wc is a wide-character code representing
a character of class "punct" in the program’s current
locale.

iswprint(wc) Tests whether wc is a wide-character code representing
a character of class "print" in the program’s current
locale.
iswalphax(3C)

iswgraph(wc) Tests whether \textit{wc} is a wide-character code representing a character of class "graph" in the program's current locale.

iswcntrl(wc) Tests whether \textit{wc} is a wide-character code representing a character of class "cntrl" in the program's current locale.

iswascii(wc) Tests whether \textit{wc} is a wide-character code representing an ASCII character.

isphonogram(wc) Tests whether \textit{wc} is a wide-character code representing a phonetic language character, excluding ASCII characters.

isideogram(wc) Tests whether \textit{wc} is a wide-character code representing an ideographic language character, excluding ASCII characters.

isenglish(wc) Tests whether \textit{wc} is a wide-character code representing an English language character, excluding ASCII characters.

isnumber(wc) Tests whether \textit{wc} is a wide-character code representing digit [0–9], excluding ASCII characters.

isspecial(wc) Tests whether \textit{wc} is a wide-character code representing a special language character, excluding ASCII characters.

\textbf{ATTRIBUTES} See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

\textbf{SEE ALSO} localedef(1), setlocale(3C), stdio(3C), ascii(5), attributes(5)
iswctype() - test character for specified class

SYNOPSIS

#include <wchar.h>

int iswctype(wint_t wc, wctype_t charclass);

DESCRIPTION

The iswctype() function determines whether the wide-character code \( wc \) has the character class \( \text{charclass} \), returning TRUE or FALSE. The iswctype() function is defined on WEOF and wide-character codes corresponding to the valid character encodings in the current locale. If the \( wc \) argument is not in the domain of the function, the result is undefined. If the value of \( \text{charclass} \) is invalid (that is, not obtained by a call to \text{wctype}(3C) or \text{charclass} is invalidated by a subsequent call to \text{setlocale}(3C) that has affected category \text{LC_CTYPE}), the result is indeterminate.

RETURN VALUES

The iswctype() function returns 0 for FALSE and non-zero for TRUE.

USAGE

There are twelve strings that are reserved for the standard character classes:

- "alnum"
- "alpha"
- "blank"
- "cntrl"
- "digit"
- "graph"
- "lower"
- "print"
- "punct"
- "space"
- "upper"
- "xdigit"

In the table below, the functions in the left column are equivalent to the functions in the right column.

<table>
<thead>
<tr>
<th>iswalnum(wc)</th>
<th>iswctype(wc, wctype(&quot;alnum&quot;))</th>
</tr>
</thead>
<tbody>
<tr>
<td>iswalpha(wc)</td>
<td>iswctype(wc, wctype(&quot;alpha&quot;))</td>
</tr>
<tr>
<td>iswcntrl(wc)</td>
<td>iswctype(wc, wctype(&quot;cntrl&quot;))</td>
</tr>
<tr>
<td>iswdigit(wc)</td>
<td>iswctype(wc, wctype(&quot;digit&quot;))</td>
</tr>
<tr>
<td>iswgraph(wc)</td>
<td>iswctype(wc, wctype(&quot;graph&quot;))</td>
</tr>
<tr>
<td>iswlower(wc)</td>
<td>iswctype(wc, wctype(&quot;lower&quot;))</td>
</tr>
<tr>
<td>iswprint(wc)</td>
<td>iswctype(wc, wctype(&quot;print&quot;))</td>
</tr>
<tr>
<td>iswpunct(wc)</td>
<td>iswctype(wc, wctype(&quot;punct&quot;))</td>
</tr>
<tr>
<td>iswspace(wc)</td>
<td>iswctype(wc, wctype(&quot;space&quot;))</td>
</tr>
<tr>
<td>iswupper(wc)</td>
<td>iswctype(wc, wctype(&quot;upper&quot;))</td>
</tr>
<tr>
<td>iswxdigit(wc)</td>
<td>iswctype(wc, wctype(&quot;xdigit&quot;))</td>
</tr>
</tbody>
</table>

The call
iswctype(3C)

iswctype(wc, wctype("blank"))

does not have an equivalent isw*( ) function.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO

iswalpha(3C), setlocale(3C), wctype(3C), attributes(5), environ(5)
killpg – send signal to a process group

#include <signal.h>

int killpg(pid_t pgrp, int sig);

The killpg() function sends the signal sig to the process group pgrp. See signal(3HEAD) for a list of signals.

The real or effective user ID of the sending process must match the real or saved set-user ID of the receiving process, unless the effective user ID of the sending process is the privileged user. A single exception is the signal SIGCONT, which may always be sent to any descendant of the current process.

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

The killpg() function will fail and no signal will be sent if:

EINVAL The sig argument is not a valid signal number.

EPERM The effective user ID of the sending process is not privileged user, and neither its real nor effective user ID matches the real or saved set-user ID of one or more of the target processes.

ESRCH No processes were found in the specified process group.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO kill(2), setpgrp(2), sigaction(2), signal(3HEAD), attributes(5)
The `lckpwdf()` and `ulckpwdf()` functions enable modification access to the password databases through the lock file. A process first uses `lckpwdf()` to lock the lock file, thereby gaining exclusive rights to modify the `/etc/passwd` or `/etc/shadow` password database. See `passwd(4)` and `shadow(4)`. Upon completing modifications, a process should release the lock on the lock file using `ulckpwdf()`. This mechanism prevents simultaneous modification of the password databases. The lock file, `/etc/.pwd.lock`, is used to coordinate modification access to the password databases `/etc/passwd` and `/etc/shadow`.

If `lckpwdf()` is successful in locking the file within 15 seconds, it returns 0. If unsuccessful (for example, `/etc/.pwd.lock` is already locked), it returns -1.

If `ulckpwdf()` is successful in unlocking the file `/etc/.pwd.lock`, it returns 0. If unsuccessful (for example, `/etc/.pwd.lock` is already unlocked), it returns -1.

These routines are for internal use only; compatibility is not guaranteed.

These routines are for internal use only; compatibility is not guaranteed.

These routines are for internal use only; compatibility is not guaranteed.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO `getpwnam(3C)`, `getspnam(3C)`, `passwd(4)`, `shadow(4)`, `attributes(5)`
**NAME**
ldexp – load exponent of a floating point number

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

double ldexp(double x, int exp);
```

**DESCRIPTION**
The ldexp() function computes the quantity \( x \times 2^{\exp} \).

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.

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 to 0.0, 0 is returned and errno may be set to ERANGE.

**ERRORS**
The ldexp() function will fail if:

| ERANGE         | The value to be returned would have caused overflow. |

The ldexp() function may fail if:

| ERANGE         | The value to be returned would have caused underflow. |

**USAGE**
An application wishing to check for error situations should set errno to 0 before calling ldexp(). If errno is non-zero on return, or the return value is NaN, an error has occurred.

**ATTRIBUTES**
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**
frexp(3C), isnan(3M), attributes(5)
lfmt(3C)

NAME
lfmt – display error message in standard format and pass to logging and monitoring services

SYNOPSIS
#include <pfmt.h>

int lfmt(FILE *stream, long flags, char *format, ... /* arg */);

DESCRIPTION
The lfmt() function retrieves a format string from a locale-specific message database (unless MM_NOGET is specified) and uses it for printf(3C) style formatting of args. The output is displayed on stream. If stream is NULL no output is displayed.

The lfmt() function encapsulates the output in the standard error message format (unless MM_NOSTD is specified, in which case the output is like that of printf()). It forwards its output to the logging and monitoring facility, even if stream is NULL. Optionally, lfmt() displays the output on the console with a date and time stamp.

If the printf() format string is to be retrieved from a message database, the format argument must have the following structure:

<catalog>:<msgnum>:<defmsg>.

If MM_NOGET is specified, only the <defmsg> field must be specified.

The <catalog> field indicates the message database that contains the localized version of the format string. This field is limited to 14 characters selected from a set of all characters values, excluding the null character (\0) and the ASCII codes for slash (/) and colon (:).

The <msgnum> field is a positive number that indicates the index of the string into the message database.

If the catalog does not exist in the locale (specified by the last call to setlocale(3C) using the LC_ALL or LC_MESSAGES categories), or if the message number is out of bound, lfmt() will attempt to retrieve the message from the C locale. If this second retrieval fails, lfmt() uses the <defmsg> field of the format argument.

If <catalog> is omitted, lfmt() will attempt to retrieve the string from the default catalog specified by the last call to setcat(3C). In this case, the format argument has the following structure:

:<msgnum>:<defmsg>.

The lfmt() function will output the message

Message not found!!

as the format string if <catalog> is not a valid catalog name, if no catalog is specified (either explicitly or with setcat()), if <msgnum> is not a valid number, or if no message could be retrieved from the message databases and <defmsg> was omitted.
The `flags` argument determines the type of output (whether the `format` should be interpreted as it is or be encapsulated in the standard message format) and the access to message catalogs to retrieve a localized version of `format`.

The `flags` argument is composed of several groups, and can take the following values (one from each group):

### Output format control

- **MM_NOSTD**: Do not use the standard message format but interpret `format` as a `printf()` format. Only catalog access control flags, console display control and logging information should be specified if `MM_NOSTD` is used; all other flags will be ignored.
- **MM_STD**: Output using the standard message format (default value is 0).

### Catalog access control

- **MM_NOGET**: Do not retrieve a localized version of `format`. In this case, only the `<defmsg>` field of `format` is specified.
- **MM_GET**: Retrieve a localized version of `format` from `<catalog>`, using `<msgid>` as the index and `<defmsg>` as the default message (default value is 0).

### Severity (standard message format only)

- **MM_HALT**: Generate a localized version of `HALT`, but donot halt the machine.
- **MM_ERROR**: Generate a localized version of `ERROR` (default value is 0).
- **MM_WARNING**: Generate a localized version of `WARNING`.
- **MM_INFO**: Generate a localized version of `INFO`.

Additional severities can be defined with the `addsev(3C)` function, using number-string pairs with numeric values in the range [5-255]. The specified severity is formed by the bitwise OR operation of the numeric value and other `flags` arguments.

If the severity is not defined, `lfmt()` uses the string `SEV=N` where `N` is the integer severity value passed in `flags`.

Multiple severities passed in `flags` will not be detected as an error. Any combination of severities will be summed and the numeric value will cause the display of either a severity string (if defined) or the string `SEV=N` (if undefined).

### Action

- **MM_ACTION**: Specify an action message. Any severity value is superseded and replaced by a localized version of `TO FIX`. 
### Console display control

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM_CONSOLE</td>
<td>Display the message to the console in addition to the specified stream.</td>
</tr>
<tr>
<td>MM_NOCONSOLE</td>
<td>Do not display the message to the console in addition to the specified stream (default value is 0).</td>
</tr>
</tbody>
</table>

### Logging information

**Major classification**
Identify the source of the condition. Identifiers are: MM_HARD (hardware), MM_SOFT (software), and MM_FIRM (firmware).

**Message source subclassification**
Identify the type of software in which the problem is spotted. Identifiers are: MM_APPL (application), MM_UTIL (utility), and MM_OPSYS (operating system).

The `lfmt()` function displays error messages in the following format:

```plaintext
label: severity: text
```

If no `label` was defined by a call to `setlabel(3C)`, the message is displayed in the format:

```plaintext
severity: text
```

If `lfmt()` is called twice to display an error message and a helpful action or recovery message, the output may appear as follows:

```plaintext
label: severity: text
label: TO FIX: text
```

### RETURN VALUES

Upon successful completion, `lfmt()` returns the number of bytes transmitted. Otherwise, it returns a negative value:

- `-1` Write the error to `stream`
- `-2` Cannot log and/or display at console.

### USAGE

Since `lfmt()` uses `gettext(3C)`, it is recommended that `lfmt()` not be used.

### EXAMPLES

**EXAMPLE 1** The following example

```c
setlabel("UX:test");
lfmt(stderr, MM_ERROR|MM_CONSOLE|MM_SOFT|MM_UTIL,
    "test:2:Cannot open file: %s\n", strerror(errno));
```

displays the message to `stderr` and to the console and makes it available for logging:

UX:test: ERROR: Cannot open file: No such file or directory

**EXAMPLE 2** The following example

```c
setlabel("UX:test");
lfmt(stderr, MM_INFO|MM_SOFT|MM_UTIL,
    "test:23:test facility is enabled\n");
```
EXAMPLE 2 The following example (Continued)

displays the message to stderr and makes it available for logging:

UX:test: INFO: test facility enabled

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO addsev(3C), gettext(3C), pfmt(3C), printf(3C), setcat(3C), setlabel(3C), setlocale(3C), attributes(5), environ(5)
localeconv() function sets the components of an object with type `struct lconv` (defined in `<locale.h>`) with the values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale (see `setlocale(3C)`). The definition of `struct lconv` is given below (the values for the fields in the "C" locale are given in comments).

```c
char *decimal_point; /* "." */
char *thousands_sep; /* "" (zero length string) */
char *grouping; /* "" */
char *int_curr_symbol; /* "" */
char *currency_symbol; /* "" */
char *mon_decimal_point; /* "" */
char *mon_thousands_sep; /* "" */
char *mon_grouping; /* "" */
char *positive_sign; /* "" */
char *negative_sign; /* "" */
char int_frac_digits; /* CHAR_MAX */
char frac_digits; /* CHAR_MAX */
char p_cs_precedes; /* CHAR_MAX */
char p_sep_by_space; /* CHAR_MAX */
char n_cs_precedes; /* CHAR_MAX */
char n_sep_by_space; /* CHAR_MAX */
char p_sign_posn; /* CHAR_MAX */
char n_sign_posn; /* CHAR_MAX */
```

The members of the structure with type `char *` are strings, any of which (except `decimal_point`) can point to a null string (""), to indicate that the value is not available in the current locale or is of zero length. The members with type `char` are non-negative numbers, any of which can be `CHAR_MAX` (defined in the `<limits.h>` header) to indicate that the value is not available in the current locale. The members are the following:

- `char *decimal_point`: The decimal-point character used to format non-monetary quantities.
- `char *thousands_sep`: The character used to separate groups of digits to the left of the decimal-point character in formatted non-monetary quantities.
- `char *grouping`: A string 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.
The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

char *int_curr_symbol
The international currency symbol applicable to the current locale, left-justified within a four-character space-padded field. The character sequences should match with those specified in ISO 4217 Codes for the Representation of Currency and Funds.

c char *currency_symbol
The local currency symbol applicable to the current locale.

c char *mon_decimal_point
The decimal point used to format monetary quantities.

c char *mon_thousands_sep
The separator for groups of digits to the left of the decimal point in formatted monetary quantities.

c char *mon_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 monetary quantity. The elements of mon_grouping are interpreted according to the rules described under grouping.

c char *positive_sign
The string used to indicate a non-negative-valued formatted monetary quantity.

c char *negative_sign
The string used to indicate a negative-valued formatted monetary quantity.

c char int_frac_digits
The number of fractional digits (those to the right of the decimal point) to be displayed in an internationally formatted monetary quantity.

c char frac_digits
The number of fractional digits (those to the right of the decimal point) to be displayed in a formatted monetary quantity.

c char p_cs_precedes
Set to 1 or 0 if the currency_symbol respectively precedes or succeeds the value for a non-negative formatted monetary quantity.

c char p_sep_by_space
Set to 1 or 0 if the currency_symbol respectively is or is not separated by a space from the value for a non-negative formatted monetary quantity.

c char n_cs_precedes
Set to 1 or 0 if the currency_symbol respectively precedes or succeeds the value for a negative formatted monetary quantity.

c char n_sep_by_space
Set to 1 or 0 if the currency_symbol respectively is or is not separated by a space from the value for a negative formatted monetary quantity.
char p_sign_posn
Set to a value indicating the positioning of the positive_sign for a non-negative formatted monetary quantity. The value of p_sign_posn is interpreted according to the following:

0 Parentheses surround the quantity and currency_symbol.
1 The sign string precedes the quantity and currency_symbol.
2 The sign string succeeds the quantity and currency_symbol.
3 The sign string immediately precedes the currency_symbol.
4 The sign string immediately succeeds the currency_symbol.

char n_sign_posn
Set to a value indicating the positioning of the negative_sign for a negative formatted monetary quantity. The value of n_sign_posn is interpreted according to the rules described under p_sign_posn.

RETURN VALUES
The localeconv() function returns a pointer to the filled-in object. The structure pointed to by the return value may be overwritten by a subsequent call to localeconv().

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

EXAMPLES

EXAMPLE 1 Rules used by four countries to format monetary quantities.

The following table illustrates the rules used by four countries to format monetary quantities.

<table>
<thead>
<tr>
<th>Country</th>
<th>Positive format</th>
<th>Negative format</th>
<th>International format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Italy</td>
<td>L.1.234</td>
<td>-1.1.234</td>
<td>ITL.1.234</td>
</tr>
<tr>
<td>Netherlands</td>
<td>F 1.234,56</td>
<td>F -1.234,56</td>
<td>NLG 1.234,56</td>
</tr>
<tr>
<td>Norway</td>
<td>kr1.234,56</td>
<td>kr1.234,56-</td>
<td>NOK 1.234,56</td>
</tr>
<tr>
<td>Switzerland</td>
<td>SFrs.1,234.56</td>
<td>SFrs.1,234.56C</td>
<td>CHF 1,234.56</td>
</tr>
</tbody>
</table>

For these four countries, the respective values for the monetary members of the structure returned by localeconv() are as follows:

<table>
<thead>
<tr>
<th></th>
<th>Italy</th>
<th>Netherlands</th>
<th>Norway</th>
<th>Switzerland</th>
</tr>
</thead>
<tbody>
<tr>
<td>int_curr_symbol</td>
<td>&quot;ITL.&quot;</td>
<td>&quot;NLG &quot;</td>
<td>&quot;NOK &quot;</td>
<td>&quot;CHF &quot;</td>
</tr>
<tr>
<td>currency_symbol</td>
<td>&quot;L.&quot;</td>
<td>&quot;F&quot;</td>
<td>&quot;kr&quot;</td>
<td>&quot;SFrs.&quot;</td>
</tr>
</tbody>
</table>
localeconv(3C)

mon_decimal_point "," "," ","
mon_thousands_sep "," "," ","
mon_grouping "\3" "\3" "\3" "\3"
positive_sign "" "" "" ""
negative_sign "," "," "," "C"
int_frac_digits 2 2 2 2
frac_digits 0 2 2 2
p_cs_precedes 1 1 1 1
p_sep_by_space 0 1 0 0
n_cs_precedes 1 1 1 1
n_sep_by_space 0 1 0 0
p_sign_posn 1 1 1 1
n_sign_posn 1 4 2 2

FILES
/usr/lib/locale/locale/LC_MONETARY/monetary
LC_MONETARY database for locale
/usr/lib/locale/locale/LC_NUMERIC/numeric
LC_NUMERIC database for locale

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO
setlocale(3C), attributes(5), environ(5)
lockf() function allows sections of a file to be locked; advisory or mandatory write locks depending on the mode bits of the file (see chmod(2)). 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. See fcntl(2) for more information about record locking.

The fildes argument is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission in order to establish locks with this function call.

The function argument is a control value that specifies the action to be taken. The permissible values for function are defined in <unistd.h> as follows:

```
#define F_ULOCK 0 /* unlock previously locked section */
#define F_LOCK 1 /* lock section for exclusive use */
#define F_TLOCK 2 /* test & lock section for exclusive use */
#define F_TEST 3 /* test section for other locks */
```

All other values of function are reserved for future extensions and will result in an error if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified section. F_LOCK and F_TLOCK both lock a section of a file if the section is available. F_ULOCK removes locks from a section of the file.

The size argument is the number of contiguous bytes to be locked or unlocked. The resource to be locked or unlocked starts at the current offset in the file and extends forward for a positive size and backward for a negative size (the preceding bytes up to but not including the current offset). If size is zero, the section from the current offset through the largest file offset is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked as such locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. Locked sections will be unlocked starting at the the point of the offset through size bytes or to the end of file if size is (off_t) 0. When this situation occurs, or if this situation occurs in adjacent sections, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and F_TLOCK requests differ only by the action taken if the resource is not available. F_LOCK 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 EAGAIN if the section is already locked by another process.
File locks are released on first close by the locking process of any file descriptor for the file.

_F_ULock_ requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an _errno_ is set to _EDEADLK_ and the requested section is not released.

An _F_ULock_ request in which _size_ is non-zero and the offset of the last byte of the requested section is the maximum value for an object of type _off_t_ , when the process has an existing lock in which _size_ is 0 and which includes the last byte of the requested section, will be treated as a request to unlock from the start of the requested section with a size equal to 0. Otherwise, an _F_ULock_ request will attempt to unlock only the requested section.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by requesting another process's locked resource. Thus calls to _lockf( )_ or _fcntl(2)_ scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. The _alarm(2)_ function may be used to provide a timeout facility in applications that require this facility.

**RETURN VALUES**

Upon successful completion, 0 is returned. Otherwise, -1 is returned and _errno_ is set to indicate the error.

**ERRORS**

The _lockf( )_ function will fail if:

- _EBADF_  
  The _fdedes_ argument is not a valid open file descriptor; or function is _F_LOCK_ or _F_TLOCK_ and _fdedes_ is not a valid file descriptor open for writing.

- _EACCES_ or _EAGAIN_  
  The function argument is _F_TLOCK_ or _F_TEST_ and the section is already locked by another process.

- _EDEADLK_  
  The function argument is _F_LOCK_ and a deadlock is detected.

- _EINTR_  
  A signal was caught during execution of the function.

- _ECOMM_  
  The _fdedes_ argument is on a remote machine and the link to that machine is no longer active.

- _EINVAL_  
  The function argument is not one of _F_LOCK_, _F_TLOCK_, _F_TEST_, or _F_ULock_; or _size_ plus the current file offset is less than 0.

- _EOVERFLOW_  
  The offset of the first, or if _size_ is not 0 then the last, byte in the requested section cannot be represented correctly in an object of type _off_t_.

Basic Library Functions 305
The `lockf()` function may fail if:

- **EAGAIN** The function argument is F_LOCK or F_TLOCK and the file is mapped with `mmap(2)`.
- **EDEADLK or ENOLCK** The function argument is F_LOCK, F_TLOCK, or F_ULOCK, and the request would cause the number of locks to exceed a system-imposed limit.
- **EOPNOTSUPP or EINVAL** The locking of files of the type indicated by the `fildes` argument is not supported.

**USAGE**

Record-locking should not be used in combination with the `fopen(3C), fread(3C), fwrite(3C)` and other `stdio` functions. Instead, the more primitive, non-buffered functions (such as `open(2)`) should be used. Unexpected results may occur in processes that do buffering in the user address space. The process may later read/write data which is/was locked. The `stdio` functions are the most common source of unexpected buffering.

The `alarm(2)` function may be used to provide a timeout facility in applications requiring it.

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

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`intro(2), alarm(2), chmod(2), close(2), creat(2), fcntl(2), mmap(2), open(2), read(2), write(2), attributes(5), lf64(5)`
NAME
_longjmp, _setjmp – non-local goto

SYNOPSIS
#include <setjmp.h>

void _longjmp(jmp_buf env, int val);
int _setjmp(jmp_buf env);

DESCRIPTION
The _longjmp() and _setjmp() functions are identical to longjmp(3C) and
setjmp(3C), respectively, with the additional restriction that _longjmp() and
_setjmp() do not manipulate the signal mask.

If _longjmp() is called even though env was never initialized by a call to
_setjmp(), or when the last such call was in a function that has since returned, the
results are undefined.

RETURN VALUES
Refer to longjmp(3C) and setjmp(3C).

ERRORS
No errors are defined.

USAGE
If _longjmp() is executed and the environment in which _setjmp() was executed
no longer exists, errors can occur. The conditions under which the environment of the
_setjmp() no longer exists include exiting the function that contains the _setjmp() call, and exiting an inner block with temporary storage. This condition might not be
detectable, in which case the _longjmp() occurs and, if the environment no longer
exists, the contents of the temporary storage of an inner block are unpredictable. This
condition might also cause unexpected process termination. If the function has
returned, the results are undefined.

Passing longjmp() a pointer to a buffer not created by setjmp( ), passing
_longjmp() a pointer to a buffer not created by _setjmp(), passing
siglongjmp(3C) a pointer to a buffer not created by sigsetjmp(3C) or passing any
of these three functions a buffer that has been modified by the user can cause all the
problems listed above, and more.

The _longjmp() and _setjmp() functions are included to support programs
written to historical system interfaces. New applications should use siglongjmp(3C)
and sigsetjmp(3C) respectively.

SEE ALSO
longjmp(3C), setjmp(3C), siglongjmp(3C), sigsetjmp(3C)
**NAME**

lsearch, lfind – linear search and update

**SYNOPSIS**

```c
#include <search.h>

void *lsearch(const void *key, void *base, size_t *nelp, size_t width,
   int (*compar)(const void *, const void *));

void *lfind(const void *key, const void *base, size_t *nelp, size_t
   width, int (*compar)(const void *, const void *));
```

**DESCRIPTION**

The lsearch() function is a linear search routine generalized from Knuth (6.1) Algorithm S. (See
The Art of Computer Programming, Volume 3, Section 6.1, by Donald E. Knuth.) It returns a
pointer into a table indicating where a datum may be found. If the datum does not
occur, it is added at the end of the table. The key argument points to the datum to be
sought in the table. The base argument points to the first element in the table. The nelp
argument points to an integer containing the current number of elements in the table.
The integer is incremented if the datum is added to the table. The width argument is
the size of an element in bytes. The compar argument is a pointer to the comparison
function that the user must supply (strcmp(3C) for example). It is called with two
arguments that point to the elements being compared. The function must return zero if
the elements are equal and non-zero otherwise.

The lfind() function is the same as lsearch() except that if the datum is not
found, it is not added to the table. Instead, a null pointer is returned.

It is important to note the following:

- the pointers to the key and the element at the base of the table 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.

**RETURN VALUES**

If the searched-for datum is found, both lsearch() and lfind() return a pointer to
it. Otherwise, lfind() returns NULL and lsearch() returns a pointer to the newly
added element.

**USAGE**

Undefined results can occur if there is not enough room in the table to add a new item.

**EXAMPLES**

**EXAMPLE 1** A sample code using the lsearch() function.

This program will read in less than TABSIZE strings of length less than ELSIZE and
store them in a table, eliminating duplicates, and then will print each entry.

```c
#include <search.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define TABSIZE 50
#define ELSIZE 120
```
EXAMPLE 1 A sample code using the lsearch() function. (Continued)

```c
main( )
{
    char line[ELSIZE]; /* buffer to hold input string */
    char tab[TABSIZE][ELSIZE]; /* table of strings */
    size_t nel = 0; /* number of entries in tab */
    int i;

    while (fgets(line, ELSIZE, stdin) != NULL &&
          nel < TABSIZE)
        (void) lsearch(line, tab, &nel, ELSIZE, mycmp);
    for( i = 0; i < nel; i++ )
        (void)fputs(tab[i], stdout);
    return 0;
}
```

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO bsearch(3C), hsearch(3C), string(3C), tsearch(3C), attributes(5)

The `madvise()` function advises the kernel that a region of user mapped memory in the range `[addr, addr + len)` will be accessed following a type of pattern. The kernel uses this information to optimize the procedure for manipulating and maintaining the resources associated with the specified mapping range.

Values for `advice` are defined in `<sys/mman.h>` as:

```c
#define MADV_NORMAL 0x0 /* No further special treatment */
#define MADV_RANDOM 0x1 /* Expect random page references */
#define MADV_SEQUENTIAL 0x2 /* Expect sequential page references */
#define MADV_WILLNEED 0x3 /* Will need these pages */
#define MADV_DONTNEED 0x4 /* Don't need these pages */
#define MADV_FREE 0x5 /* Contents can be freed */
#define MADV_ACCESS_DEFAULT 0x6 /* default access */
#define MADV_ACCESS_LWP 0x7 /* next LWP to access heavily */
#define MADV_ACCESS_MANY 0x8 /* many processes to access heavily */
```

### MADV_NORMAL

The default system characteristic where accessing memory within the address range causes the system to read data from the mapped file. The kernel reads all data from files into pages which are retained for a period of time as a "cache." System pages can be a scarce resource, so the kernel steals pages from other mappings when needed. This is a likely occurrence, but adversely affects system performance only if a large amount of memory is accessed.

### MADV_RANDOM

Tells the kernel to read in a minimum amount of data from a mapped file on any single particular access. If `MADV_NORMAL` is in effect when an address of a mapped file is accessed, the system tries to read in as much data from the file as reasonable, in anticipation of other accesses within a certain locality.

### MADV_SEQUENTIAL

Tells the system that addresses in this range are likely to be accessed only once, so the system will free the resources mapping the address range as quickly as possible. This is used in the `cat(1)` and `cp(1)` utilities.

### MADV_WILLNEED

Tells the system that a certain address range is definitely needed so the kernel will start reading the specified range into memory. This can benefit programs wanting to minimize the time needed to access memory the first time, as the kernel would need to read in from the file.
MADV_DONTNEED  Tells the kernel that the specified address range is no longer needed, so the system starts to free the resources associated with the address range.

MADV_FREE        Tells the kernel that contents in the specified address range are no longer important and the range will be overwritten. When there is demand for memory, the system will free pages associated with the specified address range. In this instance, the next time a page in the address range is referenced, it will contain all zeroes. Otherwise, it will contain the data that was there prior to the MADV_FREE call. References made to the address range will not make the system read from backing store (swap space) until the page is modified again.

This value cannot be used on mappings that have underlying file objects.

MADV_ACCESS_LWP   Tells the kernel that the next LWP to touch the specified address range will access it most heavily, so the kernel should try to allocate the memory and other resources for this range and the LWP accordingly.

MADV_ACCESS_MANY  Tells the kernel that many processes and/or LWPs will access the specified address range randomly across the machine, so the kernel should try to allocate the memory and other resources for this range accordingly.

MADV_ACCESS_DEFAULT Resets the kernel’s expectation for how the specified range will be accessed to the default.

The madvise() function should be used by applications with specific knowledge of their access patterns over a memory object, such as a mapped file, to increase system performance.

**RETURN VALUES**

Upon successful completion, madvise() returns 0; otherwise, it returns −1 and sets errno to indicate the error.

**ERRORS**

EAGAIN            Some or all mappings in the address range [addr, addr + len) are locked for I/O.

EBUSY             Some or all of the addresses in the range [addr, addr + len) are locked and MS_SYNC with the MS_INVALIDATE option is specified.

EFAULT            Some or all of the addresses in the specified range could not be read into memory from the underlying object when performing MADV_WILLNEED.
EINVAL  The \textit{addr} argument is not a multiple of the page size as returned by \texttt{sysconf}(3C), the length of the specified address range is equal to 0, or the \textit{advice} argument was invalid.

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

ENOMEM  Addresses in the range \([\textit{addr}, \textit{addr} + \textit{len}]\) are outside the valid range for the address space of a process, or specify one or more pages that are not mapped.

ESTALE  Stale NFS file handle.

\textbf{ATTRIBUTES}  See \texttt{attributes}(5) for descriptions of the following attributes:

\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{ATTRIBUTE TYPE} & \textbf{ATTRIBUTE VALUE} \\
\hline
Interface Stability & Stable \\
MT-Level & MT-Safe \\
\hline
\end{tabular}
\end{center}

\textbf{SEE ALSO}  \texttt{cat(1), cp(1), meminfo(2), mmap(2), sysconf(3C), attributes(5)}
## NAME
makecontext, swapcontext – manipulate user contexts

## SYNOPSIS
```c
cc -D__MAKECONTEXT_V2_SOURCE [ flag... ] file... [ library... ]
#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.

The `makecontext()` function modifies the context specified by `ucp`, which has been initialized using `getcontext(2)`. When this context is resumed using `swapcontext()` or `setcontext(2)`, program execution continues by calling the function `func`, passing it the arguments that follow `argc` in the `makecontext()` call. The value of `argc` must match the number of pointer-sized integer arguments passed to `func`. Otherwise the behavior is undefined.

Before a call is made to `makecontext()`, the context being modified should have a stack allocated for it. The value of `argc` must match the number of integer arguments passed to `func`, 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 initialized prior to the call to `makecontext()`. If the `uc_link` member is initialized to NULL, the thread executing `func` will exit when `func` returns. See `pthread_exit(3)`. The `swapcontext()` function saves the current context in the context structure pointed to by `oucp` and sets the context to the context structure pointed to by `ucp`.

If the `ucp` or `oucp` argument points to an illegal address, the behavior is undefined and `errno` may be set to `EFAULT`.

## RETURN VALUES
Upon successful completion, `swapcontext()` returns 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

## ERRORS
The `swapcontext()` function will fail if:

- **ENOMEM** The `ucp` argument does not have enough stack left to complete the operation.

The `swapcontext()` function may fail if:

- **EFAULT** The `ucp` or `oucp` argument points to an invalid address.

## EXAMPLES
**EXAMPLE 1** Alternate execution context on a stack whose memory was allocated using `mmap(2)`.
```c
#include <stdio.h>
#include <ucontext.h>
#include <sys/mman.h>
```
EXAMPLE 1 Alternate execution context on a stack whose memory was allocated using mmap(2). (Continued)

```c
void assign(long a, int *b)
{
    *b = (int)a;
}

int main(int argc, char **argv)
{
    ucontext_t uc, back;
    size_t sz = 0x10000;
    int value = 0;
    getcontext(&uc);
    uc.uc_stack.ss_sp = mmap(0, sz,
        PROT_READ | PROT_WRITE | PROT_EXEC,
        MAP_PRIVATE | MAP_ANON, -1, 0);
    uc.uc_stack.ss_size = sz;
    uc.uc_stack.ss_flags = 0;
    uc.uc_link = &back
    makecontext(&uc, assign, 2, 100L, &value);
    swapcontext(&back, &uc);
    printf("done %d\n", value);
    return (0);
}
```

**USAGE** These functions are useful for implementing user-level context switching between multiple threads of control within a process (co-processing). More effective multiple threads of control can be obtained by using native support for multithreading. See threads(3THR).

**ATTRIBUTES** See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO** exit(2), getcontext(2), mmap(2), sigaction(2), sigprocmask(2), threads(3THR), ucontext(3HEAD), attributes(5)
The legacy implementation of makecontext() for sparc and sparcv9 was in violation of the standard. To use the updated version with the corrected behavior, specify -D__MAKECONTEXT_V2_SOURCE when invoking the compiler. See the EXAMPLES section for the correct usage.

Future releases of Solaris will enable the corrected behavior by default, thereby eliminating the need to define __MAKECONTEXT_V2_SOURCE.
makedev(3C)

NAME
makedev, major, minor – manage a device number

SYNOPSIS
#include <sys/types.h>
#include <sys/mkdev.h>

dev_t makedev(major_t maj, minor_t min);
major_t major(dev_t device);
minor_t minor(dev_t device);

DESCRIPTION
The makedev() function returns a formatted device number on success and NODEV on failure. The maj argument is the major number. The min argument is the minor number. The makedev() function can be used to create a device number for input to mknod(2).

The major() function returns the major number component from device.

The minor() function returns the minor number component from device.

RETURN VALUES
Upon successful completion, makedev() returns a formatted device number. Otherwise, NODEV is returned and errno is set to indicate the error.

ERRORS
The makedev() function will fail if:

EINVAL One or both of the arguments maj and min is too large, or the device number created from maj and min is NODEV.

The major() function will fail if:

EINVAL The device argument is NODEV, or the major number component of device is too large.

The minor() function will fail if:

EINVAL The device argument is NODEV.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
mknod(2), stat(2), attributes(5)
malloc, calloc, free, memalign, realloc, valloc, alloca – memory allocator

#include <stdlib.h>

void *
malloc(size_t size);

void *
calloc(size_t nelem, size_t elsize);

void
free(void *ptr);

void *
memalign(size_t alignment, size_t size);

void *
realloc(void *ptr, size_t size);

void *
valloc(size_t size);

#include <alloca.h>

void *
alloca(size_t size);

The malloc() and free() functions provide a simple, general-purpose memory allocation package. The malloc() function returns a pointer to a block of at least size bytes suitably aligned for any use. If the space assigned by malloc() is overrun, the results are undefined.

The argument to free() is a pointer to a block previously allocated by malloc(), calloc(), or realloc(). After free() is executed, this space is made available for further allocation by the application, though not returned to the system. Memory is returned to the system only upon termination of the application. If ptr is a null pointer, no action occurs. If a random number is passed to free(), the results are undefined.

The calloc() function allocates space for an array of nelem elements of size elsize. The space is initialized to zeros.

The memalign() function allocates size bytes on a specified alignment boundary and returns a pointer to the allocated block. The value of the returned address is guaranteed to be an even multiple of alignment. The value of alignment must be a power of two and must be greater than or equal to the size of a word.

The realloc() function changes the size of the block pointed to by ptr to size bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If ptr is NULL, realloc() behaves like malloc() for the specified size. If size is 0 and ptr is not a null pointer, the space pointed to is made available for further allocation by the application, though not returned to the system. Memory is returned to the system only upon termination of the application.

The valloc() function has the same effect as malloc(), except that the allocated memory will be aligned to a multiple of the value returned by sysconf (_SC_PAGESIZE).
The \texttt{alloca()} function allocates \texttt{size} bytes of space in the stack frame of the caller, and returns a pointer to the allocated block. This temporary space is automatically freed when the caller returns. If the allocated block is beyond the current stack limit, the resulting behavior is undefined.

**RETURN VALUES**

Upon successful completion, each of the allocation functions returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

If there is no available memory, \texttt{malloc()}, \texttt{realloc()}, \texttt{memalign()}, \texttt{valloc()}, and \texttt{calloc()} return a null pointer. When \texttt{realloc()} is called with \texttt{size} > 0 and returns \texttt{NULL}, the block pointed to by \texttt{ptr} is left intact. If \texttt{size}, \texttt{nelem}, or \texttt{elsize} is 0, either a null pointer or a unique pointer that can be passed to \texttt{free()} is returned.

If \texttt{malloc()}, \texttt{calloc()}, or \texttt{realloc()} returns unsuccessfully, \texttt{errno} will be set to indicate the error. The \texttt{free()} function does not set \texttt{errno}.

**ERRORS**

The \texttt{malloc()}, \texttt{calloc()}, and \texttt{realloc()} functions will fail if:

- \texttt{ENOMEM} The physical limits of the system are exceeded by \texttt{size} bytes of memory which cannot be allocated.
- \texttt{EAGAIN} There is not enough memory available to allocate \texttt{size} bytes of memory; but the application could try again later.

**USAGE**

Portable applications should avoid using \texttt{valloc()} but should instead use \texttt{malloc()} or \texttt{mmap}(2). On systems with a large page size, the number of successful \texttt{valloc()} operations might be 0.

Comparative features of \texttt{malloc(3C)}, \texttt{bsdmalloc(3MALLOC)}, and \texttt{malloc(3MALLOC)} are as follows:

- The \texttt{bsdmalloc(3MALLOC)} routines afford better performance, but are space-inefficient.
- The \texttt{malloc(3MALLOC)} routines are space-efficient, but have slower performance.
- The standard, fully SCD-compliant \texttt{malloc} routines are a trade-off between performance and space-efficiency.

**ATTRIBUTES**

See \texttt{attributes(5)} for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>\texttt{malloc()}, \texttt{calloc()}, \texttt{free()}, \texttt{realloc()}, \texttt{valloc()} are Standard; \texttt{memalign()} and \texttt{alloca()} are Stable.</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

\texttt{brk(2), getrlimit(2), bsdmalloc(3MALLOC), malloc(3MALLOC), mapmalloc(3MALLOC), watchmalloc(3MALLOC), attributes(5)}
WARNINGS

Undefined results will occur if the size requested for a block of memory exceeds the maximum size of a process's heap, which can be obtained with `getrlimit(2)`.

The `alloca()` function is machine-, compiler-, and most of all, system-dependent. Its use is strongly discouraged.
malloc(3MALLOC)

NAME
malloc, free, realloc, calloc, mallopt, mallinfo – memory allocator

SYNOPSIS
cc [ flag ... ] file ... -lmalloc [ library ... ]
#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);
#include <malloc.h>
int mallopt(int cmd, int value);
struct mallinfo mallinfo(void);

DESCRIPTION
The malloc() and free() functions provide a simple general-purpose memory allocation package.

The malloc() function returns a pointer to a block of at least size bytes suitably aligned for any use.

The argument to free() is a pointer to a block previously allocated by malloc(). After free() is performed, this space is made available for further allocation, and its contents have been destroyed See mallopt() below for a way to change this behavior. If ptr is a null pointer, no action occurs.

Undefined results occur if the space assigned by malloc() is overrun or if some random number is handed to free().

The realloc() function changes the size of the block pointed to by ptr to size bytes and returns a pointer to the (possibly moved) block. The contents are unchanged up to the lesser of the new and old sizes. If ptr is a null pointer, realloc() behaves like malloc() for the specified size. If size is 0 and ptr is not a null pointer, the object it points to is freed.

The calloc() function allocates space for an array of nelem elements of size elsize. The space is initialized to zeros.

The mallopt() function provides for control over the allocation algorithm. The available values for cmd are:

M_MXFAST Set maxfast to value. The algorithm allocates all blocks below the size of maxfast in large groups and then doles them out very quickly. The default value for maxfast is 24.

M_NLBLKS Set numlblks to value. The above mentioned “large groups” each contain numlblks blocks. numlblks must be greater than 0. The default value for numlblks is 100.
M_GRAIN

Set grain to value. The sizes of all blocks smaller than maxfast are
c onsidered to be rounded up to the nearest multiple of grain. grain
must be greater than 0. The default value of grain is the smallest
number of bytes that will allow alignment of any data type. Value
will be rounded up to a multiple of the default when grain is set.

M_KEEP

Preserve data in a freed block until the next malloc(),
realloc(), or calloc(). This option is provided only for
compatibility with the old version of malloc(), and it is not
recommended.

These values are defined in the <malloc.h> header.

The malloc() function can be called repeatedly, but cannot be called after the first
 small block is allocated.

The mallinfo() function provides instrumentation describing space usage. It returns
the mallinfo structure with the following members:

unsigned long arena;  /* total space in arena */
unsigned long ordblks;  /* number of ordinary blocks */
unsigned long smblks;  /* number of small blocks */
unsigned long hblkhd;  /* space in holding block headers */
unsigned long hblks;  /* number of holding blocks */
unsigned long usmblks;  /* space in small blocks in use */
unsigned long fsmblks;  /* space in free small blocks */
unsigned long uordblks;  /* space in ordinary blocks in use */
unsigned long fordblks;  /* space in free ordinary blocks */
unsigned long keepcost;  /* space penalty if keep option */
                       /* is used */

The mallinfo structure is defined in the <malloc.h> header.

Each of the allocation routines returns a pointer to space suitably aligned (after
possible pointer coercion) for storage of any type of object.

RETURN VALUES

The malloc(), realloc(), and calloc() functions return a null pointer if there is
not enough available memory. When realloc() returns NULL, the block pointed to
by ptr is left intact. If malloc() is called after any allocation or if cmd or value are
invalid, a non-zero value is returned. Otherwise, it returns 0.

ERRORS

If malloc(), calloc(), or realloc() returns unsuccessfully, errno is set
to indicate the error:

ENOMEM  size bytes of memory exceeds the physical limits of your system,
and cannot be allocated.

EAGAIN  There is not enough memory available at this point in time to
allocate size bytes of memory; but the application could try again
later.
malloc(3MALLOC)

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

brk(2), bsdmalloc(3MALLOC), libtmalloc(3LIB), malloc(3C),
mapmalloc(3MALLOC), mtmalloc(3MALLOC), watchmalloc(3MALLOC),
attributes(5)

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 for compatibility with old applications. New
applications should include <stdlib.h> to access the prototypes for these functions.

Comparative Features of these malloc routines, bsdmalloc(3MALLOC), and
malloc(3C)

- These malloc routines are space-efficient but have slower performance.
- The bsdmalloc(3MALLOC) routines afford better performance but are
  space-inefficient.
- The standard, fully SCD-compliant malloc(3C) routines are a trade-off between
  performance and space-efficiency.

The free() function does not set errno.
mapmalloc

SYNOPSIS

cc [ flag ...] file ... -lmapmalloc [ library ...]

#include <stdlib.h>

void *malloc(size_t size);
void *calloc(size_t nelem, size_t elsize);
void free(void *ptr);
void *realloc(void *ptr, size_t size);

DESCRIPTION

The collection of malloc routines in this library use mmap(2) instead of sbrk(2) for acquiring new heap space. The routines in this library are intended to be used only if necessary, when applications must call sbrk(), but need to call other library routines that might call malloc. The algorithms used by these routines are not sophisticated. There is no reclaiming of memory.

malloc() and free() provide a simple general-purpose memory allocation package.

malloc() returns a pointer to a block of at least size bytes suitably aligned for any use.

The argument to free() is a pointer to a block previously allocated by malloc(), calloc() or realloc(). If ptr is a NULL pointer, no action occurs.

Undefined results will occur if the space assigned by malloc() is overrun or if some random number is handed to free().

calloc() allocates space for an array of nelem elements of size elsize. The space is initialized to zeros.

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.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

malloc() and realloc() will fail if there is not enough available memory.

Entry points for malloc_debug(), mallocmap(), mallocopt(), mallinfo(), memalign(), and valloc(), are empty routines, and are provided only to protect the user from mixing malloc() functions from different implementations.

RETURN VALUES

If there is no available memory, malloc(), realloc(), and calloc() return a null pointer. When realloc() returns NULL, the block pointed to by ptr is left intact. If size, nelem, or elsize is 0, a unique pointer to the arena is returned.
mapmalloc(3MALLOC)

FILES
/usr/lib/libmapmalloc

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO brk(2), getrlimit(2), mmap(2), realloc(3C), malloc(3MALLOC), attributes(5)
mblen – get number of bytes in a character

SYNOPSIS
#include <stdlib.h>

int mblen(const char *s, size_t n);

DESCRIPTION
If s is not a null pointer, mblen() determines the number of bytes constituting the character pointed to by s. It is equivalent to:

mbtowc((wchar_t *)0, s, n);

A call with s as a null pointer causes this function to return 0. The behavior of this function is affected by the LC_CTYPE category of the current locale.

RETURN VALUES
If s is a null pointer, mblen() returns 0. It s is not a null pointer, mblen() returns 0 (if s points to the null byte), the number of bytes that constitute the character (if the next n or fewer bytes form a valid character), or -1 (if they do not form a valid character) and may set errno to indicate the error. In no case will the value returned be greater than n or the value of the MB_CUR_MAX macro.

ERRORS
The mblen() function may fail if:

EILSEQ Invalid character sequence is detected.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO
mbstowcs(3C), mbtowc(3C), setlocale(3C), wcstombs(3C), wctomb(3C), attributes(5)
mbrlen(3C)

NAME  mbrlen – get number of bytes in a character (restartable)
SYNOPSIS  
#include <wchar.h>

size_t mbrlen(const char *s, size_t n, mbstate_t *ps);

DESCRIPTION  If s is not a null pointer, mbrlen() determines the number of bytes constituting the character pointed to by s. It is equivalent to:

mbstate_t internal;
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal);

If ps is a null pointer, the mbrlen() function uses its own internal mbstate_t object, which is initialized at program startup to the initial conversion state. Otherwise, the mbstate_t object pointed to by ps is used to completely describe the current conversion state of the associated character sequence. Solaris will behave as if no function defined in the Solaris Reference Manual calls mbrlen().

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

RETURN VALUES  The mbrlen() function returns the first of the following that applies:

0  If the next n or fewer bytes complete the character that corresponds to the null wide-character.
positive  If the next n or fewer bytes complete a valid character; the value returned is the number of bytes that complete the character.
(size_t)-2  If the next n bytes contribute to an incomplete but potentially valid character, and all n bytes have been processed. When n has at least the value of the MB_CUR_MAX macro, this case can only occur if s points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).
(size_t)-1  If an encoding error occurs, in which case the next n or fewer bytes do not contribute to a complete and valid character. In this case, EILSEQ is stored in errno and the conversion state is undefined.

ERRORS  The mbrlen() function may fail if:

EINVAL  The ps argument points to an object that contains an invalid conversion state.
EILSEQ  Invalid character sequence is detected.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See NOTES below</td>
</tr>
</tbody>
</table>
If `ps` is not a null pointer, `mbrlen()` uses the `mbstate_t` object pointed to by `ps` and the function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale. If `ps` is a null pointer, `mbrlen()` uses its internal `mbstate_t` object and the function is Unsafe in multithreaded applications.
mbrtowc(3C)

**NAME**
mbrtowc – convert a character to a wide-character code (restartable)

**SYNOPSIS**

```c
#include <wchar.h>

size_t mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);
```

**DESCRIPTION**

If `s` is a null pointer, the `mbrtowc()` function is equivalent to the call:

```
mbrtowc(NULL, '', 1, ps)
```

In this case, the values of the arguments `pwc` and `n` are ignored.

If `s` is not a null pointer, the `mbrtowc()` function inspects at most `n` bytes beginning at the byte pointed to by `s` to determine the number of bytes needed to complete the next character (including any shift sequences). If the function determines that the next character is completed, it determines the value of the corresponding wide-character and then, if `pwc` is not a null pointer, stores that value in the object pointed to by `pwc`. If the corresponding wide-character is the null wide-character, the resulting state described is the initial conversion state.

If `ps` is a null pointer, the `mbrtowc()` function uses its own internal `mbstate_t` object, which is initialized at program startup to the initial conversion state. Otherwise, the `mbstate_t` object pointed to by `ps` is used to completely describe the current conversion state of the associated character sequence. Solaris will behave as if no function defined in the Solaris Reference Manual calls `mbrtowc()`.

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

**RETURN VALUES**

The `mbrtowc()` function returns the first of the following that applies:

- `0`  
  If the next `n` or fewer bytes complete the character that corresponds to the null wide-character (which is the value stored).

- **positive**  
  If the next `n` or fewer bytes complete a valid character (which is the value stored); the value returned is the number of bytes that complete the character.

- `(size_t)-2`  
  If the next `n` bytes contribute to an incomplete but potentially valid character, and all `n` bytes have been processed (no value is stored). When `n` has at least the value of the `MB_CUR_MAX` macro, this case can only occur if `s` points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).

- `(size_t)-1`  
  If an encoding error occurs, in which case the next `n` or fewer bytes do not contribute to a complete and valid character (no value is stored). In this case, `EILSEQ` is stored in `errno` and the conversion state is undefined.

**ERRORS**

The `mbrtowc()` function may fail if:

---

328  man pages section 3: Basic Library Functions • Last Revised 3 Feb 1998
The `ps` argument points to an object that contains an invalid conversion state.

Invalid character sequence is detected.

### attributes(5)

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See NOTES below</td>
</tr>
</tbody>
</table>

### SEE ALSO

mbsinit(3C), setlocale(3C), attributes(5), environ(5)

### NOTES

If `ps` is not a null pointer, `mbrtowc()` uses the `mbstate_t` object pointed to by `ps` and the function can be used safely in multithreaded applications, as long as `setlocale(3C)` is not being called to change the locale. If `ps` is a null pointer, `mbrtowc()` uses its internal `mbstate_t` object and the function is Unsafe in multithreaded applications.
mbsinit(3C)

NAME mbsinit – determine conversion object status

SYNOPSIS

```c
#include <wchar.h>

int mbsinit(const mbstate_t *ps);
```

DESCRIPTION

If `ps` is not a null pointer, the `mbsinit()` function determines whether the object
pointed to by `ps` describes an initial conversion state.

RETURN VALUES

The `mbsinit()` function returns non-zero if `ps` is a null pointer, or if the pointed-to
object describes an initial conversion state; otherwise, it returns 0.

If an `mbstate_t` object is altered by any of the functions described as "restartable",
and is then used with a different character sequence, or in the other conversion
direction, or with a different `LC_CTYPE` category setting than on earlier function calls,
the behavior is undefined. See `environ(5)`.

ERRORS

No errors are defined.

USAGE

The `mbstate_t` object is used to describe the current conversion state from a
particular character sequence to a wide-character sequence (or vice versa) under the
rules of a particular setting of the `LC_CTYPE` category of the current locale.

The initial conversion state corresponds, for a conversion in either direction, to the
beginning of a new character sequence in the initial shift state. A zero-valued
`mbstate_t` object is at least one way to describe an initial conversion state. A
zero-valued `mbstate_t` object can be used to initiate conversion involving any
character sequence, in any `LC_CTYPE` category setting.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
</tbody>
</table>

SEE ALSO

`mbrlen(3C), mbtowc(3C), mbsrtowcs(3C), setlocale(3C), wcrtomb(3C),
wcsrtombs(3C), attributes(5), environ(5)`

NOTES

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

NAME
mbsrtowcs – convert a character string to a wide-character string (restartable)

SYNOPSIS
#include <wchar.h>

size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len,
                mbstate_t *ps);

DESCRIPTION
The mbsrtowcs() function converts a sequence of characters, beginning in the
conversion state described by the object pointed to by ps, from the array indirectly
pointed to by src into a sequence of corresponding wide-characters. If dst is not a null
pointer, the converted characters are stored into the array pointed to by dst.
Conversion continues up to and including a terminating null character, which is also
stored. Conversion stops early in either of the following cases:

- When a sequence of bytes is encountered that does not form a valid character.
- When len codes have been stored into the array pointed to by dst (and dst is not a
  null pointer).

Each conversion takes place as if by a call to the mbrtowc() function.

If dst is not a null pointer, the pointer object pointed to by src is assigned either a null
pointer (if conversion stopped due to reaching a terminating null character) or the
address just past the last character converted (if any). If conversion stopped due to
reaching a terminating null character, and if dst is not a null pointer, the resulting state
described is the initial conversion state.

If ps is a null pointer, the mbsrtowcs() function uses its own internal mbstate_t
object, which is initialized at program startup to the initial conversion state.
Otherwise, the mbstate_t object pointed to by ps is used to completely describe the
current conversion state of the associated character sequence. Solaris will behave as if
no function defined in the Solaris Reference Manual calls mbsrtowcs().

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

RETURN VALUES
If the input conversion encounters a sequence of bytes that do not form a valid
character, an encoding error occurs. In this case, the mbsrtowcs() function stores the
value of the macro EILSEQ in errno and returns (size_t)−1; the conversion state is
undefined. Otherwise, it returns the number of characters successfully converted, not
including the terminating null (if any).

ERRORS
The mbsrtowcs() function may fail if:

EINVAL The ps argument points to an object that contains an invalid
conversion state.

EILSEQ Invalid character sequence is detected.
mbsrtowcs(3C)

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See NOTES below</td>
</tr>
</tbody>
</table>

SEE ALSO
mbrtowc(3C), mbsinit(3C), setlocale(3C), attributes(5), environ(5)

NOTES
If ps is not a null pointer, mbsrtowcs() uses the mbstate_t object pointed to by ps and the function can be used safely in multithreaded applications, as long as setlocale(3C) is not being called to change the locale. If ps is a null pointer, mbsrtowcs() uses its internal mbstate_t object and the function is Unsafe in multithreaded applications.
NAME  mbstowcs – convert a character string to a wide-character string

SYNOPSIS  
#include <stdlib.h>

size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);

DESCRIPTION  The mbstowcs() function converts a sequence of characters from the array pointed to by s into a sequence of corresponding wide-character codes and stores not more than n wide-character codes into the array pointed to by pwcs. No characters that follow a null byte (which is converted into a wide-character code with value 0) will be examined or converted. Each character is converted as if by a call to mbtowc(3C).

No more than n elements will be modified in the array pointed to by pwcs. If copying takes place between objects that overlap, the behavior is undefined.

The behavior of this function is affected by the LC_CTYPE category of the current locale. If pwcs is a null pointer, mbstowcs() returns the length required to convert the entire array regardless of the value of n, but no values are stored.

RETURN VALUES  If an invalid character is encountered, mbstowcs() returns (size_t)-1 and may set errno to indicate the error. Otherwise, mbstowcs() returns the number of the array elements modified (or required if pwcs is NULL), not including a terminating 0 code, if any. The array will not be zero-terminated if the value returned is n.

ERRORS  The mbstowcs() function may fail if:

EILSEC Invalid byte sequence is detected.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO  mblen(3C), mbtowc(3C), setlocale(3C), wcstombs(3C), wctomb(3C), attributes(5)
mbtowc(3C)

NAME  mbtowc — convert a character to a wide-character code

SYNOPSIS
#include <stdlib.h>

int mbtowc(wchar_t *pwc, const char *s, size_t n);

DESCRIPTION
If s is not a null pointer, mbtowc() determines the number of the bytes that constitute
the character pointed to by s. It then determines the wide-character code for the value
of type wchar_t that corresponds to that character. (The value of the wide-character
code corresponding to the null byte is 0.) If the character is valid and pwc is not a null
pointer, mbtowc() stores the wide-character code in the object pointed to by pwc.

A call with s as a null pointer causes this function to return 0. The behavior of this
function is affected by the LC_CTYPE category of the current locale. At most n bytes of
the array pointed to by s will be examined.

RETURN VALUES
If s is a null pointer, mbtowc() returns 0. If s is not a null pointer, mbtowc() returns
0 (if s points to the null byte), the number of bytes that constitute the converted
character (if the next n or fewer bytes form a valid character), or −1 and may set
errno to indicate the error (if they do not form a valid character).

In no case will the value returned be greater than n or the value of the MB_CUR_MAX
macro.

ERRORS
The mbtowc() function may fail if:

EILSEQ  Invalid character sequence is detected.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO
mblen(3C), mbstowcs(3C), setlocale(3C), wcstombs(3C), wctomb(3C),
attributes(5)
NAME
mctl – memory management control

SYNOPSIS
/usr/ucb/cc[ flag ... ] file ...
#include <sys/types.h>
#include <sys/mman.h>

int mctl( addr, len, function, arg);

caddr_t addr;
size_t len;
int function;
int 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(). 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 VALUES

mctl() returns 0 on success, −1 on failure.

ERRORS

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’s 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), memcntl(2), getpagesize(3C), mlock(3C), mlockall(3C), msync(3C)

NOTES

Use of these interfaces should be restricted to only applications written on BSD
platforms. Use of these interfaces with any of the system libraries or in multi-thread
applications is unsupported.
memory(3C)

NAME
memory, memccpy, memchr, memcmp, memcpy, memmove, memset – 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);

#include <string.h>
const void *memchr(const void *s, int c, size_t n);
#include <cstring>
void *std::memchr(void *s, int c, size_t n);

DESCRIPTION
These functions operate as efficiently as possible on memory areas (arrays of bytes bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

The memccpy() function copies bytes from memory area s2 into s1, stopping after the first occurrence of c (converted to an unsigned char) has been copied, or after n bytes have been copied, whichever comes first. It returns a pointer to the byte after the copy of c in s1, or a null pointer if c was not found in the first n bytes of s2.

The memchr() function returns a pointer to the first occurrence of c (converted to an unsigned char) in the first n bytes (each interpreted as an unsigned char) of memory area s, or a null pointer if c does not occur.

The memcmp() function compares its arguments, looking at the first n bytes (each interpreted as an unsigned char), and returns an integer less than, equal to, or greater than 0, according as s1 is lexicographically less than, equal to, or greater than s2 when taken to be unsigned characters.

The memcpy() function copies n bytes from memory area s2 to s1. It returns s1.

The memmove() function copies n bytes from memory areas s2 to s1. Copying between objects that overlap will take place correctly. It returns s1.

The memset() function sets the first n bytes in memory area s to the value of c (converted to an unsigned char). It returns s.
memory(3C)

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  string(3C), attributes(5)
SYNOPSIS

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

int mkfifo(const char *path, mode_t mode);
```

DESCRIPTION

The `mkfifo()` function creates a new FIFO special file named by the pathname pointed to by `path`. The file permission bits of the new FIFO are initialized from `mode`. The file permission bits of the `mode` argument are modified by the process’s file creation mask (see `umask(2)`). Bits other than the file permission bits in `mode` are ignored.

If `path` names a symbolic link, `mkfifo()` fails and sets `errno` to `EEXIST`.

The FIFO’s user ID is set to the process’s effective user ID. The FIFO’s group ID is set to the group ID of the parent directory or to the effective group ID of the process.

The `mkfifo()` function calls `mknod(2)` to create the file.

Upon successful completion, `mkfifo()` marks for update the `st_atime`, `st_ctime`, and `st_mtime` fields of the file. Also, the `st_ctime` and `st_mtime` fields of the directory that contains the new entry are marked for update.

RETURN VALUES

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

ERRORS

The `mkfifo()` function will fail if:

- **EACCES**: A component of the path prefix denies search permission, or write permission is denied on the parent directory of the FIFO to be created.
- **EEXIST**: The named file already exists.
- **ELOOP**: A loop exists in symbolic links encountered during resolution of the `path` argument.
- **ENAMETOOLONG**: The length of the `path` argument exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX`.
- **ENOENT**: A component of the path prefix specified by `path` does not name an existing directory or `path` is an empty string.
- **ENOSPC**: The directory that would contain the new file cannot be extended or the file system is out of file-allocation resources.
- **ENOTDIR**: A component of the path prefix is not a directory.
- **EROFS**: The named file resides on a read-only file system.

The `mkfifo()` function may fail if:
mknod(3C)

EXAMPLES

EXAMPLE 1 Create a FIFO File

The following example demonstrates how to create a FIFO file named /home/cnd/mod_done with read and write permissions for the owner and read permissions for the group and others.

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

int status;
...
status = mkfifo("/home/cnd/mod_done", S_IWUSR | S_IRUSR |
               S_IRGRP | S_IROTH);
```

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

mkdir(1), chmod(2), exec(2), mknod(2), umask(2), stat(3HEAD), ufs(7FS), attributes(5), standards(5)
mkstemp(3C)

NAME
mkstemp – make a unique file name

SYNOPSIS
#include <stdlib.h>

int mkstemp(char *template);

DESCRIPTION
The mkstemp() function replaces the contents of the string pointed to by template by a unique file name, and returns a file descriptor for the file open for reading and writing. The function thus prevents any possible race condition between testing whether the file exists and opening it for use. The string in template should look like a file name with six trailing 'X's; mkstemp() replaces each 'X' with a character from the portable file name character set. The characters are chosen such that the resulting name does not duplicate the name of an existing file.

RETURN VALUES
Upon successful completion, mkstemp() returns an open file descriptor. Otherwise -1 is returned if no suitable file could be created.

ERRORS
No errors are defined.

USAGE
It is possible to run out of letters.

The mkstemp() function does not check to determine whether the file name part of template exceeds the maximum allowable file name length.

The tmpfile(3C) function is preferred over this function.

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

SEE ALSO
getpid(2), open(2), tmpfile(3C), tmpnam(3C), lf64(5), standards(5)
** mktemp(3C)  

** NAME **  

mktemp – make a unique file name  

** SYNOPSIS **  

#include <stdlib.h>  

char *mktemp(char *template);  

** DESCRIPTION **  

The mktemp() function replaces the contents of the string pointed to by template with a unique file name, and returns template. The string in template should look like a file name with six trailing 'X's; mktemp() will replace the 'X's with a character string that can be used to create a unique file name. Only 26 unique file names per thread can be created for each unique template.  

** RETURN VALUES **  

The mktemp() function will assign to template the empty string if it cannot create a unique name.  

** ATTRIBUTES **  

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

** SEE ALSO **  

mkstemp(3C), tmpfile(3C), tmpnam(3C), attributes(5)
The mktime() function converts the time represented by the tm structure pointed to by timeptr into a calendar time (the number of seconds since 00:00:00 UTC, January 1, 1970).

The tm structure contains the following members:

```c
int tm_sec; /* seconds after the minute [0, 61] */
int tm_min; /* minutes after the hour [0, 59] */
int tm_hour; /* hour since midnight [0, 23] */
int tm_mday; /* day of the month [1, 31] */
int tm_mon; /* months since January [0, 11] */
int tm_year; /* years since 1900 */
int tm_wday; /* days since Sunday [0, 6] */
int tm_yday; /* days since January 1 [0, 365] */
int tm_isdst; /* flag for daylight savings time */
```

In addition to computing the calendar time, mktime() normalizes the supplied tm structure. The original values of the tm_wday and tm_yday components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated in the definition of the structure. On successful completion, the values of the tm_wday and tm_yday components are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to be within the appropriate ranges. The final value of tm_mday is not set until tm_mon and tm_year are determined.

The tm_year member must be for year 1901 or later. Calendar times before 20:45:52 UTC, December 13, 1901 or after 03:14:07 UTC, January 19, 2038 cannot be represented. Portable applications should not try to create dates before 00:00:00 UTC, January 1, 1970 or after 00:00:00 UTC, January 1, 2038.

The original values of the components may be either greater than or less than the specified range. For example, a tm_hour of −1 means 1 hour before midnight, tm_mday of 0 means the day preceding the current month, and tm_mon of −2 means 2 months before January of tm_year.

If tm_isdst is positive, the original values are assumed to be in the alternate timezone. If it turns out that the alternate timezone is not valid for the computed calendar time, then the components are adjusted to the main timezone. Likewise, if tm_isdst is zero, the original values are assumed to be in the main timezone and are converted to the alternate timezone if the main timezone is not valid. If tm_isdst is negative, mktime() attempts to determine whether the alternate timezone is in effect for the specified time.

Local timezone information is used as if mktime() had called tzset(). See ctime(3C).
If the calendar time can be represented in an object of type `time_t`, `mktime()` returns the specified calendar time without changing `errno`. If the calendar time cannot be represented, the function returns the value (`time_t`)−1 and sets `errno` to indicate the error.

The `mktime()` function will fail if:

- `EOVERFLOW` The date represented by the input `tm` struct cannot be represented in a `time_t`. Note that the `errno` setting may change if future revisions to the standards specify a different value.

The `mktime()` function is MT-Safe in multithreaded applications, as long as no user-defined function directly modifies one of the following variables: `timezone`, `altzone`, `daylight`, and `tzname`. See `ctime(3C)`.

Note that −1 can be a valid return value for the time that is one second before the Epoch. The user should clear `errno` before calling `mktime()`. If `mktime()` then returns −1, the user should check `errno` to determine whether or not an error actually occurred.

The `mktime()` function assumes Gregorian dates. Times before the adoption of the Gregorian calendar will not match historical records.

**EXAMPLE 1** Sample code using `mktime()`.

What day of the week is July 4, 2001?

```c
#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]);
```

The `zoneinfo` timezone data files do not transition past Tue Jan 19 03:14:07 2038 UTC. Therefore for 64-bit applications using `zoneinfo` timezones, calculations beyond this date may not use the correct offset from standard time, and could return incorrect values. This affects the 64-bit version of `mktime()`.
ATTRIBUTES
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
</tbody>
</table>

SEE ALSO
ctime(3C), getenv(3C), TIMEZONE(4), attributes(5)
mlock(3C)

NAME
mlock, munlock – lock or unlock pages in memory

Default
#include <sys/mman.h>

int mlock(caddr_t addr, size_t len);
int munlock(caddr_t addr, size_t len);

Standard
#include <sys/mman.h>

conforming
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);

DESCRIPTION
The mlock() function uses the mappings established for the address range
[addr, addr + len) to identify pages to be locked in memory. If the page identified by a
mapping changes, such as occurs when a copy of a writable MAP_PRIVATE page is
made upon the first store, the lock will be transferred to the newly copied private
page.

The munlock() function removes locks established with mlock().

A given page may be locked multiple times by executing an mlock() through
different mappings. That is, if two different processes lock the same page, then the
page will remain locked until both processes remove their locks. However, within a
given mapping, page locks do not nest – multiple mlock() operations on the same
address in the same process will all be removed with a single munlock(). Of course,
a page locked in one process and mapped in another (or visible through a different
mapping in the locking process) is still locked in memory. This fact can be used to
create applications that do nothing other than lock important data in memory, thereby
avoiding page I/O faults on references from other processes in the system.

If the mapping through which an mlock() has been performed is removed, an
munlock() is implicitly performed. An munlock() is also performed implicitly
when a page is deleted through file removal or truncation.

Locks established with mlock() are not inherited by a child process after a fork()
and are not nested.

Attempts to mlock() more memory than a system-specific limit will fail.

RETURN VALUES
Upon successful completion, the mlock() and munlock() functions return 0.
Otherwise, no changes are made to any locks in the address space of the process, the
functions return −1 and set errno to indicate the error.

ERRORS
The mlock() and munlock() functions will fail if:

EINVAL The addr argument is not a multiple of the page size as returned by
sysconf(3C).
ENOMEM Addresses in the range [addr, addr + len) are invalid for the address
space of a process, or specify one or more pages which are not
mapped.
The system does not support this memory locking interface.

ENOSYS

The process’s effective user ID is not superuser.

EPERM

The mlock() function will fail if:

EAGAIN Some or all of the memory identified by the range [addr, addr + len) could not be locked because of insufficient system resources.

USAGE Because of the impact on system resources, the use of mlock() and munlock() is restricted to the superuser.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO fork(2), memcntl(2), mmap(2), plock(3C), mlockall(3C), sysconf(3C), attributes(5), standards(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 mlockall() function locks in memory all pages mapped by an address space.

The value of flags determines whether the pages to be locked are those currently
mapped by the address space, those that will be mapped in the future, or both:

MCL_CURRENT  Lock current mappings
MCL_FUTURE  Lock future mappings

If MCL_FUTURE is specified for mlockall(), mappings are locked as they are added
to the address space (or replace existing mappings), provided sufficient memory is
available. Locking in this manner is not persistent across the exec family of functions
(see exec(2)).

Mappings locked using mlockall() with any option may be explicitly unlocked
with a munlock() call (see mlock(3C)).

The munlockall() function removes address space locks and locks on mappings in
the address space.

All conditions and constraints on the use of locked memory that apply to mlock(3C)
also apply to mlockall().

Locks established with mlockall() are not inherited by a child process after a
fork(2) call, and are not nested.

RETURN VALUES  
Upon successful completion, the mlockall() and munlockall() functions return
0. Otherwise, they return -1 and set errno to indicate the error.

ERRORS  
The mlockall() and munlockall() functions will fail if:

EAGAIN  Some or all of the memory in the address space could not be
locked due to sufficient resources. This error condition applies to
mlockall() only.

EINVAL  The flags argument contains values other than MCL_CURRENT and
MCL_FUTURE.

EPERM  The process’s effective user ID is not super-user.

USAGE  
The mlockall() and munlockall() functions require super-user privileges.

ATTRIBUTES  
See attributes(5) for descriptions of the following attributes:
mlockall(3C)

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO exec(2), fork(2), memcntl(2), mmap(2), plock(3C), mlock(3C), sysconf(3C), attributes(5)
NAME
modf, modff – decompose floating-point number

SYNOPSIS
#include <math.h>

double modf(double x, double *iptr);
float modff(float x, float *iptr);

DESCRIPTION
The modf() and modff() functions break the argument x into integral and fractional
parts, each of which has the same sign as the argument. The modf() function stores
the integral part as a double in the object pointed to by iptr. The modff() function
stores the integral part as a float in the object pointed to by iptr.

RETURN VALUES
Upon successful completion, modf() and modff() return the signed fractional part
of x.

If x is NaN, NaN is returned and *iptr is set to NaN.

If the correct value would cause underflow to 0.0, modf() returns 0 and errno may
be set to ERANGE.

_ERRORS_
The modf() function may fail if:
ERANGE The result underflows.

USAGE
An application wishing to check for error situations should set errno to 0 before
calling modf(). If errno is non-zero on return, or the return value is NaN, an error
has occurred.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
frexp(3C), isnan(3M), ldexp(3C), attributes(5)
NAME | monitor – prepare process execution profile

SYNOPSIS | `#include <mon.h>`
```c
void monitor(int (*lowpc)(), int (*highpc)(), WORD *buffer, size_t bufsize, size_t nfunc);
```

DESCRIPTION | The `monitor()` function is an interface to the `profil(2)` function and is called automatically with default parameters by any program created by the `cc(1B)` utility with the `-p` option specified. Except to establish further control over profiling activity, it is not necessary to explicitly call `monitor()`.

When used, `monitor()` is called at least at the beginning and the end of a program. The first call to `monitor()` initiates the recording of two different kinds of execution-profile information: execution-time distribution and function call count. Execution-time distribution data is generated by `profil()` and the function call counts are generated by code supplied to the object file (or files) by `cc(1B)` `-p`. Both types of information are collected as a program executes. The last call to `monitor()` writes this collected data to the output file `mon.out`.

The name of the file written by `monitor()` is controlled by the environment variable `PROFDIR`. If `PROFDIR` does not exist, the file `mon.out` is created in the current directory. If `PROFDIR` exists but has no value, `monitor()` does no profiling and creates no output file. If `PROFDIR` is `dirname`, and `monitor()` is called automatically by compilation with `cc -p`, the file created is `dirname/pid.progname` where `progname` is the name of the program.

The `lowpc` and `highpc` arguments are the beginning and ending addresses of the region to be profiled.

The `buffer` argument is the address of a user-supplied array of `WORD` (defined in the header `<mon.h>`). The `buffer` argument is used by `monitor()` to store the histogram generated by `profil()` and the call counts.

The `bufsize` argument identifies the number of array elements in `buffer`.

The `nfunc` argument is the number of call count cells that have been reserved in `buffer`. Additional call count cells will be allocated automatically as they are needed.

The `bufsize` argument should be computed using the following formula:

```c
size_of_buffer =
    sizeof(struct hdr) +
    nfunc * sizeof(struct cnt) +
    ((highpc-lowpc)/BARSIZE) * sizeof(WORD) +
    sizeof(WORD) - 1;
bufsize = (size_of_buffer / sizeof(WORD));
```

where:
- `lowpc`, `highpc`, `nfunc` are the same as the arguments to `monitor()`;
- `kind` is a defined constant in the `mon.h` header file.
- **BARSIZE** is the number of program bytes that correspond to each histogram bar, or cell, of the `profil()` buffer;
- the hdr and cnt structures and the type WORD are defined in the header <mon.h>.

The default call to `monitor()` is as follows:

```c
monitor (&eprol, &etext, wbuf, wbufsz, 600);
```

where:
- `eprol` is the beginning of the user’s program when linked with `cc -p` (see `end(3C)`);
- `etext` is the end of the user’s program (see `end(3C)`);
- `wbuf` is an array of WORD with `wbufsz` elements;
- `wbufsz` is computed using the `bufsize` formula shown above with `BARSIZE` of 8;
- `600` is the number of call count cells that have been reserved in `buffer`.

These parameter settings establish the computation of an execution-time distribution histogram that uses `profil()` for the entire program, initially reserves room for 600 call count cells in `buffer`, and provides for enough histogram cells to generate significant distribution-measurement results. For more information on the effects of `bufsize` on execution-distribution measurements, see `profil(2)`.

**EXAMPLE 1** Example to stop execution monitoring and write the results to a file.

To stop execution monitoring and write the results to a file, use the following:

```c
monitor( (int (*)( ) )0, (int (*)( ) )0, (WORD *)0, 0, 0);
```

Use `prof` to examine the results.

**USAGE** Additional calls to `monitor()` after `main()` has been called and before `exit()` has been called will add to the function-call count capacity, but such calls will also replace and restart the `profil()` histogram computation.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO** `cc(1B), profil(2), end(3C), attributes(5), prof(5)`
msync() function writes all modified copies of pages over the range
[addr, addr + len] to the underlying hardware, or invalidates any copies so that further
references to the pages will be obtained by the system from their permanent storage
locations. The permanent storage for a modified MAP_SHARED mapping is the file the
page is mapped to; the permanent storage for a modified MAP_PRIVATE mapping is
its swap area.

The flags argument is a bit pattern built from the following values:

- MS_ASYNC perform asynchronous writes
- MS_SYNC perform synchronous writes
- MS_INVALIDATE invalidate mappings

If flags is MS_ASYNC or MS_SYNC, the function synchronizes the file contents to match
the current contents of the memory region.

- All write references to the memory region made prior to the call are visible by
  subsequent read operations on the file.
- All writes to the same portion of the file prior to the call may or may not be visible
  by read references to the memory region.
- Unmodified pages in the specified range are not written to the underlying
  hardware.

If flags is MS_ASYNC, the function may return immediately once all write operations
are scheduled; if flags is MS_SYNC, the function does not return until all write
operations are completed.

If flags is MS_INVALIDATE, the function synchronizes the contents of the memory
region to match the current file contents.

- All writes to the mapped portion of the file made prior to the call are visible by
  subsequent read references to the mapped memory region.
- All write references prior to the call, by any process, to memory regions mapped to
  the same portion of the file using MAP_SHARED, are visible by read references to
  the region.

If msync() causes any write to the file, then the file’s st_ctime and st_mtime fields
are marked for update.

Upon successful completion, msync() returns 0; otherwise, it returns −1 and sets
errno to indicate the error.

The msync() function will fail if:
msync(3C)

EBUSY    Some or all of the addresses in the range \([addr, addr + len)\) are locked and \(MS\_SYNC\) with the \(MS\_INVALIDATE\) option is specified.
EAGAIN   Some or all pages in the range \([addr, addr + len)\) are locked for I/O.
EINVAL   The \(addr\) argument is not a multiple of the page size as returned by \(sysconf\)(3C).
          The \(flags\) argument is not some combination of \(MS\_ASYNC\) and \(MS\_INVALIDATE\).
EIO      An I/O error occurred while reading from or writing to the file system.
ENOMEM   Addresses in the range \([addr, addr + len)\) are outside the valid range for the address space of a process, or specify one or more pages that are not mapped.
EPERM    \(MS\_INVALIDATE\) was specified and one or more of the pages is locked in memory.

USAGE   The \(msync()\) function should be used by programs that require a memory object to be in a known state, for example, in building transaction facilities.

Normal system activity can cause pages to be written to disk. Therefore, there are no guarantees that \(msync()\) is the only control over when pages are or are not written to disk.

ATTRIBUTES See \(attributes(5)\) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO \(mempct1(2), mmap(2), sysconf(3C), attributes(5), standards(5)\)
### mtmalloc, mallocctl – MT hot memory allocator

#### SYNOPSIS

```c
#include <mtmalloc.h>
cc -o a.out -lthread -lmtmalloc

void *malloc(size_t size);
void *calloc(size_t nelem, size_t elsize);
void free(void *ptr);
void *memalign(size_t alignment, size_t size);
void *realloc(void *ptr, size_t size);
void *valloc(size_t size);
void mallocctl(int cmd, long value);
```

#### DESCRIPTION

The `malloc()` and `free()` functions provide a simple general-purpose memory allocation package that is suitable for use in high performance multithreaded applications. The suggested use of this library is in multithreaded applications; it can be used for single threaded applications, but there is no advantage in doing so. This library cannot be dynamically loaded via `dlopen()` during runtime because there must be only one manager of the process heap.

The `malloc()` function returns a pointer to a block of at least `size` bytes suitably aligned for any use.

The argument to `free()` is a pointer to a block previously allocated by `malloc()`, `calloc()` or `realloc()`. After `free()` is performed this space is available for further allocation. If `ptr` is a null pointer, no action occurs.

Undefined results will occur if the space assigned by `malloc()` is overrun or if a random number is handed to `free()`. A freed pointer that is passed to `free()` will send a `SIGABRT` signal to the calling process. This behavior is controlled by `mallocctl()`.

The `calloc()` function allocates a zero-initialized space for an array of `nelem` elements of size `elsize`.

The `memalign()` function allocates `size` bytes on a specified alignment boundary and returns a pointer to the allocated block. The value of the returned address is guaranteed to be an even multiple of `alignment`. Note that the value of `alignment` must be a power of two, and must be greater than or equal to the size of a word.

The `realloc()` function changes the size of the block pointed to by `ptr` to `size` bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. If `ptr` is NULL, `realloc()` behaves like `malloc()` for the specified size. If `size` is 0 and `ptr` is not a null pointer, the object pointed to is freed.
The `valloc()` function has the same effect as `malloc()`, except that the allocated memory will be aligned to a multiple of the value returned by `sysconf(_SC_PAGESIZE)`.

After possible pointer coercion, each allocation routine returns a pointer to a space that is suitably aligned for storage of any type of object.

The `malloc()`, `realloc()`, `calloc()`, `memalign()`, and `valloc()` functions will fail if there is not enough available memory.

The `mallocctl()` function controls the behavior of the `malloc` library. The options fall into two general classes, debugging options and performance options.

- **MTDOUBLEFREE**: Allows double free of a pointer. Setting value to 1 means yes and 0 means no. The default behavior of double free results in a core dump.
- **MTDEBUGPATTERN**: Writes misaligned data into the buffer after `free()`. When the buffer is reallocated, the contents are verified to ensure that there was no access to the buffer after the `free`. If the buffer has been dirtied, a SIGABRT signal is delivered to the process. Setting value to 1 means yes and 0 means no. The default behavior is to not write misaligned data. The pattern used is `0xdeadbeef`. Use of this option results in a performance penalty.
- **MTINITBUFFER**: Writes misaligned data into the newly allocated buffer. This option is useful for detecting some accesses before initialization. Setting value to 1 means yes and 0 means no. The default behavior is to not write misaligned data to the newly allocated buffer. The pattern used is `0xbaddcafe`. Use of this option results in a performance penalty.
- **MTCHUNKSIZE**: This option changes the size of allocated memory when a pool has exhausted all available memory in the buffer. Increasing this value allocates more memory for the application. A substantial performance gain can occur because the library makes fewer calls to the OS for more memory. Acceptable number values are between 9 and 256; the default value is 9. This value is multiplied by 8192.

If there is no available memory, `malloc()`, `realloc()`, `memalign()`, `valloc()`, and `calloc()` return a null pointer. When `realloc()` is called with `size > 0` and returns `NULL`, the block pointed to by `ptr` is left intact. If `size`, `nelem`, or `elsize` is 0, either a null pointer or a unique pointer that can be passed to `free()` is returned.

If `malloc()`, `calloc()`, or `realloc()` returns unsuccessfully, `errno` will be set to indicate the error.

**RETURN VALUES**

If there is no available memory, `malloc()`, `realloc()`, `memalign()`, `valloc()`, and `calloc()` return a null pointer. When `realloc()` is called with `size > 0` and returns `NULL`, the block pointed to by `ptr` is left intact. If `size`, `nelem`, or `elsize` is 0, either a null pointer or a unique pointer that can be passed to `free()` is returned.

If `malloc()`, `calloc()`, or `realloc()` returns unsuccessfully, `errno` will be set to indicate the error.
The `malloc()`, `calloc()`, and `realloc()` functions will fail if:

- **ENOMEM** The physical limits of the system are exceeded by `size` bytes of memory which cannot be allocated.
- **EAGAIN** There is not enough memory available to allocate `size` bytes of memory; but the application could try again later.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`brk(2)`, `getrlimit(2)`, `bsdmalloc(3MALLOC)`, `dlopen(3DL)`, `malloc(3C)`, `malloc(3MALLOC)`, `mapmalloc(3MALLOC)`, `signal(3HEAD)`, `watchmalloc(3MALLOC)`, `attributes(5)`

**WARNINGS**

Undefined results will occur if the size requested for a block of memory exceeds the maximum size of a process’s heap. This information may be obtained using `getrlimit()`.

**NOTES**

Comparative Features of `malloc(3C)`, `bsdmalloc(3MALLOC)`, `malloc(3MALLOC)`, and `mtmalloc`.

- The `bsdmalloc(3MALLOC)` routines afford better performance, but are space-inefficient.
- The `malloc(3MALLOC)` routines are space-efficient, but have slower performance.
- The standard, fully SCD-compliant `malloc` routines are a trade-off between performance and space-efficiency.
- The `mtmalloc` routines provide fast, concurrent `malloc()` implementation that is space-inefficient.

The `free()` function does not set `errno`. 
NAME
ndbm, dbm_clearerr, dbm_close, dbm_delete, dbm_error, dbm_fetch, dbm_firstkey, 
dbm_nextkey, dbm_open, dbm_store – database functions

SYNOPSIS
#include <ndbm.h>

int dbm_clearerr (DBM *db);
void dbm_close (DBM *db);
int dbm_delete (DBM *db, datum key);
int dbm_error (DBM *db);
datum dbm_fetch (DBM *db, datum key);
datum dbm_firstkey (DBM *db);
datum dbm_nextkey (DBM *db);
DBM *dbm_open (const char *file, int open_flags, mode_t file_mode);
int dbm_store (DBM *db, datum key, datum content, int store_mode);

DESCRIPTION
These functions create, access and modify a database. They maintain key/content pairs
in a database. The functions will handle large databases (up to a billion blocks) and
will access a keyed item in one or two file system accesses. This package replaces the
earlier dbm(3UCB) library, which managed only a single database.

keys and contents are described by the datum typedef. A datum consists of at least two
members, dptr and dsize. The dptr member points to an object that is dsize bytes
in length. Arbitrary binary data, as well as ASCII character strings, may be stored in
the object pointed to by dptr.

The database is stored in two files. One file is a directory containing a bit map of keys
and has .dir as its suffix. The second file contains all data and has .pag as its suffix.

The dbm_open() function opens a database. The file argument to the function is
the pathname of the database. The function opens two files named file.dir and
file.pag. The open_flags argument has the same meaning as the flags argument of
open(2) except that a database opened for write-only access opens the files for read
and write access. The file_mode argument has the same meaning as the third argument
of open(2).

The dbm_close() function closes a database. The argument db must be a pointer to a
dbm structure that has been returned from a call to dbm_open().

The dbm_fetch() function reads a record from a database. The argument db is a
pointer to a database structure that has been returned from a call to dbm_open(). The
argument key is a datum that has been initialized by the application program to the
value of the key that matches the key of the record the program is fetching.

The dbm_store() function writes a record to a database. The argument db is a
pointer to a database structure that has been returned from a call to dbm_open(). The
argument key is a datum that has been initialized by the application program to the
value of the key that identifies (for subsequent reading, writing or deleting) the record the program is writing. The argument content is a datum that has been initialized by the application program to the value of the record the program is writing. The argument store_mode controls whether dbm_store() replaces any pre-existing record that has the same key that is specified by the key argument. The application program must set store_mode to either DBM_INSERT or DBM_REPLACE. If the database contains a record that matches the key argument and store_mode is DBM_REPLACE, the existing record is replaced with the new record. If the database contains a record that matches the key argument and store_mode is DBM_INSERT, the existing record is not replaced with the new record. If the database does not contain a record that matches the key argument and store_mode is either DBM_INSERT or DBM_REPLACE, the new record is inserted in the database.

The dbm_delete() function deletes a record and its key from the database. The argument db is a pointer to a database structure that has been returned from a call to dbm_open(). The argument key is a datum that has been initialized by the application program to the value of the key that identifies the record the program is deleting.

The dbm_firstkey() function returns the first key in the database. The argument db is a pointer to a database structure that has been returned from a call to dbm_open().

The dbm_nextkey() function returns the next key in the database. The argument db is a pointer to a database structure that has been returned from a call to dbm_open(). The dbm_firstkey() function must be called before calling dbm_nextkey(). Subsequent calls to dbm_nextkey() return the next key until all of the keys in the database have been returned.

The dbm_error() function returns the error condition of the database. The argument db is a pointer to a database structure that has been returned from a call to dbm_open().

The dbm_clearerr() function clears the error condition of the database. The argument db is a pointer to a database structure that has been returned from a call to dbm_open().

These database functions support key/content pairs of at least 1024 bytes.

**RETURN VALUES**

The dbm_store() and dbm_delete() functions return 0 when they succeed and a negative value when they fail.

The dbm_store() function returns 1 if it is called with a flags value of DBM_INSERT and the function finds an existing record with the same key.

The dbm_error() function returns 0 if the error condition is not set and returns a non-zero value if the error condition is set.

The return value of dbm_clearerr() is unspecified.
The `dbm_firstkey()` and `dbm_nextkey()` functions return a key datum. When the end of the database is reached, the `dptr` member of the key is a null pointer. If an error is detected, the `dptr` member of the key is a null pointer and the error condition of the database is set.

The `dbm_fetch()` function returns a content datum. If no record in the database matches the key or if an error condition has been detected in the database, the `dptr` member of the content is a null pointer.

The `dbm_open()` function returns a pointer to a database structure. If an error is detected during the operation, `dbm_open()` returns a `(DBM *)0`.

No errors are defined.

The following code can be used to traverse the database:

```c
for(key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

The `dbm` functions provided in this library should not be confused in any way with those of a general-purpose database management system. These functions do not provide for multiple search keys per entry, they do not protect against multi-user access (in other words they do not lock records or files), and they do not provide the many other useful database functions that are found in more robust database management systems. Creating and updating databases by use of these functions is relatively slow because of data copies that occur upon hash collisions. These functions are useful for applications requiring fast lookup of relatively static information that is to be indexed by a single key.

The `dptr` pointers returned by these functions may point into static storage that may be changed by subsequent calls.

The `dbm_delete()` function does not physically reclaim file space, although it does make it available for reuse.

After calling `dbm_store()` or `dbm_delete()` during a pass through the keys by `dbm_firstkey()` and `dbm_nextkey()`, the application should reset the database by calling `dbm_firstkey()` before again calling `dbm_nextkey()`.

### EXAMPLES

#### EXAMPLE 1 Using the Database Functions

The following example stores and retrieves a phone number, using the name as the key. Note that this example does not include error checking.

```c
#include <ndbm.h>
#include <stdio.h>
#include <fcntl.h>
#define NAME "Bill"
#define PHONE_NO "123-4567"
#define DB_NAME "phones"
main()
{
    DBM *db;
    datum name = {NAME, sizeof (NAME)};
```
**EXAMPLE 1 Using the Database Functions (Continued)**

```c

datum put_phone_no = {PHONE_NO, sizeof (PHONE_NO)};
datum get_phone_no;
/* Open the database and store the record */
db = dbm_open(DB_NAME, O_RDWR | O_CREAT, 0660);
(void) dbm_store(db, name, put_phone_no, DBM_INSERT);
/* Retrieve the record */
get_phone_no = dbm_fetch(db, name);
(void) printf("Name: %s, Phone Number: %s\n", name.dptr,
                    get_phone_no.dptr);
/* Close the database */
dbm_close(db);
return (0);
```

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Unsafe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

ar(1), cat(1), cp(1), tar(1), open(2), dbm(3UCB), netconfig(4), attributes(5)

**NOTES**

The .pag file will contain holes so that its apparent size may be larger than its actual content. Older versions of the UNIX operating system may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp(1), cat(1), tar(1), ar(1)) without filling in the holes.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover all key/content pairs that hash together must fit on a single block. dbm_store() will return an error in the event that a disk block fills with inseparable data.

The order of keys presented by dbm_firstkey() and dbm_nextkey() depends on a hashing function.

There are no interlocks and no reliable cache flushing; thus concurrent updating and reading is risky.

The database files (file.dir and file.pag) are binary and are architecture-specific (for example, they depend on the architecture’s byte order.) These files are not guaranteed to be portable across architectures.
nice(3UCB)

NAME    nice – change priority of a process

SYNOPSIS /usr/ucb/cc [ flag ... ] file ...
        #include<unistd.h>
        int nice(incr);
        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(2). 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 VALUES 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:

EPERM The value of incr specified was negative, and the effective user ID is not the privileged user.

SEE ALSO nice(1), renice(1), fork(2), priocntl(2), getpriority(3C)

NOTES Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-threaded applications is unsupported.
nlist(3UCB)

NAME
nlist – get entries from symbol table

SYNOPSIS
/usr/ucb/cc [ flag ... ] file ...
#include <nlist.h>

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

DESCRIPTION
nlist() examines the symbol table from the executable image 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 pointed to by nl consists of
an array of structures containing names, types and values. The n_name field of each
such structure is taken to be a pointer to a character string representing a symbol
name. The list is terminated by an entry with a NULL pointer (or a pointer to a NULL
string) in the n_name field. For each entry in nl, if the named symbol is present in the
executable image’s symbol table, its value and type are placed in the n_value and
n_type fields. If a symbol cannot be located, the corresponding n_type field of nl is
set to zero.

RETURN VALUES
Upon normal completion, nlist() returns the number of symbols that were not
located in the symbol table. If an error occurs, nlist() returns -1 and sets all of the
n_type fields in members of the array pointed to by nl to zero.

SEE ALSO
nlist(3ELF), a.out(4)

NOTES
Use of these interfaces should be restricted to only applications written on BSD
platforms. Use of these interfaces with any of the system libraries or in multi-thread
applications is unsupported.

Only the n_value field is compatibly set. Other fields in the nlist structure are
filled with the ELF (Executable and Linking Format) values (see nlist(3ELF) and
a.out(4)).
The `nl_langinfo()` function returns a pointer to a null-terminated string containing information relevant to a particular language or cultural area defined in the programs locale. The manifest constant names and values of `item` are defined by `<langinfo.h>`. For example:

```c
nl_langinfo(ABDAY_1);
```

would return a pointer to the string “Dim” if the identified language was French and a French locale was correctly installed; or “Sun” if the identified language was English.

If `setlocale(3C)` has not been called successfully, or if data for a supported language is either not available, or if `item` is not defined therein, then `nl_langinfo()` returns a pointer to the corresponding string in the C locale. In all locales, `nl_langinfo()` returns a pointer to an empty string if `item` contains an invalid setting.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

The array pointed to by the return value should not be modified by the program. Subsequent calls to `nl_langinfo()` may overwrite the array.
NAME    offsetof – offset of structure member

SYNOPSIS #include <stddef.h>

size_t offsetof(type, member-designator);

DESCRIPTION The offsetof() macro defined in <stddef.h> expands to an integral constant expression that has type size_t. The value of this expression is the offset in bytes to the structure member (designated by member-designator) from the beginning of its structure (designated by type).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO attributes(5)
The opendir() function opens a directory stream corresponding to the directory named by the `dirname` argument.

The fdopendir() function opens a directory stream for the directory file descriptor `fd`. The directory file descriptor should not be used or closed following a successful function call, as this might cause undefined results from future operations on the directory stream obtained from the call. Use closedir(3C) to close a directory stream.

The directory stream is positioned at the first entry. If the type `DIR` is implemented using a file descriptor, applications will only be able to open up to a total of \{OPEN_MAX\} files and directories. A successful call to any of the exec functions will close any directory streams that are open in the calling process. See exec(2).

Upon successful completion, opendir() and fdopendir() return a pointer to an object of type `DIR`. Otherwise, a null pointer is returned and `errno` is set to indicate the error.

The opendir() function will fail if:

- **EACCES**: Search permission is denied for the component of the path prefix of `dirname` or read permission is denied for `dirname`.
- **ELOOP**: Too many symbolic links were encountered in resolving path.
- **ENAMETOOLONG**: The length of the `dirname` argument exceeds \{PATH_MAX\}, or a path name component is longer than \{NAME_MAX\} while \{_POSIX_NO_TRUNC\} is in effect.
- **ENOENT**: A component of `dirname` does not name an existing directory or `dirname` is an empty string.
- **ENOTDIR**: A component of `dirname` is not a directory.

The fdopendir() function will fail if:

- **ENOTDIR**: The file descriptor `fd` does not reference a directory.

The opendir() function may fail if:

- **EMFILE**: There are \{OPEN_MAX\} file descriptors currently open in the calling process.
- **ENAMETOOLONG**: Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
Too many files are currently open on the system.

**USAGE**
The `opendir()` and `fdopendir()` functions should be used in conjunction with `readdir(3C)`, `closedir(3C)` and `rewinddir(3C)` to examine the contents of the directory (see the **EXAMPLES** section in `readdir(3C)`). This method is recommended for portability.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td><code>opendir()</code> is Standard; <code>fdopendir()</code> is Evolving</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**
`lstat(2)`, `symlink(2)`, `closedir(3C)`, `readdir(3C)`, `rewinddir(3C)`, `attributes(5)`
NAME perror, errno – print system error messages

SYNOPSIS #include <stdio.h>
void perror(const char *s);
#include <errno.h>
int errno;

DESCRIPTION The perror() function produces a message on the standard error output (file descriptor 2) describing the last error encountered during a call to a system or library function. The argument string s is printed, followed by a colon and a blank, followed by the message and a NEWLINE character. If s is a null pointer or points to a null string, the colon is not printed. The argument string should include the name of the program that incurred the error. The error number is taken from the external variable errno, which is set when errors occur but not cleared when non-erroneous calls are made. See intro(2).

USAGE If the application is linked with -lint1, then messages printed from this function are in the native language specified by the LC_MESSAGES locale category. See setlocale(3C).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO intro(2), fmtmsg(3C), gettext(3C), setlocale(3C), strerror(3C), attributes(5)
**NAME**
pfmt – display error message in standard format

**SYNOPSIS**
```c
#include <pfmt.h>

int pfmt(FILE *stream, long flags, char *format, ... /* arg */);
```

**DESCRIPTION**
The `pfmt()` retrieves a format string from a locale-specific message database (unless `MM_NOGET` is specified) and uses it for `printf(3C)` style formatting of `args`. The output is displayed on `stream`.

The `pfmt()` function encapsulates the output in the standard error message format (unless `MM_NOSTD` is specified, in which case the output is similar to `printf()`).

If the `printf()` format string is to be retrieved from a message database, the `format` argument must have the following structure:
```
<catalog>:<msgnum>:<defmsg>.
```

If `MM_NOGET` is specified, only the `defmsg` field must be specified.

The `catalog` field is used to indicate the message database that contains the localized version of the format string. This field must be limited to 14 characters selected from the set of all characters values, excluding `\0` (null) and the ASCII codes for `/` (slash) and `:` (colon).

The `msgnum` field is a positive number that indicates the index of the string into the message database.

If the catalog does not exist in the locale (specified by the last call to `setlocale(3C)` using the `LC_ALL` or `LC_MESSAGES` categories), or if the message number is out of bound, `pfmt()` will attempt to retrieve the message from the C locale. If this second retrieval fails, `pfmt()` uses the `defmsg` field of the `format` argument.

If `catalog` is omitted, `pfmt()` will attempt to retrieve the string from the default catalog specified by the last call to `setlocale(3C)`. In this case, the `format` argument has the following structure:
```
:<msgnum>:<defmsg>.
```

The `pfmt()` will output `Message not found!!
` as `format` string if `catalog` is not a valid catalog name, if no catalog is specified (either explicitly or with `setlocale()`), if `msgnum` is not a valid number, or if no message could be retrieved from the message databases and `defmsg` was omitted.

The `flags` argument determine the type of output (such as whether the `format` should be interpreted as is or encapsulated in the standard message format), and the access to message catalogs to retrieve a localized version of `format`.

The `flags` argument is composed of several groups, and can take the following values (one from each group):
Output format control

MM_NOSTD  Do not use the standard message format, interpret format as printf() format. Only catalog access control flags should be specified if MM_NOSTD is used; all other flags will be ignored.

MM_STD  Output using the standard message format (default value 0).

Catalog access control

MM_NOGET  Do not retrieve a localized version of format. In this case, only the defmsg field of the format is specified.

MM_GET  Retrieve a localized version of format from the catalog, using msgid as the index and defmsg as the default message (default value 0).

Severity (standard message format only)

MM_HALT  Generate a localized version of HALT, but do not halt the machine.

MM_ERROR  Generate a localized version of ERROR (default value 0).

MM_WARNING  Generate a localized version of WARNING.

MM_INFO  Generate a localized version of INFO.

Additional severities can be defined. Add-on severities can be defined with number-string pairs with numeric values from the range [5-255], using addsev(3C). The specified severity will be generated from the bitwise OR operation of the numeric value and other flags. If the severity is not defined, pfmt() uses the string SEV=N, where N is replaced by the integer severity value passed in flags.

Multiple severities passed in flags will not be detected as an error. Any combination of severities will be summed and the numeric value will cause the display of either a severity string (if defined) or the string SEV=N (if undefined).

Action

MM_ACTION  Specify an action message. Any severity value is superseded and replaced by a localized version of TO FIX.

The pfmt() function displays error messages in the following format:

| label: severity: text |

If no label was defined by a call tosetLabel(3C), the message is displayed in the format:

| severity: text |

If pfmt() is called twice to display an error message and a helpful action or recovery message, the output can look like:
Upon success, `pfmt()` returns the number of bytes transmitted. Upon failure, it returns a negative value:

\[ -1 \]

Write error to `stream`.

**EXAMPLE 1** Example of `pfmt()` function.

Example 1:

```c
setlabel("UX:test");
pfmt(stderr, MM_ERROR, "test:2:Cannot open file: %s\n", strerror(errno));
```

displays the message:

UX:test: ERROR: Cannot open file: No such file or directory

Example 2:

```c
setlabel("UX:test");
setcat("test");
pfmt(stderr, MM_ERROR, ":10:Syntax error\n");
pfmt(stderr, MM_ACTION, "55:Usage ...
");
```

displays the message

UX:test: ERROR: Syntax error
UX:test: TO FIX: Usage ...

**USAGE** Since it uses `gettext(3C)`, `pfmt()` should not be used.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-safe</td>
</tr>
</tbody>
</table>

**SEE ALSO** `addsev(3C), gettext(3C), lfmt(3C), printf(3C), setcat(3C), setlabel(3C), setlocale(3C), attributes(5), environ(5)`
NAME
plock – lock or unlock into memory process, text, or data

SYNOPSIS
#include <sys/lock.h>

int plock(int op);

DESCRIPTION
The plock() function allows the calling process to lock or unlock into memory its
text segment (text lock), its data segment (data lock), or both its text and data
segments (process lock). Locked segments are immune to all routine swapping. The
effective user ID of the calling process must be super-user to use this call.

The plock() function performs the function specified by op:

PROCLOCK  Lock text and data segments into memory (process lock).
TXTLOCK    Lock text segment into memory (text lock).
DATLOCK    Lock data segment into memory (data lock).
UNLOCK     Remove locks.

RETURN VALUES
Upon successful completion, 0 is returned. Otherwise, -1 is returned and errno
is set to indicate the error.

ERRORS
The plock() function fails and does not perform the requested operation if:

EAGAIN     Not enough memory.
EINVAL     The op argument is equal to PROCLOCK and a process lock, a text
lock, or a data lock already exists on the calling process; the op
argument is equal to TXTLOCK and a text lock or a process lock
already exists on the calling process; the op argument is equal to
DATLOCK and a data lock or a process lock already exists on the
calling process; or the op argument is equal to UNLOCK and no lock
exists on the calling process.

EPERM      The effective user of the calling process is not super-user.

USAGE
The mlock(3C) and mlockall(3C) functions are the preferred interfaces for process
locking.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
exec(2), exit(2), fork(2), memcntl(2), mlock(3C), mlockall(3C), attributes(5)
NAME  
popen, pclose – initiate a pipe to or from a process

SYNOPSIS

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
int pclose(FILE *stream);
```

DESCRIPTION

The `popen()` function creates a pipe between the calling program and the command to be executed. The arguments to `popen()` are pointers to null-terminated strings. The `command` argument consists of a shell command line. The `mode` argument is an I/O mode, either `r` for reading or `w` for writing. The value returned is a stream pointer such that one can write to the standard input of the command, if the I/O mode is `w`, by writing to the file `stream` (see `intro(3)`); and one can read from the standard output of the command, if the I/O mode is `r`, by reading from the file `stream`. Because open files are shared, a type `r` command may be used as an input filter and a type `w` as an output filter.

The environment of the executed command will be as if a child process were created within the `popen()` call using `fork(2)`. If the application is standard-conforming (see `standards(5)`), the child is invoked with the call:

```
execl("/usr/xpg4/bin/sh", "sh", "-c", command, (char *)0);
```

otherwise, the child is invoked with the call:

```
execl("/usr/bin/sh", "sh", "-c", command, (char *)0);
```

The `pclose()` function closes a stream opened by `popen()` by closing the pipe. It waits for the associated process to terminate and returns the termination status of the process running the command language interpreter. This is the value returned by `waitpid(2)`. See `wstat(3HEAD)` for more information on termination status.

RETURN VALUES

Upon successful completion, `popen()` returns a pointer to an open stream that can be used to read or write to the pipe. Otherwise, it returns a null pointer and may set `errno` to indicate the error.

Upon successful completion, `pclose()` returns the termination status of the command language interpreter as returned by `waitpid()`. Otherwise, it returns -1 and sets `errno` to indicate the error.

ERRORS

The `popen()` function may fail if:

- **EMFILE**  
  There are currently `FOPEN_MAX` or `STREAM_MAX` streams open in the calling process.

- **EINVAL**  
  The `mode` argument is invalid.

The `pclose()` function will fail if:

- **ECHILD**  
  The status of the child process could not be obtained, as described above.
The `popen()` function may also set `errno` values as described by `fork(2)` or `pipe(2)`.

If the original and `popen()` processes concurrently read or write a common file, neither should use buffered I/O. Problems with an output filter may be forestalled by careful buffer flushing, for example, with `fflush()` (see `fclose(3C)`). A security hole exists through the `IFS` and `PATH` environment variables. Full pathnames should be used (or `PATH` reset) and `IFS` should be set to space and tab (" \	").

The signal handler for `SIGCHLD` should be set to default when using `popen()`. If the process has established a signal handler for `SIGCHLD`, it will be called when the command terminates. If the signal handler or another thread in the same process issues a `wait(2)` call, it will interfere with the return value of `pclose()`. If the process's signal handler for `SIGCHLD` has been set to ignore the signal, `pclose()` will fail and `errno` will be set to `ECHILD`.

**EXAMPLE 1 popen() example**

The following program will print on the standard output (see `stdio(3C)`) the names of files in the current directory with a `.c` suffix.

```c
#include <stdio.h>
#include <stdlib.h>
main( )
{
  char *cmd = "/usr/bin/ls *.c";
  char buf[BUFSIZ];
  FILE *ptr;
  if ((ptr = popen(cmd, "r")) != NULL)
    while (fgets(buf, BUFSIZ, ptr) != NULL)
      (void) printf("%s", buf);
  (void) pclose(ptr);
  return 0;
}
```

**EXAMPLE 2 system() replacement**

The following code fragment can be used in a multithreaded process in place of the MT-Unsafe `system(3C)` function:

```c
pclose(popen(cmd, "w"));
```

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>
SEE ALSO

ksh(1), pipe(2), wait(2), waitpid(2), fclose(3C), fopen(3C), stdio(3C), system(3C), wstat(3HEAD), attributes(5), standards(5)
printf functions place output on the standard output stream `stdout`. The `fprintf()` function places output on the named output stream `stream`. The `sprintf()` function places output, followed by the null byte (`\0`), in consecutive bytes starting at `s`; it is the user's responsibility to ensure that enough storage is available. The `snprintf()` function is identical to `sprintf()` with the addition of the argument `n`, which specifies the size of the buffer referred to by `s`. The buffer is always terminated with the null byte.

Each of these functions converts, formats, and prints its arguments under control of the `format`. The `format` is a character string, beginning and ending in its initial shift state, if any. The `format` is composed of zero or more directives: ordinary characters, which are simply copied to the output stream and conversion specifications, each of which results in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the `format`. If the `format` is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Conversions can be applied to the `n`th argument after the `format` in the argument list, rather than to the next unused argument. In this case, the conversion 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 radix character in the output string. The radix character is defined by the program's locale (category `LC_NUMERIC`). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (`.`).
Each conversion specification is introduced by the % character or by the character sequence $n$, after which the following appear in sequence:

- An optional field, consisting of a decimal digit string followed by a $, specifying the next argument to be converted. If this field is not provided, the args following the last argument converted will be used.
- Zero or more flags (in any order), which modify the meaning of the conversion specification.
- An optional minimum field width. If the converted value has fewer bytes than the field width, it will be padded with spaces by default on the left; it will be padded on the right, if the left-adjustment flag (hyphen), described below, is given to the field width. The field width takes the form of an asterisk (*), described below, or a decimal integer.

If the conversion character is s, a standard-conforming application (see standards) interprets the field width as the minimum number of bytes to be printed; an application that is not standard-conforming interprets the field width as the minimum number of columns of screen display. For an application that is not standard-conforming, %10s means if the converted value has a screen width of 7 columns, 3 spaces would be padded on the right.

If the format is %ws, then the field width should be interpreted as the minimum number of columns of screen display.

- An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions (the field is padded with leading zeros); the number of digits to appear after the radix character for the e, E, and f conversions, the maximum number of significant digits for the g and G conversions; or the maximum number of bytes to be printed from a string in s and S conversions. The precision takes the form of a period (.) followed either by an asterisk (*), described below, or an optional decimal digit string, where a null digit string is treated as 0. If a precision appears with any other conversion character, the behavior is undefined.

If the conversion character is s or S, a standard-conforming application (see standards) interprets the precision as the maximum number of bytes to be written; an application that is not standard-conforming interprets the precision as the maximum number of columns of screen display. For an application that is not standard-conforming, %.5s would print only the portion of the string that would display in 5 screen columns. Only complete characters are written.

For %ws, the precision should be interpreted as the maximum number of columns of screen display. The precision takes the form of a period (.) followed by a decimal digit string; a null digit string is treated as zero. Padding specified by the precision overrides the padding specified by the field width.

- An optional h specifies that a following d, i, o, u, x, or X conversion character applies to a type short int or type unsigned short int argument (the argument will be promoted according to the integral promotions, and its value converted to type short int or unsigned short int before printing); an optional h specifying that a following n conversion character applies to a pointer to a type short int argument; an optional l (ell) specifying that a following d, i, o, u, x, or X conversion character applies to a type long int or unsigned long
**printf(3C)**

- **int argument; an optional l (ell) specifying that a following n conversion character applies to a pointer to a type long int argument; an optional ll (ell ell) specifying that a following d, i, o, u, x, or X conversion character applies to a type long long or unsigned long long argument; an optional ll (ell ell) specifying that a following n conversion character applies to a pointer to a long long argument; or an optional L specifying that a following e, E, f, g, or G conversion character applies to a type long double argument. If an h, l, ll, or L appears with any other conversion character, the behavior is undefined.**

- An optional l (ell) specifying that a following c conversion character applies to a **wint_t** argument; an optional l (ell) specifying that a following s conversion character applies to a pointer to a **wchar_t** argument.

- A **conversion character** (see below) that indicates the type of conversion to be applied.

A field width, or precision, or both may be indicated by an asterisk (*). In this case, an argument of type int supplies the field width or precision. Arguments specifying field width, or precision, or both must appear in that order before the argument, if any, to be converted. A negative field width is taken as a – flag followed by a positive field width. A negative precision is taken as if the precision were omitted. In format strings containing the %n$ form of a conversion specification, a field width or precision may be indicated by the sequence *m$, where m is a decimal integer in the range [1, NL_ARGMAX] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```c
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

The format can contain either numbered argument specifications (that is, %n$ and *m$), or unnumbered argument specifications (that is, % and *), but normally not both. The only exception to this is that % can be mixed with the %n$ form. The results of mixing numbered and unnumbered argument specifications in a format string are undefined. When numbered argument specifications are used, specifying the Nth argument requires that all the leading arguments, from the first to the (N–1)th, are specified in the format string.

**Flag Characters**

The flag characters and their meanings are:

- `'` The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %g, or %G) will be formatted with thousands' grouping characters. For other conversions the behavior is undefined. The non-monetary grouping character is used.

- `−` The result of the conversion will be left-justified within the field. The conversion will be right-justified if this flag is not specified.

- `+` The result of a signed conversion will always begin with a sign (+ or –). The conversion will begin with a sign only when a negative value is converted if this flag is not specified.
If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a space will be placed before the result. This means that if the space and + flags both appear, the space flag will be ignored.

# The value is to be converted to an alternate form. For c, d, i, s, and u conversions, the flag has no effect. For an o conversion, it increases the precision (if necessary) to force the first digit of the result to be a zero. For x or X conversion, a non-zero result will have 0x (or 0X) prepended to it. For e, E, f, g, and G conversions, the result will always contain a radix character, even if no digits follow the radix character. Without this flag, the radix character appears in the result of these conversions only if a digit follows it. For g and G conversions, trailing zeros will not be removed from the result as they normally are.

0 For d, i, o, u, x, X, 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. If the 0 and ‘ flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined.

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.

The conversion characters and their meanings are:

d, i The int argument is converted to a signed decimal in the style [-] dddd. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

o The unsigned int argument is converted to unsigned octal format in the style dddd. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

u The unsigned int argument is converted to unsigned decimal format in the style dddd. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

x The unsigned int argument is converted to unsigned hexadecimal format in the style dddd; the letters abcdef are used. The precision specifies the minimum number of digits to appear; if the value being
converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no characters.

X Behaves the same as the x conversion character except that letters ABCDEF are used instead of abcdef.

f The double argument is converted to decimal notation in the style \([-]ddd \ . \ ddd\), where the number of digits after the radix character (see setlocale(3C)) is equal to the precision specification. If the precision is missing it is taken as 6; if the precision is explicitly 0 and the # flag is not specified, no radix character appears. If a radix character appears, at least 1 digit appears before it. The value is rounded to the appropriate number of digits.

e,E The double argument is converted to the style \([-\)d. \ dded \± \ddd\), where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision. When the precision is missing it is taken as 6; if the precision is 0 and the # flag is not specified, no radix character appears. The E conversion 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.

g,G The double argument 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 an explicit precision is 0, it is taken as 1. The style used depends on the value converted: style e (or E) will be used only if the exponent resulting from the conversion is less than \(-4\) or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result. A radix character appears only if it is followed by a digit.

c The int argument is converted to an unsigned char, and the resulting byte is printed.

If an l (ell) qualifier is present, the wint_t argument is converted as if by an ls conversion specification with no precision and an argument that points to a two-element array of type wchar_t, the first element of which contains the wint_t argument to the ls conversion specification and the second element contains a null wide-character.

c Same as lc.

wc The int argument is converted to a wide character (wchar_t), and the resulting wide character is printed.

s The argument must be a pointer to an array of char. Bytes from the array are written up to (but not including) any terminating null byte. If a precision is specified, a standard-conforming application (see standards(5)) will write only the number of bytes specified by precision; an application that is not standard-conforming will write only the portion
of the string that will display in the number of columns of screen display specified by precision. If the precision is not specified, it is taken to be infinite, so all bytes up to the first null byte are printed. An argument with a null value will yield undefined results.

If an l (ell) qualifier is present, the argument must be a pointer to an array of type wchar_t. Wide-characters from the array are converted to characters (each as if by a call to the wcrtomb(3C) function, with the conversion state described by an mbstate_t object initialized to zero before the first wide-character is converted) up to and including a terminating null wide-character. The resulting characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array must contain a null wide-character. If a precision is specified, no more than that many characters (bytes) are written (including shift sequences, if any), and the array must contain a null wide-character if, to equal the character sequence length given by the precision, the function would need to access a wide-character one past the end of the array. In no case is a partial character written.

S
Same as l s.

ws
The argument must be a pointer to an array of wchar_t. Bytes from the array are written up to (but not including) any terminating null character. If the precision is specified, only that portion of the wide-character array that will display in the number of columns of screen display specified by precision will be written. If the precision is not specified, it is taken to be infinite, so all wide characters up to the first null character are printed. An argument with a null value will yield undefined results.

p
The argument must be a pointer to void. The value of the pointer is converted to a set of sequences of printable characters, which should be the same as the set of sequences that are matched by the %p conversion of the scanf(3C) function.

n
The argument must be a pointer to an integer into which is written the number of bytes written to the output standard I/O stream so far by this call to one of the printf() functions. No argument is converted.

% Print a %; no argument is converted. The entire conversion specification must be %%. If a conversion specification does not match one of the above forms, the behavior is undefined.

If a floating-point value is the internal representation for infinity, the output is [±]Infinity, where Infinity is either Infinity or Inf, depending on the desired output string length. Printing of the sign follows the rules described above.

If a floating-point value is the internal representation for “not-a-number,” the output is [±]NaN. Printing of the sign follows the rules described above.
In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by printf() and fprintf() are printed as if the putc(3C) function had been called.

The st_ctime and st_mtime fields of the file will be marked for update between the call to a successful execution of printf() or fprintf() and the next successful completion of a call to fflush(3C) or fclose(3C) on the same stream or a call to exit(3C) or abort(3C).

**RETURN VALUES**
The printf(), fprintf(), and sprintf() functions return the number of bytes transmitted (excluding the terminating null byte in the case of sprintf()).

The snprintf() function returns the number of characters formatted, that is, the number of characters that would have been written to the buffer if it were large enough. If the value of n is 0 on a call to snprintf(), an unspecified value less than 1 is returned.

Each function returns a negative value if an output error was encountered.

**ERRORS**
For the conditions under which printf() and fprintf() will fail and may fail, refer to fputc(3C) or fputwc(3C).

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.

In addition, printf() and fprintf() may fail if:

- **ENOMEM** Insufficient storage space is available.

**USAGE**
If the application calling the printf() functions has any objects of type wint_t or wchar_t, it must also include the header <wchar.h> to have these objects defined.

The sprintf() and snprintf() functions are MT-Safe in multithreaded applications. The printf() and fprintf() functions can be used safely in multithreaded applications, as long as setlocale(3C) is not being called to change the locale.

**Escape Character Sequences**
It is common to use the following escape sequences built into the C language when entering format strings for the printf() functions, but these sequences are processed by the C compiler, not by the printf() function.

- \a Alert. Ring the bell.
- \b Backspace. Move the printing position to one character before the current position, unless the current position is the start of a line.
In addition, the C language supports character sequences of the form
\octal-numberand\hex-numberwhich translates into the character represented by the octal or hexadecimal number. For example, if ASCII representations are being used, the letter 'a' may be written as \141 and 'Z' as \132. This syntax is most frequently used to represent the null character as \0. This is exactly equivalent to the numeric constant zero (0). Note that the octal number does not include the zero prefix as it would for a normal octal constant. To specify a hexadecimal number, omit the zero so that the prefix is an 'x' (uppercase 'X' is not allowed in this context). Support for hexadecimal sequences is an ANSI extension. See standards(5).

EXAMPLE 1 To print the language-independent date and time format, the following statement could be used:
printf (format, weekday, month, day, hour, min);
For American usage, format could be a pointer to the string:
"%s, %s %d, %d:%.2d"
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"
producing the message:
Sonntag, 3. Juli, 10:02

EXAMPLE 2 To print a date and time in the form Sunday, July 3, 10:02, where weekday and month are pointers to null-terminated strings:
printf("%s, %s %i, %d:%.2d", weekday, month, day, hour, min);
printf(3C)

EXAMPLE 2 To print a date and time in the form Sunday, July 3, 10:02, where weekday and month are pointers to null-terminated strings:  

EXAMPLE 3 To print pi to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

EXAMPLE 4 The following example applies only to applications which are not standard-conforming (see standards(5)). To print a list of names in columns which are 20 characters wide:

```
printf("%20s%20s%20s", lastname, firstname, middlename);
```

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
</tbody>
</table>

SEE ALSO exit(2), lseek(2), write(2), abort(3C), ecvt(3C), exit(3C), fclose(3C), fflush(3C), fputwc(3C), putc(3C), scanf(3C), setlocale(3C), stdio(3C), wcstombs(3C), wctomb(3C), attributes(5), environ(5), standards(5)
NAME | printf, fprintf, sprintf, vprintf, vfprintf, vsprintf – formatted output conversion

SYNOPSIS

```c
#include <stdio.h>

int printf(const char *format, ...);

int fprintf(FILE *stream, const char *format, va_list);

FILE *stream;
char *format;
va_dcl;

char *sprintf(char *s, const char *format, va_list);

char *s, *format;
va_dcl;

int vprintf(const char *format, va_list);

char *format;
va_list ap;

int vfprintf(FILE *stream, const char *format, va_list);

FILE *stream;
char *format;
va_list ap;

char *vsprintf(char *s, const char *format, va_list);

char *s, *format;
va_list ap;
```

DESCRIPTION

printf() places output on the standard output stream stdout. fprintf() places output on the named output stream. 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.

vprintf(), vfprintf(), and vsprintf() are the same as printf(), fprintf(), and sprintf() respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by vaargs(3HEAD).

Each of these functions converts, formats, and prints its args 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 args. The results are undefined if there are insufficient args for the format. If the format is exhausted while args remain, the excess args 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 a 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 arg. 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 arg supplies the field width or precision. The arg that is actually converted is not fetched until the conversion letter is seen, so the args specifying field width or precision must appear before the arg (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:
The integer \texttt{arg} is converted to signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x and X), respectively; the letters \texttt{abcdef} are used for \texttt{x} conversion and the letters \texttt{ABCDEF} for \texttt{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 \texttt{NULL} string.

The float or double \texttt{arg} is converted to decimal notation in the style \texttt{[-]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.

The float or double \texttt{arg} is converted in the style \texttt{[-]d.ddd\,e\pm\,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 \texttt{E} format code will produce a number with \texttt{E} instead of \texttt{e} introducing the exponent. The exponent always contains at least two digits.

The float or double \texttt{arg} is printed in style \texttt{f} or \texttt{e} (or in style \texttt{E} in the case of a \texttt{G} format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style \texttt{e} or \texttt{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 \texttt{e}, \texttt{E}, \texttt{f}, \texttt{g}, and \texttt{G} formats print IEEE indeterminate values (infinity or not-a-number) as “Infinity” or “NaN” respectively.

The character \texttt{arg} is printed.

The \texttt{arg} is taken to be a string (character pointer) and characters from the string are printed until a \texttt{NULL} character (\texttt{\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 \texttt{NULL} character are printed. A \texttt{NULL} value for \texttt{arg} will yield undefined results.

Print a \texttt{\%}; 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. Characters generated by `printf()` and `fprintf()` are printed as if `putc()` had been called.

**RETURN VALUES**

Upon success, `printf()` and `fprintf()` return the number of characters transmitted, excluding the null character. `vprintf()` and `vfprintf()` return the number of characters transmitted. `sprintf()` and `vprintf()` always return `s`. If an output error is encountered, `printf()`, `fprintf()`, `vprintf()`, and `vfprintf()` return EOF.

**EXAMPLES**

**EXAMPLE 1** Examples of the `printf` Command To Print a Date and Time

To print a date and time in the form “Sunday, July 3, 10:02,” where `weekday` and `month` are pointers to `NULL`-terminated strings:

```c
printf("%s, %s %i, %d:%.2d", weekday, month, day, hour, min);
```

**EXAMPLE 2** Examples of the `printf` Command To Print to Five Decimal Places

To print to five decimal places:

```c
printf("pi = %.5f", 4 * atan(1.0));
```

**SEE ALSO**

`econvert(3C)`, `putc(3C)`, `scanf(3C)`, `vprintf(3C)`, `vfprintf(3C)`, `vaargs(3HEAD)`

**NOTES**

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

Very wide fields (>128 characters) fail.
pset_getloadavg(3C)

NAME
pset_getloadavg — get system load averages for a processor set

SYNOPSIS
#include <sys/pset.h>
#include <sys/loadavg.h>

int pset_getloadavg(psetid_t pset, double loadavg[], int nelem);

DESCRIPTION
The pset_getloadavg() function returns the number of processes assigned to the
specified processor set that are in the system run queue, averaged over various
periods of time. Up to nelem samples are retrieved and assigned to successive elements
of loadavg[]. The system imposes a maximum of 3 samples, representing averages over
the last 1, 5, and 15 minutes, respectively.

The LOADAVG_1MIN, LOADAVG_5MIN, and LOADAVG_15MIN indices, defined in
<sys/loadavg.h>, can be used to extract the data from the appropriate element of
the loadavg[] array.

If pset is PS_NONE, the load average for processes not assigned to a processor set is
returned.

If pset is PS_MYID, the load average for the processor set to which the caller is bound
is returned. If the caller is not bound to a processor set, the result is the same as if
PS_NONE was specified.

RETURN VALUES
Upon successful completion, the number of samples actually retrieved is returned. If
the load average was unobtainable or the processor set does not exist, −1 is returned
and errno is set to indicate the error.

ERRORS
The pset_getloadavg() function will fail if:

EINVAL
The number of elements specified is less than 0, or an invalid
processor set ID was specified.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Stable</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
uptime(1), w(1), psrset(1M), prstat(1M), pset_bind(2), pset_create(2),
kstat(3KSTAT), attributes(5)
psignal(3C)

NAME  psignal, psiginfo – system signal messages

SYNOPSIS  #include <<siginfo.h>

void psignal(int sig, const char *s);
void psiginfo(siginfo_t *pinfo, char *s);

DESCRIPTION  The psignal() and psiginfo() functions produce messages on the standard error output describing a signal. The sig argument is a signal that may have been passed as the first argument to a signal handler. The pinfo argument is a pointer to a siginfo structure that may have been passed as the second argument to an enhanced signal handler. See sigaction(2). The argument string s is printed first, followed by a colon and a blank, followed by the message and a NEWLINE character.

USAGE  If the application is linked with -l<lint>, then messages printed from these functions are in the native language specified by the LC_MESSAGES locale category. See setlocale(3C).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  sigaction(2), gettext(3C), perror(3C), setlocale(3C), attributes(5), siginfo(3HEAD), signal(3HEAD)
psignal(3UCB)

<table>
<thead>
<tr>
<th>NAME</th>
<th>psignal, sys_siglist – system signal messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNOPSIS</td>
<td>/usr/ucb/cc { flag ... } file ...</td>
</tr>
<tr>
<td></td>
<td>void psignal( sig, s );</td>
</tr>
<tr>
<td></td>
<td>unsigned sig;</td>
</tr>
<tr>
<td></td>
<td>char *s;</td>
</tr>
<tr>
<td></td>
<td>char *sys_siglist[];</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>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 &lt;signal.h&gt;.</td>
</tr>
<tr>
<td></td>
<td>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 &lt;signal.h&gt; 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.</td>
</tr>
<tr>
<td>SEE ALSO</td>
<td>perror(3C), signal(3C)</td>
</tr>
<tr>
<td>NOTES</td>
<td>Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.</td>
</tr>
</tbody>
</table>

Basic Library Functions 391
The `pthread_atfork()` function declares fork handlers to be called prior to and following `fork(2)`, within the thread that called `fork()`. The order of calls to `pthread_atfork()` is significant.

Before `fork()` processing begins, the `prepare` fork handler is called. The `prepare` handler is not called if its address is `NULL`.

The `parent` fork handler is called after `fork()` processing finishes in the parent process, and the `child` fork handler is called after `fork()` processing finishes in the child process. If the address of `parent` or `child` is `NULL`, then its handler is not called.

The `prepare` fork handler is called in LIFO (last-in-first-out) order, whereas the `parent` and `child` fork handlers are called in FIFO (first-in-first-out) order. This calling order allows applications to preserve locking order.

Upon successful completion, `pthread_atfork()` returns `0`. Otherwise, an error number is returned.

The `pthread_atfork()` function will fail if:

- `ENOMEM` Insufficient table space exists to record the fork handler addresses.

Solaris threads do not offer `pthread_atfork()` functionality, though a Solaris threads application can call this interface to ensure `fork1()`-safety, since the two thread APIs are interoperable. If the application is linked with `-lpthread`, `fork()` is defined to be the same as `fork1()`; otherwise `fork()` and `fork1()` behave differently. The `pthread_atfork()` function affects only `fork1()`. See `fork(2)`.

All multithreaded applications that call `fork()` in a POSIX threads program and do more than simply call `exec(2)` in the child of the fork need to ensure that the child is protected from deadlock.

Since the "fork-one" model results in duplicating only the thread that called `fork()`, it is possible that at the time of the call another thread in the parent owns a lock. This thread is not duplicated in the child, so no thread will unlock this lock in the child. Deadlock occurs if the single thread in the child needs this lock.

The problem is more serious with locks in libraries. Since a library writer does not know if the application using the library calls `fork()`, the library must protect itself from such a deadlock scenario. If the application that links with this library calls
EXAMPLE 1 make a library safe with respect to `fork()` (Continued)

`fork()` and does not call `exec()` in the child, and if it needs a library lock that may be held by some other thread in the parent that is inside the library at the time of the fork, the application deadlocks inside the library.

The following describes how to make a library safe with respect to `fork()` by using `pthread_atfork()`.

1. Identify all locks used by the library (for example `{L1, ..., Ln}`). Identify also the locking order for these locks (for example `{L1, ..., Ln}`, as well.)

2. Add a call to `pthread_atfork(f1, f2, f3)` in the library’s `init` section. `f1`, `f2`, `f3` are defined as follows:

```c
f1() {
    /* ordered in lock order */
    pthread_mutex_lock(L1);
    pthread_mutex_lock( ... );
    pthread_mutex_lock(Ln);
}

f2() {
    pthread_mutex_unlock(L1);
    pthread_mutex_unlock( ... );
    pthread_mutex_unlock(Ln);
}

f3() {
    pthread_mutex_unlock(L1);
    pthread_mutex_unlock( ... );
    pthread_mutex_unlock(Ln);
}
```

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO `exec(2)`, `fork(2)`, `atexit(3C)`, attributes(5), standards(5)
ptsname(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 a non-negative integer.

RETURN VALUES | Upon successful completion, the function ptsname() returns a pointer to a string which is the name of the pseudo-terminal slave device. This value points to a static data area that is overwritten by each call to ptsname(). Upon failure, ptsname() returns NULL. This could occur if fildes is an invalid file descriptor or if the slave device name does not exist in the file system.

ATTRIBUTES | See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO | open(2), grantpt(3C), ttyname(3C), unlockpt(3C), attributes(5)

STREAMS Programming Guide
NAME  
putenv – change or add value to environment

SYNOPSIS
#include <stdlib.h>

int putenv(char *string);

DESCRIPTION
The putenv() function makes the value of the environment variable name equal to value by altering an existing variable or creating a new one. In either case, the string pointed to by string becomes part of the environment, so altering the string will change the environment.

The string argument points to a string of the form name=value. The space used by string is no longer used once a new string-defining name is passed to putenv().

The putenv() function uses malloc(3C) to enlarge the environment.

After putenv() is called, environment variables are not in alphabetical order.

RETURN VALUES
The putenv() function returns a non-zero value if it was unable to obtain enough space using malloc(3C) for an expanded environment. Otherwise, 0 is returned.

ERRORS
The putenv() function may fail if:
ENOOMEM Insufficient memory was available.

USAGE
The putenv() function can be safely called from multithreaded programs. Caution must be exercised when using this function and getenv(3C) in multithreaded programs. These functions examine and modify the environment list, which is shared by all threads in a program. The system prevents the list from being accessed simultaneously by two different threads. It does not, however, prevent two threads from successively accessing the environment list using putenv() or getenv().

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO exec(2), getenv(3C), malloc(3C), attributes(5), environ(5)

WARNINGS
The string argument should not be an automatic variable. It should be declared static if it is declared within a function because it cannot be automatically declared. A potential error is to call putenv() with a pointer to an automatic variable as the argument and to then exit the calling function while string is still part of the environment.
The putpwent() function is the inverse of getpwent(). See getpwnam(3C). Given a pointer to a passwd structure created by getpwent(), getpwuid(), or getpwnam(), putpwent() writes a line on the stream f that matches the format of /etc/passwd.

RETURN VALUES
The putpwent() function returns a non-zero value if an error was detected during its operation. Otherwise, it returns 0.

USAGE
The putpwent() function is of limited utility, since most password files are maintained as Network Information Service (NIS) files that cannot be updated with this function. For this reason, the use of this function is discouraged. If used at all, it should be used with putspent(3C) to update the shadow file.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Unsafe</td>
</tr>
</tbody>
</table>

SEE ALSO
getpwnam(3C), putspent(3C), attributes(5)
puts, fputs – put a string on a stream

SYNOPSIS
#include <stdio.h>

int puts(const char *s);
int fputs(const char *s, FILE *stream);

DESCRIPTION
The puts() function writes the string pointed to by s, followed by a NEWLINE character, to the standard output stream stdout (see intro(3)). The terminating null byte is not written.

The fputs() function writes the null-terminated string pointed to by s to the named output stream. The terminating null byte is not written.

The st_ctime and st_mtime fields of the file will be marked for update between the successful execution of fputs() and the next successful completion of a call to fflush(3C) or fclose(3C) on the same stream or a call to exit(2) or abort(3C).

RETURN VALUES
On successful completion, both functions return the number of bytes written; otherwise they return EOF and set errno to indicate the error.

ERRORS
Refer to fputc(3C).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
exit(2), write(2), intro(3), abort(3C), fclose(3C), ferror(3C), fflush(3C), fopen(3C), fputc(3C), printf(3C), stdio(3C), attributes(5)
putspent(3C)

NAME      putspent – write shadow password file entry
SYNOPSIS  
```c
#include <shadow.h>

int putspent(const struct spwd *p, FILE *fp);
```
DESCRIPTION The putspent() function is the inverse of getspent(). See getspnam(3C). Given a pointer to a spwd structure created by getspent() or getspnam(), putspent() writes a line on the stream fp that matches the format of /etc/shadow.

The spwd structure contains the following members:
```c
char *sp_namp;
char *sp_pwdp;
long sp_lstchg;
long sp_min;
long sp_max;
long sp_warn;
long sp_inact;
long sp_expire;
unsigned long sp_flag;
```

If the sp_min, sp_max, sp_lstchg, sp_warn, sp_inact, or sp_expire member of the spwd structure is −1, or if sp_flag is 0, the corresponding /etc/shadow field is cleared.

RETURN VALUES The putspent() function returns a non-zero value if an error was detected during its operation. Otherwise, it returns 0.

USAGE Since this function is for internal use only, compatibility is not guaranteed. For this reason, its use is discouraged. If used at all, it should be used with putpwent(3C) to update the password file.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Unsafe</td>
</tr>
</tbody>
</table>

SEE ALSO getpwnam(3C), getspnam(3C), putpwent(3C), attributes(5)
putws – convert a string of Process Code characters to EUC characters

#include <stdio.h>
#include <widec.h>

int putws(wchar_t *s);

The putws() function converts the Process Code string (terminated by a
\(\text{wchar_t}\) NULL) pointed to by s, to an Extended Unix Code (EUC) string followed
by a NEWLINE character, and writes it to the standard output stream stdout. It does
not write the terminal null character.

The putws() function returns the number of Process Code characters transformed
and written. It returns EOF if it attempts to write to a file that has not been opened for
writing.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO ferror(3C), fopen(3C), fread(3C), getws(3C), printf(3C), putwc(3C), attributes(5)
qsort

NAME
qsort – quick sort

SYNOPSIS
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const void *));

DESCRIPTION
The qsort() function is an implementation of the quick-sort algorithm. It sorts a table of data in place. The contents of the table are sorted in ascending order according to the user-supplied comparison function.

The base argument points to the element at the base of the table. The nel argument is the number of elements in the table. The width argument specifies the size of each element in bytes. The compar argument is the name of the comparison function, which is called with two arguments that point to the elements being compared.

The function must return an integer less than, equal to, or greater than zero to indicate if the first argument is to be considered less than, equal to, or greater than the second argument.

The contents of the table are sorted in ascending order according to the user supplied comparison function.

EXAMPLES
EXAMPLE 1 Program sorts.

The following program sorts a simple array:

#include <stdlib.h>
#include <stdio.h>

static int
intcompare(const void *p1, const void *p2)
{
    int i = *((int *)p1);
    int j = *((int *)p2);

    if (i > j)
        return (1);
    if (i < j)
        return (-1);
    return (0);
}

int
main()
{
    int i;
    int a[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
    size_t nelems = sizeof (a) / sizeof (int);
    qsort((void *)a, nelems, sizeof (int), intcompare);

    for (i = 0; i < nelems; i++) {
        (void) printf("%d ", a[i]);
    }
}
EXAMPLE 1 Program sorts.  

```c
(void) printf("\n");
return (0);
}
```

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO sort(1), bsearch(3C), lsearch(3C), string(3C), attributes(5)

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

The relative order in the output of two items that compare as equal is unpredictable.
raise(3C)

NAME  raise – send signal to program

SYNOPSIS  
#include <signal.h>

int raise(int sig);

DESCRIPTION  The raise() function sends the signal sig to the executing program. It uses the
kill() function to send the signal to the executing program, as follows:

kill(getpid( ), sig);

See the kill(2) manual page for a detailed list of failure conditions and the
signal(3C) manual page for a list of signals.

RETURN VALUES  Upon successful completion, 0 is returned. Otherwise, −1 is returned and errno is set
to indicate the error.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  getpid(2), kill(2), signal(3C), attributes(5)
rand, srand, rand_r – simple random-number generator

#include <stdlib.h>

int rand(void);
void srand(unsigned int seed);
int rand_r(unsigned int *seed);

The rand() function uses a multiplicative congruential random-number generator
with period $2^{32}$ that returns successive pseudo-random numbers in the range of 0 to
RAND_MAX (defined in <stdlib.h>).

The srand() function uses the argument seed as a seed for a new sequence of
pseudo-random numbers to be returned by subsequent calls to rand(). If srand() is
then called with the same seed value, the sequence of pseudo-random numbers will be
repeated. If rand() is called before any calls to srand() have been made, the same
sequence will be generated as when srand() is first called with a seed value of 1.

The rand_r() function has the same functionality as rand() except that a pointer to
a seed seed must be supplied by the caller. The seed to be supplied is not the same seed
as in srand().

The spectral properties of rand() are limited. The drand48(3C) function provides a
better, more elaborate random-number generator.

The rand() is unsafe in multithreaded applications. The rand_r() function is
MT-Safe, and should be used instead. The srand() function is unsafe in
multithreaded applications.

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

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See USAGE above.</td>
</tr>
</tbody>
</table>

SEE ALSO drand48(3C), attributes(5)
NAME
rand, srand – simple random number generator

SYNOPSIS
/usr/ucb/cc [ flag ... ] file ...
int rand()
int srand( seed );
unsigned seed;

DESCRIPTION
rand() uses a multiplicative congruential 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.

SEE ALSO
drand48(3C), rand(3C), random(3C)

NOTES
Use of these interfaces should be restricted to only applications written on BSD
platforms. Use of these interfaces with any of the system libraries or in multi-thread
applications is unsupported.

The spectral properties of rand() leave a great deal to be desired. drand48(3C) and
random(3C) 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.
random(3C)

NAME
random, srandom, initstate, setstate – pseudorandom number functions

SYNOPSIS
#include <stdlib.h>

long random(void);
void srandom(unsigned int seed);
char *initstate(unsigned int seed, char *state, size_t size);
char *setstate(const char *state);

DESCRIPTION
The random() function uses a nonlinear additive feedback random-number generator
employing a default state array size of 31 long integers to return successive
pseudo-random numbers in the range from 0 to \(2^{31} - 1\). The period of this
random-number generator is approximately \(16 \times (2^{31} - 1)\). The size of the state array
determines the period of the random-number generator. Increasing the state array size
increases the period.

The srandom() function initializes the current state array using the value of seed.

The random() and srandom() functions have (almost) the same calling sequence
and initialization properties as rand() and srand() (see rand(3C)). The difference is
that rand(3C) produces a much less random sequence—in fact, the low dozen bits
generated by rand go through a cyclic pattern. All the bits generated by random() are
usable.

The algorithm from rand() is used by srandom() to generate the 31 state integers.
Because of this, different srandom() seeds often produce, within an offset, the same
sequence of low order bits from random(). If low order bits are used directly,
random() should be initialized with setstate() using high quality random values.

Unlike srand(), srandom() does not return the old seed because the amount of
state information used is much more than a single word. Two other routines are
provided to deal with restarting/changing random number generators. With 256 bytes
of state information, the period of the random-number generator is greater than \(2^{69}\),
which should be sufficient for most purposes.

Like rand(3C), random() produces by default a sequence of numbers that can be
duplicated by calling srandom() with 1 as the seed.

The initstate() and setstate() functions handle restarting and changing
random-number generators. The initstate() function allows a state array, pointed
to by the state argument, to be initialized for future use. The size argument, which
specifies the size in bytes of the state array, is used by initstate() to decide what
type of random-number generator to use; the larger the state array, the more random
the numbers. Values for the amount of state information are 8, 32, 64, 128, and 256
bytes. Other values greater than 8 bytes are rounded down to the nearest one of these
values. For values smaller than 8, random() uses a simple linear congruential random
number generator. The seed argument specifies a starting point for the random-number
sequence and provides for restarting at the same point. The initstate() function
returns a pointer to the previous state information array.
random(3C)

If initstate() has not been called, then random() behaves as though
initstate() had been called with seed = 1 and size = 128.

If initstate() is called with size < 8, then random() uses a simple linear
congruential random number generator.

Once a state has been initialized, setstate() allows switching between state arrays.
The array defined by the state argument is used for further random-number
generation until initstate() is called or setstate() is called again. The
setstate() function returns a pointer to the previous state array.

RETURN VALUES

The random() function returns the generated pseudo-random number.

The srandom() function returns no value.

Upon successful completion, initstate() and setstate() return a pointer to the
previous state array. Otherwise, a null pointer is returned.

ERRORS

No errors are defined.

USAGE

After initialization, a state array can be restarted at a different point in one of two
ways:
- The initstate() function can be used, with the desired seed, state array, and
  size of the array.
- The setstate() function, with the desired state, can be used, followed by
  srandom() with the desired seed. The advantage of using both of these functions
  is that the size of the state array does not have to be saved once it is initialized.

EXAMPLES

EXAMPLE 1 Initialize an array.

The following example demonstrates the use of initstate() to initialize an array. It
also demonstrates how to initialize an array and pass it to setstate().

```c
#include <stdlib.h>
static unsigned int state0[32];
static unsigned int state1[32] = {
  3,
  0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342, 0x7449e56b, 0xb11ee0b7, 0x2d436b86,
  0xda672e2a, 0xa1588ca88, 0xe369f35d, 0xc622c298, 0x1588ca88, 0xe369f35d, 0x904f35f7,
  0xda672e2a, 0xa1588ca88, 0xe369f35d, 0xc622c298, 0x1588ca88, 0xe369f35d, 0x904f35f7,
  0xda672e2a, 0xa1588ca88, 0xe369f35d, 0xc622c298, 0x1588ca88, 0xe369f35d, 0x904f35f7,
  0xda672e2a, 0xa1588ca88, 0xe369f35d, 0xc622c298, 0x1588ca88, 0xe369f35d, 0x904f35f7,
  0xda672e2a, 0xa1588ca88, 0xe369f35d, 0xc622c298, 0x1588ca88, 0xe369f35d, 0x904f35f7,
  0xda672e2a, 0xa1588ca88, 0xe369f35d, 0xc622c298, 0x1588ca88, 0xe369f35d, 0x904f35f7,
  0xda672e2a, 0xa1588ca88, 0xe369f35d, 0xc622c298, 0x1588ca88, 0xe369f35d, 0x904f35f7,
  0xda672e2a, 0xa1588ca88, 0xe369f35d, 0xc622c298, 0x1588ca88, 0xe369f35d, 0x904f35f7,
  0xda672e2a, 0xa1588ca88, 0xe369f35d, 0xc622c298, 0x1588ca88, 0xe369f35d, 0x904f35f7,
};
main() {
  unsigned seed;
  int n;
  seed = 1;
  n = 128;
  (void)initstate(seed, (char *)state0, n);
  printf("random() = %d0\n", random());
```
EXAMPLE 1 Initialize an array.  

(Continued)

```
", random();
    (void) setstate((char *) state1);
    printf("random() = %d\n", random());
    }
```

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See NOTES below.</td>
</tr>
</tbody>
</table>

SEE ALSO  drand48(3C), rand(3C), attributes(5)

NOTES  The random() and srand() functions are unsafe in multithreaded applications. Use of these functions in multithreaded applications is unsupported.

For initstate() and setstate(), the state argument must be aligned on an int boundary.

Newer and better performing random number generators such as addrans() and lcrans() are available with the SUNWpro package.
The resource control block routines allow the establishment or retrieval of values from a resource control block used to transfer information using the `getrctl(2)` and `setrctl(2)` functions. Each of the routines accesses or sets the resource control block member corresponding to its name. Certain of these members are read-only and do not possess set routines.

The firing time of a resource control block is 0 if the resource control action-value has not been exceeded for its lifetime on the process. Otherwise the firing time is the value of `gethrtime(3C)` at the moment the action on the resource control value was taken.

The global actions and flags are the action and flags set by `rctladm(1M)`. These values cannot be set with `setrctl()`. Valid global actions are listed in the table below. Global flags are generally a published property of the control and are not modifiable.

```
RCTL_GLOBAL_DENY_ALWAYS
    The action taken when a control value is exceeded on this control will always include denial of the resource.
```
The action taken when a control value is exceeded on this control will always exclude denial of the resource; the resource will always be granted, although other actions can also be taken.

RCTL_GLOBAL_CPU_TIME
The valid signals available as local actions include the SIGXCPU signal.

RCTL_GLOBAL_FILE_SIZE
The valid signals available as local actions include the SIGXFSZ signal.

RCTL_GLOBAL_INFINITE
This resource control supports the concept of an unlimited value; generally true only of accumulation-oriented resources, such as CPU time.

RCTL_GLOBAL_LOWERABLE
Non-privileged callers are able to lower the value of privileged resource control values on this control.

RCTL_GLOBAL_NOACTION
No global action will be taken when a resource control value is exceeded on this control.

RCTL_GLOBAL_NOBASIC
No values with the RCPRIV_BASIC privilege are permitted on this control.

RCTL_GLOBAL_NOLOCALACTION
No local actions are permitted on this control.

RCTL_Global_SYSLOG
A standard message will be logged by the syslog() facility when any resource control value on a sequence associated with this control is exceeded.

RCTL_GLOBAL_OBSERVABLE
The resource control (generally on a task- or project-related control) does not support observational control values. An RCPRIV_BASIC privileged control value placed by a process on the task or process will generate an action only if the value is exceeded by that process.

The local action and flags are those on the current resource control value represented by this resource control block. Valid actions and flags are listed in the table below. In the case of RCTL_LOCAL_SIGNAL, the second argument to rctlblk_set_local_action() contains the signal to be sent. Similarly, the signal to be sent is copied into the integer location specified by the second argument to rctlblk_get_local_action(). A restricted set of signals is made available for normal use by the resource control facility: SIGBART, SIGXRES, SIGHUP, SIGSTOP, SIGTERM, and Sigmill. Other signals are permitted due to global properties of a specific control. Calls to setrctl() with illegal signals will fail.

RCTL_LOCAL_DENY
When this resource control value is encountered, the request for the resource will be denied. Set on all values if RCTL_GLOBAL_DENY_ALWAYS is set for this control; cleared on all values if RCTL_GLOBAL_DENY_NEVER is set for this control.
This resource control value represents a request for the maximum amount of resource for this control. If RCTL_GLOBAL_INFINITE is set for this resource control, RCTL_LOCAL_MAXIMAL indicates an unlimited resource control value, one that will never be exceeded.

RCTL_LOCAL_NOACTION
No local action will be taken when this resource control value is exceeded.

RCTL_LOCAL_SIGNAL
The specified signal, sent by rctlblk_set_local_action(), will be sent to the process that placed this resource control value in the value sequence.

The rctlblk_get_recipient_pid() function returns the value of the process ID that placed the resource control value. This ID is set by the kernel by a caller invoking setrctl().

The rctlblk_get_privilege() function returns the privilege of the resource control block. Valid privileges are RCPRIV_BASIC, RCPRIV_PRIVILEGED, and RCPRIV_SYSTEM. System resource controls are read-only. Privileged resource controls require superuser privilege to write, unless the RCTL_GLOBAL_LOWERABLE global flag is set, in which case unprivileged applications can lower the value of a privileged control.

The rctlblk_get_value() and rctlblk_set_value() functions return or establish the enforced value associated with the resource control. In cases where the process, task, or project associated with the control possesses fewer capabilities than allowable by the current value, the value returned by rctlblk_get_enforced_value() will differ from that returned by rctlblk_get_value(). This capability difference arises with processes using an address space model smaller than the maximum address space model supported by the system.

The rctlblk_size() function returns the size of a resource control block for use in memory allocation. The rctlblk_t * type is an opaque pointer whose size is not connected with that of the resource control block itself. Use of rctlblk_size() is illustrated in the example below.

RETURN VALUES
The various set routines have no return values. Incorrectly composed resource control blocks will generate errors when used with setrctl(2) or getrctl(2).

ERRORS
No error values are returned. Incorrectly constructed resource control blocks will be rejected by the system calls.

EXAMPLES
EXAMPLE 1 Display the contents of a fetched resource control block.

The following example displays the contents of a fetched resource control block.

```c
#include <rctl.h>
#include <stdio.h>
#include <stdlib.h>
```
EXAMPLE 1 Display the contents of a fetched resource control block.

```c
rctlblk_t *rblk;
int rsignal;
int raction;

if ((rblk = malloc(rctlblk_size())) == NULL) {
    (void) perror("rblk malloc");
    exit(1);
}

if (getrctl("process.max-cpu-time", NULL, rblk, RCTL_FIRST) == -1) {
    (void) perror("getrctl");
    exit(1);
}

raction = rctlblk_get_local_action(rblk, &rsignal);
(void) printf("Resource control for %s
",
    "process.max-cpu-time");
(void) printf("Process ID: %d\n",
    rctlblk_get_recipient_pid(rblk));
(void) printf("Privilege: %x\n",
    rctlblk_get_privilege(rblk),
    rctlblk_get_global_flags(rblk),
    rctlblk_get_global_action(rblk),
    rctlblk_get_local_flags(rblk),
    rctlblk_get_enforced_value(rblk));
(void) printf("Local action: %s (%d)\n",
    raction, raction == RCTL_LOCAL_SIGNAL ? rsignal : 0);
(void) printf("Value: %llu\n",
    rctlblk_get_value(rblk));
(void) printf("Enforced value: %llu\n",
    rctlblk_get_enforced_value(rblk));
```

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

rctladm(1M), getrctl(2), setrctl(2), gethrtime(3C), attributes(5)
rctl_walk(3C)

NAME  rctl_walk – visit registered rctls on current system

SYNOPSIS

```
#include <rctl.h>

int rctl_walk(int (*callback)(const char *rctlname, void *walk_data),
               void *init_data);
```

DESCRIPTION

The `rctl_walk()` function provides a mechanism for the application author to examine all active resource controls (rctls) on the current system. The `callback` function provided by the application is given the name of an rctl at each invocation and can use the `walk_data` to record its own state. The callback function should return non-zero if it encounters an error condition or attempts to terminate the walk prematurely; otherwise the callback function should return 0.

RETURN VALUES

Upon successful completion, `rctl_walk()` returns 0. It returns −1 if the `callback` function returned a non-zero value or if the walk encountered an error, in which case `errno` is set to indicate the error.

ERRORS

The `rctl_walk()` function will fail if:

- **ENOMEM** There is insufficient memory available to set up the initial data for the walk.

Other returned error values are presumably caused by the `callback` function.

EXAMPLES

**EXAMPLE 1** Count the number of rctls available on the system.

The following example counts the number of resource controls on the system.

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

typedef struct wdata {
    uint_t count;
} wdata_t;

wdata_t total_count;

int simple_callback(const char *name, void *pvt)
{
    wdata_t *w = (wdata_t *)pvt;
    w->count++;
    return 0;
}
...

total_count.count = 0;
errno = 0;
if (rctl_walk(simple_callback, &total_count) == 0)
    (void) printf("count = %u\n", total_count.count);
```
ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO `setrctl(2), attributes(5)`
readdir(3C)

NAME readdir, readdir_r – read directory

SYNOPSIS

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

struct dirent *readdir(DIR *dirp);
struct dirent *readdir_r(DIR *dirp, struct dirent *entry);
```

POSIX

```c
cc [ flag ... ] file ... -D_POSIX_PTHREAD_SEMANTICS [ library ... ]

int readdir_r(DIR *dirp, struct dirent *entry, struct dirent **result);
```

DESCRIPTION

The type DIR, which is defined in the header `<dirent.h>`, represents a directory stream, which is an ordered sequence of all the directory entries in a particular directory. Directory entries represent files; files may be removed from a directory or added to a directory asynchronously to the operation of readdir() and readdir_r().

**readdir()**

The readdir() function returns a pointer to a structure representing the directory entry at the current position in the directory stream specified by the argument `dirp`, and positions the directory stream at the next entry. It returns a null pointer upon reaching the end of the directory stream. The structure dirent defined by the `<dirent.h>` header describes a directory entry.

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(3C) or rewinddir(3C), 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(2), either the parent or child (but not both) may continue processing the directory stream using readdir(), rewinddir() or seekdir(3C). If both the parent and child processes use these functions, the result is undefined.

If the entry names a symbolic link, the value of the d_ino member is unspecified.

**readdir_r()**

The readdir_r() function initializes the dirent structure referenced by entry to represent the directory entry at the current position in the directory stream referred to by `dirp`, and positions the directory stream at the next entry.
The caller must allocate storage pointed to by `entry` to be large enough for a `dirent` structure with an array of `char d_name` member containing at least `NAME_MAX` (that is, `pathconf(_PC_NAME_MAX)`) plus one elements. `_PC_NAME_MAX` is defined in `<unistd.h>`.

The `readdir_r()` function will not return directory entries containing empty names. It is unspecified whether entries are returned for `. (dot)` or `.. (dot-dot)`.

If a file is removed from or added to the directory after the most recent call to `opendir()` or `rebuilddir()`, 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.

The POSIX version (see `standards(5)`) of the `readdir_r()` function initializes the structure referenced by `entry` and stores a pointer to this structure in `result`. 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.

**RETURN VALUES**
Upon successful completion, `readdir()` and `readdir_r()` return a pointer to an object of type `struct dirent`. When an error is encountered, a null pointer is returned and `errno` is set to indicate the error. When the end of the directory is encountered, a null pointer is returned and `errno` is not changed. The POSIX `readdir_r()` returns 0 if successful or an error number to indicate failure.

**ERRORS**
The `readdir()` function will fail if:

`EOVERFLOW` One of the values in the structure to be returned cannot be represented correctly.

The `readdir()` and `readdir_r()` functions will fail if:

`EBADF` The file descriptor determined by the `DIR` stream is no longer valid. This results if the `DIR` stream has been closed.

`ENOENT` The current file pointer for the directory is not located at a valid entry.

The `readdir()` and `readdir_r()` functions may fail if:

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

`ENOENT` The current position of the directory stream is invalid.

**USAGE**
The `readdir()` function should be used in conjunction with `opendir()`, `closedir()`, and `rebuilddir()` to examine the contents of the directory. As `readdir()` returns a null pointer both at the end of the directory and on error, an application wishing to check for error situations should set `errno` to 0, then call `readdir()`, then check `errno` and if it is non-zero, assume an error has occurred.
Applications wishing to check for error situations should set \texttt{errno} to 0 before calling \texttt{readdir()}. If \texttt{errno} is set to non-zero on return, an error occurred.

The \texttt{readdir()} and \texttt{readdir_r()} functions have transitional interfaces for 64-bit file offsets. See \texttt{1f64(5)}.  

**EXAMPLE 1** Search the current directory for the entry \texttt{name}.

The following sample code will search the current directory for the entry \texttt{name}:

```c
dirp = opendir(".");
while (dirp) {
    errno = 0;
    if ((dp = readdir(dirp)) != NULL) {
        if (strcmp(dp->d_name, name) == 0) {
            closedir(dirp);
            return FOUND;
        } else {
            if (errno == 0) {
                closedir(dirp);
                return NOT_FOUND;
            }
            closedir(dirp);
            return READ_ERROR;
        }
    } else {
        if (errno == 0) {
            closedir(dirp);
            return NOT_FOUND;
        }
        closedir(dirp);
        return READ_ERROR;
    }
}
return OPEN_ERROR;
```

**ATTRIBUTES**  
See \texttt{attributes(5)} for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See \texttt{NOTES} below.</td>
</tr>
</tbody>
</table>

**NOTES**  
When compiling multithreaded programs, see \texttt{Intro(3)},  
\texttt{Notes On Multithreaded Applications}.

The \texttt{readdir()} function is unsafe in multithreaded applications. The \texttt{readdir_r()} function is safe, and should be used instead.

Solaris 2.4 and earlier releases provided a \texttt{readdir_r()} interface as specified in \texttt{POSIX.1c Draft 6}. The final \texttt{POSIX.1c} standard changed the interface as described above. Support for the Draft 6 interface is provided for compatibility only and may not be supported in future releases. New applications and libraries should use the \texttt{POSIX} standard interface.
For POSIX.1c-compliant applications, the _POSIX_PTHREAD_SEMANTICS and _REENTRANT flags are automatically turned on by defining the _POSIX_C_SOURCE flag with a value >= 199506L.

SEE ALSO
fork(2), lstat(2), symlink(2), Intro(3), closedir(3C), opendir(3C), rewinddir(3C), seekdir(3C), attributes(5), lfs(5), standards(5)
**readdir(3UCB)**

**NAME**  readdir – read a directory entry

**SYNOPSIS**

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

struct direct *readdir(dirp);

DIR *dirp;
```

**DESCRIPTION**
The `readdir()` function returns a pointer to a structure representing the directory entry at the current position in the directory stream to which `dirp` refers, and positions the directory stream at the next entry, except on read-only file systems. It returns a NULL pointer upon reaching the end of the directory stream, or upon detecting an invalid location in the directory. The `readdir()` function shall not return directory entries containing empty names. It is unspecified whether entries are returned for dot (.) or dot-dot (..). The pointer returned by `readdir()` points to data that may be overwritten by another call to `readdir()` on the same directory stream. This data shall not be overwritten by another call to `readdir()` on a different directory stream. The `readdir()` function may buffer several directory entries per actual read operation. The `readdir()` function marks for update the `st_atime` field of the directory each time the directory is actually read.

**RETURN VALUES**
The `readdir()` function returns NULL on failure and sets `errno` to indicate the error.

**ERRORS**
The `readdir()` function will fail if one or more of the following are true:

- **EAGAIN** Mandatory file/record locking was set, `O_NDELAY` or `O_NONBLOCK` was set, and there was a blocking record lock.
- **EAGAIN** Total amount of system memory available when reading using 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_NDELAY` or `O_NONBLOCK` was set.
- **EBADF** The file descriptor determined by the `DIR` stream is no longer valid. This results if the `DIR` stream has been closed.
- **EBADMSG** Message waiting to be read on a stream is not a data message.
- **EDEADLK** The `read()` was going to go to sleep and cause a deadlock to occur.
- **EFAULT** `buf` points to an illegal address.
- **EINTR** A signal was caught during the `read()` or `readv()` function.
- **EINVAL** Attempted to read from a stream linked to a multiplexor.
- **EIO** A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its
controlling terminal, and either the process is ignoring or blocking
the SIGTTIN signal or the process group of the process is
orphaned.

ENOENT The current file pointer for the directory is not located at a valid
entry.

ENOLCK The system record lock table was full, so the read() or readv()
could not go to sleep until the blocking record lock was removed.

ENOLINK fildes is on a remote machine and the link to that machine is no
longer active.

ENXIO The device associated with fildes is a block special or character
special file and the value of the file pointer is out of range.

EOVERFLOW The value of the direct structure member d_ino cannot be
represented in an ino_t.

The readdir() function has a transitional interface for 64-bit file offsets. See l64(5).

SEE ALSO getdents(2), readdir(3C), scandir(3UCB), l64(5)

NOTES Use of these interfaces should be restricted to only applications written on BSD
platforms. Use of these interfaces with any of the system libraries or in multi-thread
applications is unsupported.
realpath(3C)

NAME
realpath – resolve pathname

SYNOPSIS
#include <stdlib.h>

char *realpath(const char *file_name, char *resolved_name);

DESCRIPTION
The realpath() function derives, from the pathname pointed to by file_name, an
absolute pathname that names the same file, whose resolution does not involve ".", ". ..", or symbolic links. The generated pathname, using PATH_MAX bytes, is stored in
the buffer pointed to by resolved_name.

The realpath() function can handle both relative and absolute path names. For
absolute path names and the relative names whose resolved name cannot be expressed
relatively (for example, . . / . . /reldir), it returns the resolved absolute name.
For the other relative path names, it returns the resolved relative name.

RETURN VALUES
On successful completion, realpath() returns a pointer to the resolved name.
Otherwise, realpath() returns a null pointer and sets errno to indicate the error,
and the contents of the buffer pointed to by resolved_name are undefined.

ERRORS
The realpath() function will fail if:

EACCES Read or search permission was denied for a component
of file_name.

EINVAL Either the file_name or resolved_name argument is a null
pointer.

EIO An error occurred while reading from the file system.

ELOOP Too many symbolic links were encountered in
resolving 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.

USAGE
The realpath() function operates on null-terminated strings.

One should have execute permission on all the directories in the given and the
resolved path.
realpath(3C)

The `realpath()` function may fail to return to the current directory if an error occurs.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`getcwd(3C), sysconf(3C), attributes(5)`
NAME | reboot – reboot system or halt processor
SYNOPSIS | 
#include <sys/reboot.h>

int reboot(int howto, char *bootargs);

DESCRIPTION | The `reboot()` function reboots the system. The `howto` argument specifies the behavior of the system while rebooting and is a mask constructed by a bitwise-inclusive-OR of flags from the following list:

- **RE_AUTOBOOT**: The machine is rebooted from the root filesystem on the default boot device. This is the default behavior. See `boot(1M)` and `kernel(1M)`.

- **RB_HALT**: The processor is simply halted; no reboot takes place. This option should be used with caution.

- **RB_ASKNAME**: Interpreted by the bootstrap program and kernel, causing the user to be asked for pathnames during the bootstrap.

- **RB_DUMP**: The system is forced to panic immediately without any further processing and a crash dump is written to the dump device (see `dumpadm(1M)`) before rebooting.

Any other `howto` argument causes the kernel file to boot.

The interpretation of the `bootargs` argument is platform-dependent.

RETURN VALUES | Upon successful completion, `reboot()` never returns. Otherwise, -1 is returned and `errno` is set to indicate the error.

ERRORS | The `reboot()` function will fail if:

- **EPERM**: The caller is not the super-user.

USAGE | Only the super-user may `reboot()` a machine.

SEE ALSO | `intro(1M), boot(1M), dumpadm(1M), halt(1M), init(1M), kernel(1M), reboot(1M), uadmin(2)`
### NAME
re_comp, re_exec – compile and execute regular expressions

### SYNOPSIS
```c
#include <re_comp.h>

char *re_comp(const char *string);
int re_exec(const char *string);
```

### DESCRIPTION
The `re_comp()` function converts a regular expression string (RE) into an internal form suitable for pattern matching. The `re_exec()` function compares the string pointed to by the `string` argument with the last regular expression passed to `re_comp()`.

If `re_comp()` is called with a null pointer argument, the current regular expression remains unchanged.

Strings passed to both `re_comp()` and `re_exec()` must be terminated by a null byte, and may include NEWLINE characters.

The `re_comp()` and `re_exec()` functions support *simple regular expressions*, which are defined on the `regexp(5)` manual page. The regular expressions of the form `\{m\}`, `\{m,\}`, or `\{m,n\}` are not supported.

### RETURN VALUES
The `re_comp()` function returns a null pointer when the string pointed to by the `string` argument is successfully converted. Otherwise, a pointer to one of the following error message strings is returned:

- No previous regular expression
- Regular expression too long
- unmatched `\`
- missing `\`
- too many `\`
- unbalanced `\`

Upon successful completion, `re_exec()` returns 1 if `string` matches the last compiled regular expression. Otherwise, `re_exec()` returns 0 if `string` fails to match the last compiled regular expression, and -1 if the compiled regular expression is invalid (indicating an internal error).

### ERRORS
No errors are defined.

### USAGE
For portability to implementations conforming to X/Open standards prior to SUS, `regcomp(3C)` and `regexec(3C)` are preferred to these functions. See `standards(5)`.

### SEE ALSO
`grep(1), regcmp(1), regcomp(3C), regexec(3C), regexpr(3GEN), regexp(5), standards(5)`

---

**re_comp(3C)**

---

Basic Library Functions 423
The `regcmp()` function compiles a regular expression (consisting of the concatenated arguments) and returns a pointer to the compiled form. The `malloc(3C)` function is used to create space for the compiled form. It is the user’s responsibility to free unneeded space so allocated. A NULL return from `regcmp()` indicates an incorrect argument. `regcmp(1)` has been written to generally preclude the need for this routine at execution time.

The `regex()` function executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. The `regex()` function returns NULL on failure or a pointer to the next unmatched character on success. A global character pointer `__loc1` points to where the match began. The `regcmp()` and `regex()` functions were mostly borrowed from the editor `ed(1)`; however, the syntax and semantics have been changed slightly. The following are the valid symbols and associated meanings.

- `[]` This group of symbols retains its meaning as described on the `regexp(5)` manual page.
- `${m}$` Matches the end of the string; \`n` matches a newline.
- `−` Within brackets the minus means `through`. For example, `[a-z]` is equivalent to `[abcd . . .xyz]`. The `-` can appear as itself only if used as the first or last character. For example, the character class expression `[][−]` matches the characters ] and -.
- `+` A regular expression followed by + means one or more times. For example, `[0–9]+` is equivalent to `[0–9][0–9]∗`.
- `{m}`, `{m,u}` Integer values enclosed in `{}` indicate the number of times the preceding regular expression is to be applied. The value m is the minimum number and u is a number, less than 256, which is the maximum. If only m is present (that is, `{m}`), it indicates the exact number of times the regular expression is to be applied. The value `{m,u}` is analogous to `{m,infinity}`. The plus (+) and star (*) operations are equivalent to `{1,}` and `{0,}` respectively.
- `( ... )$n` The value of the enclosed regular expression is to be returned. The value will be stored in the (n+1)th argument following the subject argument. At most, ten enclosed regular expressions are allowed. The `regex()` function makes its assignments unconditionally.
Parentheses are used for grouping. An operator, for example, *, +, { }, can work on a single character or a regular expression enclosed in parentheses. For example, \( (a*(cb+))*$ \). By necessity, all the above defined symbols are special. They must, therefore, be escaped with a \ (backslash) to be used as themselves.

**EXAMPLE 1** Example matching a leading newline in the subject string.

The following example matches a leading newline in the subject string pointed at by cursor.

```c
char *cursor, *newcursor, *ptr;
... newcursor = regex((ptr = regcmp("\n", (char *)0)), cursor);
free(ptr);
```

The following example matches through the string Testing3 and returns the address of the character after the last matched character (the "4"). The string Testing3 is copied to the character array ret0.

```c
char ret0[9];
char *newcursor, *name;
... name = regcmp("([A-Za-z][A-Za-z0-9]{0,7})$", (char *)0);
newcursor = regex(name, "012Testing345", ret0);
```

The following example applies a precompiled regular expression in file.i (see `regcmp(1)` against `string`.

```c
#include "file.i"
char *string, *newcursor;
... newcursor = regex(name, string);
```

**FILES**

/usr/ccs/lib/libgen.a

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

ed(1), regcmp(1), malloc(3C), attributes(5), regexp(5)

**NOTES**

The user program may run out of memory if `regcmp()` is called iteratively without freeing the vectors no longer required.

When compiling multithreaded applications, the `_REENTRANT` flag must be defined on the compile line. This flag should only be used in multithreaded applications.
# NAME
regcomp, regexec, regerror, regfree – regular expression matching

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

int regcomp(regex_t *preg, const char *pattern, int flags);
int regexec(const regex_t *preg, const char *string, size_t nmatch,
            regmatch_t pmatch[], int flags);
size_t regerror(int ercode, const regex_t *preg, char *errbuf, size_t errbuf_size);
void regfree(regex_t *preg);
```

# DESCRIPTION
These functions interpret basic and extended regular expressions (described on the `regex(5)` manual page).

The structure type `regex_t` contains at least the following member:

```
size_t re_nsub                Number of parenthesised subexpressions.
```

The structure type `regmatch_t` contains at least the following members:

```
regoff_t rm_so    Byte offset from start of `string` to start of substring.
regoff_t rm_eo    Byte offset from start of `string` of the first character after
                 the end of substring.
```

## regcomp()

The `regcomp()` function will compile the regular expression contained in the string pointed to by the `pattern` argument and place the results in the structure pointed to by `preg`. The `flags` argument is the bitwise inclusive OR of zero or more of the following flags, which are defined in the header `<regex.h>`:

- **REG_EXTENDED**: Use Extended Regular Expressions.
- **REG_ICASE**: Ignore case in match.
- **REG_NOSUB**: Report only success/fail in `regexec()`.
- **REG_NEWLINE**: Change the handling of NEWLINE characters, as described in the text.

The default regular expression type for `pattern` is a Basic Regular Expression. The application can specify Extended Regular Expressions using the `REG_EXTENDED` `flags` flag.

If the `REG_NOSUB` flag was not set in `flags`, then `regcomp()` will set `re_nsub` to the number of parenthesised subexpressions (delimited by `\(` in basic regular expressions or `()` in extended regular expressions) found in `pattern`.

## regexec()

The `regexec()` function compares the null-terminated string specified by `string` with the compiled regular expression `preg` initialized by a previous call to `regcomp()`. The `flags` argument is the bitwise inclusive OR of zero or more of the following flags, which are defined in the header `<regex.h>`:
If nmatch is zero or REG_NOSUB was set in the flags argument to regcomp(), then regexec() will ignore the pmatch argument. Otherwise, the pmatch argument must point to an array with at least nmatch elements, and regexec() will fill in the elements of that array with offsets of the substrings of string that correspond to the parenthesised subexpressions of pattern: pmatch[i].rm_so will be the byte offset of the beginning and pmatch[i].rm_eo will be one greater than the byte offset of the end of substring i. (Subexpression i begins at the ith matched open parenthesis, counting from 1.) Offsets in pmatch[0] identify the substring that corresponds to the entire regular expression. Unused elements of pmatch up to pmatch[nmatch-1] will be filled with −1. If there are more than nmatch subexpressions in pattern (pattern itself counts as a subexpression), then regexec() will still do the match, but will record only the first nmatch substrings.

When matching a basic or extended regular expression, any given parenthesised subexpression of pattern might participate in the match of several different substrings of string, or it might not match any substring even though the pattern as a whole did match. The following rules are used to determine which substrings to report in pmatch when matching regular expressions:

1. If subexpression i in a regular expression is not contained within another subexpression, and it participated in the match several times, then the byte offsets in pmatch[i] will delimit the last such match.

2. If subexpression i is not contained within another subexpression, and it did not participate in an otherwise successful match, the byte offsets in pmatch[i] will be −1. A subexpression does not participate in the match when:

   * or \{\} appears immediately after the subexpression in a basic regular expression, or *, ?, or { } appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched zero times)

   or

   | is used in an extended regular expression to select this subexpression or another, and the other subexpression matched.

3. If subexpression i is contained within another subexpression j, and i is not contained within any other subexpression that is contained within j, and a match of subexpression j is reported in pmatch[j], then the match or
non-match of subexpression \( i \) reported in \( pmatch[j] \) 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 \( \text{regexec()} \) is called, the locale is different from when the regular expression was compiled, the result is undefined.

If \( \text{REG_NEWLINE} \) is not set in \( cflags \), then a NEWLINE character in \( \text{pattern or string} \) will be treated as an ordinary character. If \( \text{REG_NEWLINE} \) is set, then newline will be treated as an ordinary character except as follows:

1. A NEWLINE character in \( \text{string} \) will not be matched by a period outside a bracket expression or by any form of a non-matching list.

2. A circumflex (^) in \( \text{pattern} \), when used to specify expression anchoring will match the zero-length string immediately after a newline in \( \text{string} \), regardless of the setting of \( \text{REG_NOTBOL} \).

3. A dollar-sign ($) in \( \text{pattern} \), when used to specify expression anchoring, will match the zero-length string immediately before a newline in \( \text{string} \), regardless of the setting of \( \text{REG_NOTEOL} \).

\( \text{regfree()} \) function frees any memory allocated by \( \text{regcomp()} \) associated with \( \text{preg} \).

The following constants are defined as error return values:

- \( \text{REG_NOMATCH} \): The \( \text{regexec()} \) function failed to match.
- \( \text{REG_BADPAT} \): Invalid regular expression.
- \( \text{REG_ECOLLATE} \): Invalid collating element referenced.
- \( \text{REG_ECTYPE} \): Invalid character class type referenced.
- \( \text{REG_EESCAPE} \): Trailing \( \backslash \) in pattern.
- \( \text{REG_ESUBREG} \): Number in \( \backslash \)digit invalid or in error.
- \( \text{REG_EBRACK} \): \([ \] \) imbalance.
- \( \text{REG_ENOSYS} \): The function is not supported.
- \( \text{REG_EPAREN} \): \( \langle \rangle \) or \( () \) imbalance.
- \( \text{REG_EBRACE} \): \( \{ \} \) imbalance.
The `regerror()` function provides a mapping from error codes returned by `regcomp()` and `regexec()` to unspecified printable strings. It generates a string corresponding to the value of the `errcode` argument, which must be the last non-zero value returned by `regcomp()` or `regexec()` with the given value of `preg`. If `errcode` is not such a value, an error message indicating that the error code is invalid is returned.

If `preg` is a NULL pointer, but `errcode` is a value returned by a previous call to `regexec()` or `regcomp()`, the `regerror()` still generates an error string corresponding to the value of `errcode`.

If the `errbuf_size` argument is not zero, `regerror()` will place the generated string into the buffer of size `errbuf_size` bytes pointed to by `errbuf`. If the string (including the terminating NULL) cannot fit in the buffer, `regerror()` will truncate the string and null-terminate the result.

If `errbuf_size` is zero, `regerror()` ignores the `errbuf` argument, and returns the size of the buffer needed to hold the generated string.

If the `preg` argument to `regexec()` or `regfree()` is not a compiled regular expression returned by `regcomp()`, the result is undefined. A `preg` is no longer treated as a compiled regular expression after it is given to `regfree()`.

See `regex(5)` for BRE (Basic Regular Expression) Anchoring.

### RETURN VALUES

On successful completion, the `regcomp()` function returns 0. Otherwise, it returns an integer value indicating an error as described in `<regex.h>`, and the content of `preg` is undefined.

On successful completion, the `regexec()` function returns 0. Otherwise it returns `REG_NOMATCH` to indicate no match, or `REG_ENOSYS` to indicate that the function is not supported.

Upon successful completion, the `regerror()` function returns the number of bytes needed to hold the entire generated string. Otherwise, it returns 0 to indicate that the function is not implemented.

The `regfree()` function returns no value.

### ERRORS

No errors are defined.

### USAGE

An application could use:
to find out how big a buffer is needed for the generated string, malloc a buffer to hold the string, and then call regerror() again to get the string (see malloc(3C)). Alternately, it could allocate a fixed, static buffer that is big enough to hold most strings, and then use malloc() to allocate a larger buffer if it finds that this is too small.

**EXAMPLES**

**EXAMPLE 1** Example to match string against the extended regular expression in pattern.

```c
#include <regex.h>

/* Match string against the extended regular expression in
 * pattern, treating errors as no match.
 */
int
match(const char *string, char *pattern) {
    int status;
    regex_t re;
    if (regcomp(&re, pattern, REG_EXTENDED | REG_NOSUB) != 0) {
        return(0); /* report error */
    }
    status = regexec(&re, string, (size_t) 0, NULL, 0);
    regfree(&re);
    if (status != 0) {
        return(0); /* report error */
    }
    return(1);
}
```

The following demonstrates how the REG_NOTBOL flag could be used with regexec() to find all substrings in a line that match a pattern supplied by a user. (For simplicity of the example, very little error checking is done.)

```c
(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);
}
```

**ATTRIBUTES** See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>ATTRIBUTE TYPE</td>
<td>ATTRIBUTE VALUE</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO
fnmatch(3C), glob(3C), malloc(3C), setlocale(3C), attributes(5), regex(5)

NOTES
The `regcomp()` function can be used safely in a multithreaded application as long as `setlocale(3C)` is not being called to change the locale.
#include <stdio.h>

int remove(const char *path);

The `remove()` function causes the file or empty directory whose name is the string pointed to by `path` to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless the file is created anew.

For files, `remove()` is identical to `unlink()`. For directories, `remove()` is identical to `rmdir()`.

See `rmdir(2)` and `unlink(2)` for a detailed list of failure conditions.

Upon successful completion, `remove()` returns 0. Otherwise, it returns −1 and sets `errno` to indicate an error.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

`rmdir(2), unlink(2), attributes(5)`
rewind(3C)

NAME
rewind – reset file position indicator in a stream

SYNOPSIS
#include <stdio.h>

void rewind(FILE *stream);

DESCRIPTION
The call:
rewind(stream)
is equivalent to:

(void) fseek(stream, 0L, SEEK_SET)
except that rewind() also clears the error indicator.

RETURN VALUES
The rewind() function returns no value.

ERRORS
Refer to fseek(3C) with the exception of EINVAL which does not apply.

USAGE
Because rewind() does not return a value, an application wishing to detect errors
should clear errno, then call rewind(), and if errno is non-zero, assume an error
has occurred.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
fseek(3C), attributes(5)
rewinddir(3C)

NAME
rewinddir — reset position of directory stream to the beginning of a directory

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

void rewinddir(DIR *dirp);

DESCRIPTION
The rewinddir() function resets the position of the directory stream to which dirp
refers to the beginning of the directory. It also causes the directory stream to refer to
the current state of the corresponding directory, as a call to opendir(3C) would have
done. If dirp does not refer to a directory stream, the effect is undefined.

After a call to the fork(2) function, either the parent or child (but not both) may
continue processing the directory stream using readdir(3C), rewinddir() or
seekdir(3C). If both the parent and child processes use these functions, the result is
undefined.

RETURN VALUES
The rewinddir() function does not return a value.

ERRORS
No errors are defined.

USAGE
The rewinddir() function should be used in conjunction with opendir(),
readdir(), and closedir(3C) to examine the contents of the directory. This method
is recommended for portability.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
fork(2), closedir(3C), opendir(3C), readdir(3C), seekdir(3C), attributes(5)
NAME | scandir, alphasort – scan a directory

SYNOPSIS | 
```c
/usr/ucb/cc [ flag... ] file...
#include <sys/types.h>
#include <sys/dir.h>
int scandir(dirname, namelist, select, dcomp);
```

```c
char *dirname;
struct direct *(*namelist[]);
int (*select(.),(*dcomp))();
int alphasort(d1, d2);
struct direct **d1, **d2;
```

DESCRIPTION | The `scandir()` function reads the directory `dirname` and builds an array of pointers to directory entries using `malloc(3C)`. 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(3C)`, which sorts the completed array. If this pointer is `NULL`, the array is not sorted.

The `alphasort()` function sorts the array alphabetically.

RETURN VALUES | The `scandir()` function returns the number of entries in the array and a pointer to the array through the parameter `namelist`. The `scandir()` function returns `-1` if the directory cannot be opened for reading or if `malloc(3C)` cannot allocate enough memory to hold all the data structures.

The `alphasort()` function returns an integer greater than, equal to, or less than `0` if the directory entry name pointed to by `d1` is greater than, equal to, or less than the directory entry name pointed to by `d2`.

USAGE | The `scandir()` and `alphasort()` functions have transitional interfaces for 64-bit file offsets. See `lf64(5)`.

SEE ALSO | `getdents(2), malloc(3C), qsort(3C), readdir(3UCB), readdir(3C), lf64(5)`

NOTES | Use of these functions should be restricted to applications written on BSD platforms. Use of these functions with any of the system libraries or in multithreaded applications is unsupported.
NAME
scanf, fscanf, sscanf, vscanf, vfscanf, vsscanf – convert 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, ...);
#include <stdarg.h>
#include <stdio.h>
int vscanf (const char *format, va_list arg);
int vfscanf (FILE *stream, const char *format, va_list arg);
int vsscanf (const char *s, const char *format, va_list arg);

DESCRIPTION
The scanf() function reads from the standard input stream stdin.

The fscanf() function reads from the named input stream stream.

The sscanf() function reads from the string s.

The vscanf(), vfscanf(), and vsscanf() functions are equivalent to the
scanf(), fscanf(), and sscanf() functions, respectively, except that instead of
being called with a variable number of arguments, they are called with an argument
list as defined by the <stdarg.h> header (see stdarg(3HEAD)). These functions do
not invoke the va_end() macro. Applications using these functions should call
va_end(ap) afterwards to clean up.

Each function reads bytes, interprets them according to a format, and stores the results
in its arguments. Each expects, as arguments, a control string format described below,
and a set of pointer arguments indicating where the converted input should be stored.
The result is undefined if there are insufficient arguments for the format. If the format
is exhausted while arguments remain, the excess arguments are evaluated but are
otherwise ignored.

Conversions can be applied to the nth argument after the format in the argument list,
rather than to the next unused argument. In this case, the conversion character % (see
below) is replaced by the sequence %n$, where n is a decimal integer in the range [1,
NL_ARGMAX]. This feature provides for the definition of format strings that select
arguments in an order appropriate to specific languages. In format strings containing
the $n$ form of conversion specifications, it is unspecified whether numbered
arguments in the argument list can be referenced from the format string more than
once.

The format can contain either form of a conversion specification, that is, % or %n$,
but the two forms cannot normally be mixed within a single format string. The only
exception to this is that %% or %* can be mixed with the %n$ form.
The `scanf()` function in all its forms allows for detection of a language-dependent radix character in the input string. The radix character is defined in the program’s locale (category `LC_NUMERIC`). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.)

The format is a character string, beginning and ending in its initial shift state, if any, composed of zero or more directives. Each directive is composed of one of the following:

- one or more white-space characters (space, tab, newline, vertical-tab or form-feed characters);
- an ordinary character (neither `%` nor a white-space character); or
- a conversion specification.

Each conversion specification is introduced by the character `%` or the character sequence `%%`, after which the following appear in sequence:

- An optional assignment-suppressing character `*`.
- An optional non-zero decimal integer that specifies the maximum field width.
- An optional size modifier `h`, `l` (ell), or `ll` (ell ell) indicating the size of the receiving object. The conversion characters `d`, `i`, and `n` must be preceded by `h` if the corresponding argument is a pointer to `short int` rather than a pointer to `int`, by `l` (ell) if it is a pointer to `long int`, or by `ll` (ell ell) if it is a pointer to `long long int`. Similarly, the conversion characters `o`, `u`, and `x` must be preceded by `h` if the corresponding argument is a pointer to `unsigned short int` rather than a pointer to `unsigned int`, by `l` (ell) if it is a pointer to `unsigned long int`, or by `ll` (ell ell) if it is a pointer to `unsigned long long int`. The conversion characters `e`, `f`, and `g` must be preceded by `l` (ell) if the corresponding argument is a pointer to `double` rather than a pointer to `float`, or by `L` if it is a pointer to `long double`. Finally, the conversion characters `c`, `s`, and `[` must be preceded by `l` (ell) if the corresponding argument is a pointer to `wchar_t` rather than a pointer to a character type. If an `h`, `l` (ell), `ll` (ell ell), or `L` appears with any other conversion character, the behavior is undefined.
- A conversion character that specifies the type of conversion to be applied. The valid conversion characters are described below.

The `scanf()` functions execute each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space characters is executed by reading input until no more valid input can be read, or up to the first byte which is not a white-space character which remains unread.

A directive that is an ordinary character is executed as follows. The next byte is read from the input and compared with the byte that comprises the directive; if the comparison shows that they are not equivalent, the directive fails, and the differing and subsequent bytes remain unread.
A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion character. A conversion specification is executed in the following steps:

Input white-space characters (as specified by `isspace(3C)`) are skipped, unless the conversion specification includes a `[`, `c`, `C`, or `n` conversion character.

An item is read from the input, unless the conversion specification includes an `n` conversion character. An input item is defined as the longest sequence of input bytes (up to any specified maximum field width, which may be measured in characters or bytes dependent on the conversion character) which is an 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 0, the execution of the conversion specification fails; this condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a `%` conversion character, the input item (or, in the case of a `%n` conversion specification, the count of input bytes) is converted to a type appropriate to the conversion character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by an `*`, the result of the conversion is placed in the object pointed to by the first argument following the `format` argument that has not already received a conversion result if the conversion specification is introduced by `%`, or in the `n`th argument if introduced by the character sequence `%n$`. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion characters are valid:

- **d**: Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of `strtol(3C)` with the value 10 for the `base` argument. In the absence of a size modifier, the corresponding argument must be a pointer to `int`.

- **i**: Matches an optionally signed integer, whose format is the same as expected for the subject sequence of `strtol()` with 0 for the `base` argument. In the absence of a size modifier, the corresponding argument must be a pointer to `int`.

- **o**: Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of `strtoul(3C)` with the value 8 for the `base` argument. In the absence of a size modifier, the corresponding argument must be a pointer to `unsigned int`.

- **u**: Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of `strtoul()` with the value 10 for the `base` argument. In the absence of a size modifier, the corresponding argument must be a pointer to `unsigned int`.

- **x**: Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of `strtoul()` with the value 16.
for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to unsigned int.

**e, f, g**  Matches an optionally signed floating-point number, whose format is the same as expected for the subject sequence of `strtod(3C)`. In the absence of a size modifier, the corresponding argument must be a pointer to float.

If the `printf(3C)` 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 `scanf()` family of functions will recognize them as input.

**s**  Matches a sequence of bytes that are not white-space characters. The corresponding argument must be a pointer to the initial byte of an array of char, signed char, or unsigned char large enough to accept the sequence and a terminating null character code, which will be added automatically.

If an 1 (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character is converted to a wide-character as if by a call to the `mbtowc(3C)` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first character is converted. The corresponding argument must be a pointer to an array of `wchar_t` large enough to accept the sequence and the terminating null wide-character, which will be added automatically.

**[**  Matches a non-empty sequence of characters from a set of expected characters (the `scanset`). The normal skip over white-space characters is suppressed in this case. The corresponding argument must be a pointer to the initial byte of an array of char, signed char, or unsigned char large enough to accept the sequence and a terminating null byte, which will be added automatically.

If an 1 (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character in the sequence is converted to a wide-character as if by a call to the `mbtowc()` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first character is converted. The corresponding argument must be a pointer to an array of `wchar_t` large enough to accept the sequence and the terminating null wide-character, which will be added automatically.

The conversion specification includes all subsequent characters in the format string up to and including the matching right square bracket (`]`). The characters between the square brackets (the `scanlist`) comprise the scanset, unless the character after the left square bracket is a circumflex (`^`), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right square bracket. If the
conversion specification begins with [ ] or [^], the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise the first right square bracket is the one that ends the conversion specification. If a - is in the scanlist and is not the first character, nor the second where the first character is a ^, nor the last character, it indicates a range of characters to be matched.

c Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the conversion specification). The corresponding argument must be a pointer to the initial byte of an array of char, signed char, or unsigned char large enough to accept the sequence. No null byte is added. The normal skip over white-space characters is suppressed in this case.

If an l (ell) qualifier is present, the input is a sequence of characters that begins in the initial shift state. Each character in the sequence is converted to a wide-character as if by a call to the mbrtowc() function, with the conversion state described by an mbstate_t object initialized to zero before the first character is converted. The corresponding argument must be a pointer to an array of wchar_t large enough to accept the resulting sequence of wide-characters. No null wide-character is added.

p Matches the set of sequences that is the same as the set of sequences that is produced by the %p conversion of the corresponding printf(3C) functions. The corresponding argument must be a pointer to a pointer to void. If the input item is a value converted earlier during the same program execution, the pointer that results will compare equal to that value; otherwise the behavior of the %p conversion is undefined.

n No input is consumed. The corresponding argument must be a pointer to the integer into which is to be written the number of bytes read from the input so far by this call to the scanf() functions. Execution of a %n conversion specification does not increment the assignment count returned at the completion of execution of the function.

C Same as lc.
S Same as ls.
%
Matches a single %; no conversion or assignment occurs. The complete conversion specification must be %.

If a conversion specification is invalid, the behavior is undefined.

The conversion characters E, G, and X are also valid and behave the same as, respectively, e, g, and x.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any bytes matching the current conversion specification (except for %n) have been read (other than leading white-space characters, where permitted),
execution of the current conversion specification terminates with an input failure. Otherwise, unless execution of the current conversion specification is terminated with a matching failure, execution of the following conversion specification (if any) is terminated with an input failure.

Reaching the end of the string in `sscanf()` is equivalent to encountering end-of-file for `fscanf()`.

If conversion terminates on a conflicting input, the offending input is left unread in the input. Any trailing white space (including newline characters) is left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the `%n` conversion specification.

The `fscanf()` and `scanf()` functions may mark the `st_atime` field of the file associated with `stream` for update. The `st_atime` field will be marked for update by the first successful execution of `fgetc(3C), fgets(3C), fread(3C), fscanf( ), getc(3C), getchar(3C), gets(3C), or scanf( ) using stream` that returns data not supplied by a prior call to `ungetc(3C).

**RETURN VALUES**

Upon successful completion, these functions return the number of successfully matched and assigned input items; this number can be 0 in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF is returned. If a read error occurs the error indicator for the stream is set, EOF is returned, and `errno` is set to indicate the error.

**ERRORS**

For the conditions under which the `scanf()` functions will fail and may fail, refer to `fgetc(3C)` or `fgetwc(3C).`

In addition, `fscanf()` may fail if:

- **EILSEQ** Input byte sequence does not form a valid character.
- **EINVAL** There are insufficient arguments.

**USAGE**

If the application calling the `scanf()` functions has any objects of type `wint_t` or `wchar_t`, it must also include the header `<wchar.h>` to have these objects defined.

**EXAMPLES**

**EXAMPLE 1** The call:

```c
int i, n; float x; char name[50];

n = sscanf("%d%f%s", &i, &x, name)
```

with the input line:

```
25 54.32E-1 Hamster
```

will assign to `n` the value 3, to `i` the value 25, to `x` the value 5.432, and `name` will contain the string `Hamster`. 
EXAMPLE 2 The call:

```c
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign 56 to \( i \), 789.0 to \( x \), skip 0123, and place the string 56\0 in \( name \). The next call to `getchar(3C)` will return the character `a`.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO

`fgetc(3C), fgets(3C), fgetwc(3C), fread(3C), isspace(3C), printf(3C), setlocale(3C), stdarg(3HEAD), strtod(3C), strtol(3C), strtoul(3C), wcrtomb(3C), ungetc(3C), attributes(5)`
**NAME**
seekdir – set position of directory stream

**SYNOPSIS**
```c
#include <sys/types.h>
#include <dirent.h>

void seekdir(DIR *dirp, long int loc);
```

**DESCRIPTION**
The `seekdir()` function sets the position of the next `readdir(3C)` operation on the directory stream specified by `dirp` to the position specified by `loc`. The value of `loc` should have been returned from an earlier call to `telldir(3C)`. The new position reverts to the one associated with the directory stream when `telldir()` was performed.

If the value of `loc` was not obtained from an earlier call to `telldir()` or if a call to `rewinddir(3C)` occurred between the call to `telldir()` and the call to `seekdir()`, the results of subsequent calls to `readdir()` are unspecified.

**RETURN VALUES**
The `seekdir()` function returns no value.

**ERRORS**
No errors are defined.

**ATTRIBUTES**
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**
open(3C), readdir(3C), rewinddir(3C), telldir(3C), attributes(5)
NAME
select, FD_SET, FD_CLR, FD_ISSET, FD_ZERO – synchronous I/O multiplexing

SYNOPSIS
#include <sys/time.h>

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

void FD_SET(int fd, fd_set *fdset);

void FD_CLR(int fd, fd_set *fdset);

int FD_ISSET(int fd, fd_set *fdset);

void FD_ZERO(fd_set *fdset);

DESCRIPTION
The 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 readfs argument is not a null pointer, it points to an object of type fd_set that
on input specifies the file descriptors to be checked for being ready to read, and on
output indicates which file descriptors are ready to read.

If the writefs argument is not a null pointer, it points to an object of type fd_set that
on input specifies the file descriptors to be checked for being ready to write, and on
output indicates which file descriptors are ready to write.

If the errorfds argument is not a null pointer, it points to an object of type fd_set that
on input specifies the file descriptors to be checked for error conditions pending, and
on output indicates which file descriptors have error conditions pending.

On successful completion, the objects pointed to by the readfs, writefs, and errorfds
arguments are modified to indicate which file descriptors are ready for reading, ready
for writing, or have an error condition pending, respectively. 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.

If the readfs, writefs, and errorfds 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, writefs, and errorfds arguments are all null pointers
and the timeout argument is a null pointer, select() blocks until interrupted by a
signal.

File descriptors associated with regular files always select true for ready to read, ready
to write, and error conditions.

On failure, the objects pointed to by the readfs, writefs, and errorfds arguments are not
modified. If the timeout interval expires without the specified condition being true for
any of the specified file descriptors, the objects pointed to by the readfs, writefs, and
errorfds arguments have all bits set to 0.

A file descriptor for a socket that is listening for connections will indicate that it is
ready for reading, when connections are available. A file descriptor for a socket that is
connecting asynchronously will indicate that it is ready for writing, when a connection
has been established.

Selecting true for reading on a socket descriptor upon which a listen(3SOCKET) call
has been performed indicates that a subsequent accept(3SOCKET) call on that
descriptor will not block.

File descriptor masks of type fd_set can be initialized and tested with the macros
FD_CLR(), FD_ISSET(), FD_SET(), and FD_ZERO().

FD_CLR(fd, &fdset) Clears the bit for the file descriptor fd in the file
descrcriptor 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 VALUES
The FD_CLR(), FD_SET(), and FD_ZERO() macros return no value. The
FD_ISSET() macro returns a non-zero value if the bit for the file descriptor fd is set in
the file descriptor set pointed to by fdset, and 0 otherwise.
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 `select()` function will fail if:

- **EBADF** One or more of the file descriptor sets specified a file descriptor that is not a valid open file descriptor.
- **EINTR** The `select()` function was interrupted before any of the selected events occurred and before the timeout interval expired.
  
  If `SA_RESTART` has been set for the interrupting signal, it is implementation-dependent whether `select()` restarts or returns with EINTR.

- **EINVAL** An invalid timeout interval was specified.

- **EINVAL** The `nfds` argument is less than 0 or greater than `FD_SETSIZE`.

- **EINVAL** One of the specified file descriptors refers to a STREAM or multiplexer that is linked (directly or indirectly) downstream from a multiplexer.

- **EINVAL** A component of the pointed-to time limit is outside the acceptable range: `t_sec` must be between 0 and 10⁸, inclusive. `t_usec` must be greater than or equal to 0, and less than 10⁶.

**USAGE**

The `poll(2)` function is preferred over this function. It must be used when the number of file descriptors exceeds `FD_SETSIZE`.

The use of a timeout does not affect any pending timers set up by `alarm(2)`, `ualarm(3C)` or `setitimer(2)`.

On successful completion, the object pointed to by the `timeout` argument may be modified.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`alarm(2)`, `fcntl(2)`, `poll(2)`, `read(2)`, `setitimer(2)`, `write(2)`, `accept(3SOCKET)`, `listen(3SOCKET)`, `ualarm(3C)`, `attributes(5)`

**NOTES**

The default value for `FD_SETSIZE` (currently 1024) is larger than the default limit on the number of open files. To accommodate 32-bit applications that wish to use a larger number of open files with `select()`, it is possible to increase this size at compile time.
by providing a larger definition of FD_SETSIZE before the inclusion of any system-supplied header. The maximum supported size for FD_SETSIZE is 65536. The default value is already 65536 for 64-bit applications.
setbuf(3C)

NAME | setbuf, setvbuf – assign buffering to a stream

SYNOPSIS | #include <stdio.h>

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

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

DESCRIPTION | The setbuf() function may be used after the stream pointed to by stream (see intro(3)) is opened but before it is read or written. It causes the array pointed to by buf to be used instead of an automatically allocated buffer. If buf is the null pointer, input/output will be completely unbuffered. The constant BUFSIZ, defined in the <stdio.h> header, indicates the size of the array pointed to by buf.

The setvbuf() function may be used after a stream is opened but before it is read or written. The type argument determines how stream will be buffered. Legal values for type (defined in <stdio.h>) are:

_IOCBF Input/output to be fully buffered.

_IOFBF Output to be line buffered; the buffer will be flushed when a NEWLINE is written, the buffer is full, or input is requested.

_IONBF Input/output to be completely unbuffered.

If buf is not the null pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. The size argument specifies the size of the buffer to be used. If input/output is unbuffered, buf and size are ignored.

For a further discussion of buffering, see stdio(3C).

RETURN VALUES | If an illegal value for type is provided, setvbuf() returns a non-zero value. Otherwise, it returns 0.

USAGE | A common source of error is allocating buffer space as an “automatic” variable in a code block, and then failing to close the stream in the same block.

When using setbuf(), buf should always be sized using BUFSIZ. If the array pointed to by buf is larger than BUFSIZ, a portion of buf will not be used. If buf is smaller than BUFSIZ, other memory may be unexpectedly overwritten.

Parts of buf will be used for internal bookkeeping of the stream and, therefore, buf will contain less than size bytes when full. It is recommended that stdio(3C) be used to handle buffer allocation when using setvbuf().

ATTRIBUTES | See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

448 man pages section 3: Basic Library Functions • Last Revised 30 Dec 1996
SEE ALSO | fopen(3C), getc(3C), malloc(3C), putc(3C), stdio(3C), attributes(5)
The `setbuffer()` and `setlinebuf()` functions assign buffering to a stream. The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered, many characters are saved and written as a block; when it is line buffered, characters are saved until either a NEWLINE is encountered or input is read from stdin. The `fflush(3C)` function may be used to force the block out early. Normally all files are block buffered. A buffer is obtained from `malloc(3C)` upon the first `getc(3C)` or `putc(3C)` performed on the file. If the standard stream `stdout` refers to a terminal, it is line buffered. The standard stream `stderr` is unbuffered by default.

The `setbuffer()` function can be used after a stream `iop` has been opened but before it is read or written. It uses the character array `abuf` whose size is determined by the `asize` argument instead of an automatically allocated buffer. If `abuf` is the null pointer, input/output will be completely unbuffered. A manifest constant `BUFSIZ`, defined in the `<stdio.h>` header, tells how large an array is needed:

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

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

A stream can be changed from unbuffered or line buffered to block buffered by using `freopen(3C)`. A stream can be changed from block buffered or line buffered to unbuffered by using `freopen(3C)` followed by `setbuf(3C)` with a buffer argument of NULL.

The `setlinebuf()` function returns no useful value.

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.
setcat – define default catalog

**SYNOPSIS**

```c
#include <pfmt.h>

char *setcat(const char *catalog);
```

**DESCRIPTION**

The `setcat()` function defines the default message catalog to be used by subsequent calls to `gettext(3C)`, `lfmt(3C)`, or `pfmt(3C)` that do not explicitly specify a message catalog.

The `catalog` argument must be limited to 14 characters. These characters must be selected from a set of all characters values, excluding `\0` (null) and the ASCII codes for `/` (slash) and `:` (colon).

The `setcat()` function assumes that the catalog exists. No checking is done on the argument.

A null pointer passed as an argument will result in the return of a pointer to the current default message catalog name. A pointer to an empty string passed as an argument will cancel the default catalog.

If no default catalog is specified, or if `catalog` is an invalid catalog name, subsequent calls to `gettext(3C)`, `lfmt(3C)`, or `pfmt(3C)` that do not explicitly specify a catalog name will use `Message not found!!
` as default string.

**RETURN VALUES**

Upon successful completion, `setcat()` returns a pointer to the catalog name. Otherwise, it returns a null pointer.

**EXAMPLES**

**EXAMPLE 1** Example of `setcat()` function.

```c
setcat("test");
gettext("\:10", "hello world\n")
```

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`gettext(3C)`, `lfmt(3C)`, `pfmt(3C)`, `setlocale(3C)`, `attributes(5)`, `environ(5)`
setjmp(3C)

NAME

setjmp, sigsetjmp, longjmp, siglongjmp – non-local goto

SYNOPSIS

#include <setjmp.h>

int setjmp(jmp_buf env);

int sigsetjmp(sigjmp_buf env, int savemask);

void longjmp(jmp_buf env, int val);

void siglongjmp(sigjmp_buf env, int val);

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in a
low-level subroutine of a program.

The setjmp() function saves its stack environment in env for later use by
longjmp().

The sigsetjmp() function saves the calling process’s registers and stack
environment (see sigaltstack(2)) in env for later use by siglongjmp(). If savemask
is non-zero, the calling process’s signal mask (see sigprocmask(2)) and scheduling
parameters (see priocntl(2)) are also saved.

The longjmp() function restores the environment saved by the last call of setjmp()
with the corresponding env argument. After longjmp() completes, program
execution continues as if the corresponding call to setjmp() had just returned the
value val. The caller of setjmp() must not have returned in the interim. The
longjmp() function cannot cause setjmp() to return the value 0. If longjmp() is
invoked with a second argument of 0, setjmp() will return 1. At the time of the
second return from setjmp(), all external and static variables have values as of the
time longjmp() is called (see EXAMPLES).

The siglongjmp() function restores the environment saved by the last call of
sigsetjmp() with the corresponding env argument. After siglongjmp() completes, program
execution continues as if the corresponding call to sigsetjmp() had just returned the
value val. The siglongjmp() function cannot cause sigsetjmp() to return the value 0. If siglongjmp() is invoked with a second
argument of 0, sigsetjmp() will return 1. At the time of the second return from
sigsetjmp(), all external and static variables have values as of the time
siglongjmp() was called.

If a signal-catching function interrupts sleep(3C) and calls siglongjmp() to restore
an environment saved prior to the sleep() call, the action associated with SIGALRM
and time it is scheduled to be generated are unspecified. It is also unspecified whether
the SIGALRM signal is blocked, unless the process’s signal mask is restored as part of
the environment.

The siglongjmp() function restores the saved signal mask if and only if the env
argument was initialized by a call to the sigset jmp() function with a non-zero
savemask argument.
The values of register and automatic variables are undefined. Register or automatic
variables whose value must be relied upon must be declared as volatile.

**RETURN VALUES**
If the return is from a direct invocation, `setjmp()` and `sigsetjmp()` return 0. If the
return is from a call to `longjmp()`, `setjmp()` returns a non-zero value. If the return
is from a call to `siglongjmp()`, `sigsetjmp()` returns a non-zero value.

After `longjmp()` is completed, program execution continues as if the corresponding
invocation of `setjmp()` had just returned the value specified by `val`. The `longjmp()`
function cannot cause `setjmp()` to return 0; if `val` is 0, `setjmp()` returns 1.

After `siglongjmp()` is completed, program execution continues as if the
 corresponding invocation of `sigsetjmp()` had just returned the value specified by
`val`. The `siglongjmp()` function cannot cause `sigsetjmp()` to return 0; if `val` is 0,
`sigsetjmp()` returns 1.

**EXAMPLES**

**EXAMPLE 1** Example of `setjmp()` and `longjmp()` functions.

The following example uses both `setjmp()` and `longjmp()` to return the flow of
control to the appropriate instruction block:

```c
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>
#include <unistd.h>
jmp_buf env; static void signal_handler();

main( ) {
    int returned_from_longjump, processing = 1;
    unsigned int time_interval = 4;
    if ((returned_from_longjump = setjmp(env)) != 0) {
        switch (returned_from_longjump) {
            case SIGINT: printf("longjumped from interrupt \%d\n",SIGINT);
                break;
            case SIGALRM: printf("longjumped from alarm \%d\n",SIGALRM);
                break;
        }
    (void) signal(SIGINT, signal_handler);
    (void) signal(SIGALRM, signal_handler);
    alarm(time_interval);
    while (processing)
        { printf(" waiting for you to INTERRUPT (cntrl-C) ...\n\n");
            sleep(1);
            /* end while forever loop */
        }
    }
}

static void signal_handler(sig)
    int sig; { switch (sig) { case SIGINT: ... /* process for interrupt */
        longjmp(env,sig);
            /* break never reached */
        case SIGALRM: ... /* process for alarm */
    }}
```
EXAMPLE 1 Example of `setjmp()` and `longjmp()` functions. (Continued)

```c
longjmp(env,sig);
/* break never reached */
default: exit(sig);
}
```

When this example is compiled and executed, and the user sends an interrupt signal, the output will be:

`longjumped from interrupt`

Additionally, every 4 seconds the alarm will expire, signalling this process, and the output will be:

`longjumped from alarm`

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Unsafe</td>
</tr>
</tbody>
</table>

SEE ALSO `getcontext(2), priocntl(2), sigaction(2), sigaltstack(2), sigprocmask(2), signal(3C), attributes(5)`

WARNINGS

If `longjmp()` or `siglongjmp()` are called even though `env` was never primed by a call to `setjmp()` or `sigsetjmp()`, or when the last such call was in a function that has since returned, the results are undefined.
setjmp, longjmp, _setjmp, _longjmp – non-local goto

#include <setjmp.h>

int setjmp(env);
jmp_buf env;

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

int _setjmp(env);
jmp_buf env;

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

The setjmp() and longjmp() functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

The setjmp() function saves its stack environment in env for later use by longjmp(). 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).

The longjmp() function 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 setjmp() and longjmp() functions save and restore the signal mask while _setjmp() and _longjmp() manipulate only the C stack and registers.

None of these functions save or restore any floating-point status or control registers.
EXAMPLE 1 Examples of setjmp() and longjmp().

The following example uses both setjmp() and longjmp() to return the flow of control to the appropriate instruction block:

```c
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>
#include <unistd.h>
jmp_buf env; static void signal_handler();
main( ) {
    int returned_from_longjump, processing = 1;
    unsigned int time_interval = 4;
    if ((returned_from_longjump = setjmp(env)) != 0)
        switch (returned_from_longjump) {
            case SIGINT:
                printf("longjumped from interrupt \%d\n",SIGINT);
                break;
            case SIGALRM:
                printf("longjumped from alarm \%d\n",SIGALRM);
                break;
        }
    (void) signal(SIGINT, signal_handler);
    (void) signal(SIGALRM, signal_handler);
    alarm(time_interval);
    while (processing) {
        printf(" waiting for you to INTERRUPT (cntrl-C) ...\n");
        sleep(1);
    } /* end while forever loop */
}
static void signal_handler(sig)
    int sig;
{
    switch (sig) {
        case SIGINT: ... /* process for interrupt */
            longjmp(env,sig);
            /* break never reached */
        case SIGALRM: ... /* process for alarm */
            longjmp(env,sig);
            /* break never reached */
        default: exit(sig);
    }
}
```

When this example is compiled and executed, and the user sends an interrupt signal, the output will be:

longjumped from interrupt

Additionally, every 4 seconds the alarm will expire, signalling this process, and the output will be:

longjumped from alarm

SEE ALSO cc(1B), sigvec(3UCB), set jmp(3C), signal(3C)
### NOTES
Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

### BUGS
The `setjmp()` function 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(1B)`), the compiler will not attempt to assign variables to registers in routines that call `setjmp()`.

The `longjmp()` function never causes `setjmp()` to return 0, so programmers should not depend on `longjmp()` being able to cause `setjmp()` to return 0.
setkey(3C)

NAME  setkey – set encoding key

SYNOPSIS  #include <stdlib.h>
            void setkey(const char *key);

DESCRIPTION  The setkey() function provides (rather primitive) access to the hashing algorithm employed by the crypt(3C) function. The argument of setkey() is an array of length 64 bytes containing only the bytes with numerical value of 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored; this gives a 56-bit key which is used by the algorithm. This is the key that will be used with the algorithm to encode a string block passed to encrypt(3C).

RETURN VALUES  No values are returned.

ERRORS  The setkey() function will fail if:

        ENOSYS  The functionality is not supported on this implementation.

USAGE  In some environments, decoding may not be implemented. This is related to U.S. Government restrictions on encryption and decryption routines: the DES decryption algorithm cannot be exported outside the U.S.A. Historical practice has been to ship a different version of the encryption library without the decryption feature in the routines supplied. Thus the exported version of encrypt() does encoding but not decoding.

        Because setkey() does not return a value, applications wishing to check for errors should set errno to 0, call setkey(), then test errno and, if it is non-zero, assume an error has occurred.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  crypt(3C), encrypt(3C), attributes(5)
NAME

setlabel – define the label for pfmt() and lfmt()

SYNOPSIS

#include <pfmt.h>

int setlabel(const char *label);

DESCRIPTION

The setlabel() function defines the label for messages produced in standard format by subsequent calls to lfmt(3C) and pfmt(3C).

The label argument is a character string no more than 25 characters in length.

No label is defined before setlabel() is called. The label should be set once at the beginning of a utility and remain constant. A null pointer or an empty string passed as argument will reset the definition of the label.

RETURN VALUE

Upon successful completion, setlabel() returns 0; otherwise, it returns a non-zero value.

EXAMPLES

The following code (without previous call to setlabel()):

pfmt(stderr, MM_ERROR, "test:2:Cannot open file\n");
setlabel("UX:test");
pfmt(stderr, MM_ERROR, "test:2:Cannot open file\n");

will produce the following output:

ERROR: Cannot open file
UX:test: ERROR: Cannot open file

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

g getopt(3C), lfmt(3C), pfmt(3C), attributes(5)
setlocale(3C)

NAME
setlocale – modify and query a program’s locale

SYNOPSIS
#include <locale.h>

char *setlocale(int category, const char *locale);

DESCRIPTION
The setlocale() function selects the appropriate piece of the program’s locale as specified by the category and locale arguments. The category argument may have the following values: LC_CTYPE, LC_NUMERIC, LC_TIME, LC_COLLATE, LC_MONETARY, LC_MESSAGES, and LC_ALL. These names are defined in the <locale.h> header. The LC_ALL variable names all of a program’s locale categories.

The LC_CTYPE variable affects the behavior of character handling functions such as isdigit(3C) and tolower(3C), and multibyte character functions such as mbtowc(3C) and wctomb(3C).

The LC_NUMERIC variable affects the decimal point character and thousands separator character for the formatted input/output functions and string conversion functions.

The LC_TIME variable affects the date and time format as delivered by asctime(3C) cftime(3C) getdate(3C) strftime(3C) and strptime(3C).

The LC_COLLATE variable affects the sort order produced by collating functions such as strcoll(3C) and strxfrm(3C).

The LC_MONETARY variable affects the monetary formatted information returned by localeconv(3C).

The LC_MESSAGES variable affects the behavior of messaging functions such as dgettext(3C), gettext(3C), and gettxt(3C).

A value of "C" for locale specifies the traditional UNIX system behavior. At program startup, the equivalent of

setlocale(LC_ALL, "C")

is executed. This has the effect of initializing each category to the locale described by the environment "C".

A value of "" for locale specifies that the locale should be taken from environment variables. The order in which the environment variables are checked for the various categories is given below:

<table>
<thead>
<tr>
<th>Category</th>
<th>1st Env Var</th>
<th>2nd Env Var</th>
<th>3rd Env Var</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_CTYPE:</td>
<td>LC_ALL</td>
<td>LC_CTYPE</td>
<td>LANG</td>
</tr>
<tr>
<td>LC_COLLATE:</td>
<td>LC_ALL</td>
<td>LC_COLLATE</td>
<td>LANG</td>
</tr>
</tbody>
</table>
If a pointer to a string is given for locale, `setlocale()` attempts to set the locale for the given category to `locale`. If `setlocale()` succeeds, `locale` is returned. If `setlocale()` fails, a null pointer is returned and the program's locale is not changed.

For category `LC_ALL`, the behavior is slightly different. If a pointer to a string is given for `locale` and `LC_ALL` is given for category, `setlocale()` attempts to set the locale for all the categories to `locale`. The `locale` may be a simple locale, consisting of a single locale, or a composite locale. If the locales for all the categories are the same after all the attempted locale changes, `setlocale()` will return a pointer to the common simple locale. If there is a mixture of locales among the categories, `setlocale()` will return a composite locale.

**RETURN VALUES**

Upon successful completion, `setlocale()` returns the string associated with the specified category for the new locale. Otherwise, `setlocale()` returns a null pointer and the program's locale is not changed.

A null pointer for `locale` causes `setlocale()` to return a pointer to the string associated with the category for the program's current locale. The program's locale is not changed.

The string returned by `setlocale()` is such that a subsequent call with that string and its associated category will restore that part of the program's locale. The string returned must not be modified by the program, but may be overwritten by a subsequent call to `setlocale()`.

**ERRORS**

No errors are defined.

**FILES**

`/usr/lib/locale/locale` locale database directory for `locale`

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

---

```
setlocale(3C)

<table>
<thead>
<tr>
<th>Category</th>
<th>1st Env Var</th>
<th>2nd Env Var</th>
<th>3rd Env Var</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_CTIME</td>
<td>LC_ALL</td>
<td>LC_CTIME</td>
<td>LANG</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>LC_ALL</td>
<td>LC_NUMERIC</td>
<td>LANG</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>LC_ALL</td>
<td>LC_MONETARY</td>
<td>LANG</td>
</tr>
<tr>
<td>LC_MESSAGES</td>
<td>LC_ALL</td>
<td>LC_MESSAGES</td>
<td>LANG</td>
</tr>
</tbody>
</table>
```
To change locale in a multithreaded application, `setlocale()` should be called prior to using any locale-sensitive routine. Using `setlocale()` to query the current locale is safe and can be used anywhere in a multithreaded application.

It is the user's responsibility to ensure that mixed locale categories are compatible. For example, setting `LC_CTYPE=C` and `LC_TIME=ja` (where `ja` indicates Japanese) will not work, because Japanese time cannot be represented in the "C" locale's ASCII codeset.

Internationalization functions by `setlocale()` are supported only when the dynamic linking version of `libc` has been linked with the application. If the static linking version of `libc` has been linked with the application, `setlocale()` can handle only C and POSIX locales.
NAME  sigblock, sigmask, sigpause, sigsetmask – block signals

SYNOPSIS  

```
/usr/cc [ <flag> ... ] file ...
#include <signal.h>

int sigblock(mask);
int mask;

int sigmask(signum);
int signum;

int sigpause(int mask);
int mask;

int sigsetmask(mask);
int mask;
```

DESCRIPTION  sigblock, sigmask, sigpause, sigsetmask – block signals

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. sigblock() returns the previous mask. The previous mask may be restored using sigsetmask().

sigpause() assigns mask to the set of masked signals and then waits for a signal to arrive; on return the set of masked signals is restored. mask is usually 0 to indicate that no signals are now to be blocked. sigpause() always terminates by being interrupted, returning −1 and setting errno to EINTR.

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.

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().

It is not possible to block SIGKILL, SIGSTOP, or SIGCONT, this restriction is silently imposed by the system.

RETURN VALUES  sigblock() and sigsetmask() return the previous set of masked signals.
sigpause() returns −1 and sets errno to EINTR.

SEE ALSO  kill(2), sigaction(2), signal(3C), sigvec(3C)

NOTES  Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.
### NAME

sigfpe – signal handling for specific SIGFPE codes

### SYNOPSIS

```c
#include <floatingpoint.h>
#include <siginfo.h>

sigfpe_handler_type sigfpe(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, dumping core using `abort(3C)`, or default handling respectively. Default handling is to dump core using `abort(3C)`.

*code* is usually one of the five IEEE 754-related SIGFPE codes:

- `FPE_FLTRES` - floating-point inexact result
- `FPE_FLTDIV` - floating-point division by zero
- `FPE_FLTUND` - floating-point underflow
- `FPE_FLTOVF` - floating-point overflow
- `FPE_FLTINV` - floating-point invalid operation

Three steps are required to intercept an IEEE 754-related SIGFPE code with `sigfpe()`:

1. Set up a handler with `sigfpe()`.
2. Enable the relevant IEEE 754 trapping capability in the hardware, perhaps by using assembly-language instructions.
3. Perform a floating-point operation that generates the intended IEEE 754 exception.

`sigfpe()` never changes floating-point hardware mode bits affecting IEEE 754 trapping. No IEEE 754-related SIGFPE signals will be generated unless those hardware mode bits are enabled.

SIGFPE signals can be handled using `sigfpe()`, `sigaction(2)` or `signal(3C)`. In a particular program, to avoid confusion, use only one of these interfaces to handle SIGFPE signals.

### EXAMPLES

#### EXAMPLE 1 Example Of A User-Specified Signal Handler

A user-specified signal handler might look like this:

```c
#include <floatingpoint.h>
#include <siginfo.h>
#include <ucontext.h>

/*
 * The sample_handler prints out a message then commits suicide.
 */
void
sample_handler(int sig, siginfo_t *sip, ucontext_t *uap) {
    char *label;
    switch (sip->si_code) {
```
EXAMPLE 1 Example Of A User-Specified Signal Handler  (Continued)

```c
    case FPE_FLTINV: label = "invalid operand"; break;
    case FPE_FLTRES: label = "inexact"; break;
    case FPE_FLTDIV: label = "division-by-zero"; break;
    case FPE_FLTUND: label = "underflow"; break;
    case FPE_FLTOVF: label = "overflow"; break;
    default: label = "???"; break;
}
    fprintf(stderr, "FP exception %s (0x%x) occurred at address %p.
        " label, sip->si_code, (void *) sip->si_addr);
    abort();
}
and it might be set up like this:

#include <floatingpoint.h>
#include <siginfo.h>
#include <ucontext.h>
extern void sample_handler(int, siginfo_t *, ucontext_t *);
main(void) {
    sigfpe_handler_type hdl, old_handler1, old_handler2;
    /*
    * save current fp_overflow and fp_invalid handlers; set the new
    * fp_overflow handler to sample_handler( ) and set the new
    * fp_invalid handler to SIGFPE_ABORT (abort on invalid)
    */
    hdl = (sigfpe_handler_type) sample_handler;
    old_handler1 = sigfpe(FPE_FLTOVF, hdl);
    old_handler2 = sigfpe(FPE_FLTINV, SIGFPE_ABORT);
    ...
    /* restore old fp_overflow and fp_invalid handlers
    */
    sigfpe(FPE_FLTOVF, old_handler1);
    sigfpe(FPE_FLTINV, old_handler2);
}
```

FILES
/usr/include/floatingpoint.h
/usr/include/siginfo.h

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
sigaction(2), abort(3C), signal(3C), attributes(5), floatingpoint(3HEAD)

DIAGNOSTICS
sigfpe() returns BADSIG if code is not zero or a defined SIGFPE code.
**NAME**
siginterrupt – allow signals to interrupt functions

**SYNOPSIS**
```
/usr/ucb/cc [ flag ... ] file ...
```
```
int siginterrupt( sig, flag );
```

```c
type sig, flag;
```

**DESCRIPTION**
siginterrupt() is used to change the function restart behavior when a function is interrupted by the specified signal. If the flag is false (0), then functions 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(3C)` routine is used.

If the flag is true, (1), then restarting of functions is disabled. If a function is interrupted by the specified signal and no data has been transferred, the function will return −1 with `errno` set to EINTR. Interrupted functions 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.

**NOTES**
Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-threaded applications is unsupported.

This library routine uses an extension of the `sigvec(3UCB)` function that is not available in 4.2 BSD, hence it should not be used if backward compatibility is needed.

**RETURN VALUES**
A 0 value indicates that the call succeeded. A −1 value indicates that the call failed and `errno` is set to indicate the error.

**ERRORS**
siginterrupt() may return the following error:

EINVAL     `sig` is not a valid signal.

**SEE ALSO**
sigblock(3UCB), sigvec(3UCB), signal(3C)
NAME | signal, sigset, sighold, sigrelse, sigignore, sigpause – simplified signal management for application processes

SYNOPSIS | #include <signal.h>

void (*signal (int sig, void (*disp)(int))(int);
void (*sigset(int sig, void (*disp)(int))(int);
int sighold(int sig);
int sigrelse(int sig);
int sigignore(int sig);
int sigpause(int sig);

DESCRIPTION | These functions provide simplified signal management for application processes. See signal(3HEAD) for an explanation of general signal concepts.

The signal() and sigset() functions modify signal dispositions. The sig argument specifies the signal, which may be any signal except SIGKILL and SIGSTOP. The disp argument specifies the signal’s disposition, which may be SIG_DFL, SIG_IGN, or the address of a signal handler. If signal() is used, disp is the address of a signal handler, and sig is not SIGILL, SIGTRAP, or SIGPWR, the system first sets the signal’s disposition to SIG_DFL before executing the signal handler. If sigset() is used and disp is the address of a signal handler, the system adds sig to the calling process’s signal mask before executing the signal handler; when the signal handler returns, the system restores the calling process’s signal mask to its state prior to the delivery of the signal. In addition, if sigset() is used and disp is equal to SIG_HOLD, sig is added to the calling process’s signal mask and the signal’s disposition remains unchanged.

The sighold() function adds sig to the calling process’s signal mask.

The sigrelse() function removes sig from the calling process’s signal mask.

The sigignore() function sets the disposition of sig to SIG_IGN.

The sigpause() function removes sig from the calling process’s signal mask and suspends the calling process until a signal is received.

RETURN VALUES | Upon successful completion, signal() returns the signal’s previous disposition. Otherwise, it returns SIG_ERR and sets errno to indicate the error.

Upon successful completion, sigset() returns SIG_HOLD if the signal had been blocked or the signal’s previous disposition if it had not been blocked. Otherwise, it returns SIG_ERR and sets errno to indicate the error.

Upon successful completion, sighold(), sigrelse(), sigignore(), and sigpause(), return 0. Otherwise, they return −1 and set errno to indicate the error.

ERRORS | These functions fail if:
EINTR A signal was caught during the execution of sigpause().

EINVAL The value of the sig argument is not a valid signal or is equal to SIGKILL or SIGSTOP.

**USAGE**

The sighold() function used in conjunction with sigrelse() or sigpause() may be used to establish critical regions of code that require the delivery of a signal to be temporarily deferred.

If signal() or sigset() is used to set SIGCHLD’s disposition to a signal handler, SIGCHLD will not be sent when the calling process’s children are stopped or continued.

If any of the above functions are used to set SIGCHLD’s disposition to SIG_IGN, the calling process’s child processes will not create zombie processes when they terminate (see exit(2)). If the calling process subsequently waits for its children, it blocks until all of its children terminate; it then returns −1 with errno set to ECHILD (see wait(2) and waitid(2)).

The system guarantees that if more than one instance of the same signal is generated to a process, at least one signal will be received. It does not guarantee the reception of every generated signal.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

exit(2), kill(2), pause(2), sigaction(2), sigsend(2), wait(2), waitid(2), signal(3HEAD), attributes(5)
signal() is a simplified interface to the more general sigvec(3UCB) 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 (bus error, etc.), 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. See sigvec(3UCB) for a complete list of the signals.

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 • or a dagger. Signals marked with • are discarded if the action is SIG_DFL; signals marked with a dagger 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 functions, terminating the call prematurely, the call is automatically restarted. In particular this can occur during a read(2) or write(2) on a slow device (such as a terminal; but not a file) and during a wait(2).

The value of signal() is the previous (or initial) value of func for the particular signal.

After a fork(2) or vfork(2) the child inherits all signals. An exec(2) resets all caught signals to the default action; ignored signals remain ignored.

The previous action is returned on a successful call. Otherwise, −1 is returned and errno is set to indicate the error.

signal() will fail and no action will take place if the following occurs:
EINVAL  

sig is not a valid signal number, or is SIGKILL or SIGSTOP.

SEE ALSO  

kill(1), exec(2), fcntl(2), fork(2), getitimer(2), getrlimit(2), kill(2),
ptrace(2), read(2), sigaction(2), wait(2), write(2), abort(3C), setjmp(3UCB),
sigblock(3UCB), sigstack(3UCB), sigvec(3UCB), wait(3UCB), setjmp(3C),
signal(3C), signal(3HEAD), termio(7I)

NOTES  

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-threaded applications is unsupported.

The handler routine, func, can be declared:

void handler( signum) int signum; Here signum is the signal number. See
sigvec(3UCB) for more details.
NAME

sigsetops, sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – manipulate sets of signals

SYNOPSIS

#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(sigset_t *set, int signo);

DESCRIPTION

These functions manipulate sigset_t data types, representing the set of signals supported by the implementation.

The sigemptyset() function initializes the set pointed to by set to exclude all signals defined by the system.

The sigfillset() function initializes the set pointed to by set to include all signals defined by the system.

The sigaddset() function adds the individual signal specified by the value of signo to the set pointed to by set.

The sigdelset() function deletes the individual signal specified by the value of signo from the set pointed to by set.

The sigismember() function checks whether the signal specified by the value of signo is a member of the set pointed to by set.

Any object of type sigset_t must be initialized by applying either sigemptyset() or sigfillset() before applying any other operation.

RETURN VALUES

Upon successful completion, the sigismember() function returns 1 if the specified signal is a member of the specified set, or 0 if it is not.

Upon successful completion, the other functions return 0. Otherwise −1 is returned and errno is set to indicate the error.

ERRORS

The sigaddset(), sigdelset(), and sigismember() functions will fail if:

EINVAL The value of the signo argument is not a valid signal number.

The sigfillset() function will fail if:

EFAULT The set argument specifies an invalid address.
ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

sigaction(2), sigpending(2), sigprocmask(2), sigsuspend(2), attributes(5), signal(3HEAD)
sigstack – set and/or get alternate signal stack context

#include <signal.h>

int sigstack(struct sigstack *ss, struct sigstack *oss);

The sigstack() function allows the calling process to indicate to the system an area of its address space to be used for processing signals received by the process.

If the ss argument is not a null pointer, it must point to a sigstack structure. The length of the application-supplied stack must be at least SIGSTKSZ bytes. If the alternate signal stack overflows, the resulting behavior is undefined. (See USAGE below.)

- The value of the ss_onstack member indicates whether the process wants the system to use an alternate signal stack when delivering signals.
- The value of the ss_sp member indicates the desired location of the alternate signal stack area in the process' address space.
- If the ss argument is a null pointer, the current alternate signal stack context is not changed.

If the oss argument is not a null pointer, it points to a sigstack structure in which the current alternate signal stack context is placed. The value stored in the ss_onstack member of oss will be non-zero if the process is currently executing on the alternate signal stack. If the oss argument is a null pointer, the current alternate signal stack context is not returned.

When a signal's action indicates its handler should execute on the alternate signal stack (specified by calling sigaction(2)), sigstack() checks to see if the process is currently executing on that stack. If the process is not currently executing on the alternate signal stack, the system arranges a switch to the alternate signal stack for the duration of the signal handler's execution.

After a successful call to one of the exec functions, there are no alternate signal stacks in the new process image.

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

The sigstack() function will fail if:

EPERM An attempt was made to modify an active stack.

A portable application, when being written or rewritten, should use sigaltstack(2) instead of sigstack().

The direction of stack growth is not indicated in the historical definition of struct sigstack. The only way to portably establish a stack pointer is for the application to determine stack growth direction, or to allocate a block of storage and set the stack pointer.
sigstack() may assume that the size of the signal stack is 
SIGSTKSZ as found in <signal.h>. An application that would like to specify a 
signal stack size other than SIGSTKSZ should use sigaltstack().

Applications should not use longjmp(3C) to leave a signal handler that is running on 
a stack established with sigstack(). Doing so may disable future use of the signal 
stack. For abnormal exit from a signal handler, siglongjmp(3C), setcontext(2), or 
swapcontext(3C) may be used. These functions fully support switching from one 
stack to another.

The sigstack() function requires the application to have knowledge of the 
underlying system’s stack architecture. For this reason, sigaltstack(2) is 
recommended over this function.

SEE ALSO fork(2), _longjmp(3C), longjmp(3C), setjmp(3C), sigaltstack(2), 
siglongjmp(3C), sigsetjmp(3C)
The `sigstack()` function 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(3UCB)` 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:

```c
char *ss_sp; /* signal stack pointer */
int ss_onstack; /* current status */
```

The `ss_sp` member is the initial value to be assigned to the stack pointer when the system switches the process to the signal stack. Note that, on machines where the stack grows downwards in memory, this is not the address of the beginning of the signal stack area. The `ss_onstack` member is zero or non-zero depending on whether the process is currently executing on the signal stack or not.

If `nss` is not a null pointer, `sigstack()` sets the signal stack state to the value in the `sigstack()` structure pointed to by `nss`. If `nss` 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`.

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

The `sigstack()` function will fail and the signal stack context will remain unchanged if one of the following occurs.

- **EFAULT**: Either `nss` or `oss` points to memory that is not a valid part of the process address space.

**SEE ALSO**
- `sigaltstack(2)`, `sigvec(3UCB)`, `signal(3C)`

**WARNINGS**

Signal stacks are not "grown" automatically, as is done for the normal stack. If the stack overflows unpredictable results may occur.

**NOTES**

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-threaded applications is unsupported.
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 ORing 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:

```c
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(3UCB) rather than delivering the signal on the current stack.

If nvec 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 nvec is a NULL pointer, sigvec() does not change the handler, mask, or flags for the specified signal.

The mask specified in nvec 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 nvec 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 <signal.h>:

- SIGHUP    hangup
- SIGINT    interrupt
- SIGQUIT*   quit
- SIGILL*    illegal instruction
- SIGTRAP*   trace trap
- SIGABRT*   abort (generated by abort(3C) routine)
- SIGFPE*    arithmetic exception
- SIGKILL    kill (cannot be caught, blocked, or ignored)
- SIGBUS*    bus error
- SIGSEGV*   segmentation violation
- SIGSYS*    bad argument to function
- SIGPIPE    write on a pipe or other socket with no one to read it
- SIGALRM    alarm clock
- SIGTERM    software termination signal
- SIGURG*    urgent condition present on socket
- SIGSTOP**  stop (cannot be caught, blocked, or ignored)
SIGTSTP** stop signal generated from keyboard
SIGCONT* continue after stop (cannot be blocked)
SIGCHLD* child status has changed
SIGTTIN** background read attempted from control terminal
SIGTTOU** background write attempted to control terminal
SIGIO* I/O is possible on a descriptor (see fcntl(2))
SIGXCPU cpu time limit exceeded (see getrlimit(2))
SIGXFSZ file size limit exceeded (see getrlimit(2))
SIGVTALRM virtual time alarm; see setitimer() on getitimer(2)
SIGPROF profiling timer alarm; see setitimer() on getitimer(2)
SIGWINCH* window changed (see termio(7I))
SIGLOST resource lost (see lockd(1M))
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(2) 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, SIGFWR, 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 4.2BSD, 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 * or **. Signals marked with * are discarded if the action is SIG_DFL; signals marked with ** 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 (see core(4)).

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 functions, 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 4.2BSD, hence it should not be used if backward compatibility is needed. The affected functions are `read(2)` or `write(2)` on a slow device (such as a terminal or pipe or other socket, but not a file) and during a `wait(2)`.

After a `fork(2)` or `vfork(2)` the child inherits all signals, the signal mask, the signal stack, and the restart/interrupt and reset-signal-handler flags.

The `execve(2)` 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 functions 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`.

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 `nvec` 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`.

**SEE ALSO**

`intro(2)`, `exec(2)`, `fcntl(2)`, `fork(2)`, `getitimer(2)`, `getrlimit(2)`, `ioctl(2)`, `kill(2)`, `ptrace(2)`, `read(2)`, `umask(2)`, `vfork(2)`, `wait(2)`, `write(2)`, `setjmp(3C)`, `sigblock(3UCB)`, `sigstack(3UCB)`, `signal(3UCB)`, `wait(3UCB)`, `signal(3C)`, `core(4)`, `streamio(7I)`, `termio(7I)`

**NOTES**

Use of these interfaces should be restricted to only applications written on BSD platforms. Use of these interfaces with any of the system libraries or in multi-thread applications is unsupported.

`SIGPOLL` is a synonym for `SIGIO`. A `SIGIO` will be issued when a file descriptor corresponding to a STREAMS (see `intro(2)`) file has a “selectable” event pending. Unless that descriptor has been put into asynchronous mode (see `fcntl(2)`), a process may specifically request that this signal be sent using the `I_SETSIG ioctl(2)` call (see `streamio(7I)`). Otherwise, the process will never receive `SIGPOLL`s.

The handler routine can be declared:

```c
void handler(int sig, int code, struct sigcontext *scp, char *addr);
```

Here `sig` is the signal number; `code` is a parameter of certain signals that provides additional detail; `scp` is a pointer to the `sigcontext` structure (defined in `signal.h`), used to restore the context from before the signal; and `addr` is additional address information.
sigvec(3UCB)

The signals SIGKILL, SIGSTOP, and SIGCONT cannot be ignored.
sleep – suspend execution for an interval of time

#include <unistd.h>

unsigned int sleep(unsigned int seconds);

The current process is suspended from execution for the number of seconds specified by the argument. The actual suspension time may be less than that requested because any caught signal will terminate the sleep() following execution of that signal’s catching routine. Also, the suspension time may be longer than requested by an arbitrary amount because of the scheduling of other activity in the system. The value returned by sleep() will be the “unslept” amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested sleep() time, or premature arousal because of another caught signal.

In a single-threaded program (one not linked with -lthread or -lpthread), the routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling sleep(). If the sleep() time exceeds the time until such alarm signal, the process sleeps only until the alarm signal would have occurred. The caller’s alarm catch routine is executed just before the sleep() routine returns. But if the sleep() time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have without the intervening sleep().

In a multithreaded program (one linked with -lthread or -lpthread), the routine is implemented with a call to the nanosleep(3RT) function and does not modify the state of the alarm signal.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

alarm(2), pause(2), signal(3C), attributes(5)

In a single-threaded program, the SIGALRM signal should not be blocked or ignored during a call to sleep(). This restriction does not apply to a multithreaded program.

In a multithreaded program, only the invoking thread is suspended from execution.
NAME  
sleep – suspend execution for interval

SYNOPSIS  
/usr/ucb/cc [ flag ... ] file ...

    int 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.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  
alarm(2), getitimer(2), longjmp(3C), siglongjmp(3C), sleep(3C), usleep(3C),
attributes(5)

NOTES  
Use of these interfaces should be restricted to only applications written on BSD
platforms. Use of these interfaces with any of the system libraries or in multi-thread
applications is unsupported.

SIGALRM should not be blocked or ignored during a call to sleep(). Only a prior
call to alarm(2) should generate SIGALRM for the calling process during a call to
sleep(). A signal-catching function should not interrupt a call to sleep() to call
siglongjmp(3C) or longjmp(3C) to restore an environment saved prior to the
sleep() call.

WARNINGS  
sleep() is slightly incompatible with alarm(2). Programs that do not execute for at
least one second of clock time between successive calls to sleep() indefinitely delay
the alarm signal. Use sleep(3C). Each sleep(3C) call postpones the alarm signal that
would have been sent during the requested sleep period to occur one second later.

482  man pages section 3: Basic Library Functions • Last Revised 12 Feb 1993
NAME ssignal, gsignal – software signals

SYNOPSIS

```c
#include <signal.h>

void (*ssignal (int sig, int (*action)(int)))(int);
int gsignal(int sig);
```

DESCRIPTION

The `ssignal()` and `gsignal()` functions implement a software facility similar to `signal(3C)`. This facility is made available to users for their own purposes.

`ssignal()` Software signals made available to users are associated with integers in the inclusive range 1 through 17. A call to `ssignal()` associates a procedure, `action`, with the software signal `sig`; the software signal, `sig`, is raised by a call to `gsignal()`. Raising a software signal causes the action established for that signal to be taken.

The first argument to `ssignal()` is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user-defined) action function or one of the manifest constants `SIG_DFL` (default) or `SIG_IGN` (ignore). The `ssignal()` function returns the action previously established for that signal type; if no action has been established or the signal number is illegal, `ssignal()` returns `SIG_DFL`.

`gsignal()` The `gsignal()` raises the signal identified by its argument, `sig`.

If an action function has been established for `sig`, then that action is reset to `SIG_DFL` and the action function is entered with argument `sig`. The `gsignal()` function returns the value returned to it by the action function.

If the action for `sig` is `SIG_IGN`, `gsignal()` returns the value 1 and takes no other action.

If the action for `sig` is `SIG_DFL`, `gsignal()` returns the value 0 and takes no other action.

If `sig` has an illegal value or no action was ever specified for `sig`, `gsignal()` returns the value 0 and takes no other action.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Unsafe</td>
</tr>
</tbody>
</table>

SEE ALSO `raise(3C), signal(3C), attributes(5)`
The \texttt{stack\_getbounds()} function retrieves the stack boundaries that the calling thread is currently operating on. If the thread is currently operating on the alternate signal stack, this function will retrieve the bounds of that stack.

If successful, \texttt{stack\_getbounds()} sets the \texttt{ss\_sp} member of the \texttt{stack\_t} structure pointed to by \texttt{sp} to the base of the stack region and the \texttt{ss\_size} member to its size (maximum extent) in bytes. The \texttt{ss\_flags} member is set to \texttt{SS\_ONSTACK} if the calling thread is executing on its alternate signal stack, and zero otherwise.

Upon successful completion, \texttt{stack\_getbounds()} returns 0. Otherwise, \texttt{-1} is returned and \texttt{errno} is set to indicate the error.

The \texttt{stack\_getbounds()} function will fail if:

- \texttt{EFAULT} The \texttt{sp} argument does not refer to a valid address.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

See also \texttt{getustack(2)}, \texttt{sigaction(2)}, \texttt{sigaltstack(2)}, \texttt{stack\_setbounds(3C)}, attributes(5)
include <ucontext.h>
void * _stack_grow(void *addr);

The _stack_grow() function indicates to the system that the stack is about to be extended to the address specified by addr. If extending the stack to this address would violate the stack boundaries as retrieved by stack_getbounds(3C), a SIGSEGV is raised.

If the disposition of SIGSEGV is SIG_DFL, the process is terminated and a core dump is generated. If the application has installed its own SIGSEGV handler to run on the alternate signal stack, the signal information passed to the handler will be such that a call to stack_violation(3C) with these parameters returns 1.

The addr argument is a biased stack pointer value. See the Solaris 64-bit Developer’s Guide.

This function has no effect if the specified address, addr, is within the bounds of the current stack.

If the _stack_grow() function succeeds and does not detect a stack violation, it returns addr.

No errors are defined.

The _stack_grow() function does not actually adjust the stack pointer register. The caller is responsible for manipulating the stack pointer register once _stack_grow() returns.

The _stack_grow() function is typically invoked by code created by the compilation environment prior to executing code that modifies the stack pointer. It can also be used by hand-written assembly routines to allocate stack-based storage safely.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO stack_getbounds(3C), stack_inbounds(3C), stack_violation(3C), attributes(5)

Solaris 64-bit Developer’s Guide
stack_inbounds(3C)

NAME stack_inbounds – determine if address is within stack boundaries

SYNOPSIS #include <ucontext.h>

int stack_inbounds(void *addr);

DESCRIPTION The stack_inbounds() function returns a boolean value indicating whether the address specified by addr is within the boundaries of the stack of the calling thread. The address is compared to the stack boundary information returned by a call to stack_getbounds(3C).

RETURN VALUES The stack_inbounds() function returns 0 to indicate that addr is not within the current stack bounds, or a non-zero value to indicate that addr is within the stack bounds.

ERRORS No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO stack_getbounds(3C), attributes(5)
The `stack_setbounds()` function updates the current base and bounds of the stack for the current thread to the bounds specified by the `stack_t` structure pointed to by `sp`. The `ss_sp` member refers to the virtual address of the base of the stack memory. The `ss_size` member refers to the size of the stack in bytes. The `ss_flags` member must be set to 0.

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

The `stack_setbounds()` function will fail if:

- `EFAULT` The `sp` argument does not refer to a valid address or the `ss_sp` member of the `stack_t` structure pointed to by `sp` points to an illegal address.
- `EINVAL` The `ss_sp` member of the `stack_t` structure pointed to by `sp` is not properly aligned, the `ss_size` member is too small or is not properly aligned, or the `ss_flags` member is non-zero.

The `stack_setbounds()` function is intended for use by applications that are managing their own alternate stacks.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

See also `getustack(2), _stack_grow(3C), stack_getbounds(3C), stack_inbounds(3C), stack_violation(3C), attributes(5)`
### stack_violation(3C)

**NAME**  
`stack_violation` – determine stack boundary violation event

**SYNOPSIS**  
```
#include <ucontext.h>

int stack_violation(int sig, const siginfo_t *sip, const ucontext_t *ucp);
```

**DESCRIPTION**  
The `stack_violation()` function returns a boolean value indicating whether the signal, `sig`, and accompanying signal information, `sip`, and saved context, `ucp`, represent a stack boundary violation event or a stack overflow.

**RETURN VALUES**  
The `stack_violation()` function returns 0 if the signal does not represent a stack boundary violation event and 1 if the signal does represent a stack boundary violation event.

**ERRORS**  
No errors are defined.

**EXAMPLES**  
**EXAMPLE 1** Set up a signal handler to run on an alternate stack.

The following example sets up a signal handler for SIGSEGV to run on an alternate signal stack. For each signal it handles, the handler emits a message to indicate if the signal was produced due to a stack boundary violation.

```c
#include <stdlib.h>
#include <unistd.h>
#include <ucontext.h>
#include <signal.h>

static void handler(int sig, siginfo_t *sip, void *p)
{
    ucontext_t *ucp = p;
    const char *str;
    if (stack_violation(sig, sip, ucp))
        str = "stack violation.\n";
    else
        str = "no stack violation.\n";
    (void) write(STDERR_FILENO, str, strlen(str));
    exit(1);
}

int main(int argc, char **argv)
{
    struct sigaction sa;
    stack_t altstack;

    altstack.ss_size = SIGSTKSZ;
    altstack.ss_sp = malloc(SIGSTKSZ);
    altstack.ss_flags = 0;

    (void) sigaltstack(&altstack, NULL);
}
```
EXAMPLE 1

Set up a signal handler to run on an alternate stack.

(Continued)

```c
sa.sa_sigaction = handler;
(void) sigfillset(&sa.sa_mask);
sa.sa_flags = SA_ONSTACK | SA_SIGINFO;
(void) sigaction(SIGSEGV, &sa, NULL);

/*
 * The application is now set up to use stack_violation(3C).
 */
return (0);
```

USAGE

An application typically uses `stack_violation()` in a signal handler that has been installed for SIGSEGV using `sigaction(2)` with the SA_SIGINFO flag set and is configured to run on an alternate signal stack.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

`sigaction(2), sigaltstack(2), stack_getbounds(3C), stack_inbounds(3C), stack_setbounds(3C), attributes(5)`
stdio(3C)

NAME  stdio – standard buffered input/output package

SYNOPSIS

#include <stdio.h>
extern FILE *stdin;
extern FILE *stdout;
extern FILE *stderr;

DESCRIPTION

The functions described in the entries of 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(3C) and putchar(3C), and the
higher-level routines fgetc(3C), fgets(3C), fprintf(3C), fputc(3C), fputs(3C),
freopen(3C), fscanf(3C), fwrite(3C), gets(3C), getw(3C), printf(3C), puts(3C),
putw(3C), and scanf(3C) all use or act as if they use getc() and putc(); they can
be freely intermixed.

A file with associated buffering is called a stream (see intro(3)) and is declared to be a
pointer to a defined type FILE. The fopen(3C) function creates certain descriptive
data for a stream and returns a pointer to designate the stream in all further
transactions. Normally, there are three open streams with constant pointers declared in
the <stdio.h> header and associated with the standard open files:

stdin    standard input file
stdout   standard output file
stderr   standard error file

The following symbolic values in <unistd.h> define the file descriptors that will be
associated with the C-language stdin, stdout and stderr when the application is
started:

STDIN_FILENO  Standard input value  0  stdin
STDOUT_FILENO Standard output value  1  stdout
STDERR_FILENO Standard error value  2  stderr

The constant NULL designates a null pointer.

The integer-constant EOF is returned upon end-of-file or error by most integer
functions that deal with streams (see the individual descriptions for details).

The integer constant BUFSIZ specifies the size of the buffers used by the particular
implementation.

The integer constant FILENAME_MAX specifies the number of bytes needed to hold the
longest pathname of a file allowed by the implementation. If the system does not
impose a maximum limit, this value is the recommended size for a buffer intended to
hold a file’s pathname.

490  man pages section 3: Basic Library Functions • Last Revised 22 Jan 1993
The integer constant `FOPEN_MAX` specifies the minimum number of files that the implementation guarantees can be open simultaneously. Note that no more than 255 files may be opened using `fopen()`, and only file descriptors 0 through 255 can be used in a stream.

The functions and constants mentioned in the entries of section 3S of this manual are declared in that header and need no further declaration. The constants and the following “functions” are implemented as macros (redeclaration of these names is perilous): `getc()`, `getchar()`, `putc()`, `putchar()`, `ferror(3C)`, `feof(3C)`, `clearerr(3C)`, and `fileno(3C)`. There are also function versions of `getc()`, `getchar()`, `putc()`, `putchar()`, `ferror()`, `feof()`, `clearerr()`, and `fileno()`. Output streams, with the exception of the standard error stream `stderr`, are by default buffered if the output refers to a file and line-buffered if the output refers to a terminal. The standard error output stream `stderr` is by default unbuffered, but use of `freopen()` (see `fopen(3C)`) will cause it to become buffered or line-buffered. When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as written; when it is buffered, many characters are saved up and written as a block. When it is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested). The `setbuf()` or `setvbuf()` functions (both described on the `setbuf(3C)` manual page) may be used to change the stream’s buffering strategy.

A single open file description can be accessed both through streams and through file descriptors. Either a file descriptor or a stream will be called a `handle` on the open file description to which it refers; an open file description may have several handles.

Handles can be created or destroyed by user action without affecting the underlying open file description. Some of the ways to create them include `fcntl(2)`, `dup(2)`, `fdopen(3C)`, `fileno(3C)` and `fork(2)` (which duplicates existing ones into new processes). They can be destroyed by at least `fclose(3C)` and `close(2)`, and by the `exec` functions (see `exec(2)`), which close some file descriptors and destroy streams.

A file descriptor that is never used in an operation and could affect the file offset (for example `read(2)`, `write(2)`, or `lseek(2)`) is not considered a handle in this discussion, but could give rise to one (as a consequence of `fdopen()`, `dup()`, or `fork()`, for example). This exception does include the file descriptor underlying a stream, whether created with `fopen()` or `fdopen()`, as long as it is not used directly by the application to affect the file offset. (The `read()` and `write()` functions implicitly affect the file offset; `lseek()` explicitly affects it.)

If two or more handles are used, and any one of them is a stream, their actions shall be coordinated as described below. If this is not done, the result is undefined.

A handle that is a stream is considered to be closed when either an `fclose()` or `freopen(3C)` is executed on it (the result of `freopen()` is a new stream for this discussion, which cannot be a handle on the same open file description as its previous
value) or when the process owning that stream terminates the exit(2) or abort(3C). A file descriptor is closed by close(), _exit() (see exit(2)), or by one of the exec functions when FD_CLOEXEC is set on that file descriptor.

For a handle to become the active handle, the actions below must be performed between the last other user of the first handle (the current active handle) and the first other user of the second handle (the future active handle). The second handle then becomes the active handle. All activity by the application affecting the file offset on the first handle shall be suspended until it again becomes the active handle. (If a stream function has as an underlying function that affects the file offset, the stream function will be considered to affect the file offset. The underlying functions are described below.)

The handles need not be in the same process for these rules to apply. Note that after a fork(), two handles exist where one existed before. The application shall assure that, if both handles will ever be accessed, that they will both be in a state where the other could become the active handle first. The application shall prepare for a fork() exactly as if it were a change of active handle. (If the only action performed by one of the processes is one of the exec functions or _exit(), the handle is never accessed in that process.)

1. For the first handle, the first applicable condition below shall apply. After the actions required below are taken, the handle may be closed if it is still open.
   a. If it is a file descriptor, no action is required.
   b. If the only further action to be performed on any handle to this open file description is to close it, no action need be taken.
   c. If it is a stream that is unbuffered, no action need be taken.
   d. If it is a stream that is line-buffered and the last character written to the stream was a newline (that is, as if a putc(‘\n’) was the most recent operation on that stream), no action need be taken.
   e. If it is a stream that is open for writing or append (but not also open for reading), either an fflush() shall occur or the stream shall be closed.
   f. If the stream is open for reading and it is at the end of the file (feof(3C) is true), no action need be taken.
   g. If the stream is open with a mode that allows reading and the underlying open file description refers to a device that is capable of seeking, either an fflush() shall occur or the stream shall be closed.
   h. Otherwise, the result is undefined.
2. For the second handle: if any previous active handle has called a function that explicitly changed the file offset, except as required above for the first handle, the application shall perform a lseek() or an fseek(3C) (as appropriate to the type of the handle) to an appropriate location.
3. If the active handle ceases to be accessible before the requirements on the first handle above have been met, the state of the open file description becomes undefined. This might occur, for example, during a fork() or an _exit().
4. The `exec` functions shall be considered to make inaccessible all streams that are open at the time they are called, independent of what streams or file descriptors may be available to the new process image.

5. Implementation shall assure that an application, even one consisting of several processes, shall yield correct results (no data is lost or duplicated when writing, all data is written in order, except as requested by seeks) when the rules above are followed, regardless of the sequence of handles used. If the rules above are not followed, the result is unspecified. When these rules are followed, it is implementation defined whether, and under what conditions, all input is seen exactly once.

All the `stdio` functions are safe unless they have the `_unlocked` suffix. Each `FILE` pointer has its own lock to guarantee that only one thread can access it. In the case that output needs to be synchronized, the lock for the `FILE` pointer can be acquired before performing a series of `stdio` operations. For example:

```c
FILE iop;
flockfile(iop);
fprintf(iop, "hello ");
fprintf(iop, "world");
putc(iop, 'a');
funlockfile(iop);
```

will print everything out together, blocking other threads that might want to write to the same file between calls to `fprintf()`.

An unlocked interface is available in case performance is an issue. For example:

```c
flockfile(iop);
while (!feof(iop)) {
  *c++ = getc_unlocked(iop);
}
funlockfile(iop);
```

Invalid stream pointers usually cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

**See also**

- `close(2)`
- `lseek(2)`
- `open(2)`
- `pipe(2)`
- `read(2)`
- `write(2)`
- `ctermid(3C)`
- `cuserid(3C)`
- `fclose(3C)`
- `ferror(3C)`
- `fopen(3C)`
- `fread(3C)`
- `fseek(3C)`
- `flockfile(3C)`
- `getc(3C)`
- `gets(3C)`
- `popen(3C)`
- `printf(3C)`
- `putc(3C)`
- `puts(3C)`
- `scanf(3C)`
- `setbuf(3C)`
- `system(3C)`
- `tmpfile(3C)`
- `tmpnam(3C)`
- `ungetc(3C)`
str2sig(3C)

NAME     str2sig, sig2str – translation between signal name and signal number

SYNOPSIS  #include <signal.h>

int str2sig(const char *str, int *signum);
int sig2str(int signum, char *str);

DESCRIPTION The str2sig() function translates the signal name str to a signal number, and stores that result in the location referenced by signum. The name in str can be either the symbol for that signal, without the "SIG" prefix, or a decimal number. All the signal symbols defined in <sys/signal.h> are recognized. This means that both "CLD" and "CHLD" are recognized and return the same signal number, as do both "POLL" and "IO". For access to the signals in the range SIGRTMIN to SIGRTMAX, the first four signals match the strings "RTMIN", "RTMIN+1", "RTMIN+2", and "RTMIN+3" and the last four match the strings "RTMAX-3", "RTMAX-2", "RTMAX-1", and "RTMAX".

The sig2str() function translates the signal number signum to the symbol for that signal, without the "SIG" prefix, and stores that symbol at the location specified by str. The storage referenced by str should be large enough to hold the symbol and a terminating null byte. The symbol SIG2STR_MAX defined by <signal.h> gives the maximum size in bytes required.

RETURN VALUES The str2sig() function returns 0 if it recognizes the signal name specified in str; otherwise, it returns −1.

The sig2str() function returns 0 if the value signum corresponds to a valid signal number; otherwise, it returns −1.

EXAMPLES EXAMPLE 1 A sample program using the str2sig() function.

```c
int i;
char buf[SIG2STR_MAX];  /*storage for symbol */
str2sig("KILL", &i);    /*stores 9 in i */
str2sig("9", &i);       /* stores 9 in i */
sig2str(SIGKILL, buf);  /* stores "KILL" in buf */
sig2str(9, buf);        /* stores "KILL" in buf */
```

SEE ALSO kill(1), strsignal(3C)
NAME
strcoll – string collation

SYNOPSIS
#include <string.h>

int strcoll(const char *s1, const char *s2);

DESCRIPTION
Both strcoll() and strxfrm(3C) provide for locale-specific string sorting.
strcoll() is intended for applications in which the number of comparisons per
string is small. When strings are to be compared a number of times, strxfrm(3C) is a
more appropriate function because the transformation process occurs only once.

RETURN VALUES
Upon successful completion, strcoll() returns an integer greater than, equal to, or
less than zero in direct correlation to whether string s1 is greater than, equal to, or less
than the string s2. The comparison is based on strings interpreted as appropriate to the
program’s locale for category LC_COLLATE (see setlocale(3C)).

On error, strcoll() may set errno, but no return value is reserved to indicate an
error.

ERRORS
The strcoll() function may fail if:

EINVAL The s1 or s2 arguments contain characters outside the domain of
the collating sequence.

FILES
/usr/lib/locale/locale/locale.so.*
LC_COLLATE database for locale

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO
localedef(1), setlocale(3C), string(3C), strxfrm(3C), wstrxfrm(3C),
attributes(5), environ(5)

NOTES
The strcoll() function can be used safely in multithreaded applications, as long as
setlocale(3C) is not being called to change the locale.
**NAME**
strerror – get error message string

**SYNOPSIS**
```c
#include <string.h>

char *strerror(int errnum);
```

**DESCRIPTION**
The `strerror()` function maps the error number in `errnum` to an error message string, and returns a pointer to that string. It uses the same set of error messages as `perror(3C).` The returned string should not be overwritten.

**RETURN VALUES**
The `strerror()` function returns the string “Unknown error” if `errnum` is out of range.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Standard</td>
</tr>
<tr>
<td></td>
<td>Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**
`gettext(3C), perror(3C), setlocale(3C), attributes(5)`

**NOTES**
If the application is linked with `-lintl`, then messages returned from this function are in the native language specified by the `LC_MESSAGES` locale category; see `setlocale(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
- a required conversion character that determines the conversion to be performed.

Flags
One or more of the following optional flags can be specified to control the conversion:

- `=f` An `=` followed by a single character `f` which is used as the numeric fill character. The fill character must be representable in a single byte in order to work with precision and width counts. The default numeric fill character is the space character. This flag does not affect field width filling which always uses the space character. This flag is ignored unless a left precision (see below) is specified.

- `^` Do not format the currency amount with grouping characters. The default is to insert the grouping characters if defined for the current locale.

- `+` or `{` Specify the style of representing positive and negative currency amounts. Only one of `+` or `{` may be specified. If `+` is specified, the locale’s equivalent of `+` and `{` are used (for example, in the U.S.A.: 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 with the numeric fill character (see the =f flag above).

If grouping has not been suppressed with the ‘^’ flag, and it is defined for the current locale, grouping separators are inserted before the fill characters (if any) are added. Grouping separators are not applied to fill characters even if the fill character is a digit.

To ensure alignment, any characters appearing before or after the number in the formatted output such as currency or sign symbols are padded as necessary with space characters to make their positive and negative formats an equal length.

Right Precision | \.p
---|---
A period followed by a decimal digit string \( p \) specifying the number of digits after the radix character. If the value of the right precision \( p \) is zero, no radix character appears. If a right precision is not included, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

Conversion Characters | The conversion characters and their meanings are:
---|---
i | The `double` argument is formatted according to the locale’s international currency format (for example, in the U.S.A.: USD 1,234.56).
n | The `double` argument is formatted according to the locale’s national currency format (for example, in the U.S.A.: $1,234.56).
% | Convert to a %; no argument is converted. The entire conversion specification must be %%. 

Locale Information | The `LC_MONETARY` category of the program’s locale affects the behavior of this function including the monetary radix character (which may be different from the numeric radix character affected by the `LC_NUMERIC` category), the grouping separator, the currency symbols and formats. The international currency symbol should be in conformance with the ISO 4217: 1987 standard.

RETURN VALUES | If the total number of resulting bytes (including the terminating null byte) is not more than `maxsize`, `strfmon()` returns the number of bytes placed into the array pointed to by `s`, not including the terminating null byte. Otherwise, –1 is returned, the contents of the array are indeterminate, and `errno` is set to indicate the error.

ERRORS | The `strfmon()` function will fail if:
---|---
ENOSYS | The function is not supported.

498  man pages section 3: Basic Library Functions • Last Revised 20 Dec 1996
Conversion stopped due to lack of space in the buffer.

**EXAMPLE 1** A sample output of `strfmon()`.

Given a locale for the U.S.A. and the values 123.45, −123.45, and 3456.781:

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Output</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>%n</td>
<td>$123.45</td>
<td>default formatting</td>
</tr>
<tr>
<td></td>
<td>-$123.45</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$3,456.78</td>
<td></td>
</tr>
<tr>
<td>%11n</td>
<td>$123.45</td>
<td>right align within an 11 character field</td>
</tr>
<tr>
<td></td>
<td>-$123.45</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$3,456.78</td>
<td></td>
</tr>
<tr>
<td>%#5n</td>
<td>$123.45</td>
<td>aligned columns for values up to 99,999</td>
</tr>
<tr>
<td></td>
<td>-$123.45</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$3,456.78</td>
<td></td>
</tr>
<tr>
<td>%=*#5n</td>
<td>$***123.45</td>
<td>specify a fill character</td>
</tr>
<tr>
<td></td>
<td>-$***123.45</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$*3,456.78</td>
<td></td>
</tr>
<tr>
<td>%=0#5n</td>
<td>$000123.45</td>
<td>fill characters do not use</td>
</tr>
<tr>
<td></td>
<td>-$000123.45</td>
<td>grouping even if the fill character is a digit</td>
</tr>
<tr>
<td></td>
<td>$03,456.78</td>
<td></td>
</tr>
<tr>
<td>%^#5n</td>
<td>$123.45</td>
<td>disable the grouping separator</td>
</tr>
<tr>
<td></td>
<td>-$123.45</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$3456.78</td>
<td></td>
</tr>
<tr>
<td>%^#5.0n</td>
<td>$123</td>
<td>round off to whole units</td>
</tr>
<tr>
<td></td>
<td>-$123</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$3457</td>
<td></td>
</tr>
<tr>
<td>%^#5.4n</td>
<td>$123.4500</td>
<td>increase the precision</td>
</tr>
<tr>
<td></td>
<td>-$123.4500</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$3456.7810</td>
<td></td>
</tr>
</tbody>
</table>
strfmon(3C)

EXAMPLE 1 A sample output of strfmon(). (Continued)

<table>
<thead>
<tr>
<th>Conversion Specification</th>
<th>Output</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>%(#5n $123.45)</td>
<td>123.45</td>
<td>use an alternative pos/neg style</td>
</tr>
<tr>
<td>$3,456.78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%(#5n 123.45)</td>
<td>123.45</td>
<td>disable the currency symbol</td>
</tr>
<tr>
<td>(123.45)</td>
<td>123.45</td>
<td></td>
</tr>
<tr>
<td>3,456.78</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO
localeconv(3C), setlocale(3C), attributes(5)

NOTES
This function can be used safely in multithreaded applications, as long as setlocale(3C) is not called to change the locale.
strft ime(3C)

NAME
strftime, cftime, asctime – 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 asctime(char *s, const char *format, const struct tm *timeptr);

DESCRIPTION
The strftime(), asctime(), and cftime() functions place bytes into the array
pointed to by s as controlled by the string pointed to by format. The format string
consists of zero or more conversion specifications and ordinary characters. A
conversion specification consists of a ’%‘ (percent) character and one or two
terminating conversion characters that determine the conversion specification’s
behavior. All ordinary characters (including the terminating null byte) are copied
unchanged into the array pointed to by s. If copying takes place between objects that
overlap, the behavior is undefined. For strftime(), no more than maxsize bytes are
placed into the array.

If format is (char *)0, then the locale’s default format is used. For strftime() the
default format is the same as %c; for cftime() and asctime() the default format
is the same as %C. cftime() and asctime() 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 asctime(), and by the time represented by clock for
cftime().

%% Same as %.
%a Locale’s abbreviated weekday name.
%A Locale’s full weekday name.
%b Locale’s abbreviated month name.
%B Locale’s full month name.
%c Locale’s appropriate date and time representation.
%C Locale’s date and time representation as produced by date(1).
%C Century number (the year divided by 100 and truncated to an integer as a
decimal number [1,99]); single digits are preceded by 0; see standards(5).
%d Day of month [1,31]; single digits are preceded by 0.
%D Date as %m/%d/%y.
%e Day of month [1,31]; single digits are preceded by a space.
**strftime(3C)**

<table>
<thead>
<tr>
<th>Format Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%g</td>
<td>Week-based year within century [00,99].</td>
</tr>
<tr>
<td>%G</td>
<td>Week-based year, including the century [0000,9999].</td>
</tr>
<tr>
<td>%h</td>
<td>Locale’s abbreviated month name.</td>
</tr>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) [0,23]; single digits are preceded by 0.</td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) [1,12]; single digits are preceded by 0.</td>
</tr>
<tr>
<td>%j</td>
<td>Day number of year [1,366]; single digits are preceded by 0.</td>
</tr>
<tr>
<td>%k</td>
<td>Hour (24-hour clock) [0,23]; single digits are preceded by a blank.</td>
</tr>
<tr>
<td>%l</td>
<td>Hour (12-hour clock) [1,12]; single digits are preceded by a blank.</td>
</tr>
<tr>
<td>%m</td>
<td>Month number [1,12]; single digits are preceded by 0.</td>
</tr>
<tr>
<td>%M</td>
<td>Minute [00,59]; leading 0 is permitted but not required.</td>
</tr>
<tr>
<td>%n</td>
<td>Insert a NEWLINE.</td>
</tr>
<tr>
<td>%p</td>
<td>Locale’s equivalent of either a.m. or p.m.</td>
</tr>
<tr>
<td>%r</td>
<td>Appropriate time representation in 12-hour clock format with %p.</td>
</tr>
<tr>
<td>%R</td>
<td>Time as %H:%M.</td>
</tr>
<tr>
<td>%S</td>
<td>Seconds [00,61]; the range of values is [00,61] rather than [00,59] to allow for the occasional leap second and even more occasional double leap second.</td>
</tr>
<tr>
<td>%t</td>
<td>Insert a TAB.</td>
</tr>
<tr>
<td>%T</td>
<td>Time as %H:%M:%S.</td>
</tr>
<tr>
<td>%u</td>
<td>Weekday as a decimal number [1,7], with 1 representing Monday. See NOTES below.</td>
</tr>
<tr>
<td>%U</td>
<td>Week number of year as a decimal number [00,53], with Sunday as the first day of week 1.</td>
</tr>
<tr>
<td>%V</td>
<td>The ISO 8601 week number as a decimal number [01,53]. In the ISO 8601 week-based system, weeks begin on a Monday and week 1 of the year is the week that includes both January 4th and the first Thursday of the year. If the first Monday of January is the 2nd, 3rd, or 4th, the preceding days are part of the last week of the preceding year. See NOTES below.</td>
</tr>
<tr>
<td>%W</td>
<td>Weekday as a decimal number [0,6], with 0 representing Sunday.</td>
</tr>
<tr>
<td>%W</td>
<td>Week number of year as a decimal number [00,53], with Monday as the first day of week 1.</td>
</tr>
<tr>
<td>%x</td>
<td>Locale’s appropriate date representation.</td>
</tr>
<tr>
<td>%X</td>
<td>Locale’s appropriate time representation.</td>
</tr>
<tr>
<td>%y</td>
<td>Year within century [00,99].</td>
</tr>
</tbody>
</table>
%Y Year, including the century (for example 1993).

%Z Time zone name or abbreviation, or no bytes if no time zone information exists.

If a conversion specification does not correspond to any of the above or to any of the modified conversion specifications listed below, the behavior is undefined and 0 is returned.

The difference between %U and %W (and also between modified conversion specifications %OU and %OW) lies in which day is counted as the first of the week. Week number 1 is the first week in January starting with a Sunday for %U or a Monday for %W. Week number 0 contains those days before the first Sunday or Monday in January for %U and %W, respectively.

Some conversion specifications can be modified by the E and O modifiers to indicate that an alternate format or specification should be used rather than the one normally used by the unmodified conversion specification. If the alternate format or specification does not exist in the current locale, the behavior will be as if the unmodified specification were used.

%Ec Locale’s alternate appropriate date and time representation.

%EC Name of the base year (period) in the locale’s alternate representation.

%Eg Offset from %EC of the week-based year in the locale’s alternative representation.

%EG Full alternative representation of the week-based year.

%Ex Locale’s alternate date representation.

%EX Locale’s alternate time representation.

%Ey Offset from %EC (year only) in the locale’s alternate representation.

%EY Full alternate year representation.

%Od Day of the month using the locale’s alternate numeric symbols.

%Oe Same as %Od.

%Og Week-based year (offset from %C) in the locale’s alternate representation and using the locale’s alternate numeric symbols.

%OH Hour (24-hour clock) using the locale’s alternate numeric symbols.

%OI Hour (12-hour clock) using the locale’s alternate numeric symbols.

%Om Month using the locale’s alternate numeric symbols.

%OM Minutes using the locale’s alternate numeric symbols.

%OS Seconds using the locale’s alternate numeric symbols.

% Ou Weekday as a number in the locale’s alternate numeric symbols.
strftime(3C)

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%U</td>
<td>Week number of the year (Sunday as the first day of the week) using the locale’s alternate numeric symbols.</td>
</tr>
<tr>
<td>%w</td>
<td>Number of the weekday (Sunday=0) using the locale’s alternate numeric symbols.</td>
</tr>
<tr>
<td>%W</td>
<td>Week number of the year (Monday as the first day of the week) using the locale’s alternate numeric symbols.</td>
</tr>
<tr>
<td>%Y</td>
<td>Year (offset from %C) in the locale’s alternate representation and using the locale’s alternate numeric symbols.</td>
</tr>
</tbody>
</table>

**Selecting the Output Language**

By default, the output of `strftime()`, `cftime()`, and `asctime()` appear in U.S. English. The user can request that the output of `strftime()`, `cftime()`, or `asctime()` be in a specific language by setting the `LC_TIME` category using `setlocale()`.

**Time Zone**

Local time zone information is used as though `tzset(3C)` were called.

**RETURN VALUES**

The `strftime()`, `cftime()`, and `asctime()` functions return the number of characters placed into the array pointed to by `s`, not including the terminating null character. If the total number of resulting characters including the terminating null character is more than `maxsize`, `strftime()` returns 0 and the contents of the array are indeterminate.

**EXAMPLES**

**EXAMPLE 1** An example of the `strftime()` function.

The following example illustrates the use of `strftime()` for the POSIX locale. It shows what the string in `str` would look like if the structure pointed to by `imptr` contains the values corresponding to Thursday, August 28, 1986 at 12:44:36.

```c
strftime (str, strsize, "%A %b %d %j", tmpr)
```

This results in `str` containing "Thursday Aug 28 240".

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

**SEE ALSO**

date(1), ctime(3C), mktime(3C), setlocale(3C), strftime(3C), tzset(3C), TIMEZONE(4), zoneinfo(4), attributes(5), environ(5), standards(5)
The conversion specification for `%u` was changed in the Solaris 8 release. This change was based on the XPG4 specification.

If using the `%Z` specifier and zoneinfo timezones and if the input date is outside the range 20:45:52 UTC, December 13, 1901 to 03:14:07 UTC, January 19, 2038, the timezone name may not be correct.
#include <string.h>

int strcasecmp(const char *s1, const char *s2);
int strnccccmp(const char *s1, const char *s2, size_t n);

#include <string.h>

char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
size_t strlcat(char *dst, const char *src, size_t dstsize);
char *strchr(const char *s, int c);
char *strchr(const char *s, int c);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);

char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
size_t strlcpy(char *dst, const char *src, size_t dstsize);
size_t strcspn(const char *s1, const char *s2);
size_t strspn(const char *s1, const char *s2);

char *strdup(const char *s1);
size_t strlen(const char *s);
char *strpbrk(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 *s1, const char *s2, char **lasts);

#include <cstring>
const char *strchr(const char *s, int c);
const char *strpbrk(const char *s1, const char *s2);
const char *strchr(const char *s, int c);
const char *strstr(const char *s1, const char *s2);
#include <cstring>
char *std::strchr(char *s, int c);
char *std::strpbrk(char *s1, const char *s2);
char *std::strrchr(char *s, int c);
char *std::strstr(char *s1, const char *s2);

DESCRIPTION

The arguments s, s1, and s2 point to strings (arrays of characters terminated by a null character). The 
strcat(), strncat(), strlcat(), strcpy(), strncpy(), strlcpy(), strtok(), and strtok_r() functions all alter their first argument.
These functions do not check for overflow of the array pointed to by the first argument.

strcasecmp(), strncasecmp()
The strcasecmp() and strncasecmp() functions are case-insensitive versions of 
strcmp() and strncmp() respectively, described below. They assume the ASCII character set and ignore differences in case when comparing lower and upper case characters.

strcat(), strncat(), strlcat() The strcat() function appends a copy of string s2, including the terminating null character, to the end of string s1. The strncat() function appends at most n characters. Each returns a pointer to the null-terminated result. The initial character of s2 overrides the null character at the end of s1.

The strlcat() function appends at most (dstsize-strlen(dst)-1) characters of src to dst (dstsize being the size of the string buffer dst). If the string pointed to by dst contains a null-terminated string that fits into dstsize bytes when strlcat() is called, the string pointed to by dst will be a null-terminated string that fits in dstsize bytes (including the terminating null character) when it completes, and the initial character of src will override the null character at the end of dst. If the string pointed to by dst is longer than dstsize bytes when strlcat() is called, the string pointed to by dst will not be changed. The function returns the sum the of lengths of the two strings strlen(dst)+strlen(src). Buffer overflow can be checked as follows:

```c
if (strlcat(dst, src, dstsize) >= dstsize)
    return -1;
```

strchr(), strrchr() The strchr() function returns a pointer to the first occurrence of c (converted to a char) in string s, or a null pointer if c does not occur in the string. The strrchr() function returns a pointer to the last occurrence of c. The null character terminating a string is considered to be part of the string.

strcmp(), strncmp() The strcmp() function compares two strings byte-by-byte, according to the ordering of your machine's character set. The function returns an integer greater than, equal to, or less than 0, if the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2 respectively. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes that differ in the strings being compared. The strncmp() function makes the same comparison but looks at a maximum of n bytes. Bytes following a null byte are not compared.
The `strcpy()` function copies string `s2` to `s1`, including the terminating null character, stopping after the null character has been copied. The `strncpy()` function copies exactly \( n \) bytes, truncating `s2` or adding null characters to `s1` if necessary. The result will not be null-terminated if the length of `s2` is \( n \) or more. Each function returns `s1`.

The `strlcpy()` function copies at most `dstsize-1` characters (\( dstsize \) being the size of the string buffer `dst`) from `src` to `dst`, truncating `src` if necessary. The result is always null-terminated. The function returns `strlen(src)`. Buffer overflow can be checked as follows:

```c
if (strlcpy(dst, src, dstsize) >= dstsize)
    return -1;
```

The `strcspn()` function returns the length of the initial segment of string `s1` that consists entirely of characters not from string `s2`. The `strspn()` function returns the length of the initial segment of string `s1` that consists entirely of characters from string `s2`.

The `strdup()` function returns a pointer to a new string that is a duplicate of the string pointed to by `s1`. The returned pointer can be passed to `free()`. The space for the new string is obtained using `malloc()`. If the new string cannot be created, a null pointer is returned and `errno` may be set to `ENOMEM` to indicate that the storage space available is insufficient.

The `strlen()` function returns the number of bytes in `s`, not including the terminating null character.

The `strpbrk()` function returns a pointer to the first occurrence in string `s1` of any character from string `s2`, or a null pointer if no character from `s2` exists in `s1`.

The `strstr()` function locates the first occurrence of the string `s2` (excluding the terminating null character) in string `s1` and returns a pointer to the located string, or a null pointer if the string is not found. If `s2` points to a string with zero length (that is, the string ""), the function returns `s1`.

The `strtok()` function can be used to break the string pointed to by `s1` into a sequence of tokens, each of which is delimited by one or more characters from the string pointed to by `s2`. The `strtok()` function considers the string `s1` to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string `s2`. The first call (with pointer `s1` specified) returns a pointer to the first character of the first token, and will have written a null character into `s1` immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument being a null pointer) will work through the string `s1` immediately following that token. In this way subsequent calls will work through the string `s1` until no tokens remain. The separator string `s2` may be different from call to call. When no token remains in `s1`, a null pointer is returned.
The `strtok_r()` function has the same functionality as `strtok()` except that a pointer to a string placeholder `lasts` must be supplied by the caller. The `lasts` pointer is to keep track of the next substring in which to search for the next token.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See NOTES below.</td>
</tr>
</tbody>
</table>

**SEE ALSO**
malloc(3C), setlocale(3C), strxfrm(3C), attributes(5)

**NOTES**

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

All of these functions assume the default locale "C." For some locales, `strxfrm()` should be applied to the strings before they are passed to the functions.

The `strcasecmp()`, `strcat()`, `strchr()`, `strcmp()`, `strcpy()`, `strcspn()`, `strdup()`, `strlen()`, `strncasecmp()`, `strncat()`, `strncmp()`, `strncpy()`, `strpbrk()`, `strstr()`, `strspn()` functions are MT-Safe in multithreaded applications.

The `strtok()` function is Unsafe in multithreaded applications. The `strtok_r()` function should be used instead.
string_to_decimal(3C)

NAME  string_to_decimal, file_to_decimal, func_to_decimal – parse characters into decimal record

SYNOPSIS  
```c
#include <floatingpoint.h>

void string_to_decimal(char **pc, int nmax, int fortran_conventions, 
decimal_record *pd, enum decimal_string_form *pform, char **pechar);

void func_to_decimal(char **pc, int nmax, int fortran_conventions, 
decimal_record *pd, enum decimal_string_form *pform, char **pechar, 
int (*pget) (void), int *pnread, int (*punget)(int c));

#include <stdio.h>

void file_to_decimal(char **pc, int nmax, int fortran_conventions, 
decimal_record *pd, enum decimal_string_form *pform, char **pechar, 
FILE *pf, int *pnread);
```

DESCRIPTION  
The `char_to_decimal` functions parse a numeric token from at most `nmax` characters in a string `*pc` or file `*pf` or function `(*pget)()` into a decimal record `*pd`, classifying the form of the string in `*pform` and `*pechar`. The accepted syntax is intended to be sufficiently flexible to accommodate many languages: whitespace value or whitespace sign value, where whitespace is any number of characters defined by `isspace` in `<ctype.h>`, sign is either of `+-`, and value can be `number`, `nan`, or `inf`. `inf` can be `INF` (`inf_form`) or INFINITY (`infinity_form`) without regard to case. `nan` can be `NAN` (`nan_form`) or `NAN(nstring)` (`nanstring_form`) without regard to case; `nstring` is any string of characters not containing `'('` or `NULL`; `nstring` is copied to `pd->ds` and, currently, not used subsequently. `number` consists of `significand` or `significand efield` where `significand` must contain one or more digits and may contain one point; possible forms are

- `digits`  (int_form)
- `digits.`  (intdot_form)
- `digits`  (dotfrac_form)
- `digits.digits`  (intdotfrac_form)

`efield` consists of `echar digits` or `echar sign digits`, where `echar` one of `[Ee]`, and `digits` contains one or more digits.

When `fortran_conventions` is nonzero, additional input forms are accepted according to various Fortran conventions:

- 0  no Fortran conventions
- 1  Fortran list-directed input conventions
- 2  Fortran formatted input conventions, ignore blanks (BN)
- 3  Fortran formatted input conventions, blanks are zeros (BZ)

When `fortran_conventions` is nonzero, `echar` may also be one of `[DdQq]`, and `efield` may also have the form
sign digits.

When `fortran_conventions` >= 2, blanks may appear in the `digits` strings for the integer, fraction, and exponent fields and may appear between `echar` and the exponent sign and after the infinity and NaN forms. If `fortran_conventions` == 2, the blanks are ignored. When `fortran_conventions` == 3, the blanks that appear in `digits` strings are interpreted as zeros, and other blanks are ignored.

When `fortran_conventions` is zero, the current locale’s decimal point character is used as the decimal point; when `fortran_conventions` is nonzero, the period is used as the decimal point.

The form of the accepted decimal string is placed in `*pform`. If an `efield` is recognized, `*pechar is set to point to the `echar`.

On input, `*pc points to the beginning of a character string buffer of length >= `nmax`. On output, `*pc points to a character in that buffer, one past the last accepted character. `string_to_decimal()` gets its characters from the buffer; `file_to_decimal()` gets its characters from `*pf` and records them in the buffer, and places a null after the last character read. `func_to_decimal()` gets its characters from an int function (`*pget`).

The scan continues until no more characters could possibly fit the acceptable syntax or until `nmax` characters have been scanned. If the `nmax` limit is not reached then at least one extra character will usually be scanned that is not part of the accepted syntax. `file_to_decimal()` and `func_to_decimal()` set `*pnread` to the number of characters read from the file; if greater than `nmax`, some characters were lost. If no characters were lost, `file_to_decimal()` and `func_to_decimal()` attempt to push back, with `ungetc()` or (`*punget`), as many as possible of the excess characters read, adjusting `*pnread` accordingly. If all unget calls are successful, then `*pc` will be `NULL`. No push back will be attempted if (`*punget`) is `NULL`.

Typical declarations for `*pget()` and `*punget()` are:

```c
int xget(void)
    { ... }
int (*pget)(void) = xget;
int xunget(int c)
    { ... }
int (*punget)(int) = xunget;
```

If no valid number was detected, `pd->fpclass` is set to `fp_signaling`, `*pc` is unchanged, and `*pform` is set to `invalid_form`.

`atof(3C)` and `strtol(3C)` use `string_to_decimal()`. `scanf(3C)` uses `file_to_decimal()`.
string_to_decimal(3C)

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

ctype(3), localeconv(3), scanf(3), setlocale(3), strtod(3), ungetc(3), attributes(5)
NAME

strptime – date and time conversion

SYNOPSIS

#include <time.h>

char *strptime(const char *buf, const char *format, struct tm *tm);

Non-zeroing Behavior

cc [flag ...] file ... -D_STRPTIME_DONTZERO [library ...]

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.

The format argument is composed of zero or more conversion specifications. Each conversion specification is composed of a "%" (percent) character followed by one or two conversion characters which specify the replacement required. One or more white space characters (as specified by isspace(3)) may precede or follow a conversion specification. There must be white-space or other non-alphanumeric characters between any two conversion specifications.

A non-zeroing version of strptime(), described below under Non-zeroing Behavior, is provided if _STRPTIME_DONTZERO is defined.

The following conversion specifications are supported:

% %
Same as %.

%a %a
Day of week, using the locale’s weekday names; either the abbreviated or full name may be specified.

%A %a
Same as %a.

%b %b
Month, using the locale’s month names; either the abbreviated or full name may be specified.

%B %b
Same as %b.

%c %c
Locale’s appropriate date and time representation.

%C %C
Century number (the year divided by 100 and truncated to an integer as a decimal number [1,99]); single digits are preceded by 0. If %C is used without the %y specifier, strptime() assumes the year offset is zero in whichever century is specified. Note the behavior of %C in the absence of %y is not specified by any of the standards or specifications described on the standards(5) manual page, so portable applications should not depend on it. This behavior may change in a future release.

%d %d
Day of month [1,31]; leading zero is permitted but not required.

%D %m/%d/%y
Date as %m/%d/%y.

%e %e
Same as %d.

%h %h
Same as %b.
### strftime(3C)

<table>
<thead>
<tr>
<th>Format Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%H</td>
<td>Hour (24-hour clock) [0,23]; leading zero is permitted but not required.</td>
</tr>
<tr>
<td>%I</td>
<td>Hour (12-hour clock) [1,12]; leading zero is permitted but not required.</td>
</tr>
<tr>
<td>%j</td>
<td>Day number of the year [1,366]; leading zeros are permitted but not required.</td>
</tr>
<tr>
<td>%m</td>
<td>Month number [1,12]; leading zero is permitted but not required.</td>
</tr>
<tr>
<td>%M</td>
<td>Minute [0-59]; leading zero is permitted but not required.</td>
</tr>
<tr>
<td>%n</td>
<td>Any white space.</td>
</tr>
<tr>
<td>%p</td>
<td>Locale’s equivalent of either a.m. or p.m.</td>
</tr>
<tr>
<td>%r</td>
<td>Appropriate time representation in the 12-hour clock format with %p.</td>
</tr>
<tr>
<td>%R</td>
<td>Time as %H:%M.</td>
</tr>
<tr>
<td>%S</td>
<td>Seconds [0,61]; leading zero is permitted but not required. The range of values is [00,61] rather than [00,59] to allow for the occasional leap second and even more occasional double leap second.</td>
</tr>
<tr>
<td>%t</td>
<td>Any white space.</td>
</tr>
<tr>
<td>%T</td>
<td>Time as %H:%M:%S.</td>
</tr>
<tr>
<td>%U</td>
<td>Week number of the year as a decimal number [0,53], with Sunday as the first day of the week; leading zeros are permitted but not required.</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday as a decimal number [0,6], with 0 representing Sunday.</td>
</tr>
<tr>
<td>%W</td>
<td>Week number of the year as a decimal number [0,53], with Monday as the first day of the week; leading zero is permitted but not required.</td>
</tr>
<tr>
<td>%x</td>
<td>Locale’s appropriate date representation.</td>
</tr>
<tr>
<td>%X</td>
<td>Locale’s appropriate time representation.</td>
</tr>
<tr>
<td>%y</td>
<td>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.</td>
</tr>
<tr>
<td>%Y</td>
<td>Year, including the century (for example, 1993) [1–9999].</td>
</tr>
<tr>
<td>%Z</td>
<td>Timezone name or no characters if no time zone information exists. Local timezone information is used as though strftime() called tzset() (see ctime(3C)). Errors may not be detected. This behavior is subject to change in a future release.</td>
</tr>
</tbody>
</table>
Some conversion specifications can be modified by the \texttt{E} and \texttt{O} modifier characters to indicate that an alternate format or specification should be used rather than the one normally used by the unmodified specification. If the alternate format or specification does not exist in the current locale, the behavior will be as if the unmodified conversion specification were used.

\texttt{%Ec} Locale's alternate appropriate date and time representation.
\texttt{%EC} Name of the base year (era) in the locale's alternate representation.
\texttt{%Ex} Locale's alternate date representation.
\texttt{%EX} Locale's alternate time representation.
\texttt{%Ey} Offset from \texttt{%EC} (year only) in the locale's alternate representation.
\texttt{%EY} Full alternate year representation.
\texttt{%Od} Day of the month using the locale's alternate numeric symbols.
\texttt{%Oe} Same as \texttt{%Od}.
\texttt{%OH} Hour (24-hour clock) using the locale's alternate numeric symbols.
\texttt{%OI} Hour (12-hour clock) using the locale's alternate numeric symbols.
\texttt{%Om} Month using the locale's alternate numeric symbols.
\texttt{%OM} Minutes using the locale's alternate numeric symbols.
\texttt{%OS} Seconds using the locale's alternate numeric symbols.
\texttt{%OU} Week number of the year (Sunday as the first day of the week) using the locale's alternate numeric symbols.
\texttt{%OW} Number of the weekday (Sunday=0) using the locale's alternate numeric symbols.
\texttt{%OW} Week number of the year (Monday as the first day of the week) using the locale's alternate numeric symbols.
\texttt{%Oy} Year (offset from \texttt{%C}) in the locale's alternate representation and using the locale's alternate numeric symbols.

A conversion specification that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the specification, the specification fails, and the differing and subsequent characters remain unscanned.

A series of specifications composed of \texttt{\%n}, \texttt{\%t}, white-space characters or any combination is executed by scanning up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned. White space is defined by \texttt{isspace(3C)}. 

\textbf{Basic Library Functions}
Any other conversion specification is executed by scanning characters until a character matching the next specification is scanned, or until no more characters can be scanned. These characters, except the one matching the next specification, are then compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate `tm` structure members are set to values corresponding to the locale information. If no match is found, `strptime()` fails and no more characters are scanned.

The month names, weekday names, era names, and alternate numeric symbols can consist of any combination of upper and lower case letters. The user can request that the input date or time specification be in a specific language by setting the `LC_TIME` category using `setlocale(3C).

In addition to the behavior described above by various standards, the Solaris implementation of `strptime()` provides the following extensions. These may change at any time in the future. Portable applications should not depend on these extended features:

- If `_STRPTIME_DONTZERO` is not defined, the `tm` struct is zeroed on entry and `strptime()` updates the fields of the `tm` struct associated with the specifiers in the format string.
- If `_STRPTIME_DONTZERO` is defined, `strptime()` does not zero the `tm` struct on entry. Additionally, for some specifiers, `strptime()` will use some values in the input `tm` struct to recalculate the date and re-assign the appropriate members of the `tm` struct.

The following describes extended features regardless of whether `_STRPTIME_DONTZERO` is defined or not defined:

- If `%j` is specified, `tm_yday` is set; if year is given, and if month and day are not given, `strptime()` calculates and sets `tm_mon`, `tm_mday`, and `tm_year`.
- If `%U` or `%W` is specified and if weekday and year are given and month and day of month are not given, `strptime()` calculates and sets `tm_mon`, `tm_mday`, `tm_wday`, and `tm_year`.

The following describes extended features when `_STRPTIME_DONTZERO` is not defined:

- If `%C` is specified and `%Y` is not specified, `strptime()` assumes 0 as the year offset, then calculates the year, and assigns `tm_year`.

The following describes extended features when `_STRPTIME_DONTZERO` is defined:

- If `%C` is specified and `%Y` is not specified, `strptime()` assumes the year offset of the year value of the `tm_year` member of the input `tm` struct, then calculates the year and assigns `tm_year`.
- If `%j` is specified and neither `%Y`, `%Y`, nor `%C` are specified, and neither month nor day of month are specified, `strptime()` assumes the year value given by the value of the `tm_year` field of the input `tm` struct. Then, in addition to setting...
strftime(3C)

strftime(3C) uses day-of-year and year values to calculate the month and day-of-month, and assigns tm_mon and tm_mday.

- If %% or %w is specified, and if weekday and/or year are not given, and month and day of month are not given, strftime() will assume the weekday value and/or the year value as the value of the tm_wday field and/or tm_year field of the input tm struct. Then, strftime() will calculate the month and day-of-month and assign tm_mon, tm_mday, and/or tm_year.

- If %p is specified and if hour is not specified, strftime() will reference, and if needed, update the tm_hour member. If the am_pm input is p.m. and the input tm_hour value is between 0 - 11, strftime() will add 12 hours and update tm_hour. If the am_pm input is a.m. and input tm_hour value is between 12 - 23, strftime() will subtract 12 hours and update tm_hour.

RETURN VALUES

Upon successful completion, strftime() returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned.

USAGE

Several “same as” formats, and the special processing of white-space characters are provided in order to ease the use of identical format strings for strftime(3C) and strftime().

The strftime() function tries to calculate tm_year, tm_mon, and tm_mday when given incomplete input. This allows the struct tm created by strftime() to be passed to mktime(3C) to produce a time_t value for dates and times that are representable by a time_t. As an example, since mktime() ignores tm_yday, strftime() calculates tm_mon and tm_mday as well as filling in tm_yday when %j is specified without otherwise specifying a month and day within month.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO
cftime(3C), ctime(3C), getdate(3C), isspace(3C), mktime(3C), setlocale(3C), strftime(3C), attributes(5), environ(5), standards(5)
strsignal(3C)

NAME
strsignal – get name of signal

SYNOPSIS
#include <string.h>

char *strsignal(int sig);

DESCRIPTION
The strsignal() function maps the signal number in sig to a string describing the
signal and returns a pointer to that string. It uses the same set of the messages as
psignal(3C). The returned string should not be overwritten.

RETURN VALUES
The strsignal() function returns NULL if sig is not a valid signal number.

USAGE
If the application is linked with -lint1, messages returned from this function are in
the native language specified by the LC_MESSAGES locale category; see
setlocale(3C).

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
gettext(3C), psignal(3C), setlocale(3C), str2sig(3C), attributes(5)
NAME
strtof, atof – convert string to double-precision number

SYNOPSIS
#include <stdlib.h>

double strtof(const char *str, char **endptr);
double atof(const char *str);

DESCRIPTION
The strtof() 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(3C)); a subject sequence interpreted as a floating-point constant; and a final
string of one or more unrecognized 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.

atof() The atof(str) function call is equivalent to strtof(str, (char **)NULL).

RETURN VALUES
Upon successful completion, strtof() returns the converted value. 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, `HUGE` is returned (according to the sign of the value), and `errno` is set to `ERANGE`. When the `-xc` or `-xa` compilation options are used, `HUGE_VAL` is returned instead of `HUGE`.

If the correct value would cause an underflow, 0 is returned and `errno` is set to `ERANGE`.

If `str` is NaN, then `atof()` returns NaN.

**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.

**USAGE**

Because 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set `errno` to 0, then call `strtod()`, then check `errno` and if it is non-zero, assume an error has occurred.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`isspace(3C)`, `localeconv(3C)`, `scanf(3C)`, `setlocale(3C)`, `strtol(3C)`, attributes(5), standards(5)

**NOTES**

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

The DESCRIPTION and RETURN VALUES sections above are very similar to the wording used by the Single UNIX Specification version 2 and the 1989 C Standard to describe the behavior of the `strtod()` function. Since some users have reported that they find the description confusing, the following notes may be helpful.

1. The `strtod()` function does not modify the string pointed to by `str` and does not `malloc()` space to hold the decomposed portions of the input string.
2. If `endptr` is not `(char **)NULL`, `strtod()` will set the pointer pointed to by `endptr` to the first byte of the "final string of unrecognized characters". (If all input characters were processed, the pointer pointed to by `endptr` will be set to point to the null character at the end of the input string.)
3. If `strtod()` returns 0.0, one of the following occurred:
   a. The "subject sequence" was not an empty string, but evaluated to 0.0. (In this case, `errno` will be left unchanged.)
b. The "subject sequence" was an empty string. (In this case, the Single UNIX Specification version 2 allows errno to be set to EINVAL or to be left unchanged. The C Standard does not specify any specific behavior in this case.)

c. The "subject sequence" specified a numeric value that would cause a floating point underflow. (In this case, errno may be set to ERANGE or may be left unchanged.) Note that the standards do not require that implementations distinguish between these three cases. An application can determine case (b) by making sure that there are no leading white-space characters in the string pointed to by str and giving strtok() an endptr that is not (char **)NULL. If endptr points to the first character of str when strtok() returns, you have detected case (b). Case (c) can be detected by looking for a non-zero digit before the exponent part of the "subject sequence". Note, however, that the decimal-point character is locale-dependent.

4. If strtok() returns +HUGE_VAL or −HUGE_VAL, one of the following occurred:

a. If +HUGE_VAL is returned and errno is set to ERANGE, a floating point overflow occurred while processing a positive value.

b. If −HUGE_VAL is returned and errno is set to ERANGE, a floating point overflow occurred while processing a negative value.

c. If strtok() does not set errno to ERANGE, the value specified by the "subject string" converted to +HUGE_VAL or −HUGE_VAL, respectively. Note that if errno is set to ERANGE when strtok() is called, case (c) is indistinguishable from cases (a) and (b).
string conversion routines

#include <stdlib.h>

long strtol(const char *str, char **endptr, int base);
long long strtoll(const char *str, char **endptr, int base);
long atol(const char *str);
long long atoll(const char *str);
int atoi(const char *str);
char *lltostr(long long value, char *endptr);
char *ulltostr(unsigned long long value, char *endptr);

The **strtol()** function converts the initial portion of the string pointed to by *str* to a type long int representation.

The **strtoll()** function converts the initial portion of the string pointed to by *str* to a type long long representation.

Both functions first decompose the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by *isspace(3C)*); a subject sequence interpreted as an integer represented in some radix determined by the value of *base*; and a final string of one or more unrecognized characters, including the terminating null byte of the input string. They then attempt to convert the subject sequence to an integer and return the result.

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a + or − sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a + or − sign. The letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.
If the subject sequence has the expected form and the value of base is 0, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

In other than the POSIX locale, additional implementation-dependent subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of str is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

### atol(), atoll() and atoi()

Except for behavior on error, atol() is equivalent to: 
```
 strtol(str, (char **)NULL, 10).
```

Except for behavior on error, atoll() is equivalent to: 
```
strtoll(str, (char **)NULL, 10).
```

Except for behavior on error, atoi() is equivalent to: 
```
(int) strtol(str, (char **)NULL, 10).
```

### lltostr() and ulltostr()

The lltostr() function returns a pointer to the string represented by the long long value. The endptr argument is assumed to point to the byte following a storage area into which the decimal representation of value is to be placed as a string. The lltostr() function converts value to decimal and produces the string, and returns a pointer to the beginning of the string. No leading zeros are produced, and no terminating null is produced. The low-order digit of the result always occupies memory position endptr−1. The behavior of lltostr() is undefined if value is negative. A single zero digit is produced if value is 0.

The ulltostr() function is similar to lltostr() except that value is an unsigned long long.

### RETURN VALUES

Upon successful completion, strtol(), strtoll(), atol(), atoll(), and atoi() return the converted value, if any. If no conversion could be performed, strtol() and strtoll() return 0 and errno may be set to EINVAL.

If the correct value is outside the range of representable values, strtol() returns LONG_MAX or LONG_MIN and strtoll() returns LLONG_MAX or LLONG_MIN (according to the sign of the value), and errno is set to ERANGE.

Upon successful completion, lltostr() and ulltostr() return a pointer to the converted string.

### ERRORS

The strtol() and strtoll() functions will fail if:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERANGE</td>
<td>The value to be returned is not representable. The strtol() and strtoll() functions may fail if:</td>
</tr>
</tbody>
</table>
strtol(3C)

EINVAl  The value of base is not supported.

USAGE  Because 0, LONG_MIN, LONG_MAX, LLONG_MIN, and LLONG_MAX are returned on error and are also valid returns on success, an application wishing to check for error situations should set errno to 0, call the function, then check errno and if it is non-zero, assume an error has occurred.

The strtol() function no longer accepts values greater than LONG_MAX or LLONG_MAX as valid input. Use strtoul(3C) instead.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  isalpha(3C), isspace(3C), scanf(3C), strtol(3C), strtoul(3C), attributes(5)
NAME
strtol, strtoull – convert string to unsigned long

SYNOPSIS
#include <stdlib.h>

unsigned long strtol(const char *str, char **endptr, int base);
unsigned long long strtoull(const char *str, char **endptr, int base);

DESCRIPTION
The strtol() function converts the initial portion of the string pointed to by str to a type unsigned long int representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by isspace(3C)); a subject sequence interpreted as an integer represented in some radix determined by the value of base; and a final string of one or more unrecognised characters, including the terminating null byte of the input string. Then it attempts to convert the subject sequence to an unsigned integer, and returns the result.

If the value of base is 0, the expected form of the subject sequence is that of a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a + or - sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 to 15 respectively.

If the value of base is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by base, optionally preceded by a + or - sign. The letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of base are permitted. If the value of base is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of base is 0, the sequence of characters starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

In other than the POSIX locale, additional implementation-dependent subject sequence forms may be accepted.
If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of \textit{str} is stored in the object pointed to by \textit{endptr}, provided that \textit{endptr} is not a null pointer.

The \texttt{strtoull()} function is identical to \texttt{strtoul()} except that it returns the value represented by \textit{str} as an unsigned long long.

**RETURN VALUES**

Upon successful completion \texttt{strtoul()} returns the converted value, if any. If no conversion could be performed, 0 is returned and \textit{errno} may be set to \texttt{EINVAL}. If the correct value is outside the range of representable values, \texttt{ULONG_MAX} is returned and \textit{errno} is set to \texttt{ERANGE}.

**ERRORS**

The \texttt{strtoul()} function will fail if:

- \texttt{EINVAL} The value of \textit{base} is not supported.
- \texttt{ERANGE} The value to be returned is not representable.

The \texttt{strtoul()} function may fail if:

- \texttt{EINVAL} No conversion could be performed.

**USAGE**

Because 0 and \texttt{ULONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set \textit{errno} to 0, then call \texttt{strtoul()}, then check \textit{errno} and if it is non-zero, assume an error has occurred.

Unlike \texttt{strtod(3C)} and \texttt{strtol(3C)}, \texttt{strtoul()} must always return a non-negative number; so, using the return value of \texttt{strtoul()} for out-of-range numbers with \texttt{strtoul()} could cause more severe problems than just loss of precision if those numbers can ever be negative.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

\texttt{isalpha(3C)}, \texttt{isspace(3C)}, \texttt{scanf(3C)}, \texttt{strtod(3C)}, \texttt{strtol(3C)}, attributes(5)
# strtows, wstostr — code conversion for Process Code and File Code

## Synopsis

```c
#include <widec.h>

wchar_t *strtwos(wchar_t *dst, const char *src);
char *wstostr(char *dst, const wchar_t *src);
```

## Description


The `strtwos()` function takes a character string `src`, converts it to a Process Code string, terminated by a Process Code null, and places the result into `dst`.

The `wstostr()` function takes the Process Code string pointed to by `src`, converts it to a character string, and places the result into `dst`.

## Return Values

The `strtwos()` function returns the Process Code string if it completes successfully. Otherwise, a null pointer will be returned and `errno` will be set to `EILSEQ`.

The `wstostr()` function returns the File Code string if it completes successfully. Otherwise, a null pointer will be returned and `errno` will be set to `EILSEQ`.

## See Also

`wstring(3C)`
The `strxfrm()` function transforms the string pointed to by `s2` and places the resulting string into the array pointed to by `s1`. The transformation is such that if `strcmp(3C)` is applied to two transformed strings, it returns a value greater than, equal to or less than 0, corresponding to the result of `strcoll(3C)` applied to the same two original strings. No more than `n` bytes are placed into the resulting array pointed to by `s1`, including the terminating null byte. If `n` is 0, `s1` is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

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 failure, `strxfrm()` returns `(size_t) -1`.

The transformation function is such that two transformed strings can be ordered by `strcmp(3C)` as appropriate to collating sequence information in the program’s locale (category `LC_COLLATE`).

The fact that when `n` is 0, `s1` is permitted to be a null pointer, is useful to determine the size of the `s1` array prior to making the transformation.

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(3C)`, then check `errno` and if it is non-zero, assume an error has occurred.

This issue is aligned with the ANSI C standard; this does not affect compatibility with XPG3 applications. Reliable error detection by this function was never guaranteed.
The `strxfrm()` function can be used safely in a multithreaded application, as long as `setlocale(3C)` is not being called to change the locale.

### Attributes

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

### SEE ALSO

`localedef(1), setlocale(3C), strcmp(3C), strcoll(3C), wscoll(3C), attributes(5), environ(5), standards(5)`

### NOTES

The `strxfrm()` function can be used safely in a multithreaded application, as long as `setlocale(3C)` is not being called to change the locale.
The `swab()` function copies `nbytes` bytes, which are pointed to by `src`, to the object pointed to by `dest`, exchanging adjacent bytes. The `nbytes` argument should be even. If `nbytes` is odd `swab()` copies and exchanges `nbytes-1` bytes and the disposition of the last byte is unspecified. If copying takes place between objects that overlap, the behavior is undefined. If `nbytes` is negative, `swab()` does nothing.

No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

See also attributes(5), standards(5)
The `sync_instruction_memory()` function performs whatever steps are required to make instructions modified by a program executable.

Some processor architectures, including some SPARC processors, have separate and independent instruction and data caches which are not kept consistent by hardware. For example, if the instruction cache contains an instruction from some address and the program then stores a new instruction at that address, the new instruction may not be immediately visible to the instruction fetch mechanism. Software must explicitly invalidate the instruction cache entries for new or changed mappings of pages that might contain executable instructions. The `sync_instruction_memory()` function performs this function, and/or any other functions needed to make modified instructions between `addr` and `addr+len` visible. A program should call `sync_instruction_memory()` after modifying instructions and before executing them.

On processors with unified caches (one cache for both instructions and data) and pipelines which are flushed by a branch instruction, such as the x86 architecture, the function may do nothing and just return.

The changes are immediately visible to the thread calling `sync_instruction_memory()` when the call returns, even if the thread should migrate to another processor during or after the call. The changes become visible to other threads in the same manner that stores do; that is, they eventually become visible, but the latency is implementation-dependent.

The result of executing `sync_instruction_memory()` are unpredictable if `addr` through `addr+len-1` are not valid for the address space of the program making the call.

No values are returned.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

See also attributes(5)
NAME
syscall – indirect system call

SYNOPSIS
/usr/ucb/cc [ flag ... ] file ...
#include <sys/syscall.h>

int syscall(number, arg, ...);

DESCRIPTION
syscall() performs the function whose assembly language interface has the
specified number, and arguments arg ... Symbolic constants for functions can be
found in the header <sys/syscall.h>.

RETURN VALUES
On error syscall() returns -1 and sets the external variable errno (see intro(2)).

FILES
<sys/syscall.h>

SEE ALSO
intro(2), pipe(2)

NOTES
Use of these interfaces should be restricted to only applications written on BSD
platforms. Use of these interfaces with any of the system libraries or in multi-thread
applications is unsupported.

WARNINGS
There is no way to use syscall() to call functions such as pipe(2) which return
values that do not fit into one hardware register.

Since many system calls are implemented as library wrappers around traps to the
kernel, these calls may not behave as documented when called from syscall(),
which bypasses these wrappers. For these reasons, using syscall() is not
recommended.
The `sysconf()` function provides a method for an application to determine
the current value of a configurable system limit or option (variable).

The `name` argument represents the system variable to be queried. The following table
lists the minimal set of system variables from `<limits.h>` and `<unistd.h>` that can
be returned by `sysconf()` and the symbolic constants defined in `<unistd.h>` that
are the corresponding values used for `name` on the SPARC and x86 platforms.

<table>
<thead>
<tr>
<th>Name</th>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SC_2_C_BIND</td>
<td>_POSIX2_C_BIND</td>
<td>Supports the C language binding option</td>
</tr>
<tr>
<td>_SC_2_C_DEV</td>
<td>_POSIX2_C_DEV</td>
<td>Supports the C language development utilities option</td>
</tr>
<tr>
<td>_SC_2_C_VERSION</td>
<td>_POSIX2_C_VERSION</td>
<td>Integer value indicates version of ISO POSIX-2 standard (Commands)</td>
</tr>
<tr>
<td>_SC_2_CHAR_TERM</td>
<td>_POSIX2_CHAR_TERM</td>
<td>Supports at least one terminal</td>
</tr>
<tr>
<td>_SC_2_FORT_DEV</td>
<td>_POSIX2_FORT_DEV</td>
<td>Supports FORTRAN Development Utilities Option</td>
</tr>
<tr>
<td>_SC_2_FORT_RUN</td>
<td>_POSIX2_FORT_RUN</td>
<td>Supports FORTRAN Run-time Utilities Option</td>
</tr>
<tr>
<td>_SC_2_LOCALEDEF</td>
<td>_POSIX2_LOCALEDEF</td>
<td>Supports creation of locales by the localedef utility</td>
</tr>
<tr>
<td>_SC_2_SW_DEV</td>
<td>_POSIX2_SW_DEV</td>
<td>Supports Software Development Utility Option</td>
</tr>
<tr>
<td>_SC_2_UPE</td>
<td>_POSIX2_UPE</td>
<td>Supports User Portability Utilities Option</td>
</tr>
<tr>
<td>_SC_2_VERSION</td>
<td>_POSIX2_VERSION</td>
<td>Integer value indicates version of ISO POSIX-2 standard (C language binding)</td>
</tr>
<tr>
<td>_SC_AIO_LISTIO_MAX</td>
<td>AIO_LISTIO_MAX</td>
<td>Max number of I/O operations in a single list I/O call supported</td>
</tr>
<tr>
<td>_SC_AIO_MAX</td>
<td>AIO_MAX</td>
<td>Max number of outstanding asynchronous I/O operations supported</td>
</tr>
<tr>
<td>_SC_AIO_PRIO_DELTA_MAX</td>
<td>AIO_PRIO_DELTA_MAX</td>
<td>Max amount by which process can decrease</td>
</tr>
</tbody>
</table>
its asynchronous
I/O priority level
from its own
scheduling priority
Max size of argv[]
plus envp[]
Asynchronous I/O
Max number of
functions that can
be registered with
atexit()
Number of physical
memory pages not
currently in use by
system
Maximum obase values
allowed by bc
Max number of
elements permitted
in array by bc
Max scale value
allowed by bc
Max length of string
constant allowed by
bc
Max processes
allowed to a UID
Ticks per second
(cclock_t)
Max number of
weights that can be
assigned to entry of
the LC_COLLATE order
keyword in locale
definition file
Max possible
processor ID
Max number of timer
expiration overruns
Max number of
parentheses by expr
Supports File
Synchronization
Max size of group
entry buffer
Max size of password
entry buffer
Max number of iovec
structures available
to one process for
use with readv() and
writev()
Job control
supported?
Max length of input
line
Max length of login

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SC_ARG_MAX</td>
<td>ARG_MAX Max size of argv[] plus envp[]</td>
</tr>
<tr>
<td>_SC_ASYNCIO_MAX</td>
<td>_POSIX_ASYNCHRONOUS_IO Supports Asynchronous I/O</td>
</tr>
<tr>
<td>_SC_ATEXIT_MAX</td>
<td>ATEXIT_MAX Max number of functions that can be registered with atexit()</td>
</tr>
<tr>
<td>_SC_AVPHYS_PAGES</td>
<td>Number of physical memory pages not currently in use by system</td>
</tr>
<tr>
<td>_SC_BC_BASE_MAX</td>
<td>BC_BASE_MAX Maximum obase values allowed by bc</td>
</tr>
<tr>
<td>_SC_BC_DIM_MAX</td>
<td>BC_DIM_MAX Max number of elements permitted in array by bc</td>
</tr>
<tr>
<td>_SC_BC_SCALE_MAX</td>
<td>BC_SCALE_MAX Max scale value allowed by bc</td>
</tr>
<tr>
<td>_SC_BC_STRING_MAX</td>
<td>BC_STRING_MAX Max length of string constant allowed by bc</td>
</tr>
<tr>
<td>_SC_CHILD_MAX</td>
<td>CHILD_MAX Max processes allowed to a UID</td>
</tr>
<tr>
<td>_SC_CLK_TCK</td>
<td>CLK_TCK Ticks per second (clock_t)</td>
</tr>
<tr>
<td>_SC_COLL_WEIGHTS_MAX</td>
<td>COLL_WEIGHTS_MAX Max number of weights that can be assigned to entry of the LC_COLLATE order keyword in locale definition file</td>
</tr>
<tr>
<td>_SC_CPUID_MAX</td>
<td>Max possible processor ID</td>
</tr>
<tr>
<td>_SC_DELAYTIMER_MAX</td>
<td>DELAYTIMER_MAX Max number of timer expiration overruns</td>
</tr>
<tr>
<td>_SC_EXPR_NEST_MAX</td>
<td>EXPR_NEST_MAX Max number of parentheses by expr</td>
</tr>
<tr>
<td>_SC_FSYNC</td>
<td>_POSIX_FSYNC Supports File Synchronization</td>
</tr>
<tr>
<td>_SC_GETGR_R_SIZE_MAX</td>
<td>NSS_BUFLEN_GROUP Max size of group entry buffer</td>
</tr>
<tr>
<td>_SC_GETPW_R_SIZE_MAX</td>
<td>NSS_BUFLEN_PASSWD Max size of password entry buffer</td>
</tr>
<tr>
<td>_SC_IOV_MAX</td>
<td>IOV_MAX Max number of iovec structures available to one process for use with readv() and writev()</td>
</tr>
<tr>
<td>_SC_JOB_CONTROL</td>
<td>_POSIX_JOB_CONTROL Job control supported?</td>
</tr>
<tr>
<td>_SC_LINE_MAX</td>
<td>LINE_MAX Max length of input line</td>
</tr>
<tr>
<td>_SC/Login_Name_MAX</td>
<td>LOGNAME_MAX + 1 Max length of login</td>
</tr>
<tr>
<td>name</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>_SC_LOGNAME_MAX</td>
<td>LOGNAME_MAX</td>
</tr>
<tr>
<td>_SC_MAPPED_FILES</td>
<td>_POSIX_MAPPED_FILES</td>
</tr>
<tr>
<td>_SC_MMAP</td>
<td>_POSIX_MEMLOCK</td>
</tr>
<tr>
<td>_SC_MEMLOCK_RANGE</td>
<td>_POSIX_MEMLOCK_RANGE</td>
</tr>
<tr>
<td>_SC_MEMORY_PROTECTION</td>
<td>_POSIX_MEMORY_PROTECTION</td>
</tr>
<tr>
<td>_SC_MESSAGE_PASSING</td>
<td>_POSIX_MESSAGE_PASSING</td>
</tr>
<tr>
<td>_SC_MQ_OPEN_MAX</td>
<td>MQ_OPEN_MAX</td>
</tr>
<tr>
<td>_SC_MQ_PRIO_MAX</td>
<td>MQ_PRIO_MAX</td>
</tr>
<tr>
<td>_SC_NGROUPS_MAX</td>
<td>NGROUPS_MAX</td>
</tr>
<tr>
<td>_SC_NGROUPS_MAX</td>
<td>NGROUPS_MAX</td>
</tr>
<tr>
<td>_SC_NPROCESSORS_CONF</td>
<td>Number of processors configured</td>
</tr>
<tr>
<td>_SC_NPROCESSORS_MAX</td>
<td>Max number of processors supported by platform</td>
</tr>
<tr>
<td>_SC_NPROCESSORS_ONLN</td>
<td>Number of processors online</td>
</tr>
<tr>
<td>_SC_OPEN_MAX</td>
<td>OPEN_MAX</td>
</tr>
<tr>
<td>_SC_PAGESIZE</td>
<td>PAGESIZE</td>
</tr>
<tr>
<td>_SC_PAGE_SIZE</td>
<td>PAGESIZE</td>
</tr>
<tr>
<td>_SC_PASS_MAX</td>
<td>PASS_MAX</td>
</tr>
<tr>
<td>_SC_PHYS_PAGES</td>
<td>Total number of pages of physical memory in system</td>
</tr>
<tr>
<td>_SC_PRIORITY_SCHEDULING</td>
<td>_POSIX_PRIORITY_SCHEDULING</td>
</tr>
<tr>
<td>_SC_REALTIME_SIGNALS</td>
<td>_POSIX_REALTIME_SIGNALS</td>
</tr>
<tr>
<td>_SC_RTSIG_MAX</td>
<td>RTSIG_MAX</td>
</tr>
<tr>
<td>_SC_SAVED_IDS</td>
<td>_POSIX_SAVED_IDS</td>
</tr>
</tbody>
</table>

Supports Memory Mapped Files
Max pid value
Supports Process Memory Locking
Supports Range Memory Locking
Supports Memory Protection
Supports Message Passing
Max number of open message queues a process can hold
Max number of message priorities supported
Max simultaneous groups to which one can belong
Number of processors configured
Max number of processors supported by platform
Number of processors online
Max open files per process
System memory page size
Same as _SC_PAGESIZE
Max number of significant bytes in a password
Total number of pages of physical memory in system
Supports Prioritized I/O
Supports Process Scheduling
Max number of repeated occurrences of a regular expression permitted when using interval notation \{m,n\}
Supports Realtime Signals
Max number of realtime signals reserved for application use
Saved IDs (seteuid())
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Supported?</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>_SC_SEM_NSEMS_MAX</td>
<td>SEM_NSEMS_MAX Max number of POSIX semaphores a process can have</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_SEM_VALUE_MAX</td>
<td>SEM_VALUE_MAX Max value a POSIX semaphore can have</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_SEMAPHORES</td>
<td>POSIX_SEMAPHORES Supports Semaphores</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_SHARED_MEMORY_OBJECTS</td>
<td>POSIX_SHARED_MEMORY_OBJECTS Memory Objects</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_SIGQUEUE_MAX</td>
<td>SIGQUEUE_MAX Max number of queued signals that a process can send and have pending at receiver(s) at a time</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_STACK_PROT</td>
<td>STREAM_MAX Default stack protection</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_STREAM_MAX</td>
<td>STREAM_MAX Number of streams one process can have open at a time</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_SUSTRUCTURED_IO</td>
<td>POSIX_SUSTRUCTURED_IO Supports Synchronized I/O</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_THREAD_ATTR_STACKADDR</td>
<td>POSIX_THREAD_ATTR_STACKADDR Stack Address</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_THREAD_ATTR_STACKSIZE</td>
<td>POSIX_THREAD_ATTR_STACKSIZE Stack Size</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_THREAD_DESTRUCTOR_ITERATIONS</td>
<td>PTHREAD_DESTRUCTOR_ITERATIONS Number attempts made to destroy thread-specific data on thread exit</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_THREAD_KEYS_MAX</td>
<td>PTHREAD_KEYS_MAX Max number of data keys per process</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_THREAD_PRIORITY_PROTECT</td>
<td>POSIX_THREAD_PRIORITY_PROTECT Supports Priority Protection option</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_THREAD_PRIORITY_SCHEDULING</td>
<td>POSIX_THREAD_PRIORITY_SCHEDULING Supports Thread Scheduling option</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_THREAD_PROCESS_SHARED</td>
<td>POSIX_THREAD_PROCESS_SHARED Process-Shared Synchronization option</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_THREAD_STACK_MIN</td>
<td>PTHREAD_STACK_MIN Min byte size of thread stack storage</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_THREAD_THREADS_MAX</td>
<td>PTHREAD_THREADS_MAX Max number of threads per process supported</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_THREADS</td>
<td>POSIX_THREADS Supports Threads option</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_TIMER_MAX</td>
<td>TIMER_MAX Max number of timer per process supported</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_TIMERS</td>
<td>POSIX_TIMERS Supports Timers</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>_SC_TTY_NAME_MAX</td>
<td>TTYNAME_MAX Max length of tty</td>
<td>supported?</td>
<td></td>
</tr>
<tr>
<td>Symbol</td>
<td>Definition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_TZNAME_MAX</td>
<td>Max number of bytes supported for name of a time zone</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_VERSION</td>
<td>POSIX.1 version supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_XBS5_ILP32_OFF32</td>
<td>Indicates support for X/Open ILP32 w/32-bit offset build environment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_XBS5_ILP32_OFFBIG</td>
<td>Indicates support for X/Open ILP32 w/64-bit offset build environment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_XBS5_LP64_OFF64</td>
<td>Indicates support of X/Open LP64, 64-bit offset build environment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_XBS5_LP64_OFF64</td>
<td>Same as _SC_XBS5_LP64_OFF64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_XOPEN_CRYPT</td>
<td>Supports X/Open Encryption Feature Group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_XOPEN_ENH_I18N</td>
<td>Supports X/Open Enhanced Internationalization Feature Group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_XOPEN_LEGACY</td>
<td>Supports X/Open Legacy Feature Group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_XOPEN_REALTIME</td>
<td>Supports X/Open POSIX Realtime Feature Group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_XOPEN_REALTIME_THREADS</td>
<td>Supports X/Open POSIX Realtime Threads Feature Group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_XOPEN_SHM</td>
<td>Supports X/Open Shared Memory Feature Group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_XOPEN_UNIX</td>
<td>Supports X/Open CAE Specification, August 1994, System Interfaces and</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Headers, Issue 4, Version 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_XOPEN_VERSION</td>
<td>Integer value indicates version of X/Open Portability Guide to which</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>implementation conforms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>_SC_XOPEN_XCU VERSION</td>
<td>Integer value indicates version of XCU specification to which implementation conforms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Upon successful completion, \texttt{sysconf()} returns the current variable value on the system. The value returned will not be more restrictive than the corresponding value described to the application when it was compiled with the implementation's \texttt{<limits.h>}, \texttt{<unistd.h>} or \texttt{<time.h>}. The value will not change during the lifetime of the calling process.

If \textit{name} is an invalid value, \texttt{sysconf()} returns \texttt{-1} and sets \texttt{errno} to indicate the error. If the variable corresponding to \textit{name} is associated with functionality that is not supported by the system, \texttt{sysconf()} returns \texttt{-1} without changing the value of \texttt{errno}.

Calling \texttt{sysconf()} with the following returns \texttt{-1} without setting \texttt{errno}, because no maximum limit can be determined. The system supports at least the minimum values and can support higher values depending upon system resources.

\begin{verbatim}
Variable                  Minimum supported value
_SC_AIO_MAX               _POSIX_AIO_MAX
_SC_ATEXIT_MAX            32
_SC_THREAD_THREADS_MAX    _POSIX_THREAD_THREADS_MAX
_SC_THREAD_KEYS_MAX       _POSIX_THREAD_KEYS_MAX
_SC_THREAD_DESTRUCTOR_ITERATIONS _POSIX_THREAD_DESTRUCTOR_ITERATIONS
\end{verbatim}

The following SPARC and x86 platform variables return \texttt{EINVAL}:

\begin{verbatim}
_SC_COHER_BLKSIZE   _SC_DCACHE_ASSOC
_SC_DCACHE_BLKSIZE _SC_DCACHE_LINESZ
_SC_DCACHE_SZ      _SC_DCACHE_TBLKSZ
_SC_ICACHE_ASSOC    _SC_ICACHE_BLKSIZE
_SC_ICACHE_LINESZ  _SC_ICACHE_SZ
_SC_SPLIT_CACHE
\end{verbatim}

\textbf{ERRORS}  

The \texttt{sysconf()} function will fail if:

- \texttt{EINVAL} The value of the \textit{name} argument is invalid.

\textbf{ATTRIBUTES}  

See \texttt{attributes(5)} for descriptions of the following attributes:

\begin{verbatim}
<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>SPARC and x86</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe, Async-Signal-Safe</td>
</tr>
</tbody>
</table>
\end{verbatim}

\textbf{SEE ALSO}  

\texttt{fpathconf(2), seteuid(2), setrlimit(2), attributes(5), standards(5)}

\textbf{NOTES}  

A call to \texttt{setrlimit()} can cause the value of \texttt{OPEN_MAX} to change.

Multiplying \texttt{sysconf(_SC_PHYS_PAGES)} or \texttt{sysconf(_SC_AVPHYS_PAGES)} by \texttt{sysconf(_SC_PAGESIZE)} to determine memory amount in bytes can exceed the maximum values representable in a long or unsigned long.

The value of \texttt{CLK_TCK} can be variable and it should not be assumed that \texttt{CLK_TCK} is a compile-time constant.
The \_SC\_PHYS\_PAGES and \_SC\_AVPHYS\_PAGES variables are specific to Solaris 2.3 or compatible releases.
syslog(3C)

NAME  syslog, openlog, closelog, setlogmask – control system log

SYNOPSIS
#include <syslog.h>

void openlog(const char *ident, int logopt, int facility);
void syslog(int priority, const char *message, .../* arguments */);
void closelog(void);
int setlogmask(int maskpri);

DESCRIPTION
The syslog() function sends a message to syslogd(1M), which, depending on the
configuration of /etc/syslog.conf, logs it in an appropriate system log, writes it to
the system console, forwards it to a list of users, or forwards it to syslogd on another
host over the network. The logged message includes a message header and a message
body. The message header consists of a facility indicator, a severity level indicator, a
timestamp, a tag string, and optionally the process ID.

The message body is generated from the message and following arguments in the same
manner as if these were arguments to printf(3UCB), except that occurrences of %m in
the format string pointed to by the message argument are replaced by the error
message string associated with the current value of errno. A trailing NEWLINE
character is added if needed.

Values of the priority argument are formed by ORing together a severity level value and
an optional facility value. If no facility value is specified, the current default facility
value is used.

Possible values of severity level include:

<table>
<thead>
<tr>
<th>Severity Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG_EMERG</td>
<td>A panic condition. This is normally broadcast to all users.</td>
</tr>
<tr>
<td>LOG_ALERT</td>
<td>A condition that should be corrected immediately, such as a corrupted system database.</td>
</tr>
<tr>
<td>LOG_CRIT</td>
<td>Critical conditions, such as hard device errors.</td>
</tr>
<tr>
<td>LOG_ERR</td>
<td>Errors.</td>
</tr>
<tr>
<td>LOG_WARNING</td>
<td>Warning messages.</td>
</tr>
<tr>
<td>LOG_NOTICE</td>
<td>Conditions that are not error conditions, but that may require special handling.</td>
</tr>
<tr>
<td>LOG_INFO</td>
<td>Informational messages.</td>
</tr>
<tr>
<td>LOG_DEBUG</td>
<td>Messages that contain information normally of use only when debugging a program.</td>
</tr>
</tbody>
</table>

The facility indicates the application or system component generating the message.
Possible facility values include:
LOG_KERN  Messages generated by the kernel. These cannot be
generated by any user processes.

LOG_USER  Messages generated by random user processes. This is
the default facility identifier if none is specified.

LOG_MAIL  The mail system.

LOG_DAEMON  System daemons, such as in.ftpd(1M).

LOG_AUTH  The authorization system: login(1), su(1M),
getty(1M).

LOG_LPR  The line printer spooling system: lpr(1B), lpc(1B).

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(1), at(1), cron(1M).

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 openlog() function sets process attributes that affect subsequent calls to
syslog(). The ident argument is a string that is prepended to every message. The
logopt argument indicates logging options. Values for logopt are constructed by a
bitwise-inclusive OR of zero or more of the following:

LOG_PID  Log the process ID with each message. This is useful
for identifying specific daemon processes (for daemons
that fork).

LOG_CONS  Write messages to the system console if they cannot be
sent to syslogd(1M). This option is safe to use in
daemon processes that have no controlling terminal,
since syslog() forks before opening the console.

LOG_NDELAY  Open the connection to syslogd(1M) immediately.
Normally the open is delayed until the first message is
logged. This is useful for programs that need to manage the order in which file descriptors are allocated.

**LOG_ODELAY**

Delay open until `syslog()` is called.

**LOG_NOWAIT**

Do not wait for child processes that have been forked to log messages onto the console. This option should be used by processes that enable notification of child termination using `SIGCHLD`, since `syslog()` may otherwise block waiting for a child whose exit status has already been collected.

The facility argument encodes a default facility to be assigned to all messages that do not have an explicit facility already encoded. The initial default facility is `LOG_USER`.

The `openlog()` and `syslog()` functions may allocate a file descriptor. It is not necessary to call `openlog()` prior to calling `syslog()`.

The `closelog()` function closes any open file descriptors allocated by previous calls to `openlog()` or `syslog()`.

The `setlogmask()` function sets the log priority mask for the current process to `maskpri` and returns the previous mask. If the `maskpri` argument is 0, the current log mask is not modified. Calls by the current process to `syslog()` with a priority not set in `maskpri` are rejected. The mask for an individual priority `pri` is calculated by the macro `LOG_MASK(pri)`; the mask for all priorities up to and including `toppri` is given by the macro `LOG_UPT(toppri)`. The default log mask allows all priorities to be logged.

Symbolic constants for use as values of the `logopt`, `facility`, `priority`, and `maskpri` arguments are defined in the `<syslog.h>` header.

**RETURN VALUES**

The `setlogmask()` function returns the previous log priority mask. The `closelog()`, `openlog()` and `syslog()` functions return no value.

**ERRORS**

No errors are defined.

**EXAMPLES**

**EXAMPLE 1** Example of LOG_ALERT message.

This call logs a message at priority LOG_ALERT:

```c
syslog(LOG_ALERT, "who: internal error 23");
```

The FTP daemon `ftpd` would make this call to `openlog()` to indicate that all messages it logs should have an identifying string of `ftpd`, should be treated by `syslogd(1M)` as other messages from system daemons are, should include the process ID of the process logging the message:

```c
openlog("ftpd", LOG_PID, LOG_DAEMON);
```
EXAMPLE 1 Example of LOG_ALERT message.  (Continued)

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(1M) as other messages to the facility
LOG_LOCAL2 are:

    syslog(LOG_INFO|LOG_LOCAL2, "error: %m");

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  at(1), crontab(1), logger(1), login(1), lpc(1B), lpr(1B), cron(1M), getty(1M),
in.ftpd(1M), su(1M), syslogd(1M), printf(3UCB), syslog.conf(4), attributes(5)
#include <stdlib.h>

int system(const char *string);

system() function causes string to be given to the shell as input, as if string had been typed as a command at a terminal. The invoker waits until the shell has completed, then returns the exit status of the shell in the format specified by waitpid(2).

If string is a null pointer, system() checks if the shell exists and is executable. If the shell is available, system() returns a non-zero value; otherwise, it returns 0. If the application is standard-conforming (see standards(5)), system() uses /usr/xpg4/bin/sh (see ksh(1)); otherwise system() uses /usr/bin/sh (see sh(1)).

The system() function executes vfork(2) to create a child process that in turn invokes one of the exec family of functions (see exec(2)) on the shell to execute string. If vfork() or the exec function fails, system() returns −1 and sets errno to indicate the error.

The system() function manipulates the signal handlers for SIGINT, SIGQUIT, and SIGCHLD. For this reason it is not safe to call system() in a multithreaded process. Concurrent calls to system() will interfere destructively with the disposition of these signals, even if they are not manipulated by other threads in the application. See popen(3C) for a replacement for system() that is thread-safe.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Unsafe</td>
</tr>
</tbody>
</table>

SEE ALSO

ksh(1), sh(1), exec(2), vfork(2), waitpid(2), popen(3C), attributes(5), standards(5)
NAME

tcdrain – wait for transmission of output

SYNOPSIS

#include <termios.h>

int tcdrain(int fildes);

DESCRIPTION

The tcdrain() function waits until all output written to the object referred to by fildes is transmitted. The fildes argument is an open file descriptor associated with a terminal.

Any attempts to use tcdrain() from a process which is a member of a background process group on a fildes associated with its controlling terminal, will cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation, and no signal is sent.

RETURN VALUES

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

ERRORS

The tcdrain() function will fail if:

EBADF  The fildes argument is not a valid file descriptor.
EINTR  A signal interrupted tcdrain().
ENOTTY  The file associated with fildes is not a terminal.

The tcdrain() function may fail if:

EIO  The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe, and Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

tcflush(3C), attributes(5), termio(7I)
**NAME**
tcflow — suspend or restart the transmission or reception of data

**SYNOPSIS**
```c
#include <termios.h>

int tcflow(int fildes, int action);
```

**DESCRIPTION**
The `tcflow()` function suspends transmission or reception of data on the object referred to by `fildes`, depending on the value of `action`. The `fildes` argument is an open file descriptor associated with a terminal.

- If `action` is TCOOFF, output is suspended.
- If `action` is TCOON, suspended output is restarted.
- If `action` is TCIOFF, the system transmits a STOP character, which is intended to cause the terminal device to stop transmitting data to the system.
- If `action` is TCION, the system transmits a START character, which is intended to cause the terminal device to start transmitting data to the system.

The default on the opening of a terminal file is that neither its input nor its output are suspended.

Attempts to use `tcflow()` from a process which is a member of a background process group on a `fildes` associated with its controlling terminal, will cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation, and no signal is sent.

**RETURN VALUES**
Upon successful completion, 0 is returned. Otherwise, -1 is returned and `errno` is set to indicate the error.

**ERRORS**
The `tcflow()` function will fail if:

- EBADF       The `fildes` argument is not a valid file descriptor.
- EINVAL      The `action` argument is not a supported value.
- ENOTTY      The file associated with `fildes` is not a terminal.

The `tcflow()` function may fail if:

- EIO         The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe, Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**
tcsendbreak(3C), attributes(5), termio(7I)
NAME
tcflush – flush non-transmitted output data, non-read input data or both

SYNOPSIS
#include <termios.h>

int tcflush(int fildes, int queue_selector);

DESCRIPTION
Upon successful completion, tcflush() discards data written to the object referred to by fildes (an open file descriptor associated with a terminal) but not transmitted, or data received but not read, depending on the value of queue_selector:

- If queue_selector is TCIFLUSH it flushes data received but not read.
- If queue_selector is TCOFLUSH it flushes data written but not transmitted.
- If queue_selector is TCIOFLUSH it flushes both data received but not read and data written but not transmitted.

Attempts to use tcflush() from a process which is a member of a background process group on a fildes associated with its controlling terminal, will cause the process group to be sent a SIGTTOU signal. If the calling process is blocking or ignoring SIGTTOU signals, the process is allowed to perform the operation, and no signal is sent.

RETURN VALUES
Upon successful completion, 0 is returned. Otherwise, −1 is returned and errno is set to indicate the error.

ERRORS
The tcflush() function will fail if:

- EBADF The fildes argument is not a valid file descriptor.
- EINVAL The queue_selector argument is not a supported value.
- ENOTTY The file associated with fildes is not a terminal.

The tcflush() function may fail if:

- EIO The process group of the writing process is orphaned, and the writing process is not ignoring or blocking SIGTTOU.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe, and Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO tcdrain(3C), attributes(5), termio(7I)
NAME  
tcgetattr – get the parameters associated with the terminal

SYNOPSIS
#include <termios.h>

int tcgetattr(int fildes, struct termios *termios_p);

DESCRIPTION
The tcgetattr() function gets the parameters associated with the terminal referred to by fildes and stores them in the termios structure (see termio(7I)) referenced by termios_p. The fildes argument is an open file descriptor associated with a terminal.

The termios_p argument is a pointer to a termios structure.

The tcgetattr() operation is allowed from any process.

If the terminal device supports different input and output baud rates, the baud rates stored in the termios structure returned by tcgetattr() reflect the actual baud rates, even if they are equal. If differing baud rates are not supported, the rate returned as the output baud rate is the actual baud rate. If the terminal device does not support split baud rates, the input baud rate stored in the termios structure will be 0.

RETURN VALUES
Upon successful completion, 0 is returned. Otherwise, −1 is returned and errno is set to indicate the error.

ERRORS
The tcgetattr() function will fail if:

EBADF       The fildes argument is not a valid file descriptor.
ENOTTY      The file associated with fildes is not a terminal.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe, and Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
tcsetattr(3C), attributes(5), termio(7I)
The `tcgetpgrp()` function will return the value of the process group ID of the foreground process group associated with the terminal. If there is no foreground process group, `tcgetpgrp()` returns a value greater than 1 that does not match the process group ID of any existing process group.

The `tcgetpgrp()` function is allowed from a process that is a member of a background process group; however, the information may be subsequently changed by a process that is a member of a foreground process group.

Upon successful completion, `tcgetpgrp()` returns the value of the process group ID of the foreground process associated with the terminal. Otherwise, −1 is returned and `errno` is set to indicate the error.

The `tcgetpgrp()` function will fail if:
- `EBADF` The `fd` 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 `attributes(5)` for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe, and Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**
- `setpgid(2)`, `setsid(2)`, `tcsetpgrp(3C)`, `attributes(5)`, `termio(7I)`
tcgetsid(3C)

NAME
tcgetsid – get process group ID for session leader for controlling terminal

SYNOPSIS
#include <termios.h>

pid_t tcgetsid(int fildes);

DESCRIPTION
The tcgetsid() function obtains the process group ID of the session for which the terminal specified by fildes is the controlling terminal.

RETURN VALUES
Upon successful completion, tcgetsid() returns the process group ID associated with the terminal. Otherwise, a value of (pid_t)-1 is returned and errno is set to indicate the error.

ERRORS
The tcgetsid() function will fail if:

EACCES The fildes argument is not associated with a controlling terminal.
EBADF The fildes argument is not a valid file descriptor.
ENOTTY The file associated with fildes is not a terminal.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
attributes(5), termio(7I)
tcsendbreak – send a “break” for a specific duration

SYNOPSIS

```c
#include <termios.h>

int tcsendbreak(int fildes, int duration);
```

DESCRIPTION

The `fildes` argument is an open file descriptor associated with a terminal.

If the terminal is using asynchronous serial data transmission, `tcsendbreak()` will cause transmission of a continuous stream of zero-valued bits for a specific duration. If `duration` is 0, it will cause transmission of zero-valued bits for at least 0.25 seconds, and not more than 0.5 seconds. If `duration` is not 0, it behaves in a way similar to `tcdrain(3C)`.

If the terminal is not using asynchronous serial data transmission, it sends data to generate a break condition or returns without taking any action.

Attempts to use `tcsendbreak()` from a process which is a member of a background process group on a `fildes` associated with its controlling terminal will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation, and no signal is sent.

RETURN VALUES

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

ERRORS

The `tcsendbreak()` function will fail if:

- `EBADF` The `fildes` argument is not a valid file descriptor.
- `ENOTTY` The file associated with `fildes` is not a terminal.

The `tcsendbreak()` function may fail if:

- `EIO` The process group of the writing process is orphaned, and the writing process is not ignoring or blocking `SIGTTOU`.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe, and Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO `tcdrain(3C)`, `attributes(5)`, `termio(7I)`
**NAME**

tcsetattr – set the parameters associated with the terminal

**SYNOPSIS**

```c
#include <termios.h>

int tcsetattr(int filedes, int optional_actions, const struct termios *termios_p);
```

**DESCRIPTION**

The `tcsetattr()` function sets the parameters associated with the terminal referred to by the open file descriptor `filedes` (an open file descriptor associated with a terminal) from the `termios` structure (see `termios(7I)`) referenced by `termios_p` as follows:

- If `optional_actions` is `TCSANOW`, the change will occur immediately.
- If `optional_actions` is `TCSADRAIN`, the change will occur after all output written to `filedes` is transmitted. This function should be used when changing parameters that affect output.
- If `optional_actions` is `TCSAFLUSH`, the change will occur after all output written to `filedes` is transmitted, and all input so far received but not read will be discarded before the change is made.

If the output baud rate stored in the `termios` structure pointed to by `termios_p` is the zero baud rate, B0, the modem control lines will no longer be asserted. Normally, this will disconnect the line.

If the input baud rate stored in the `termios` structure pointed to by `termios_p` is 0, the input baud rate given to the hardware will be the same as the output baud rate stored in the `termios` structure.

The `tcsetattr()` function will return successfully if it was able to perform any of the requested actions, even if some of the requested actions could not be performed. It will set all the attributes that implementation supports as requested and leave all the attributes not supported by the implementation unchanged. If no part of the request can be honoured, it will return -1 and set `errno` to `EINVAL`. If the input and output baud rates differ and are a combination that is not supported, neither baud rate is changed. A subsequent call to `tcgetattr(3C)` will return the actual state of the terminal device (reflecting both the changes made and not made in the previous `tcsetattr()` call). The `tcsetattr()` function will not change the values in the `termios` structure whether or not it actually accepts them.

The effect of `tcsetattr()` is undefined if the value of the `termios` structure pointed to by `termios_p` was not derived from the result of a call to `tcgetattr(3C)` on `filedes`; an application should modify only fields and flags defined by this document between the call to `tcgetattr(3C)` and `tcsetattr()`, leaving all other fields and flags unmodified.

No actions defined by this document, other than a call to `tcsetattr()` or a close of the last file descriptor in the system associated with this terminal device, will cause any of the terminal attributes defined by this document to change.
Attempts to use `tcsetattr()` from a process which is a member of a background process group on a `fd` associated with its controlling terminal, will cause the process group to be sent a `SIGTTOU` signal. If the calling process is blocking or ignoring `SIGTTOU` signals, the process is allowed to perform the operation, and no signal is sent.

**Usage**
If trying to change baud rates, applications should call `tcsetattr()` then call `tcgetattr()` in order to determine what baud rates were actually selected.

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

**Errors**
The `tcsetattr()` function will fail if:
- **EBADF** The `fd` argument is not a valid file descriptor.
- **EINVAL** The `optional_actions` argument is not a supported value, or an attempt was made to change an attribute represented in the `termios` structure to an unsupported value.
- **ENOTTY** The file associated with `fd` is not a terminal.

The `tcsetattr()` function may fail if:
- **EIO** The process group of the writing process is orphaned, and the writing process is not ignoring or blocking `SIGTTOU`.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe, and Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**See Also**
`cfgetispeed(3C), tcgetattr(3C), attributes(5), termio(7I)`
tcsetpgrp(3C)

NAME

tcsetpgrp – set foreground process group ID

SYNOPSIS

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

int tcsetpgrp(int fildes, pid_t pgid_id);

DESCRIPTION

If the process has a controlling terminal, tcsetpgrp() will set the foreground
process group ID associated with the terminal to pgid_id. The file associated with fildes
must be the controlling terminal of the calling process and the controlling terminal
must be currently associated with the session of the calling process. The value of
pgid_id must match a process group ID of a process in the same session as the calling
process.

RETURN VALUES

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

ERRORS

The tcsetpgrp() function will fail if:
EBADF The fildes argument is not a valid file descriptor.
EINVAL This implementation does not support the value in the pgid_id
argument.
ENOTTY The calling process does not have a controlling terminal, or the file
is not the controlling terminal, or the controlling terminal is no
longer associated with the session of the calling process.
EIO The process is not ignoring or holding SIGTTOU and is a member
of an orphaned process group.
EPERM The value of pgid_id does not match the process group ID of a
process in the same session as the calling process.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe, and Async-Signal-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

tcgetpgrp(3C), attributes(5), termio(7I)
NAME

tell – return a file offset for a file descriptor

SYNOPSIS

#include <unistd.h>

off_t tell(int fd);

DESCRIPTION

The tell() function obtains the current value of the file-position indicator for the file descriptor fd.

RETURN VALUES

Upon successful completion, tell() returns the current value of the file-position indicator for fd measured in bytes from the beginning of the file.

Otherwise, it returns −1 and sets errno to indicate the error.

ERRORS

The tell() function will fail if:

EBADF The file descriptor fd is not an open file descriptor.
EOVERFLOW The current file offset cannot be represented correctly in an object of type off_t.
ESPIPE The file descriptor fd is associated with a pipe or FIFO.

USAGE

The tell() function is equivalent to lseek(fd, 0, SEEK_CUR).

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO

lseek(2), attributes(5)
telldir(3C)

NAME telldir – current location of a named directory stream

SYNOPSIS #include <dirent.h>

    long int telldir(DIR *dirp);

DESCRIPTION The telldir() function obtains the current location associated with the directory stream specified by dirp.

    If the most recent operation on the directory stream was a seekdir(3C), the directory position returned from the telldir() is the same as that supplied as a loc argument for seekdir().

RETURN VALUES Upon successful completion, telldir() returns the current location of the specified directory stream.

ERRORS The telldir() function will fail if:

    EOVERFLOW The current location of the directory cannot be stored in an object of type long.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO opendir(3C), readdir(3C), seekdir(3C), attributes(5)
### NAME
termios – general terminal interface

### SYNOPSIS
```c
#include <termios.h>

int tcgetattr(int fildes, struct termios *termios_p);
int tcsetattr(int fildes, int optional_actions, const struct termios *termios_p);
int tcsendbreak(int fildes, int duration);
int tcdrain(int fildes);
int tcflush(int fildes, int queue_selector);
int tcflow(int fildes, int action);
speed_t cfgetospeed(const struct termios *termios_p);
int cfsetospeed(struct termios *termios_p, speed_t speed);
speed_t cfgetispeed(const struct termios *termios_p);
int cfsetispeed(struct termios *termios_p, speed_t speed);

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

pid_t tcgetpgrp(int fildes);
int tcsetpgrp(int fildes, pid_t pgid);
pid_t tcgetsid(int fildes);
```

### DESCRIPTION
These functions describe a general terminal interface for controlling asynchronous communications ports. A more detailed overview of the terminal interface can be found in `termio(7I)`, which also describes an `ioctl(2)` interface that provides the same functionality. However, the function interface described by these functions is the preferred user interface.

Each of these functions is now described on a separate manual page.

### SEE ALSO
`ioctl(2), cfgetispeed(3C), cfgetospeed(3C), cfsetispeed(3C), cfsetospeed(3C),
cfgetpgrp(3C), tcdrain(3C), tcflush(3C), tcflow(3C), tcgetattr(3C),
tcsetattr(3C),
tcgetpgrp(3C), tcgetsid(3C), tcsendbreak(3C), tcsetattr(3C),
tcgetpgrp(3C), tcsendbreak(3C), termio(7I)`
times(3UCB)

NAME    times – get process times

SYNOPSIS /usr/ucb/cc [ flag ... ] file ...
#include <sys/param.h>
#include <sys/types.h>
#include <sys/times.h>

int times(tmsp);
    register struct tms *tmsp;

DESCRIPTION The times() function returns time-accounting information for the current process
and for the terminated child processes of the current process. All times are reported in
clock ticks. The number of clock ticks per second is defined by the variable CLK_TCK,
found in the header <limits.h>.

A structure with the following members is returned by times():

    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.

RETURN VALUES Upon successful completion, times() returns 0. Otherwise, it returns -1.

SEE ALSO time(1), time(2), wait(2), getrusage(3C)

NOTES Use of these interfaces should be restricted to only applications written on BSD
platforms. Use of these interfaces with any of the system libraries or in multi-threaded
applications is unsupported.

The times() function has been superseded by getrusage(3C).
tmpfile – create a temporary file

#include <stdio.h>

FILE *tmpfile(void);

The tmpfile() function creates a temporary file and opens a corresponding stream. The file will automatically be deleted when all references to the file are closed. The file is opened as in fopen(3C) for update (w+).

The largest value that can be represented correctly in an object of type off_t will be established as the offset maximum in the open file description.

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.

The tmpfile() function will fail if:

EINTR A signal was caught during the execution of tmpfile().

EMFILE There are OPEN_MAX file descriptors currently open in the calling process.

ENFILE The maximum allowable number of files is currently open in the system.

ENOSPC The directory or file system which would contain the new file cannot be expanded.

The tmpfile() function may fail if:

EMFILE There are FOPEN_MAX streams currently open in the calling process.

ENOMEM Insufficient storage space is available.

The stream refers to a file which is unlinked. If the process is killed in the period between file creation and unlinking, a permanent file may be left behind.

The tmpfile() function has a transitional interface for 64-bit file offsets. See l64(5).

SEE ALSO unlink(2), fopen(3C), tmpnam(3C), l64(5)
tmpnam(3C)

NAME    tmpnam, tmpnam_r, tempnam – create a name for a temporary file

SYNOPSIS #include <stdio.h>

char *tmpnam(char *s);
char *tmpnam_r(char *s);
char *tempnam(const char *dir, const char *pfx);

DESCRIPTION These functions generate file names that can be used safely for a temporary file.

tmpnam() The tmpnam() function always generates a file name using the path prefix defined as
 P_tmpdir in the <stdio.h> header. On Solaris systems, the default value for
 P_tmpdir is /var/tmp. If s is NULL, tmpnam() leaves its result in 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.

tmpnam_r() The tmpnam_r() function has the same functionality as tmpnam() except that if s is a
 null pointer, the function returns NULL.

tempnam() The tempnam() function allows the user to control the choice of a directory. The
 argument dir points to the name of the directory in which the file is to be created. If dir
 is NULL or points to a string that is not a name for an appropriate directory, the path
 prefix defined as P_tmpdir in the <stdio.h> header is used. If that directory is not
 accessible, /tmp is used. If, however, the TMPDIR environment variable is set in the
 user’s environment, its value is used as the temporary-file directory.

Many applications prefer that temporary files have certain initial character sequences
 in their names. The pfx argument may be NULL or point to a string of up to five
 characters to be used as the initial characters of the temporary-file name.

Upon successful completion, tempnam() uses malloc(3) to allocate space for a
 string, puts the generated pathname in that space, and returns a pointer to it. The
 pointer is suitable for use in a subsequent call to free(). If tempnam() cannot return
 the expected result for any reason (for example, malloc() failed), or if none of the
 above-mentioned attempts to find an appropriate directory was successful, a null
 pointer is returned and errno is set to indicate the error.

ERRORS The tempnam() function will fail if:

ENOMEM Insufficient storage space is available.

USAGE These functions generate a different file name each time they are called.

Files created using these functions and either fopen(3) or creat(2) are temporary
 only in the sense that they reside in a directory intended for temporary use, and their
 names are unique. It is the user’s responsibility to remove the file when its use is
 ended.
If called more than TMP_MAX (defined in `<stdio.h>`) times in a single process, these functions start recycling previously used names.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using these functions or `mktemp(3C)` and the file names are chosen to render duplication by other means unlikely.

The `tmpnam()` function is unsafe in multithreaded applications. The `tempnam()` function is safe in multithreaded applications and should be used instead.

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

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See USAGE above.</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`creat(2), unlink(2), fopen(3C), free(3C), malloc(3C), mktemp(3C), tmpfile(3C), attributes(5)`
toascii(3C)

NAME  toascii – translate integer to a 7-bit ASCII character

SYNOPSIS  #include <ctype.h>
           
           int toascii(int c);

DESCRIPTION  The toascii() function converts its argument into a 7-bit ASCII character.

RETURN VALUES  The toascii() function returns the value (c & 0x7f).

ERRORS  No errors are returned.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO  isascii(3C), attributes(5)
NAME
_tolower – transliterate upper-case characters to lower-case

SYNOPSIS
#include <ctype.h>

int _tolower(int c);

DESCRIPTION
The _tolower() macro is equivalent to tolower(3C) except that the argument c must be an upper-case letter.

RETURN VALUES
On successful completion, _tolower() returns the lower-case letter corresponding to the argument passed.

ERRORS
No errors are defined.

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

ATTRIBUTE TYPE | ATTRIBUTE VALUE
----------------|------------------
MT-Level        | MT-Safe
CSI            | Enabled

SEE ALSO
isupper(3C), tolower(3C), attributes(5)
tolower(3C)

NAME
tolower – transliterate upper-case characters to lower-case

SYNOPSIS
#include <ctype.h>

int tolower(int c);

DESCRIPTION
The tolower() function has as a domain a type int, the value of which is
representable as an unsigned char or the value of EOF. If the argument has any
other value, the argument is returned unchanged. If the argument of tolower() represents an upper-case letter, and there exists a corresponding lower-case letter (as
defined by character type information in the program locale category LC_CTYPE),
the result is the corresponding lower-case letter. All other arguments in the domain are
returned unchanged.

RETURN VALUES
On successful completion, tolower() returns the lower-case letter corresponding to
the argument passed. Otherwise, it returns the argument unchanged.

ERRORS
No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO	_tolower(3C), setlocale(3C), attributes(5)
NAME
_toupper – transliterate lower-case characters to upper-case

SYNOPSIS
#include <ctype.h>

int _toupper(int c);

DESCRIPTION
The _toupper() macro is equivalent to toupper(3C) except that the argument c
must be a lower-case letter.

RETURN VALUES
On successful completion, _toupper() returns the upper-case letter corresponding to
the argument passed.

ERRORS
No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO
islower(3C), toupper(3C), attributes(5)
The `toupper()` function has as a domain a type `int`, the value of which is representable as an `unsigned char` or the value of `EOF`. If the argument has any other value, the argument is returned unchanged. If the argument of `toupper()` represents a lower-case letter, and there exists a corresponding upper-case letter (as defined by character type information in the program locale category `LC_CTYPE`), the result is the corresponding upper-case letter. All other arguments in the domain are returned unchanged.

On successful completion, `toupper()` returns the upper-case letter corresponding to the argument passed.

No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO: `toupper(3C)`, `setlocale(3C)`, `attributes(5)`
#include <wctype.h>

wint_t towctrans(wint_t wc, wctrans_t desc);

The `towctrans()` function maps the wide character `wc` using the mapping described by `desc`. The current setting of the LC_CTYPE category shall be the same as during the call to `wctrans()` that returned the value `desc`.

The function call `towctrans(wc, wctrans("tolower"))` behaves the same as `towlower(wc)`.

The function call `towctrans(wc, wctrans("toupper"))` behaves the same as `towupper(wc)`.

The `towctrans()` function returns the mapped value of `wc`, using the mapping described by `desc`; otherwise, it returns `wc` unchanged.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

See also `setlocale(3C), wctrans(3C), attributes(5)`
towlower(3C)

NAME
towlower – transliterate upper-case wide-character code to lower-case

SYNOPSIS
#include <wchar.h>

wint_t towlower(wint_t wc);

DESCRIPTION
The towlower() function has as a domain a type wint_t, the value of which must
be a character representable as a wchar_t, and must be a wide-character code
corresponding to a valid character in the current locale or the value of WEOF. If the
argument has any other value, the argument is returned unchanged. If the argument
of towlower() represents an upper-case wide-character code, and there exists a
corresponding lower-case wide-character code (as defined by character type
information in the program locale category LC_CTYPE), the result is the
corresponding lower-case wide-character code. All other arguments in the domain are
returned unchanged.

RETURN VALUES
On successful completion, towlower() returns the lower-case letter corresponding to
the argument passed. Otherwise, it returns the argument unchanged.

ERRORS
No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO
iswalpha(3C), setlocale(3C), towupper(3C), attributes(5)
towupper (3C)

NAME
towupper – transliterate lower-case wide-character code to upper-case

SYNOPSIS
#include <wchar.h>

wint_t towupper(wint_t wc);

DESCRIPTION
The towupper() function has as a domain a type wint_t, the value of which must
be a character representable as a wchar_t, and must be a wide-character code
corresponding to a valid character in the current locale or the value of WEOF. If the
argument has any other value, the argument is returned unchanged. If the argument
of towupper() represents a lower-case wide-character code (as defined by character
type information in the program locale category LC_CTYPE), the result is the
corresponding upper-case wide-character code. All other arguments in the domain are
returned unchanged.

RETURN VALUES
Upon successful completion, towupper() returns the upper-case letter corresponding
to the argument passed. Otherwise, it returns the argument unchanged.

ERRORS
No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO
iswalpha(3C), setlocale(3C), tolower(3C), attributes(5)
truncat(3C)

NAME    truncate, ftruncate – set a file to a specified length

SYNOPSIS

#include <unistd.h>

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

DESCRIPTION

The truncate() function causes the regular file named by path to have a size of
length bytes.

The ftruncate() function causes the regular file referenced by fildes to have a size of
length bytes.

The effect of ftruncate() and truncate() on other types of files is unspecified. If
the file previously was larger than length, the extra data is lost. If it was previously
shorter than length, bytes between the old and new lengths are read as zeroes. With
ftruncate(), the file must be open for writing; for truncate(), the process must
have write permission for the file.

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.

RETURN VALUES

Upon successful completion, ftruncate() and truncate() return 0. Otherwise, −1
is returned and errno is set to indicate the error.

ERRORS

The ftruncate() and truncate() functions will fail if:

EINTR   A signal was caught during execution.
EINVAL   The length argument was less than 0.
EFBIG or EINVAL   The length argument was greater than the maximum
file size.
EIO   An I/O error occurred while reading from or writing to
a file system.

The truncate() function will fail if:

EACCES   A component of the path prefix denies search
permission, or write permission is denied on the file.
EFAULT  The path argument points outside the process’ allocated
address space.
EINVAL   The path argument is not an ordinary file.
truncate(3C)

<table>
<thead>
<tr>
<th>EISDIR</th>
<th>The named file is a directory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELOOP</td>
<td>Too many symbolic links were encountered in</td>
</tr>
<tr>
<td></td>
<td>resolving path.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>The maximum number of file descriptors available to</td>
</tr>
<tr>
<td></td>
<td>the process has been reached.</td>
</tr>
<tr>
<td>ENAMETOOLONG</td>
<td>The length of the specified pathname exceeds</td>
</tr>
<tr>
<td></td>
<td>PATH_MAX bytes, or the length of a component of the</td>
</tr>
<tr>
<td></td>
<td>pathname exceeds NAME_MAX bytes.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>A component of path does not name an existing file or</td>
</tr>
<tr>
<td></td>
<td>path is an empty string.</td>
</tr>
<tr>
<td>ENFILE</td>
<td>Additional space could not be allocated for the system</td>
</tr>
<tr>
<td></td>
<td>file table.</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>A component of the path prefix of path is not a</td>
</tr>
<tr>
<td></td>
<td>directory.</td>
</tr>
<tr>
<td>ENOLINK</td>
<td>The path argument points to a remote machine and the</td>
</tr>
<tr>
<td></td>
<td>link to that machine is no longer active.</td>
</tr>
<tr>
<td>EROFS</td>
<td>The named file resides on a read-only file system.</td>
</tr>
</tbody>
</table>

The **ftruncate()** function will fail if:

<table>
<thead>
<tr>
<th>EAGAIN</th>
<th>The file exists, mandatory file/record locking is set, and there are outstanding record locks on the file (see chmod(2)).</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF or EINVAL</td>
<td>The fd parameter is not a file descriptor open for writing.</td>
</tr>
<tr>
<td>EFBIG</td>
<td>The file is a regular file and length is greater than the offset maximum established in the open file description associated with fd.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The fd parameter references a file that was opened without write permission.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The fd parameter does not correspond to an ordinary file.</td>
</tr>
<tr>
<td>ENOLINK</td>
<td>The fd parameter points to a remote machine and the link to that machine is no longer active.</td>
</tr>
</tbody>
</table>

The **truncate()** function may fail if:

| ENAMETOOLONG | Pathname resolution of a symbolic link produced an intermediate result whose                                               |

**USAGE** The **truncate()** and **ftruncate()** functions have transitional interfaces for 64-bit file offsets. See **flag64(5)**.
ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  chmod(2), fcntl(2), open(2), attributes(5), l64(5)
The tsearch(), tfind(), tdelete(), and twalk() functions are routines for manipulating binary search trees. They are generalized from Knuth (6.2.2) Algorithms T and D. All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to the elements being compared. It returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The tsearch() function is used to build and access the tree. The key argument is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to *key (the value pointed to by key), a pointer to this found datum is returned. Otherwise, *key is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. The rootp argument points to a variable that points to the root of the tree. A null value for the variable pointed to by rootp denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

Like tsearch(), tfind() will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, tfind() will return a null pointer. The arguments for tfind() are the same as for tsearch().

The tdelete() function deletes a node from a binary search tree. The arguments are the same as for tsearch(). The variable pointed to by rootp will be changed if the deleted node was the root of the tree. tdelete() returns a pointer to the parent of the deleted node, or a null pointer if the node is not found.

The twalk() function traverses a binary search tree. The root argument is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) action is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type

typedef enum { preorder, postorder, endorder, leaf } VISIT;(defined in <search.h>), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a
RETURN VALUES

If the node is found, both \texttt{tsearch()} and \texttt{tfind()} return a pointer to it. If not, 
\texttt{tfind()} returns a null pointer, and \texttt{tsearch()} returns a pointer to the inserted 
item.

A null pointer is returned by \texttt{tsearch()} if there is not enough space available to 
create a new node.

A null pointer is returned by \texttt{tsearch()}, \texttt{tfind()} and \texttt{tdelete()} if \texttt{rootp} is a null 
pointer on entry.

The \texttt{tdelete()} function returns a pointer to the parent of the deleted node, or a null 
pointer if the node is not found.

The \texttt{twalk()} function returns no value.

ERRORS

No errors are defined.

USAGE

The \texttt{root} argument to \texttt{twalk()} is one level of indirection less than the \texttt{rootp} arguments 
to \texttt{tsearch()} and \texttt{tdelete()}.

There are two nomenclatures used to refer to the order in which tree nodes are visited. 
\texttt{tsearch()} uses preorder, postorder and endorder to refer respectively to visiting a 
node before any of its children, after its left child and before its right, and after both its 
children. The alternate nomenclature uses preorder, inorder and postorder to refer to 
the same visits, which could result in some confusion over the meaning of postorder.

If the calling function alters the pointer to the root, results are unpredictable.

EXAMPLES

\textbf{EXAMPLE 1} A sample program of using \texttt{tsearch} function.

The following code reads in strings and stores structures containing a pointer to each 
string and a count of its length. It then walks the tree, printing out the stored strings 
and their lengths in alphabetical order.

\begin{verbatim}
#include <string.h>
#include <stdio.h>
#include <search.h>

struct node {
    char *string;
    int length;
};

char string_space[10000];
struct node nodes[500];
void *root = NULL;
\end{verbatim}
EXAMPLE 1 A sample program of using `tsearch` function.  

```c
int node_compare(const void *node1, const void *node2) {
    return strcmp(((const struct node *) node1)->string,
                   ((const struct node *) node2)->string);
}

void print_node(const void *node, VISIT order, int level) {
    if (order == preorder || order == leaf) {
        printf("length=%d, string=%20s\n",
               (*(struct node **)node)->length,
               (*(struct node **)node)->string);
    }
}

main() {
    char *strptr = string_space;
    struct node *nodeptr = nodes;
    int i = 0;

    while (gets(strptr) != NULL && i++ < 500) {
        nodeptr->string = strptr;
        nodeptr->length = strlen(strptr);
        (void) tsearch((void *)nodeptr,
                        &root, node_compare);
        strptr += nodeptr->length + 1;
        nodeptr++;
    }
    twalk(root, print_node);
}
```

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  `bsearch(3C), hsearch(3C), lsearch(3C), attributes(5)`
ttyname(3C)

NAME   ttyname, ttyname_r – find pathname of a terminal

SYNOPSIS
#include <unistd.h>
char *ttyname(int fd);  
char *ttyname_r(int fd, char *name, int namelen);

POSIX  cc [ flag ...] file ... -D_POSIX_PTHREAD_SEMANTICS [ library ... ]
int ttyname_r(int fd, char *name, size_t namesize);

DESCRIPTION  The ttyname() function returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor fd. The return value may point to static data whose content is overwritten by each call.

The ttyname_r() function has the same functionality as ttyname() except that the caller must supply a buffer name with length namelen to store the result; this buffer must be at least _POSIX_PATH_MAX in size (defined in <limits.h>). The POSIX version (see standards(5)) of ttyname_r() takes a namesize parameter of type size_t.

RETURN VALUES   Upon successful completion, ttyname() and ttyname_r() return a pointer to a string. Otherwise, a null pointer is returned and errno is set to indicate the error.

The POSIX ttyname_r() returns zero if successful, or the error number upon failure.

ERRORS   The ttyname_r() function will fail if:
ERANGE      The size of the buffer is smaller than the result to be returned.

The ttyname() function may fail if:
EBADF        The fd argument is not a valid file descriptor.
ENOTTY       The fd argument does not refer to a terminal device.

FILES   /dev/* device file

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See NOTES below.</td>
</tr>
</tbody>
</table>

SEE ALSO   Intro(3), gettext(3C), setlocale(3C), attributes(5), standards(5)

NOTES   When compiling multithreaded programs, see Intro(3),
Notes On Multithreaded Applications.

If the application is linked with -lnt1, then messages printed from this function are in the native language specified by the LC_MESSAGES locale category; see setlocale(3C).

576 man pages section 3: Basic Library Functions • Last Revised 20 Mar 1997
The return value points to static data whose content is overwritten by each call.

The `ttyname()` is Unsafe in multithreaded applications. The `ttyname_r()` function is MT-Safe, and should be used instead.

Solaris 2.4 and earlier releases provided definitions of the `ttyname_r()` interface as specified in POSIX.1c Draft 6. The final POSIX.1c standard changed the interface as described above. Support for the Draft 6 interface is provided for compatibility only and may not be supported in future releases. New applications and libraries should use the POSIX standard interface.
ttyslot(3C)

NAME ttyslot – find the slot of the current user in the user accounting database

SYNOPSIS

#include <stdlib.h>

int ttyslot(void);

DESCRIPTION The ttyslot() function returns the index of the current user’s entry in the user accounting database, /var/adm/utmpx. The current user’s entry is an entry for which the utline member matches the name of a terminal device associated with any of the process’s file descriptors 0, 1 or 2. The index is an ordinal number representing the record number in the database of the current user’s entry. The first entry in the database is represented by the return value 0.

RETURN VALUES Upon successful completion, ttyslot() returns the index of the current user’s entry in the user accounting database. If an error was encountered while searching for the terminal name or if none of the above file descriptors are associated with a terminal device, −1 is returned.

FILES /var/adm/utmpx user access and accounting information

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO getutent(3C), ttyname(3C), utmpx(4), attributes(5)
#include <unistd.h>

useconds_t ualarm(useconds_t useconds, useconds_t interval);

The `ualarm()` function causes the SIGALRM signal to be generated for the calling process after the number of real-time microseconds specified by the `useconds` argument has elapsed. When the `interval` argument is non-zero, repeated timeout notification occurs with a period in microseconds specified by the `interval` argument. If the notification signal, SIGALRM, is not caught or ignored, the calling process is terminated.

Because of scheduling delays, resumption of execution when the signal is caught may be delayed an arbitrary amount of time.

Interactions between `ualarm()` and either `alarm(2)` or `sleep(3C)` are unspecified.

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.

No errors are defined.

The `ualarm()` function is a simplified interface to `setitimer(2)`, and uses the ITIMER_REAL interval timer.

`alarm(2)`, `setitimer(2)`, `sighold(3C)`, `signal(3C)`, `sleep(3C)`, `usleep(3C)`
The umem_alloc() function returns a pointer to a block of size bytes suitably aligned for any variable type. The initial contents of memory allocated using umem_alloc() is undefined. The flags argument determines the behavior of umem_alloc() if it is unable to fulfill the request. The flags argument can take the following values:

- **UMEM_DEFAULT**
  - Return NULL on failure.

- **UMEM_NOFAIL**
  - Call an optional callback (set with umem_nofail_callback()) on failure. The callback takes no arguments and can finish by:
    - returning UMEM_CALLBACK_RETRY, in which case the allocation will be retried. If the allocation fails, the callback will be invoked again.
    - returning UMEM_CALLBACK_EXIT(status), in which case exit(2) is invoked with status as its argument. The exit() function is called only once. If multiple threads return from the UMEM_NOFAIL callback with UMEM_CALLBACK_EXIT(status), one will call exit() while the other blocks until exit() terminates the program.
    - invoking a context-changing function (setcontext(2)) or a non-local jump (longjmp(3C) or siglongjmp(3C), or ending the current thread of control (thr_exit(3THR) or pthread_exit(3THR). The application is responsible for any necessary cleanup. The state of libumem remains consistent.

If no callback has been set or the callback has been set to NULL, umem_alloc(..., UMEM_NOFAIL) behaves as though the callback returned UMEM_CALLBACK_EXIT(255).
The libumem library can call callbacks from any place that a UMEM_NOFAIL allocation is issued. In multithreaded applications, callbacks are expected to perform their own concurrency management.

The function call `umem_alloc(0, flag)` always returns NULL. The function call `umem_free(NULL, 0)` is allowed.

The `umem_zalloc()` function has the same semantics as `umem_alloc()`, but the block of memory is initialized to zeros before it is returned.

The `umem_free()` function frees blocks previously allocated using `umem_alloc()` and `umem_zalloc()`. The buffer address and size must exactly match the original allocation. Memory must not be returned piecemeal.

The `umem_nofail_callback()` function sets the process-wide UMEM_NOFAIL callback. See the description of UMEM_NOFAIL for more information.

The `malloc()`, `calloc()`, `free()`, `memalign()`, `realloc()`, and `valloc()` functions are as described in `malloc(3C)`. The libumem library provides these functions for backwards compatibility with the standard functions.

**ENVIRONMENT VARIABLES**

See `umem_debug(3MALLOC)` for environment variables that effect the debugging features of the libumem library.

**UMEM_OPTIONS**

Contains a list of comma-separated options. Unrecognized options are ignored. The options that are supported are:

- `backend=sbrk`
- `backend=mmap`

Set the underlying function used to allocate memory. This option can be set to `sbrk` (the default) for an `sbrk(2)`-based source or `mmap` for an `mmap(2)`-based source. If set to a value that is not supported, `sbrk` will be used.

**EXAMPLE 1** Using the `umem_alloc()` function

```c
#include <stdio.h>
#include <umem.h>
...
char *buf = umem_alloc(1024, UMEM_DEFAULT);
if (buf == NULL) {
    fprintf(stderr, "out of memory\n");
}
```
EXAMPLE 1 Using the umem_alloc() function (Continued)

```c
    return (1);
}
/* cannot assume anything about buf’s contents */
...
umem_free(buf, 1024);
...
```

EXAMPLE 2 Using the umem_zalloc() function

```c
#include <stdio.h>
#include <umem.h>
...
char *buf = umem_zalloc(1024, UMEM_DEFAULT);
if (buf == NULL) {
    fprintf(stderr, "out of memory\n");
    return (1);
}
/* buf contains zeros */
...
umem_free(buf, 1024);
...
```

EXAMPLE 3 Using UMEM_NOFAIL

```c
#include <stdlib.h>
#include <stdio.h>
#include <umem.h>
/* Note that the allocation code below does not have to
* check for umem_alloc() returning NULL
*/
int
my_failure_handler(void)
{
    (void) fprintf(stderr, "out of memory\n");
    return (UMEM_CALLBACK_EXIT(255));
}
...
umem_nofail_callback(my_failure_handler);
...
int i;
char *buf[100];
for (i = 0; i < 100; i++)
    buf[i] = umem_alloc(1024 * 1024, UMEM_NOFAIL);
...
for (i = 0; i < 100; i++)
    umem_free(buf[i], 1024 * 1024);
...
EXAMPLE 4 Using UMEM_NOFAIL in a multithreaded application

```c
#define _REENTRANT
#include <thread.h>
#include <stdio.h>
#include <umem.h>

void *
start_func(void *the_arg)
{
    int *info = (int *)the_arg;
    char *buf = umem_alloc(1024 * 1024, UMEM_NOFAIL);
    /* does not need to check for buf == NULL */
    buf[0] = 0;
    ...
    /*
     * if there were other UMEM_NOFAIL allocations,
     * we would need to arrange for buf to be
     * umem_free()ed upon failure.
     */
    ...
    umem_free(buf, 1024 * 1024);
    return (the_arg);
}
...
int
my_failure_handler(void)
{
    /* terminate the current thread with status NULL */
    thr_exit(NULL);
}
...
umem_nofail_callback(my_failure_handler);
...
int my_arg;
thread_t tid;
void *status;
(void) thr_create(NULL, NULL, start_func, &my_arg, 0,
                  NULL);
...
while (thr_join(0, &tid, &status) != 0)
{
    if (status == NULL) {
        (void) fprintf(stderr, "thread %d ran out of memory\n",
                       tid);
    }
    ...
}
```

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:
umem_alloc(3MALLOC)

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>malloc(), calloc(), free(), realloc(), and valloc() are Standard. memalign() is Stable. umem_alloc(), umem_zalloc(), umem_free(), and umem_nofail_callback() are Evolving.</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO exit(2), mmap(2), sbrk(2), bsdmalloc(3MALLOC), libumem(3LIB), longjmp(3C), malloc(3C), malloc(3MALLOC), mapmalloc(3MALLOC), pthread_exit(3THR), thr_exit(3THR), umem_cache_create(3MALLOC), umem_debug(3MALLOC), watchmalloc(3MALLOC), attributes(5), standards(5)

Solaris Modular Debugger Guide

WARNINGS Any of the following can cause undefined results:
- Passing a pointer returned from umem_alloc() or umem_zalloc() to free() or realloc().
- Passing a pointer returned from malloc(), calloc(), valloc(), memalign(), or realloc() to umem_free().
- Writing past the end of a buffer allocated using umem_alloc() or umem_zalloc().
- Performing UMEM_NOFAIL allocations from an atexit(3C) handler.

If the UMEM_NOFAIL callback performs UMEM_NOFAIL allocations, infinite recursion can occur.

NOTES The following list compares the features of the malloc(3C), bsdmalloc(3MALLOC), malloc(3MALLOC), mtmalloc(3MALLOC), and the libumem functions.
- The malloc(3C), bsdmalloc(3MALLOC), and malloc(3MALLOC) functions have no support for concurrency. The libumem and mtmalloc(3MALLOC) functions support concurrent allocations.
- The bsdmalloc(3MALLOC) functions afford better performance but are space-inefficient.
- The malloc(3MALLOC) functions are space-efficient but have slower performance.
- The standard, fully SCD-compliant malloc(3C) functions are a trade-off between performance and space-efficiency.
- The mtmalloc(3MALLOC) functions provide fast, concurrent malloc() implementations that are not space-efficient.
- The libumem functions provide a fast, concurrent allocation implementation that in most cases is more space-efficient than mtmalloc(3MALLOC).
umem_cache_create, umem_cache_destroy, umem_cache_alloc, umem_cache_free – allocation cache manipulation

SYNOPSIS

cc [ flag ... ] file ... -lumem [ library ... ]
#include <umem.h>

umem_cache_t * umem_cache_create(char *debug_name, size_t bufsize,
size_t align, umem_constructor_t *constructor, umem_destructor_t *
*destructor, umem_reclaim_t *reclaim, void *callback_data, vmem_t *
*source, int cflags);

void umem_cache_destroy(umem_cache_t *cache);

void umem_cache_alloc(umem_cache_t *cache, int flags);

void umem_cache_free(umem_cache_t *cache, void *buffer);

DESCRIPTION

These functions create, destroy, and use an "object cache". An object cache is a collection of buffers of a single size, with optional content caching enabled by the use of callbacks (see Cache Callbacks). Object caches are MT-Safe. Multiple allocations and freeing of memory from different threads can proceed simultaneously. Object caches are faster and use less space per buffer than malloc(3MALLOC) and umem_alloc(3MALLOC). For more information about object caching, see “The Slab Allocator: An Object-Caching Kernel Memory Allocator” and “Magazines and vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources”.

The umem_cache_create() function creates object caches. Once a cache has been created, objects can be requested from and returned to the cache using umem_cache_alloc() and umem_cache_free(), respectively. A cache with no outstanding buffers can be destroyed with umem_cache_destroy().

Creating and Destroying Caches

The umem_cache_create() function creates a cache of objects and takes as arguments the following:

d.debug_name A human-readable name for debugging purposes.
bufsize The size, in bytes, of the buffers in this cache.
align The minimum alignment required for buffers in this cache. This parameter must be a power of 2. If 0, it is replaced with the minimum required alignment for the current architecture.
constructor The callback to construct an object.
derestructor The callback to destroy an object.
reclaim The callback to reclaim objects.
callback_data An opaque pointer passed to the callbacks.
source This parameter must be NULL.
cflags This parameter must be either 0 or UMC_NODEBUG. If UMC_NODEBUG, all debugging features are disabled for
Each cache can have up to three associated callbacks:

- `int constructor(void *buffer, void *callback_data, int flags);`
- `void destructor(void *buffer, void *callback_data);`
- `void reclaim(void *callback_data);`

The `callback_data` argument is always equal to the value passed to `umem_cache_create()`, thereby allowing a client to use the same callback functions for multiple caches, but with customized behavior.

The reclaim callback is called when the umem function is requesting more memory from the operating system. This callback can be used by clients who retain objects longer than they are strictly needed (for example, caching non-active state). A typical reclaim callback might return to the cache ten per cent of the unneeded buffers.

The constructor and destructor callbacks enable the management of buffers with the constructed state. The constructor takes as arguments a buffer with undefined contents, some callback data, and the flags to use for any allocations. This callback should transform the buffer into the constructed state.

The destructor callback takes as an argument a constructed object and prepares it for return to the general pool of memory. The destructor should undo any state that the constructor created. For debugging, the destructor can also check that the buffer is in the constructed state, to catch incorrectly freed buffers. See `umem_debug(3MALLOC)` for further information on debugging support.

The `umem_cache_destroy()` function destroys an object cache. If the cache has any outstanding allocations, the behavior is undefined.

Allocating Objects

The `umem_cache_alloc()` function takes as arguments:

- `cache` a cache pointer
- `flags` flags that determine the behavior if `umem_cache_alloc()` is unable to fulfill the allocation request

If successful, `umem_cache_alloc()` returns a pointer to the beginning of an object of `bufsize` length.

There are three cases to consider:

- A new buffer needed to be allocated. If the cache was created with a constructor, it is applied to the buffer and the resulting object is returned.
- The object cache was able to use a previously freed buffer. If the cache was created with a constructor, the object is returned unchanged from when it was freed.
- The allocation of a new buffer failed. The `flags` argument determines the behavior:
umem_cache_create(3MALLOC)

| UMEM_DEFAULT   | The umem_cache_alloc() function returns NULL if the allocation fails. |
| UMEM_NOFAIL    | The umem_cache_alloc() function cannot return NULL. A callback is used to determine what action occurs. See umem_alloc(3MALLOC) for more information. |

Freeing Objects

The umem_cache_free() function takes as arguments:

- **cache**: a cache pointer
- **buf**: a pointer previously returned from umem_cache_alloc(). This argument must not be NULL.

If the cache was created with a constructor callback, the object must be returned to the constructed state before it is freed.

Undefined behavior results if an object is freed multiple times, if an object is modified after it is freed, or if an object is freed to a cache other than the one from which it was allocated.

Caches with Constructors

When a constructor callback is in use, there is essentially a contract between the cache and its clients. The cache guarantees that all objects returned from umem_cache_alloc() will be in the constructed state, and the client guarantees that it will return the object to the constructed state before handing it to umem_cache_free().

RETURN VALUES

Upon failure, the umem_cache_create() function returns a null pointer.

ERRORS

The umem_cache_create() function will fail if:

- **EAGAIN**: There is not enough memory available to allocate the cache data structure.
- **EINVAL**: The debug_name argument is NULL, the align argument is not a power of two or is larger than the system pagesize, or the bufsize argument is 0.
- **ENOMEM**: The libumem library could not be initialized, or the bufsize argument is too large and its use would cause integer overflow to occur.

EXAMPLES

**EXAMPLE 1** Use a fixed-size structure with no constructor callback.

```c
#include <umem.h>

typedef struct my_obj {
    long my_data1;
} my_obj_t;
/*
```
EXAMPLE 1 Use a fixed-size structure with no constructor callback. (Continued)

* my_objs can be freed at any time. The contents of
* my_data1 is undefined at allocation time.
*/

umem_cache_t *my_obj_cache;

... my_obj_cache = umem_cache_create("my_obj", sizeof (my_obj_t),
   0, NULL, NULL, NULL, NULL, NULL, 0);
... my_obj_t *cur = umem_cache_alloc(my_obj_cache, UMEM_DEFAULT);
... /* use cur */
... umem_cache_free(my_obj_cache, cur);
...

EXAMPLE 2 Use an object with a mutex.

#define _REENTRANT
#include <synch.h>
#include <umem.h>

typedef struct my_obj {
   mutex_t my_mutex;
   long my_data;
} my_obj_t;

/*
* my_objs can only be freed when my_mutex is unlocked.
*/

int my_obj_constructor(void *buf, void *ignored, int flags)
{
   my_obj_t *myobj = buf;
   (void) mutex_init(&my_obj->my_mutex, USYNC_THREAD, NULL);
   return (0);
}

void my_objDestructor(void *buf, void *ignored)
{
   my_obj_t *myobj = buf;
   (void) mutex_destroy(&my_obj->my_mutex);
}

umem_cache_t *my_obj_cache;

... my_obj_cache = umem_cache_create("my_obj", sizeof (my_obj_t),
   0, my_obj_constructor, my_objDestructor, NULL, NULL,
EXAMPLE 2 Use an object with a mutex. (Continued)

    NULL, 0);

    my_obj_t *cur = umem_cache_alloc(my_obj_cache, UMEM_DEFAULT);
    cur->my_data = 0; /* cannot assume anything about my_data */
    ...
    umem_cache_free(my_obj_cache, cur);
    ...

EXAMPLE 3 Use a more complex object with a mutex.

#define _REENTRANT
#include <assert.h>
#include <synch.h>
#include <umem.h>

typedef struct my_obj {
    mutex_t my_mutex;
    cond_t my_cv;
    struct bar *my_barlist;
    unsigned my_refcount;
} my_obj_t;

/*
 * my_objs can only be freed when my_barlist == NULL,
 * my_refcount == 0, there are no waiters on my_cv, and
 * my_mutex is unlocked.
 */

int
my_obj_constructor(void *buf, void *ignored, int flags)
{
    my_obj_t *myobj = buf;

    (void) mutex_init(&myobj->my_mutex, USYNC_THREAD, NULL);
    (void) cond_init(&myobj->my_cv, USYNC_THREAD, NULL);
    myobj->my_barlist = NULL;
    myobj->my_refcount = 0;

    return (0);
}

void
my_obj_destructor(void *buf, void *ignored)
{
    my_obj_t *myobj = buf;

    assert(myobj->my_refcount == 0);
    assert(myobj->my_barlist == NULL);
    (void) cond_destroy(&myobj->my_cv);
    (void) mutex_destroy(&myobj->my_mutex);
}

umem_cache_t *my_obj_cache;
EXAMPLE 3 Use a more complex object with a mutex.  (Continued)

...  
my_obj_cache = umem_cache_create("my_obj", sizeof (my_obj_t),  
0, my_obj_constructor, my_obj_destructor, NULL, NULL,  
NULL, 0);  
...  
my_obj_t *cur = umem_cache_alloc(my_obj_cache, UMEM_DEFAULT);  
...  
/* use cur */  
...  
umem_cache_free(my_obj_cache, cur);  
...

EXAMPLE 4 Use objects with a subordinate buffer while reusing callbacks.

#include assert.h>
#include umem.h>

typedef struct my_obj {
  char *my_buffer;
  size_t my_size;
} my_obj_t;

/*
 * my_size and the my_buffer pointer should never be changed
 */

int
my_obj_constructor(void *buf, void *arg, int flags)
{
  size_t sz = (size_t)arg;
  my_obj_t *myobj = buf;
  if (((myobj->my_buffer = umem_alloc(sz, flags)) == NULL)  
      return (1));
  my_size = sz;
  return (0);
}

void
my_obj_destructor(void *buf, void *arg)
{
  size_t sz = (size_t)arg;
  my_obj_t *myobj = buf;
  assert(sz == buf->my_size);
  umem_free(myobj->my_buffer, sz);
}

...  
umem_cache_t *my_obj_4k_cache;
EXAMPLE 4 Use objects with a subordinate buffer while reusing callbacks.  

```c
umem_cache_t *my_obj_8k_cache;
...
my_obj_cache_4k = umem_cache_create("my_obj_4k", sizeof (my_obj_t),
    0, my_obj_constructor, my_obj_destructor, NULL, (void *)4096,
    NULL, 0);
my_obj_cache_8k = umem_cache_create("my_obj_8k", sizeof (my_obj_t),
    0, my_obj_constructor, my_obj_destructor, NULL, (void *)8192,
    NULL, 0);
...
my_obj_t *my_obj_4k = umem_cache_alloc(my_obj_4k_cache,
    UMEM_DEFAULT);
my_obj_t *my_obj_8k = umem_cache_alloc(my_obj_8k_cache,
    UMEM_DEFAULT);
/* no assumptions should be made about the contents of the buffers */
...*/
/* make sure to return them to the correct cache */
umem_cache_free(my_obj_4k_cache, my_obj_4k);
umem_cache_free(my_obj_8k_cache, my_obj_8k);
...
```

See the EXAMPLES section of umem_alloc(3MALLOC) for examples involving the UMEM_NOFAIL flag.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Evolving</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
setcontext(2), atexit(3C), libumem(3LIB), longjmp(3C), swapcontext(3C),
thr_exit(3THR), umem_alloc(3MALLOC), umem_debug(3MALLOC),
attributes(5)

Bonwick, Jeff, “The Slab Allocator: An Object-Caching Kernel Memory Allocator”,

Bonwick, Jeff and Jonathan Adams, “Magazines and vmem: Extending the Slab
Allocator to Many CPUs and Arbitrary Resources”, Proceedings of the Summer 2001
Usenix Conference.

WARNINGS
Any of the following can cause undefined results:

- Destroying a cache that has outstanding allocated buffers.
- Using a cache after it has been destroyed.
- Calling umem_cache_free() on the same buffer multiple times.
- Passing a NULL pointer to umem_cache_free().
umem_cache_create(3MALLOC)

- Writing past the end of a buffer.
- Reading from or writing to a buffer after it has been freed.
- Performing UMEM_NOFAIL allocations from an atexit(3C) handler.

Per-cache callbacks can be called from a variety of contexts. The use of functions that modify the active context, such as setcontext(2), swapcontext(3C), and thr_exit(3THR), or functions that are unsafe for use in multithreaded applications, such as longjmp(3C) and siglongjmp(3C), result in undefined behavior.

A constructor callback that performs allocations must pass its flags argument unchanged to umem_alloc(3MALLOC) and umem_cache_alloc(). Any allocations made with a different flags argument results in undefined behavior. The constructor must correctly handle the failure of any allocations it makes.

NOTES

Object caches make the following guarantees about objects:

- If the cache has a constructor callback, it is applied to every object before it is returned from umem_cache_alloc() for the first time.
- If the cache has a constructor callback, an object passed to umem_cache_free() and later returned from umem_cache_alloc() is not modified between the two events.
- If the cache has a destructor, it is applied to all objects before their underlying storage is returned.

No other guarantees are made. In particular, even if there are buffers recently freed to the cache, umem_cache_alloc() can fail.
### NAME
umem_debug – debugging features of the umem library

### SYNOPSIS
```bash
cc [ flag... ] file... -lumem [ library... ]
#include <umem.h>
```

### DESCRIPTION
The libumem library provides debugging features that detect memory leaks, buffer overruns, multiple frees, use of uninitialized data, use of freed data, and many other common programming errors. The activation of the run-time debugging features is controlled by environment variables.

When the library detects an error, it writes a description of the error to an internal buffer that is readable with the `::umem_status mdb(1)` command and then calls `abort(3C)`.

### ENVIRONMENT VARIABLES

<table>
<thead>
<tr>
<th>UMEM_DEBUG</th>
<th>This variable contains a list of comma-separated options. Unrecognized options are ignored. Possible options include:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>audit[=frames]</code></td>
<td>This option enables the recording of auditing information, including thread ID, high-resolution time stamp, and stack trace for the last action (allocation or free) on every allocation. If transaction logging (see <code>UMEM_LOGGING</code>) is enabled, this auditing information is also logged.</td>
</tr>
</tbody>
</table>

The `frames` parameter sets the number of stack frames recorded in the auditing structure. The upper bound for frames is implementation-defined. If a larger value is requested, the upper bound is used instead.

If `frames` is not specified or is not an integer, the default value of 15 is used.

This option also enables the `guards` option.

<table>
<thead>
<tr>
<th><code>contents[=count]</code></th>
<th>If auditing and contents logging (see <code>UMEM_LOGGING</code>) are enabled, the first <code>count</code> bytes of each buffer are logged when they are freed. If a buffer is shorter than <code>count</code> bytes, it is logged in its entirety.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>default</code></td>
<td>This option is equivalent to <code>audit,contents,guards</code>.</td>
</tr>
</tbody>
</table>

| `guards` | This option enables filling allocated and freed buffers with special patterns to help detect the use of uninitialized data and previously freed buffers. It also enables an 8-byte redzone after each buffer that contains `0xfeedfacefeedfaceULL`. |
When an object is freed, it is filled with 0xdeadbeef. When an object is allocated, the 0xdeadbeef pattern is verified and replaced with 0xbaddcafe. The redzone is checked every time a buffer is allocated or freed.

For caches with either constructors or destructors, or both, umem_cache_alloc(3MALLOC) and umem_cache_free(3MALLOC) apply the cache’s constructor and destructor, respectively, instead of caching constructed objects. The presence of assert(3C)s in the destructor verifying that the buffer is in the constructed state can be used to detect any objects returned in an improper state. See umem_cache_create(3MALLOC) for details.

verbose
The library writes error descriptions to standard error before aborting. These messages are not localized.

UMEM_LOGGING
To be enabled, this variable should be set to a comma-separated list of in-memory logs. The logs available are:

transaction[=size]
If the audit debugging option is set (see UMEM_DEBUG), the audit structures from previous transactions are entered into this log.

contents[=size]
If the audit debugging option is set, the contents of objects are recorded in this log as they are freed.

If the “contents” debugging option was not set, 256 bytes of each freed buffer are saved.

fail[=size]
Records are entered into this log for every failed allocation.

For any of these options, if size is not specified, the default value of 64k is used. The size parameter must be an integer that can be qualified with K, M, G, or T to specify kilobytes, megabytes, gigabytes, or terabytes, respectively.

Logs that are not listed or that have either a size of 0 or an invalid size are disabled.

The log is disabled if during initialization the requested amount of storage cannot be allocated.
ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Unstable</td>
</tr>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
mdb(1), abort(3C), signal(3C), umem_cache_create(3MALLOC), attributes(5)

Solaris Modular Debugger Guide

WARNINGS
When libumem aborts the process using abort(3C), any existing signal handler for SIGABRT is called. If the signal handler performs allocations, undefined behavior can result.

NOTES
Some of the debugging features work only for allocations smaller than 16 kilobytes in size. Allocations larger than 16 kilobytes could have reduced support.

Activating any of the library’s debugging features could significantly increase the library’s memory footprint and decrease its performance.
ungetc(3C)

NAME     ungetc – push byte back into input stream

SYNOPSIS  
#include <stdio.h>

int ungetc(int c, FILE *stream);

DESCRIPTION
The ungetc() function pushes the byte specified by c (converted to an unsigned
char) back onto the input stream pointed to by stream. The pushed-back bytes will be
returned by subsequent reads on that stream in the reverse order of their pushing. A
successful intervening call (with the stream pointed to by stream) to a file-positioning
function (fseek(3C), fsetpos(3C) or rewind(3C)) discards any pushed-back bytes
for the stream. The external storage corresponding to the stream is unchanged.

Four bytes of push-back are guaranteed. If ungetc() is called too many times on the
same stream without an intervening read or file-positioning operation on that stream,
the operation may fail.

If the value of c equals that of the macro EOF, the operation fails and the input stream
is unchanged.

A successful call to ungetc() clears the end-of-file indicator for the stream. The value
of the file-position indicator for the stream after reading or discarding all pushed-back
bytes will be the same as it was before the bytes were pushed back. The file-position
indicator is decremented by each successful call to ungetc(); if its value was 0 before
a call, its value is indeterminate after the call.

RETURN VALUES
Upon successful completion, ungetc() returns the byte pushed back after
conversion. Otherwise it returns EOF.

ERRORS
No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
read(2), intro(3), fseek(3C), fsetpos(3C), getc(3C), setbuf(3C), stdio(3C),
attributes(5)
ungetwc – push wide-character code back into input stream

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

wint_t ungetwc(wint_t wc, FILE *stream);

The ungetwc() function pushes the character corresponding to the wide character code specified by wc back onto the input stream pointed to by stream. The pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by stream) to a file-positioning function (fseek(3C), fsetpos(3C) or rewind(3C)) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

One character of push-back is guaranteed. If ungetwc() is called too many times on the same stream without an intervening read or file-positioning operation on that stream, the operation may fail.

If the value of wc equals that of the macro WEOF, the operation fails and the input stream is unchanged.

A successful call to ungetwc() clears the end-of-file indicator for the stream. The value of the file-position indicator for the stream after reading or discarding all pushed-back characters will be the same as it was before the characters were pushed back. The file-position indicator is decremented (by one or more) by each successful call to ungetwc(); if its value was 0 before a call, its value is indeterminate after the call.

Upon successful completion, ungetwc() returns the wide-character code corresponding to the pushed-back character. Otherwise it returns WEOF.

The ungetwc() function may fail if:

EILSEQ An invalid character sequence is detected, or a wide-character code does not correspond to a valid character.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO read(2), fseek(3C), fsetpos(3C), rewind(3C), setbuf(3C), attributes(5)
unlockpt(3C)

NAME  unlockpt – unlock a pseudo-terminal master/slave pair

SYNOPSIS  
#include <stdlib.h>

int unlockpt(int fildes);

DESCRIPTION  
The unlockpt() function unlocks the slave pseudo-terminal device associated with the master to which fildes refers.

Portable applications must call unlockpt() before opening the slave side of a pseudo-terminal device.

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

ERRORS  
The unlockpt() function may fail if:

EBADF  The fildes argument is not a file descriptor open for writing.
EINVAL  The fildes argument is not associated with a master pseudo-terminal device.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  
open(2), grantpt(3C), ptsname(3C), attributes(5)

STREAMS Programming Guide
usleep – suspend execution for interval in microseconds

#include <unistd.h>

int usleep(useconds_t useconds);

The `usleep()` function suspends the caller from execution for the number of microseconds specified by the `useconds` argument. (A microsecond is .000001 seconds.) Because of other activity, or because of the time spent in processing the call, the actual suspension time may be longer than the amount of time specified.

If the value of `useconds` is 0, then the call has no effect.

In a single-threaded program (one not linked with `-lthread` or `-lpthread`), the `usleep()` function uses the process’s realtime interval timer to indicate to the system when the process should be woken up.

There is one real-time interval timer for each process. The `usleep()` function will not interfere with a previous setting of this timer. If the process has set this timer prior to calling `usleep()`, and if the time specified by `useconds` equals or exceeds the interval timer’s prior setting, the caller will be woken up shortly before the timer was set to expire.

Interactions between `usleep()` and either `alarm(2)` or `sleep(3C)` are unspecified.

In a multithreaded program (one linked with `-lthread` or `-lpthread`), `usleep()` is implemented by a call to `nanosleep(3RT)` and does not modify the state of the alarm signal or the realtime interval timer. There is no interaction between this version of `usleep()` and either `alarm(2)` or `sleep(3C)`.

On completion, `usleep()` returns 0. There are no error returns.

No errors are returned.

The `usleep()` function is included for its historical usage. The `nanosleep(3RT)` function is preferred over this function.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

In a multithreaded program, only the invoking thread is suspended from execution.
**NAME**
vfprintf, vwprintf, vswprintf – wide-character formatted output of a stdarg argument list

**SYNOPSIS**
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vwprintf(const wchar_t *format, va_list arg);
int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);
int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list arg);

**DESCRIPTION**
The vwprintf(), vfwprintf(), and vswprintf() functions are the same as wprintf(), fprintf(), and sprintf() respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by <stdarg.h>. See stdarg(3HEAD).

These functions do not invoke the va_end() macro. However, as these functions do invoke the va_arg() macro, the value of ap after the return is indeterminate.

**RETURN VALUES**
Refer to fprintf(3C).

**ERRORS**
Refer to fprintf(3C).

**USAGE**
Applications using these functions should call va_end(ap) afterwards to clean up.

**ATTRIBUTES**
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
</tbody>
</table>

**SEE ALSO**
fprintf(3C), setlocale(3C), attributes(5), stdarg(3HEAD)

**NOTES**
The vwprintf(), vfwprintf(), and vswprintf() functions can be used safely in multithreaded applications, as long as setlocale(3C) is not being called to change the locale.
NAME
vlfmt – display error message in standard format and pass to logging and monitoring services

SYNOPSIS
#include <pfmt.h>
#include <stdarg.h>

int vlfmt(FILE *stream, long flag, const char *format, va_list ap);

DESCRIPTION
The vlfmt() function is identical to lfmt(3C), except that it is called with an argument list as defined by <stdarg.h>.

The <stdarg.h> header defines the type va_list and a set of macros for advancing through a list of arguments whose number and types may vary. The ap argument is of type va_list. This argument is used with the <stdarg.h> macros va_start(), va_arg(), and va_end(). See stdarg(3HEAD). The example in the EXAMPLES section below demonstrates their use with vlfmt().

RETURN VALUES
Upon successful completion, vlfmt() returns the number of bytes transmitted. Otherwise, −1 is returned if there was a write error to stream, or −2 is returned if unable to log and/or display at console.

EXAMPLES
EXAMPLE 1 Use of vlfmt() to write an errlog() routine.

The following example demonstrates how vlfmt() could be used to write an errlog() routine. The va_list() macro is used as the parameter list in a function definition. The va_start(ap,...) call, where ap is of type va_list, must be invoked before any attempt to traverse and access unnamed arguments. Calls to va_arg(ap, atype) traverse the argument list. Each execution of va_arg() expands to an expression with the value and type of the next argument in the list ap, which is the same object initialized by va_start(). The atype argument is the type that the returned argument is expected to be. The va_end(ap) macro must be invoked when all desired arguments have been accessed. The argument list in ap can be traversed again if va_start() is called again after va_end(). In the example below, va_arg() is executed first to retrieve the format string passed to errlog(). The remaining errlog() arguments (arg1, arg2, ...) are passed to vlfmt() in the argument ap.

#include <pfmt.h>
#include <stdarg.h>

/*
 * errlog should be called like
 * errlog(log_info, format, arg1, ...);
 */
void errlog(long log_info, ...) { 
    va_list ap;
    char *format;
    va_start(ap,);
    format = va_arg(ap, char *);
    (void) vlfmt(stderr, log_info|MM_ERROR, format, ap);
    va_end(ap);
    (void) abort();
}
EXAMPLE 1 Use of vlfmt() to write an errlog() routine.  (Continued)

Since vlfmt() uses getsxt(3C), it is recommended that vlfmt() not be used.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
gettxt(3C), lfmt(3C), attributes(5),stdarg(3HEAD)
vpfmt(3C)

NAME
vpfmt – display error message in standard format and pass to logging and monitoring services

SYNOPSIS
#include <pfmt.h>
#include <stdarg.h>

int vpfmt(FILE *stream, long flag, const char *format, va_list ap);

DESCRIPTION
The vpfmt() function is identical to pfmt(3C), except that it is called with an argument list as defined by <stdarg.h>.

The <stdarg.h> header defines the type va_list and a set of macros for advancing through a list of arguments whose number and types may vary. The ap argument is of type va_list. This argument is used with the <stdarg.h> macros va_start(), va_arg(), and va_end(). See stdarg(3HEAD). The example in the EXAMPLES section below demonstrates their use with vpfmt().

RETURN VALUES
Upon successful completion, vpfmt() returns the number of bytes transmitted. Otherwise, −1 is returned if there was a write error to stream.

EXAMPLES
EXAMPLE 1 Use of vpfmt() to write an error routine.

The following example demonstrates how vpfmt() could be used to write an error() routine. The va_alist() macro is used as the parameter list in a function definition. The va_start(ap, ...) call, where ap is of type va_list, must be invoked before any attempt to traverse and access unnamed arguments. Calls to va_arg(ap, atype) traverse the argument list. Each execution of va_arg() expands to an expression with the value and type of the next argument in the list ap, which is the same object initialized by va_start(). The atype argument is the type that the returned argument is expected to be. The va_end(ap) macro must be invoked when all desired arguments have been accessed. The argument list in ap can be traversed again if va_start() is called again after va_end(). In the example below, va_arg() is executed first to retrieve the format string passed to error(). The remaining error() arguments (arg1, arg2, ...) are passed to vpfmt() in the argument ap.

#include <pfmt.h>
#include <stdarg.h>
/*
 *   error should be called like
 *   error(format, arg1, ...);
 */
void error(...)
{
    va_list ap;
    char *format;
    va_start(ap, );
    format = va_arg(ap, char *);
    (void) vpfmt(stderr, MM_ERROR, format, ap);
    va_end(ap);
    (void) abort();
}
**EXAMPLE 1** Use of *vpfmt()* to write an error routine.  

**USAGE**
Since *vpfmt()* uses *gettext(3C)*, it is recommended that *vpfmt()* not be used.

**ATTRIBUTES**
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**
*gettext(3C), printf(3C), attributes(5), stdarg(3HEAD)*
vprintf(3C)

NAME
vprintf, vfprintf, vsprintf, vsnprintf – print formatted output of a variable argument list

SYNOPSIS
#include <stdio.h>
#include <stdarg.h>

int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *s, const char *format, va_list ap);
int vsnprintf(char *s, size_t n, const char *format, va_list ap);

DESCRIPTION
The vprintf(), vfprintf(), vsprintf() and vsnprintf() functions 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 in the <stdarg.h> header. See printf(3C) and stdarg(3HEAD).

The <stdarg.h> header defines the type va_list and a set of macros for advancing through a list of arguments whose number and types may vary. The argument ap to the vprintf family of functions is of type va_list. This argument is used with the <stdarg.h> header file macros va_start(), va_arg(), and va_end() (see stdarg(3HEAD)). The EXAMPLES section below demonstrates the use of va_start() and va_end() with vprintf().

The macro va_alist() is used as the parameter list in a function definition, as in the function called error() in the example below. The macro va_start(ap, parmN), where ap is of type va_list and parmN is the rightmost parameter (just before ...), must be called before any attempt to traverse and access unnamed arguments is made. The va_end(ap) macro must be invoked when all desired arguments have been accessed. The argument list in ap can be traversed again if va_start() is called again after va_end(). In the example below, the error() arguments (arg1, arg2, ...) are passed to vfprintf() in the argument ap.

RETURN VALUES
The vprintf(), vfprintf(), and vsprintf() functions return the number of characters transmitted (not including \0 in the case of vsprintf()). The vsnprintf() function returns the number of characters formatted, that is, the number of characters that would have been written to the buffer if it were large enough. Each function returns a negative value if an output error was encountered.

ERRORS
The vprintf() and vfprintf() functions will fail if either the stream is unbuffered or the stream’s buffer needed to be flushed and:

EFBIG
The file is a regular file and an attempt was made to write at or beyond the offset maximum.

EXAMPLES
EXAMPLE 1 Using vprintf() to write an error routine.

The following demonstrates how vprintf() could be used to write an error routine:
**EXAMPLE 1** Using `vprintf()` to write an error routine.  

```c
#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);
    /* print out remainder of message */
    (void) vfprintf(stderr, format, ap);
    va_end(ap);
    (void) abort;
}
```

**ATTRIBUTES**  
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**  
printf(3C), attributes(5), stdarg(3HEAD)
NAME
vsyslog – log message with a stdarg argument list

SYNOPSIS
#include <syslog.h>
#include <stdarg.h>

int vsyslog(int priority, const char *message, va_list ap);

DESCRIPTION
The vsyslog() function is identical to syslog(3C), except that it is called with an
argument list as defined by stdarg(3HEAD) rather than with a variable number of
arguments.

EXAMPLES
EXAMPLE 1 Use vsyslog() to write an error routine.

The following demonstrates how vsyslog() can be used to write an error routine.

#include <syslog.h>
#include <stdarg.h>

/*
 * error should be called like:
 *   error(pri, function_name, format, arg1, arg2...);
 */

void error(int pri, char *function_name, char *format, ...)
{
    va_list args;
    va_start(args, format);
    /* log name of function causing error */
    (void) syslog(pri, "ERROR in %s.", function_name);
    /* log remainder of message */
    (void) vsyslog(pri, format, args);
    va_end(args);
    (void) abort( );
}

main()
{
    error(LOG_ERR, "main", "process %d is dying", getpid());
}

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
stdarg(3HEAD), syslog(3C), attributes(5)
The wait3() function delays its caller until a signal is received or one of its child processes terminates or stops due to tracing. If any child process has died or stopped due to tracing and this has not already been reported, return is immediate, returning the process ID and status of one of those children. If that child process has died, it is discarded. If there are no children, −1 is returned immediately. If there are only running or stopped but reported children, the calling process is blocked.

If statusp is not a null pointer, then on return from a successful wait3() call, the status of the child process is stored in the integer pointed to by statusp. *statusp indicates the cause of termination and other information about the terminated process in the following manner:

- If the low-order 8 bits of *statusp are equal to 0177, the child process has stopped; the 8 bits higher up from the low-order 8 bits of *statusp contain the number of the signal that caused the process to stop. See signal(3HEAD).
- If the low-order 8 bits of *statusp are non-zero and are not equal to 0177, the child process terminated due to a signal; the low-order 7 bits of *statusp contain the number of the signal that terminated the process. In addition, if the low-order seventh bit of *statusp (that is, bit 0200) is set, a ‘core image’ of the process was produced; see signal(3HEAD).
- Otherwise, the child process terminated due to an exit() call; the 8 bits higher up from the low-order 8 bits of *statusp contain the low-order 8 bits of the argument that the child process passed to exit(); see exit(2).

The options argument is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in <sys/wait.h>:

- WNOHANG Execution of the calling process is not suspended if status is not immediately available for any child process.
- WUNTRACED The status of any child processes that are stopped, and whose status has not yet been reported since they stopped, are also reported to the requesting process.

If rusage is not a null pointer, a summary of the resources used by the terminated process and all its children is returned. Only the user time used and the system time used are currently available. They are returned in the ru_utime and ru_stime, members of the rusage structure, respectively.
When the WNOHANG option is specified and no processes have status to report, 
wait3() returns 0. The WNOHANG and WUNTRACED options may be combined by the 
bitwise OR operation of the two values.

The wait4() function is an extended interface. With a pid argument of 0, it is 
equivalent to wait3(). If pid has a nonzero value, then wait4() returns status only 
for the indicated process ID, but not for any other child processes. The status can be 
evaluated using the macros defined by wstat(3XFN).

**RETURN VALUES**

If wait3() or wait4() returns due to a stopped or terminated child process, the 
process ID of the child is returned to the calling process. Otherwise, −1 is returned and 
errno is set to indicate the error.

If wait3() or wait4() return due to the delivery of a signal to the calling process, 
−1 is returned and errno is set to EINTR. If WNOHANG was set in options, it has at least 
one child process specified by pid for which status is not available, and status is not 
available for any process specified by pid, 0 is returned. Otherwise, −1 is returned and 
errno is set to indicate the error.

The wait3() and wait4() functions return 0 if WNOHANG is specified and there are 
no stopped or exited children, and return the process ID of the child process if they 
return due to a stopped or terminated child process. Otherwise, they return −1 and set 
errno to indicate the error.

**ERRORS**

The wait3() and wait4() functions will fail and return immediately if:

ECHILD The calling process has no existing unwaited-for child processes.

EFAULT The statusp or rusage arguments point to an illegal address.

EINTR The function was interrupted by a signal. The value of the location 
pointed to by statusp is undefined.

EINVAL The value of options is not valid.

The wait4() function may fail if:

ECHILD The process specified by pid does not exist or is not a child of the 
calling process.

The wait3() and wait4() functions will terminate prematurely, return −1, and set 
errno to EINTR upon the arrival of a signal whose SA_RESTART bit in its flags field 
is not set (see sigaction(2)).

**SEE ALSO**

kill(1), exit(2), wait(2), waitid(2), waitpid(2), getrusage(3C), signal(3C), proc(4), signal(3HEAD), wstat(3XFN)

**NOTES**

If a parent process terminates without waiting on its children, the initialization process 
(process ID = 1) inherits the children.
The `wait3()` and `wait4()` functions are automatically restarted when a process receives a signal while awaiting termination of a child process, unless the `SA_RESTART` bit is not set in the flags for that signal.
NAME
wait, wait3, wait4, waitpid, WIFSTOPPED, WIFSIGNALED, WIFEXITED – wait for process to terminate or stop

SYNOPSIS
/usr/ucb/cc [ flag ... ] file ...
#include <sys/wait.h>
int wait( statusp );
int *statusp;
int waitpid( pid, statusp, options );
int pid;
int *statusp;
int options;
#include <sys/time.h>
#include <sys/resource.h>
int wait3( statusp, options, rusage );
int *statusp;
int options;
struct rusage *rusage;
int wait4( pid, statusp, options, rusage );
int pid;
int *statusp;
int options;
struct rusage *rusage;
WIFSTOPPED( status );
int status;
WIFSIGNALED( status );
int status;
WIFEXITED( status );
int status;

DESCRIPTION
wait() delays its caller until a signal is received or one of its child processes terminates or stops due to tracing. If any child process has died or stopped due to tracing and this has not been reported using wait(), return is immediate, returning the process ID and exit status of one of those children. If that child process has died, it is discarded. If there are no children, return is immediate with the value −1 returned. If there are only running or stopped but reported children, the calling process is blocked.

If status is not a NULL pointer, then on return from a successful wait() call the status of the child process whose process ID is the return value of wait() is stored in the wait() union pointed to by status. The w_status member of that union is an int; it indicates the cause of termination and other information about the terminated process in the following manner:

wait(3UCB)
If the low-order 8 bits of `w_status` are equal to 0177, the child process has stopped; the 8 bits higher up from the low-order 8 bits of `w_status` contain the number of the signal that caused the process to stop. See `ptrace(2)` and `sigvec(3UCB)`.

If the low-order 8 bits of `w_status` are non-zero and are not equal to 0177, the child process terminated due to a signal; the low-order 7 bits of `w_status` contain the number of the signal that terminated the process. In addition, if the low-order seventh bit of `w_status` (that is, bit 0200) is set, a “core image” of the process was produced; see `sigvec(3UCB)`.

Otherwise, the child process terminated due to an `exit()` call; the 8 bits higher up from the low-order 8 bits of `w_status` contain the low-order 8 bits of the argument that the child process passed to `exit();` see `exit(2)`.

`waitpid()` behaves identically to `wait()` if `pid` has a value of −1 and `options` has a value of zero. Otherwise, the behavior of `waitpid()` is modified by the values of `pid` and `options` as follows:

`pid` specifies a set of child processes for which status is requested. `waitpid()` only returns the status of a child process from this set.

- If `pid` is equal to −1, status is requested for any child process. In this respect, `waitpid()` is then equivalent to `wait()`.
- If `pid` is greater than zero, it specifies the process ID of a single child process for which status is requested.
- If `pid` is equal to zero, status is requested for any child process whose process group ID is equal to that of the calling process.
- If `pid` is less than −1, status is requested for any child process whose process group ID is equal to the absolute value of `pid`.

`options` is constructed from the bitwise inclusive OR of zero or more of the following flags, defined in the header `<sys/wait.h>`:

- **WNOHANG** `waitpid()` does not suspend execution of the calling process 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, are also reported to the requesting process.

`wait3()` is an alternate interface that allows both non-blocking status collection and the collection of the status of children stopped by any means. The `status` parameter is defined as above. The `options` parameter is used to indicate the call should not block if there are no processes that have status to report (`WNOHANG`), and/or that children of the current process that are stopped due to a `SIGTTIN`, `SIGTTOU`, `SIGTSTP`, or `SIGSTOP` signal are eligible to have their status reported as well (`WUNTRACED`). A terminated child is discarded after it reports status, and a stopped process will not...
report its status more than once. If rusage is not a NULL pointer, a summary of the resources used by the terminated process and all its children is returned. Only the user time used and the system time used are currently available. They are returned in rusage.ru_utime and rusage.ru_stime, respectively.

When the WNOHANG option is specified and no processes have status to report, wait3() returns 0. The WNOHANG and WUNTRACED options may be combined by ORing the two values.

wait4() is another alternate interface. With a pid argument of 0, it is equivalent to wait3(). If pid has a nonzero value, then wait4() returns status only for the indicated process ID, but not for any other child processes.

WIFSTOPPED, WIFSIGNALED, WIFEXITED, are macros that take an argument status, of type int, as returned by wait(), or wait3(), or wait4(). WIFSTOPPED evaluates to true (1) when the process for which the wait() call was made is stopped, or to false (0) otherwise. WIFSIGNALED evaluates to true when the process was terminated with a signal. WIFEXITED evaluates to true when the process exited by using an exit(2) call.

RETURN VALUES

If wait() or waitpid() returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of −1 is returned and errno is set to indicate the error.

If wait() or waitpid() return due to the delivery of a signal to the calling process, a value of −1 is returned and errno is set to EINTR. If waitpid() function 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, a value of zero is returned. Otherwise, a value of −1 is returned, and errno is set to indicate the error.

wait3() and wait4() returns 0 if WNOHANG is specified and there are no stopped or exited children, and returns the process ID of the child process if it returns due to a stopped or terminated child process. Otherwise, they returns a value of −1 and sets errno to indicate the error.

ERRORS

wait(), wait3() or wait4() will fail and return immediately if one or more of the following are true:

ECHILD The calling process has no existing unwaited-for child processes.

EFAULT The status or rusage arguments point to an illegal address.

waitpid() may set errno to:

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 statusp is undefined.
EINVAL The value of `options` is not valid.

`wait()`, `wait3()` and `wait4()` will terminate prematurely, return −1, and set
`errno` to `EINTR` upon the arrival of a signal whose `SV_INTERRUPT` bit in its flags
field is set (see `sigvec(3UCB)` and `siginterrupt(3UCB)`). `signal(3UCB)`, sets this
bit for any signal it catches.

**SEE ALSO**

`exit(2)`, `ptrace(2)`, `wait(2)`, `waitpid(2)`, `getrusage(3C)`, `siginterrupt(3UCB)`,
`signal(3UCB)`, `sigvec(3UCB)`, `signal(3C)`

**NOTES**

Use of these interfaces should be restricted to only applications written on BSD
platforms. Use of these interfaces with any of the system libraries or in multi-thread
applications is unsupported.

If a parent process terminates without waiting on its children, the initialization process
(process ID = 1) inherits the children.

`wait()`, `wait3()` and `wait4()` are automatically restarted when a process
receives a signal while awaiting termination of a child process, unless the
`SV_INTERRUPT` bit is set in the flags for that signal.

Calls to `wait()` with an argument of 0 should be cast to type `int *`, as in:

```c
wait((int *)0)
```

Previous SunOS releases used `union wait*statusp` and `union wait status` in
place of `int *statusp` and `int status`. The union contained a member `w_status`
that could be treated in the same way as `status`.

Other members of the `wait` union could be used to extract this information more
conveniently:

- If the `w_stopval` member had the value `WSTOPPED`, the child process had
  stopped; the value of the `w_stopsig` member was the signal that stopped the
  process.
- If the `w_termsig` member was non-zero, the child process terminated due to a
  signal; the value of the `w_termsig` member was the number of the signal that
  terminated the process. If the `w_coredump` member was non-zero, a core dump
  was produced.
- Otherwise, the child process terminated due to a call to `exit()`. The value of the
  `w_retcode` member was the low-order 8 bits of the argument that the child
  process passed to `exit()`.

`union wait` is obsolete in light of the new specifications provided by `IEEE Std
1003.1-1988` and endorsed by `SVID89` and `XPG3`. SunOS Release 4.1 supports
union wait for backward compatibility, but it will disappear in a future release.
The `walkcontext`() function walks the call stack pointed to by `uptr`, which can be obtained by a call to `getcontext(2)` or from a signal handler installed with the `SA_SIGINFO` flag. The `walkcontext`() function calls the user-supplied function `operate_func` for each routine found on the call stack and each signal handler invoked. The user function is passed three arguments: the PC at which the call or signal occurred, the signal number that occurred at this PC (0 if no signal occurred), and the third argument passed to `walkcontext()`. If the user function returns a non-zero value, `walkcontext()` returns without completing the callstack walk.

The `printstack()` function uses `walkcontext()` to print a symbolic stack trace to the specified file descriptor. This is useful for reporting errors from signal handlers. The `printstack()` function uses `dladdr()` (see `dladdr(3DL)`) to obtain symbolic symbol names. As a result, only global symbols are reported as symbol names by `printstack()`.

Upon successful completion, `walkstack()` and `printstack()` return 0. If `walkstack()` cannot read the stack or the stack trace appears corrupted, both functions return -1.

No error values are defined.

The `walkcontext()` function is typically used to obtain information about the call stack for error reporting, performance analysis, or diagnostic purposes. Many library functions are not Async-Signal-Safe and should not be used from a signal handler. If `walkcontext()` is to be called from a signal handler, careful programming is required. In particular, `stdio(3C)` and `malloc(3C)` cannot be used.

The `printstack()` function is Async-Signal-Safe and can be called from a signal handler. The output format from `printstack()` is unstable, as it varies with the scope of the routines.

Tail-call optimizations on SPARC eliminate stack frames that would otherwise be present. For example, if the code is of the form

```c
#include <stdio.h>

main()
{
    bar();
    exit(0);
}

bar()
```

...


```c
{
    int a;
    a = foo(fileno(stdout));
    return (a);
}

foo(int file)
{
    printstack(file);
}
```

compiling without optimization will yield a stack trace of the form

```
/tmp/q:foo+0x8
/tmp/q:bar+0x14
/tmp/q:main+0x4
/tmp/q:_start+0xb8
```

whereas with higher levels of optimization the output is

```
/tmp/q:main+0x10
/tmp/q:_start+0xb8
```

since both the call to `foo()` in main and the call to `bar()` in `foo()` are handled as
tail calls that perform a return or restore in the delay slot. For further information, see
`The SPARC Architecture Manual`.

**Attributes**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Stability</td>
<td>Stable</td>
</tr>
<tr>
<td>MT-Level</td>
<td>Async-Signal-Safe</td>
</tr>
</tbody>
</table>

**See Also**

`intro(2), getcontext(2), sigaction(2), dladdr(3DL), siginfo(3HEAD), attributes(5)`

NAME  
watchmalloc, cfreeto memalign, valloc – debugging memory allocator

SYNOPSIS
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
void *memalign(size_t alignment, size_t size);
void *valloc(size_t size);
void *calloc(size_t nelem, size_t elsize);

#include <malloc.h>

int mallopt(int cmd, int value);

DESCRIPTION
The collection of malloc() functions in this shared object are an optional replacement for the standard versions of the same functions in the system C library. See malloc(3C). They provide a more strict interface than the standard versions and enable enforcement of the interface through the watchpoint facility of /proc. See proc(4).

Any dynamically linked application can be run with these functions in place of the standard functions if the following string is present in the environment (see ld.so.1):

LD_PRELOAD=watchmalloc.so.1

The individual function interfaces are identical to the standard ones as described in malloc(3C). However, laxities provided in the standard versions are not permitted when the watchpoint facility is enabled (see WATCHPOINTS below):

- Memory may not be freed more than once.
- A pointer to freed memory may not be used in a call to realloc().
- A call to malloc() immediately following a call to free() will not return the same space.
- Any reference to memory that has been freed yields undefined results.

To enforce these restrictions partially, without great loss in speed as compared to the watchpoint facility described below, a freed block of memory is overwritten with the pattern 0xdeadbeef before returning from free(). The malloc() function returns with the allocated memory filled with the pattern 0xbaddcafe as a precaution against applications incorrectly expecting to receive back unmodified memory from the last free(). The calloc() function always returns with the memory zero-filled.
watchmalloc(3MALLOC)

Entry points for mallopt() and mallinfo() are provided as empty routines, and are present only because some malloc() implementations provide them.

WATCHPOINTS

The watchpoint facility of /proc can be applied by a process to itself. The functions in watchmalloc.so.1 use this feature if the following string is present in the environment:

MALLOC_DEBUG=WATCH

This causes every block of freed memory to be covered with WA_WRITE watched areas. If the application attempts to write any part of freed memory, it will trigger a watchpoint trap, resulting in a SIGTRAP signal, which normally produces an application core dump.

A header is maintained before each block of allocated memory. Each header is covered with a watched area, thereby providing a red zone before and after each block of allocated memory (the header for the subsequent memory block serves as the trailing red zone for its preceding memory block). Writing just before or just after a memory block returned by malloc() will trigger a watchpoint trap.

Watchpoints incur a large performance penalty. Requesting MALLOC_DEBUG=WATCH can cause the application to run 10 to 100 times slower, depending on the use made of allocated memory.

Further options are enabled by specifying a comma-separated string of options:

MALLOC_DEBUG=WATCH,RW,STOP

WATCH

Enables WA_WRITE watched areas as described above.

RW

Enables both WA_READ and WA_WRITE watched areas. An attempt either to read or write freed memory or the red zones will trigger a watchpoint trap. This incurs even more overhead and can cause the application to run up to 1000 times slower.

STOP

The process will stop showing a FLTWATCH machine fault if it triggers a watchpoint trap, rather than dumping core with a SIGTRAP signal. This allows a debugger to be attached to the live process at the point where it underwent the watchpoint trap. Also, the various /proc tools described in proc(1) can be used to examine the stopped process.

One of WATCH or RW must be specified, else the watchpoint facility is not engaged. RW overrides WATCH. Unrecognized options are silently ignored.
Sizes of memory blocks allocated by malloc() are rounded up to the worst-case alignment size, 8 bytes for 32-bit processes and 16 bytes for 64-bit processes. Accessing the extra space allocated for a memory block is technically a memory violation but is in fact innocuous. Such accesses are not detected by the watchpoint facility of watchmalloc.

Interposition of watchmalloc.so.1 fails innocuously if the target application is statically linked with respect to its malloc() functions.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO proc(1), bsdmalloc(3MALLOC), calloc(3C), free(3C), malloc(3C), malloc (3MALLOC), mapmalloc(3MALLOC), memalign(3C), realloc(3C), valloc(3C), libmapmalloc(3LIB), proc(4), attributes(5)
wcrtomb — convert a wide-character code to a character (restartable)

#include <stdio.h>

size_t wcrtomb(char *, wchar_t wc, mbstate_t *ps);

If s is a null pointer, the wcrtomb() function is equivalent to the call:
wcrtomb(buf, L'\0', ps)where buf is an internal buffer.

If s is not a null pointer, the wcrtomb() function determines the number of bytes
needed to represent the character that corresponds to the wide-character given by wc
(including any shift sequences), and stores the resulting bytes in the array whose first
element is pointed to by s. At most MB_CUR_MAX bytes are stored. If wc is a null
wide-character, a null byte is stored, preceded by any shift sequence needed to restore
the initial shift state. The resulting state described is the initial conversion state.

If ps is a null pointer, the wcrtomb() function uses its own internal mbstate_t
object, which is initialized at program startup to the initial conversion state.
Otherwise, the mbstate_t object pointed to by ps is used to completely describe the
conversion state of the associated character sequence. Solaris will behave as if
no function defined in the Solaris Reference Manual calls wcrtomb().

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

The wcrtomb() function returns the number of bytes stored in the array object
(including any shift sequences). When wc is not a valid wide-character, an encoding
error occurs. In this case, the function stores the value of the macros EILSEQ in errno
and returns (size_t)−1; the conversion state is undefined.

The wcrtomb() function may fail if:

EINVAL The ps argument points to an object that contains an invalid
conversion state.

EILSEQ Invalid wide-character code is detected.

If ps is not a null pointer, wcrtomb() uses the mbstate_t object pointed to by ps and
the function can be used safely in multithreaded applications, as long as
setlocale(3C) is not being called to change the locale. If ps is a null pointer,
w.crtomb() uses its internal mbstate_t object and the function is Unsafe in
multithreaded applications.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See NOTES below</td>
</tr>
</tbody>
</table>
SEE ALSO mbsinit(3C), setlocale(3C), attributes(5), environ(5)

wctomb(3C)
wcscoll(3C)

NAME    wcscoll, wscoll — wide character string comparison using collating information

SYNOPSIS  
#include <wchar.h>

int wcscoll(const wchar_t *ws1, const wchar_t *ws2);
int wscoll(const wchar_t *ws1, const wchar_t *ws2);

DESCRIPTION  
The wcscoll() and wscoll() functions compare the wide character string pointed to by ws1 to the wide character string pointed to by ws2, both interpreted as appropriate to the LC_COLLATE category of the current locale.

RETURN VALUES  
Upon successful completion, wcscoll() and wscoll() return an integer greater than, equal to, or less than 0, depending upon whether the wide character string pointed to by ws1 is greater than, equal to, or less than the wide character string pointed to by ws2, when both are interpreted as appropriate to the current locale. On error, wcscoll() and wscoll() may set errno, but no return value is reserved to indicate an error.

ERRORS  
The wcscoll() and wscoll() functions may fail if:

EINVAL    The ws1 or ws2 arguments contain wide character codes outside the domain of the collating sequence.
ENOSYS    The function is not supported.

USAGE  
Because no return value is reserved to indicate an error, an application wishing to check for error situations should set errno to 0, call either wcscoll() or wscoll(), then check errno and if it is non-zero, assume an error has occurred.

The wcsxfrm(3C) and wcscmp(3C) functions should be used for sorting large lists.

The wcscoll() and wscoll() functions can be used safely in multithreaded applications as long as setlocale(3C) is not being called to change the locale.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO  
setlocale(3C), wcscmp(3C), wcsxfrm(3C), attributes(5)
wcsftime – convert date and time to wide character string

#include <wchar.h>

size_t wcsftime(wchar_t *wcs, size_t maxsize, const char *format, const struct tm *timptr);

size_t wcsftime(wchar_t *wcs, size_t maxsize, const wchar_t *format, const struct tm *timptr);

The `wcsftime()` function is equivalent to the `strftime(3C)` function, except that:

- The argument `wcs` points to the initial element of an array of wide-characters into which the generated output is to be placed.
- The argument `maxsize` indicates the maximum number of wide-characters to be placed in the output array.
- The argument `format` is a wide-character string and the conversion specifications are replaced by corresponding sequences of wide-characters.
- The return value indicates the number of wide-characters placed in the output array.

If copying takes place between objects that overlap, the behavior is undefined.

If the total number of resulting wide character codes (including the terminating null wide-character code) is no more than `maxsize`, `wcsftime()` returns the number of wide-character codes placed into the array pointed to by `wcs`, not including the terminating null wide-character code. Otherwise, 0 is returned and the contents of the array are indeterminate.

The `wcsftime()` function uses `malloc(3C)` and should `malloc()` fail, `errno` will be set by `malloc()`.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

See also `malloc(3C), setlocale(3C), strftime(3C), attributes(5), standards(5)`

Notes

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

NAME
wcsrtombs – convert a wide-character string to a character string (restartable)

SYNOPSIS
#include <wchar.h>

size_t wcsrtombs(char *dst, const wchar_t **src, size_t len,
                 mbstate_t *ps);

DESCRIPTION
The wcsrtombs() function converts a sequence of wide-characters from the array indirectly pointed to by src into a sequence of corresponding characters, beginning in the conversion state described by the object pointed to by ps. If dst is not a null pointer, the converted characters are then stored into the array pointed to by dst. Conversion continues up to and including a terminating null wide-character, which is also stored. Conversion stops earlier in the following cases:

- When a code is reached that does not correspond to a valid character.
- When the next character would exceed the limit of len total bytes to be stored in the array pointed to by dst (and dst is not a null pointer).

Each conversion takes place as if by a call to the wcrtomb() function.

If dst is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide-character) or the address just past the last wide-character converted (if any). If conversion stopped due to reaching a terminating null wide-character, the resulting state described is the initial conversion state.

If ps is a null pointer, the wcsrtombs() function uses its own internal mbstate_t object, which is initialized at program startup to the initial conversion state. Otherwise, the mbstate_t object pointed to by ps is used to completely describe the current conversion state of the associated character sequence. Solaris will behave as if no function defined in the Solaris Reference Manual calls wcsrtombs().

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

RETURN VALUES
If conversion stops because a code is reached that does not correspond to a valid character, an encoding error occurs. In this case, the wcsrtombs() function stores the value of the macro EILSEQ in errno and returns (size_t)-1; the conversion state is undefined. Otherwise, it returns the number of bytes in the resulting character sequence, not including the terminating null (if any).

ERRORS
The wcsrtombs() function may fail if:

EINVAL
The ps argument points to an object that contains an invalid conversion state.

EILSEQ
A wide-character code does not correspond to a valid character.

USAGE
If ps is not a null pointer, wcsrtombs() uses the mbstate_t object pointed to by ps and the function can be used safely in multithreaded applications, as long as setlocale(3C) is not being called to change the locale. If ps is a null pointer,
wcsrtombs() uses its internal mbstate_t object and the function is Unsafe in multithreaded applications.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>See NOTES below</td>
</tr>
</tbody>
</table>

SEE ALSO
mbsinit(3C), setlocale(3C), wcrtomb(3C), attributes(5), environ(5)
wcsstr(3C)

NAME  wcsstr – find a wide-character substring

SYNOPSIS  #include <wchar.h>

wchar_t *wcsstr(const wchar_t *ws1, const wchar_t *ws2);

ISO C++  #include <wchar.h>

const wchar_t *wcsstr(const wchar_t *ws1, const wchar_t *ws2);

#include <cwchar>

wchar_t *std::wcsstr(wchar_t *ws1, const wchar_t *ws2);

DESCRIPTION  The wcsstr() function locates the first occurrence in the wide-character string
pointed to by ws1 of the sequence of wide-characters (excluding the terminating null
wide-character) in the wide-character string pointed to by ws2.

RETURN VALUES  On successful completion, wcsstr() returns a pointer to the located wide-character
string, or a null pointer if the wide-character string is not found.

If ws2 points to a wide-character string with zero length, the function returns ws1.

ERRORS  No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  wszchr(3C), attributes(5)
The wcstod() and wstod() functions convert the initial portion of the wide character string pointed to by *nptr to double representation. They first decompose the input wide character string into three parts: an initial, possibly empty, sequence of white-space wide character codes (as specified by iswspace(3C)); a subject sequence interpreted as a floating-point constant; and a final wide-character string of one or more unrecognised wide-character codes, including the terminating null wide character code of the input wide character string. They then attempt to convert the subject sequence to a floating-point number, and return 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 program’s 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 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 *nptr is stored in the object pointed to by *endptr, provided that *endptr is not a null pointer.

The watof(str) function is equivalent to wstod(str, (wchar_t **)NULL).
The `wcstod()` and `wstod()` functions 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, `+HUGE_VAL` is returned (according to the sign of the value), and `errno` is set to `ERANGE`.

If the correct value would cause underflow, 0 is returned, and `errno` is set to `ERANGE`.

The `wcstod()` and `wstod()` functions will fail if:

- **ERANGE**: The value to be returned would cause overflow or underflow.

The `wcstod()` and `wcstod()` functions may fail if:

- **EINVAL**: No conversion could be performed.

**USAGE**

Because 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set `errno` to 0 call `wcstod()` or `wstod()`, then check `errno` and if it is non-zero, assume an error has occurred.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`iswspace(3C), localeconv(3C), scanf(3C), setlocale(3C), wcstol(3C), attributes(5)`
The `wcstol()` and `wstol()` functions convert the initial portion of the wide character string pointed to by `nptr` to `long int` representation. They first decompose the input wide character string into three parts: an initial, possibly empty, sequence of white-space wide-character codes (as specified by `iswspace(3C)`), 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 unrecognised wide character codes, including the terminating null wide-character code of the input wide character string. They then attempt to convert the subject sequence to an integer, and return the result.

If the value of `base` is 0, the expected form of the subject sequence is that of a decimal constant, octal constant or hexadecimal constant, any of which may be preceded by a '+' or '-' sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix '0' optionally followed by a sequence of the digits '0' to '7' only. A hexadecimal constant consists of the prefix '0x' or '0X' followed by a sequence of the decimal digits and letters 'a' (or 'A') to 'f' (or 'F') with values 10 to 15 respectively.

If the value of `base` is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a '+' or '-' sign, but not including an integer suffix. The letters from 'a' (or 'A') to 'z' (or 'Z') inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the wide-character code representations of '0x' or '0X' may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide character string, starting with the first non-white-space wide-character code, that is of the expected form. The subject sequence contains no wide-character codes if the input wide character string is empty or consists entirely of white-space wide-character code, or if the first non-white-space wide-character code is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of `base` is 0, the sequence of wide-character codes starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it is used as the base for conversion, ascribing to each letter its
value as given above. If the subject sequence begins with a minus sign (-), the value resulting from the conversion is negated. A pointer to the final wide character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

In other than the POSIX locale, additional implementation-dependent subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The *watol()* function is equivalent to *wstol(str, (wchar_t **)NULL, 10)*.

The *watoll()* function is the long-long (double long) version of *watoi()*.

The *watoi()* function is equivalent to (int)watoi().

Upon successful completion, *wcstol()* and *wstol()* return the converted value, if any. If no conversion could be performed, 0 is returned, and *errno* may be set to indicate the error. If the correct value is outside the range of representable values, \{LONG_MAX\} or \{LONG_MIN\} is returned (according to the sign of the value), and *errno* is set to ERANGE.

The *wcstol()* and *wstol()* functions will fail if:

EINVAL    The value of *base* is not supported.
ERANGE    The value to be returned is not representable.

The *wcstol()* and *wstol()* functions may fail if:

EINVAL    No conversion could be performed.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  iswalpha(3C), iswspace(3C), scanf(3C), wcstod(3C), attributes(5)

NOTES

Because 0, \{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, call *wcstol()* or *wstol()* , then check *errno* and if it is non-zero assume an error has occurred.

Truncation from long long to long can take place upon assignment or by an explicit cast.
NAME
wcstombs – convert a wide-character string to a character string

SYNOPSIS
#include <stdlib.h>

size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);

DESCRIPTION
The wcstombs() function converts the sequence of wide-character codes from the
array pointed to by pwcs into a sequence of characters and stores these characters into
the array pointed to by s, stopping if a character would exceed the limit of n total
bytes or if a null byte is stored. Each wide-character code is converted as if by a call to
wctomb(3C).

The behavior of this function is affected by the LC_CTYPE category of the current
locale.

No more than n bytes will be modified in the array pointed to by s. If copying takes
place between objects that overlap, the behavior is undefined. If s is a null pointer,
wcstombs() returns the length required to convert the entire array regardless of the
value of n, but no values are stored.

RETURN VALUES
If a wide-character code is encountered that does not correspond to a valid character
(of one or more bytes each), wcstombs() returns (size_t)-1. Otherwise,
wcstombs() returns the number of bytes stored in the character array, not including
any terminating null byte. The array will not be null-terminated if the value returned
is n.

ERRORS
The wcstombs() function may fail if:

EILSEQ A wide-character code does not correspond to a valid character.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO
mblen(3C), mbstowcs(3C), mbtowc(3C), setlocale(3C), wctomb(3C),
attributes(5)
wcstoul() function converts the initial portion of the wide character string pointed to by `nptr` to unsigned long int representation. It first decomposes the input wide-character string into three parts: an initial, possibly empty, sequence of white-space wide-character codes (as specified by the function `iswspace(3C)`); 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. It then attempts to convert the subject sequence to an unsigned integer, and returns the result.

If the value of `base` is 0, the expected form of the subject sequence is that of a decimal constant, an octal constant, or a hexadecimal constant, any of which may be preceded by a `+` or a `−` sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix `0`, optionally followed by a sequence of the digits `0` to `7` only. A hexadecimal constant consists of the prefix `0x` or `0X`, followed by a sequence of the decimal digits and letters `a` (or `A`) to `f` (or `F`), with values 10 to 15, respectively.

If the value of `base` is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by `base`, optionally preceded by a `+` or a `−` sign, but not including an integer suffix. The letters from `a` (or `A`) to `z` (or `Z`) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the wide-character codes `0x` or `0X` may optionally precede the sequence of letters and digits, following the sign, if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first wide-character code that is not a white space and is of the expected form. The subject sequence contains no wide-character codes if the input wide-character string is empty or consists entirely of white-space wide-character codes, or if the first wide-character code that is not a white space is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of `base` is 0, the sequence of wide-character codes starting with the first digit is interpreted as an integer constant. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide character string is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

In other than the POSIX locale, additional 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.

Because 0 and ULONG_MAX are returned on error and 0 is also a valid return on success, an application wishing to check for error situations should set errno to 0, call wcstoul(), then check errno and if it is non-zero, assume an error has occurred.

**RETURN VALUE**
Upon successful completion, wcstoul() returns the converted value, if any, and does not change the setting of errno. If no conversion could be performed, 0 is returned and errno may be set to indicate the error. If the correct value is outside the range of representable values, ULONG_MAX is returned and errno is set to ERANGE.

**ERRORS**
The wcstoul() function will fail if:
- EINVAL The value of base is not supported.
- ERANGE The value to be returned is not representable.

The wcstoul() function may fail if:
- EINVAL No conversion could be performed.

**USAGE**
Unlike wcstod(3C) and wcstol(3C), wcstoul() must always return a non-negative number; using the return value of wcstoul() for out-of-range numbers with wcstoul() could cause more severe problems than just loss of precision if those numbers can ever be negative.

**ATTRIBUTES**
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

**SEE ALSO**
isspace(3C), iswalpha(3C), scanf(3C), wcstod(3C), wcstol(3C), attributes(5)
NAME
wcstring, wcscat, wscat, wcscn cat, wscn cat, wscmp, wscmp, wcscmp, wscncmp, wcscpy, wscpy, wcscnpy, wscnpy, wcslen, wslen, wcscr, wscr, wscrchr, wscrchr, windex, windexe, wcspbrk, wsspbrk, wssbrk, wcscws, wsswsc, wcsspn, wsspnn, wcsspn, wsscsnp, wcstok, wstok – wide-character string operations

SYNOPSIS
#include <wchar.h>

wchar_t * wcscat (wchar_t * ws1, const wchar_t * ws2);
wchar_t * wcscn cat (wchar_t * ws1, const wchar_t * ws2, size_t n);
int wcscmp (const wchar_t * ws1, const wchar_t * ws2);
int wcscncmp (const wchar_t * ws1, const wchar_t * ws2, size_t n);
wchar_t * wscpy (wchar_t * ws1, const wchar_t * ws2);
wchar_t * wcsncpy (wchar_t * ws1, const wchar_t * ws2, size_t n);
size_t wcslen (const wchar_t * ws);
wchar_t * wcschr (const wchar_t * ws, wchar_t wc);
wchar_t * wcsrchr (const wchar_t * ws, wchar_t wc);
wchar_t * wcspbrk (const wchar_t * ws1, const wchar_t * ws2);
wchar_t * wcswcs (const wchar_t * ws1, const wchar_t * ws2);
size_t wcsspn (const wchar_t * ws1, const wchar_t * ws2);
size_t wcscspn (const wchar_t * ws1, const wchar_t * ws2);
wchar_t * wcstok (wchar_t * ws1, const wchar_t * ws2);
wchar_t * wcstok (wchar_t * ws1, const wchar_t * ws2, wchar_t ** ptr);

#include <widec.h>
wchar_t * wscat (wchar_t * ws1, const wchar_t * ws2);
wchar_t * wsncat (wchar_t * ws1, const wchar_t * ws2, size_t n);
int wscmp (const wchar_t * ws1, const wchar_t * ws2);
int wsncmp (const wchar_t * ws1, const wchar_t * ws2, size_t n);
wchar_t * wscpy (wchar_t * ws1, const wchar_t * ws2);
wchar_t * wsncpy (wchar_t * ws1, const wchar_t * ws2, size_t n);
size_t wslen (const wchar_t * ws);
wchar_t * wschr (const wchar_t * ws, wchat_t wc);
wchar_t * wsrchr (const wchar_t * ws, wchat_t wc);
wchar_t * wspbrk (const wchar_t * ws1, const wchar_t * ws2);
size_t wsspn (const wchar_t * ws1, const wchar_t * ws2);

634  man pages section 3: Basic Library Functions • Last Revised 28 Jan 1998
These functions operate on wide-character strings terminated by wchar_t NULL characters. During appending or copying, these routines do not check for an overflow condition of the receiving string. In the following, ws, ws1, and ws2 point to wide-character strings terminated by a wchar_t NULL.

\textbf{wcscat()}, \textbf{wcat()}

The \texttt{wcscat()} and \texttt{wcat()} functions append a copy of the wide-character string pointed to by \texttt{ws2} (including the terminating null wide-character code) to the end of the wide-character string pointed to by \texttt{ws1}. The initial wide-character code of \texttt{ws2} overwrites the null wide-character code at the end of \texttt{ws1}. If copying takes place between objects that overlap, the behavior is undefined. Both functions return \texttt{ws1}; no return value is reserved to indicate an error.

\textbf{wcsncat()}, \textbf{wsncat()}

The \texttt{wcsncat()} and \texttt{wsncat()} functions append not more than \texttt{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 \texttt{ws2} to the end of the wide-character string pointed to by \texttt{ws1}. The initial wide-character code of \texttt{ws2} overwrites the null wide-character code at the end of \texttt{ws1}. A terminating null wide-character code is always appended to the result. Both functions return \texttt{ws1}; no return value is reserved to indicate an error.

\textbf{wcscmp()}, \textbf{wscmp()}

The \texttt{wcscmp()} and \texttt{wscmp()} functions compare the wide-character string pointed to by \texttt{ws1} to the wide-character string pointed to by \texttt{ws2}. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared. Upon completion, both functions return an integer greater than, equal to, or less than zero, if the wide-character string pointed to by \texttt{ws1} is greater than, equal to, or less than the wide-character string pointed to by \texttt{ws2}. 
The `wcscmp()` and `wsncmp()` functions compare not more than \( n \) wide-character codes (wide-character codes that follow a null wide character code are not compared) from the array pointed to by `ws1` to the array pointed to by `ws2`. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared. Upon successful completion, both functions return an integer greater than, equal to, or less than zero, if the possibly null-terminated array pointed to by `ws1` is greater than, equal to, or less than the possibly null-terminated array pointed to by `ws2`.

The `wcscpy()` and `wscpy()` functions copy the wide-character string pointed to by `ws2` (including the terminating null wide-character code) into the array pointed to by `ws1`. If copying takes place between objects that overlap, the behavior is undefined. Both functions return `ws1`; no return value is reserved to indicate an error.

The `wcsncpy()` and `wsncpy()` functions copy not more than \( n \) wide-character codes (wide-character codes that follow a null wide character code are not copied) from the array pointed to by `ws2` to the array pointed to by `ws1`. If copying takes place between objects that overlap, the behavior is undefined. If the array pointed to by `ws2` is a wide-character string that is shorter than \( n \) wide-character codes, null wide-character codes are appended to the copy in the array pointed to by `ws1`, until a total \( n \) wide-character codes are written. Both functions return `ws1`; no return value is reserved to indicate an error.

The `wcslen()` and `wslen()` functions compute the number of wide-character codes in the wide-character string to which `ws` points, not including the terminating null wide-character code. Both functions return `ws`; no return value is reserved to indicate an error.

The `wcschr()` and `wschr()` functions locate the first occurrence of `wc` in the wide-character string pointed to by `ws`. The value of `wc` must be a character representable as a type `wchar_t` and must be a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code is considered to be part of the wide-character string. Upon completion, both functions return a pointer to the wide-character code, or a null pointer if the wide-character code is not found.

The `wcsrchr()` and `wsrchr()` functions locate the last occurrence of `wc` in the wide-character string pointed to by `ws`. The value of `wc` must be a character representable as a type `wchar_t` and must be a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code is considered to be part of the wide-character string. Upon successful completion, both functions return a pointer to the wide-character code, or a null pointer if `wc` does not occur in the wide-character string.

The `windex()` and `wrindex()` functions behave the same as `wschr()` and `wsrchr()`, respectively.
### wcspbrk(), wspbrk()

The `wcspbrk()` and `wspbrk()` functions locate the first occurrence in the wide-character string pointed to by `ws1` of any wide-character code from the wide-character string pointed to by `ws2`. Upon successful completion, the function returns a pointer to the wide-character code, or a null pointer if no wide-character code from `ws2` occurs in `ws1`.

### wcswcs()

The `wcswcs()` function locates the first occurrence in the wide-character string pointed to by `ws1` of the sequence of wide-character codes (excluding the terminating null wide-character code) in the wide-character string pointed to by `ws2`. Upon successful completion, the function returns a pointer to the located wide-character string, or a null pointer if the wide-character string is not found. If `ws2` points to a wide-character string with zero length, the function returns `ws1`.

### wcsspn(), wsspn()

The `wcsspn()` and `wsspn()` functions compute the length of the maximum initial segment of the wide-character string pointed to by `ws1` which consists entirely of wide-character codes from the wide-character string pointed to by `ws2`. Both functions return the length `ws1`; no return value is reserved to indicate an error.

### wcscspn(), wscspn()

The `wcscspn()` and `wscspn()` functions compute the length of the maximum initial segment of the wide-character string pointed to by `ws1` which consists entirely of wide-character codes not from the wide-character string pointed to by `ws2`. Both functions return the length of the initial substring of `ws1`; no return value is reserved to indicate an error.

### wcstok(), wstok()

A sequence of calls to the `wcstok()` and `wstok()` functions break the wide-character string pointed to by `ws1` into a sequence of tokens, each of which is delimited by a wide-character code from the wide-character string pointed to by `ws2`.

The third argument points to a caller-provided `wchar_t` pointer into which the `wcstok()` function stores information necessary for it to continue scanning the same wide-character string. This argument is not available with the XPG4 and SUS versions of `wcstok()`, nor is it available with the `wstok()` function. See `standards(5)`.

The first call in the sequence searches the wide-character string pointed to by `ws1` for the first wide-character code that is not contained in the current separator string pointed to by `ws2`. If no such wide-character code is found, then there are no tokens in the wide-character string pointed to by `ws1`, and `wcstok()` and `wstok()` return a null pointer. If such a wide-character code is found, it is the start of the first token.

The `wcstok()` and `wstok()` functions then search from that point for a wide-character code that is contained in the current separator string. If no such wide-character code is found, the current token extends to the end of the wide-character string pointed to by `ws1`, and subsequent searches for a token will...
return a null pointer. If such a wide-character code is found, it is overwritten by a null
wide character, which terminates the current token. The wcstok() and wstok() functions save a pointer to the following wide-character code, from which the next
search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts
searching from the saved pointer and behaves as described above.

Upon successful completion, both functions return a pointer to the first wide-character
code of a token. Otherwise, if there is no token, a null pointer is returned.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO malloc(3C), string(3C), wcswidth(3C), wcwidth(3C), attributes(5), standards(5)
# wcwidth(3C)

## NAME
wcswidth – number of column positions of a wide-character string

## SYNOPSIS
```c
#include <wchar.h>

int wcswidth(const wchar_t *pwcs, size_t n);
```

## DESCRIPTION
The `wcswidth()` function determines the number of column positions required for `n` wide-character codes (or fewer than `n` wide-character codes if a null wide-character code is encountered before `n` wide-character codes are exhausted) in the string pointed to by `pwcs`.

## RETURN VALUES
The `wcswidth()` function either returns 0 (if `pwcs` points to a null wide-character code), or returns the number of column positions to be occupied by the wide-character string pointed to by `pwcs`, or returns -1 (if any of the first `n` wide-character codes in the wide-character string pointed to by `pwcs` is not a printing wide-character code).

## ERRORS
No errors are defined.

## ATTRIBUTES
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

## SEE ALSO
setlocale(3C), wcwidth(3C), attributes(5)
The `wcsxfrm()` and `wsxfrm()` functions transform the wide character string pointed to by `ws2` and place the resulting wide character string into the array pointed to by `ws1`. The transformation is such that if either the `wcscmp(3C)` or `wscmp(3C)` functions are applied to two transformed wide strings, they return a value greater than, equal to, or less than 0, corresponding to the result of the `wcscoll(3C)` or `wscoll(3C)` function applied to the same two original wide character strings. No more than `n` wide-character codes are placed into the resulting array pointed to by `ws1`, including the terminating null wide-character code. If `n` is 0, `ws1` is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

The `wcsxfrm()` and `wsxfrm()` functions return the length of the transformed wide character string (not including the terminating null wide-character code). If the value returned is `n` or more, the contents of the array pointed to by `ws1` are indeterminate.

On error, `wcsxfrm()` and `wsxfrm()` return `(size_t)-1` and set `errno` to indicate the error.

The `wcsxfrm()` and `wsxfrm()` functions may fail if:

- **EINVAL** The wide character string pointed to by `ws2` contains wide-character codes outside the domain of the collating sequence.
- **ENOSYS** The function is not supported.

The transformation function is such that two transformed wide character strings can be ordered by the `wcscmp()` or `wscmp()` functions as appropriate to collating sequence information in the program’s locale (category `LC_COLLATE`).

The fact that when `n` is 0, `ws1` is permitted to be a null pointer, is useful to determine the size of the `ws1` array prior to making the transformation.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set `errno` to 0, call `wcsxfrm()` or `wsxfrm()`, then check `errno` and if it is non-zero, assume an error has occurred.

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

See `attributes(5)` for descriptions of the following attributes:
### wcsxfrm(3C)

<table>
<thead>
<tr>
<th>MT-Level</th>
<th>MT-Safe with exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

**SEE ALSO**

setlocale(3C), wcscmp(3C), wscoll(3C), wscmp(3C), wscoll(3C), attributes(5)
#include <stdio.h>
#include <wchar.h>

int wctob (wint_t c);

The wctob() function determines whether c corresponds to a member of the extended character set whose character representation is a single byte when in the initial shift state.

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

The wctob() function returns EOF if c does not correspond to a character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character.

No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
</tbody>
</table>

SEE ALSO btowc(3C), setlocale(3C), attributes(5), environ(5)

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

NAME
wctomb – convert a wide-character code to a character

SYNOPSIS
#include <stdlib.h>

int wctomb(char *s, wchar_t wchar);

DESCRIPTION
The wctomb() function determines the number of bytes needed to represent
the character corresponding to the wide-character code whose value is wchar. It stores the
character representation (possibly multiple bytes) in the array object pointed to by s (if
s is not a null pointer). At most MB_CUR_MAX bytes are stored.

A call with s as a null pointer causes this function to return 0. The behavior of this
function is affected by the LC_CTYPE category of the current locale.

RETURN VALUES
If s is a null pointer, wctomb() returns 0 value. If s is not a null pointer, wctomb()
returns -1 if the value of wchar does not correspond to a valid character, or returns the
number of bytes that constitute the character corresponding to the value of wchar.

In no case will the value returned be greater than the value of the MB_CUR_MAX macro.

ERRORS
No errors are defined.

USAGE
The wctomb() function can be used safely in a multithreaded application, as long as
setlocale(3C) is not being called to change the locale.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO
mblen(3C), mbstowcs(3C), mbtowc(3C), setlocale(3C), wcstombs(3C),
attributes(5)
**NAME**

wctrans – define character mapping

**SYNOPSIS**

```c
#include <wctype.h>

wctrans_t wctrans(const char *charclass);
```

**DESCRIPTION**

The `wctrans()` function is defined for valid character mapping names identified in the current locale. The `charclass` is a string identifying a generic character mapping name for which codeset-specific information is required. The following character mapping names are defined in all locales – "tolower" and "toupper".

The function returns a value of type `wctrans_t`, which can be used as the second argument to subsequent calls of `towctrans(3C)`. The `wctrans()` function determines values of `wctrans_t` according to the rules of the coded character set defined by character mapping information in the program’s locale (category `LC_CTYPE`). The values returned by `wctrans()` are valid until a call to `setlocale(3C)` that modifies the category `LC_CTYPE`.

**RETURN VALUES**

The `wctrans()` function returns 0 if the given character mapping name is not valid for the current locale (category `LC_CTYPE`), otherwise it returns a non-zero object of type `wctrans_t` that can be used in calls to `towctrans(3C)`.

**ERRORS**

The `wctrans()` function may fail if:

- `EINVAL` The character mapping name pointed to by `charclass` is not valid in the current locale.

**ATTRIBUTES**

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

**SEE ALSO**

`setlocale(3C)`, `towctrans(3C)`, `attributes(5)`
NAME  wctype – define character class

SYNOPSIS

```c
#include <wchar.h>

wctype_t wctype(const char *charclass);
```

DESCRIPTION

The `wctype()` function is defined for valid character class names as defined in the current locale. The `charclass` is a string identifying a generic character class for which codeset-specific type information is required. The following character class names are defined in all locales:

<table>
<thead>
<tr>
<th>charclass</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>alnum</td>
<td>alpha</td>
</tr>
<tr>
<td>cntrl</td>
<td>digit</td>
</tr>
<tr>
<td>lower</td>
<td>print</td>
</tr>
<tr>
<td>space</td>
<td>upper</td>
</tr>
</tbody>
</table>

Additional character class names defined in the locale definition file (category `LC_CTYPE`) can also be specified.

The function returns a value of type `wctype_t`, which can be used as the second argument to subsequent calls of `iswctype(3C)`. `wctype()` determines values of `wctype_t` according to the rules of the coded character set defined by character type information in the program’s locale (category `LC_CTYPE`). The values returned by `wctype()` are valid until a call to `setlocale(3C)` that modifies the category `LC_CTYPE`.

RETURN VALUES

The `wctype()` function returns 0 if the given character class name is not valid for the current locale (category `LC_CTYPE`); otherwise it returns an object of type `wctype_t` that can be used in calls to `iswctype()`.

ATTRIBUTES

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO

`iswctype(3C)`, `setlocale(3C)`, `attributes(5)`
wcwidth(3C)

NAME wcwidth – number of column positions of a wide-character code

SYNOPSIS #include <wchar.h>

    int wcwidth(wchar_t wc);

DESCRIPTION The wcwidth() function determines the number of column positions required for the wide character wc. The value of wc must be a character representable as a wchar_t, and must be a wide-character code corresponding to a valid character in the current locale.

RETURN VALUES The wcwidth() function either returns 0 (if wc is a null wide-character code), or returns the number of column positions to be occupied by the wide-character code wc, or returns −1 (if wc does not correspond to a printing wide-character code).

ERRORS No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe with exceptions</td>
</tr>
<tr>
<td>CSI</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

SEE ALSO setlocale(3C), wcswidth(3C), attributes(5)
NAME
wmemchr – find a wide-character in memory

SYNOPSIS

ISO C++
#include <wchar.h>

const wchar_t *wmemchr(const wchar_t *ws, wchar_t wc, size_t n);
#include <cwchar>
wchar_t *std::wmemchr(wchar_t *ws, wchar_t wc, size_t n);

DESCRIPTION
The wmemchr() function locates the first occurrence of \textit{wc} in the initial \textit{n} wide-characters of the object pointed to by \textit{ws}. This function is not affected by locale and all wchar_t values are treated identically. The null wide-character and wchar_t values not corresponding to valid characters are not treated specially.

If \textit{n} is 0, \textit{ws} must be a valid pointer and the function behaves as if no valid occurrence of \textit{wc} is found.

RETURN VALUES
The wmemchr() function returns a pointer to the located wide-character, or a null pointer if the wide-character does not occur in the object.

ERRORS
No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
wmemcmp(3C), wmemcpy(3C), wmemmove(3C), wmemset(3C), attributes(5)
wmemcmp(3C)

NAME    wmemcmp – compare wide-characters in memory

SYNOPSIS
```c
#include <wchar.h>

int wmemcmp(const wchar_t *ws1, const wchar_t *ws2, size_t n);
```

DESCRIPTION
The wmemcmp() function compares the first n wide-characters of the object pointed to by ws1 to the first n wide-characters of the object pointed to by ws2. This function is not affected by locale and all wchar_t values are treated identically. The null wide-character and wchar_t values not corresponding to valid characters are not treated specially.

If n is zero, ws1 and ws2 must be a valid pointers and the function behaves as if the two objects compare equal.

RETURN VALUES
The wmemcmp() function returns an integer greater than, equal to, or less than 0, accordingly as the object pointed to by ws1 is greater than, equal to, or less than the object pointed to by ws2.

ERRORS
No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO wmemchr(3C), wmemcpy(3C), wmemmove(3C), wmemset(3C), attributes(5)
NAME
wmemcpy – copy wide-characters in memory

SYNOPSIS
#include <wchar.h>

wchar_t *wmemcpy(wchar_t *ws1, const wchar_t *ws2, size_t n);

DESCRIPTION
The wmemcpy() function copies n wide-characters from the object pointed to by ws2 to the object pointed to be ws1. This function is not affected by locale and all wchar_t values are treated identically. The null wide-character and wchar_t values not corresponding to valid characters are not treated specially.

If n is zero, ws1 and ws2 must be a valid pointers, and the function copies zero wide-characters.

RETURN VALUES
The wmemcpy() function returns the value of ws1.

ERRORS
No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
wmemchr(3C), wmemcmp(3C), wmemmove(3C), wmemset(3C), attributes(5)
wmemmove(3C)

NAME  wmemmove – copy wide-characters in memory with overlapping areas
SYNOPSIS  
```c
#include <wchar.h>
wchar_t *wmemmove(wchar_t *ws1, const wchar_t *ws2, size_t n);
```
DESCRIPTION  The `wmemmove()` function copies `n` wide-characters from the object pointed to by `ws2` to the object pointed to by `ws1`. Copying takes place as if the `n` wide-characters from the object pointed to by `ws2` are first copied into a temporary array of `n` wide-characters that does not overlap the objects pointed to by `ws1` or `ws2`, and then the `n` wide-characters from the temporary array are copied into the object pointed to by `ws1`.

This function is not affected by locale and all `wchar_t` values are treated identically. The null wide-character and `wchar_t` values not corresponding to valid characters are not treated specially.

If `n` is 0, `ws1` and `ws2` must be a valid pointers, and the function copies zero wide-characters.

RETURN VALUES  The `wmemmove()` function returns the value of `ws1`.

ERRORS  No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  `wmemchr(3C), wmemcmp(3C), wmemcpy(3C), wmemset(3C), attributes(5)"
NAME  wmemset – set wide-characters in memory

SYNOPSIS  #include <wchar.h>

        wchar_t *wmemset(wchar_t *ws, wchar_t wc, size_t n);

DESCRIPTION  The wmemset() function copies the value of wc into each of the first n wide-characters of the object pointed to by ws. This function is not affected by locale and all wchar_t values are treated identically. The null wide-character and wchar_t values not corresponding to valid characters are not treated specially.

        If n is 0, ws must be a valid pointer and the function copies zero wide-characters.

RETURN VALUES  The wmemset() functions returns the value of ws.

ERRORS  No errors are defined.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO  wmemchr(3C), wmemcmp(3C), wmemcpy(3C), wmemmove(3C), attributes(5)
wordexp(3C)

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, subject to quoting, and places
the list of expanded words into the structure pointed to by pwordexp.

The wordfree() function frees any memory allocated by wordexp() associated
with pwordexp.

words Argument
The words argument is a pointer to a string containing one or more words to be
expanded. The expansions will be the same as would be performed by the shell if
words were the part of a command line representing the arguments to a utility.
Therefore, words must not contain an unquoted NEWLINE or any of the unquoted
shell special characters:
| & ; < >

except in the context of command substitution. It also must not contain unquoted
parentheses or braces, except in the context of command or variable substitution. If the
argument words contains an unquoted comment character (number sign) that is the
beginning of a token, wordexp() may treat the comment character as a regular
character, or may interpret it as a comment indicator and ignore the remainder of
words.

pwordexp Argument
The structure type wordexp_t is defined in the header <wordexp.h> and includes at
least the following members:

size_t we_wordc
Count of words matched by words.

char **we_wordv
Pointer to list of expanded words.

size_t we_offs
Slots to reserve at the beginning of
pwordexp->we_wordv.

The wordexp() function stores the number of generated words into
pwordexp->we_wordc and a pointer to a list of pointers to words in
pwordexp->we_wordv. Each individual field created during field splitting is a
separate word in the pwordexp->we_wordv list. The words are in order. The first
pointer after the last word pointer will be a null pointer.

It is the caller’s responsibility to allocate the storage pointed to by pwordexp. The
wordexp() function allocates other space as needed, including memory pointed to by
pwordexp->we_wordv. The wordfree() function frees any memory associated with
pwordexp from a previous call to wordexp().
The \texttt{flags} argument is used to control the behavior of \texttt{wordexp()}. The value of \texttt{flags} is the bitwise inclusive OR of zero or more of the following constants, which are defined in \texttt{<wordexp.h>}:

- \texttt{WRDE_APPEND} Append words generated to the ones from a previous call to \texttt{wordexp()}. 
- \texttt{WRDE_DOFFS} Make use of \texttt{pwordexp->we_offs}. If this flag is set, \texttt{pwordexp->we_offs} is used to specify how many NULL pointers to add to the beginning of \texttt{pwordexp->we_wordv}. In other words, \texttt{pwordexp->we_wordv} will point to \texttt{pwordexp->we_offs} NULL pointers, followed by \texttt{pwordexp->we_wordc} word pointers, followed by a NULL pointer.
- \texttt{WRDE_NOCMD} Fail if command substitution is requested.
- \texttt{WRDE_REUSE} The \texttt{pwordexp} argument was passed to a previous successful call to \texttt{wordexp()}, and has not been passed to \texttt{wordfree()}. The result will be the same as if the application had called \texttt{wordfree()} and then called \texttt{wordexp()} without \texttt{WRDE_REUSE}.
- \texttt{WRDE_SHOWERR} Do not redirect \texttt{stderr} to \texttt{/dev/null}.
- \texttt{WRDE_UNDEF} Report error on an attempt to expand an undefined shell variable.

The \texttt{WRDE_APPEND} flag can be used to append a new set of words to those generated by a previous call to \texttt{wordexp()}. The following rules apply when two or more calls to \texttt{wordexp()} are made with the same value of \texttt{pwordexp} and without intervening calls to \texttt{wordfree()}:

1. The first such call must not set \texttt{WRDE_APPEND}. All subsequent calls must set it.
2. All of the calls must set \texttt{WRDE_DOFFS}, or all must not set it.
3. After the second and each subsequent call, \texttt{pwordexp->we_wordv} will point to a list containing the following:
   a. zero or more NULL pointers, as specified by \texttt{WRDE_DOFFS} and \texttt{pwordexp->we_offs}.
   b. pointers to the words that were in the \texttt{pwordexp->we_wordv} list before the call, in the same order as before.
   c. pointers to the new words generated by the latest call, in the specified order.
4. The count returned in \texttt{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 \texttt{wordexp()}, but if it does it must reset them to the original value before a subsequent call, using the same \texttt{pwordexp} value, to \texttt{wordfree()} or \texttt{wordexp()} with the \texttt{WRDE_APPEND} or \texttt{WRDE_REUSE} flag.

If \texttt{words} contains an unquoted:
NEWLINE | & ; < > ( ) { } in an inappropriate context, wordexp() will fail, and the number of expanded words will be zero.

Unless WRDE_SHOWERR is set in flags, wordexp() will redirect stderr to /dev/null for any utilities executed as a result of command substitution while expanding words.

If WRDE_SHOWERR is set, wordexp() may write messages to stderr if syntax errors are detected while expanding words. If WRDE_DOFFS is set, then pwordexp->we_offs must have the same value for each wordexp() call and wordfree() call using a given pwordexp.

The following constants are defined as error return values:

- **WRDE_BADCHAR** One of the unquoted characters:
  
  NEWLINE | & ; < > ( ) { } appears in words in an inappropriate context.

- **WRDE_BADVAL** Reference to undefined shell variable when WRDE_UNDEF is set in flags.

- **WRDE_CMDSUB** Command substitution requested when WRDE_NOCMD was set in flags.

- **WRDE_NOSPACE** Attempt to allocate memory failed.

- **WRDE_SYNTAX** Shell syntax error, such as unbalanced parentheses or unterminated string.

**RETURN VALUES**

On successful completion, wordexp() returns 0. Otherwise, a non-zero value as described in <wordexp.h> is returned to indicate an error. If wordexp() returns the value WRDE_NOSPACE, then pwordexp->we_wordc and pwordexp->we_wordv will be updated to reflect any words that were successfully expanded. In other cases, they will not be modified.

The wordfree() function returns no value.

**ERRORS**

No errors are defined.

**USAGE**

This function is intended to be used by an application that wants to do all of the shell’s expansions on a word or words obtained from a user. For example, if the application prompts for a filename (or list of filenames) and then uses wordexp() to process the input, the user could respond with anything that would be valid as input to the shell.

The WRDE_NOCMD flag is provided for applications that, for security or other reasons, want to prevent a user from executing shell command. Disallowing unquoted shell special characters also prevents unwanted side effects such as executing a command or writing a file.
See attributes(5) for descriptions of the following attributes:

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO fnmatch(3C), glob(3C), attributes(5)
wsprintf() function outputs a Process Code string ending with a Process Code (wchar_t) null character. It is the user’s responsibility to allocate enough space for this wchar_t string.

This returns the number of Process Code characters (excluding the null terminator) that have been written. The conversion specifications and behavior of wsprintf() are the same as the regular sprintf(3C) function except that the result is a Process Code string for wsprintf(), and on Extended Unix Code (EUC) character string for sprintf().

Upon successful completion, wsprintf() returns the number of characters printed. Otherwise, a negative value is returned.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

See also wsscanf(3C), printf(3C), scanf(3C), sprintf(3C), attributes(5)
wsscanf — formatted input conversion

#include <stdio.h>
#include <widec.h>

int wsscanf(wchar_t *s, const char *format, /* pointer */ ...);

The wsscanf() function reads Process Code characters from the Process Code string s, interprets them according to the format, and stores the results in its arguments. It expects, as arguments, a control string format, and a set of pointer arguments indicating where the converted input should be stored. The results are undefined if there are insufficient args for the format. If the format is exhausted while args remain, the excess args are simply ignored.

The conversion specifications and behavior of wsscanf() are the same as the regular sscanf(3C) function except that the source is a Process Code string for wsscanf() and on Extended Unix Code (EUC) character string for sscanf(3C).

Upon successful completion, wsscanf() returns the number of characters matched. Otherwise, it returns a negative value.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

See also: wsscanf(3C), printf(3C), scanf(3C), attributes(5)
NAME
wstring, wscasecmp, wsncasecmp, wsdup, wscol – Process Code string operations

SYNOPSIS
#include <widec.h>

int wscasecmp(const wchar_t *s1, const wchar_t *s2);
int wsncasecmp(const wchar_t *s1, const wchar_t *s2, int n);
wchar_t *wsdup(const wchar_t *s);
int wscol(const wchar_t *s);

DESCRIPTION
These functions operate on Process Code strings terminated by wchar_t null characters. During appending or copying, these routines do not check for an overflow condition of the receiving string. In the following, s, s1, and s2 point to Process Code strings terminated by a wchar_t null.

wscasecmp(), wsncasecmp()
The wscasecmp() function compares its arguments, ignoring case, and returns an integer greater than, equal to, or less than 0, depending upon whether s1 is lexicographically greater than, equal to, or less than s2. It makes the same comparison but compares at most n Process Code characters. The four Extended Unix Code (EUC) codesets are ordered from lowest to highest as 0, 2, 3, 1 when characters from different codesets are compared.

wsdup() The wsdup() function returns a pointer to a new Process Code string, which is a duplicate of the string pointed to by s. The space for the new string is obtained using malloc(3C). If the new string cannot be created, a null pointer is returned.

wscol() The wscol() function returns the screen display width (in columns) of the Process Code string s.

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

<table>
<thead>
<tr>
<th>ATTRIBUTE TYPE</th>
<th>ATTRIBUTE VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT-Level</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

SEE ALSO
malloc(3C), string(3C), wchar(3C), attributes(5)
Index

b
binary output — fwrite, 181

Numbers and Symbols
__fbufsize — interfaces to stdio FILE structure, 109
__flbf — interfaces to stdio FILE structure, 109
__fpending — interfaces to stdio FILE structure, 109
__fpurge — interfaces to stdio FILE structure, 109
__freadable — interfaces to stdio FILE structure, 109
__freading — interfaces to stdio FILE structure, 109
__fsetlocking — interfaces to stdio FILE structure, 109
__fwritable — interfaces to stdio FILE structure, 109
__fwriting — interfaces to stdio FILE structure, 109

A
abort — terminate the process abnormally, 21
abs — return absolute value of integer, 22
accounting, time accounting for current process — times, 558
acquire and release stream lock — flockfile, 131
funlockfile, 131
additional severities, define — addsev, 23
address of symbol, get address in shared object or executable — dladdr, 94
addsev — define additional severities, 23
addseverity — build a list of severity levels for an application for use with formattedmsg, 24
alarm, schedule signal after interval in microseconds — ualarm, 579
allocation cache manipulation —
  umem_cache_alloc, 585
  umem_cache_create, 585
  umem_cache_destroy, 585
  umem_cache_free, 585
alphasort — scan a directory, 435
applications
  build a list of severity levels for use with formattedmsg — addseverity, 24
display a message on stderr or system console — formattedmsg, 133
get entries from symbol table — nlist, 363
arithmetic, compute the quotient and remainder — div, 75
arithmetic, 48-bit integer, generate uniformly distributed pseudo-random numbers —
drand48, 96
asctime — convert date and time to string, 501
assert — verify program assertion, 26
associate a stream with a file descriptor —
  fdopen, 115
### Basic Library Functions

- **atexit** — register a function to run at process termination or object unloading, 27
- **atof** — convert string to double-precision number, 519
- **atoi** — string conversion routines, 522
- **atol** — string conversion routines, 522
- **atoll** — string conversion routines, 522
- **attropen** — open a file, 28
- **basename** — return the last element of path name, 29
- **bcmp** — operates on variable length strings of bytes, 35
- **bcopy** — operates on variable length strings of bytes, 35
- **binary search of sorted table** — **bsearch**, 33
- **binary search trees, manage** — **tdelete**, 573
  - **tfind**, 573
  - **tsearch**, 573
  - **twalk**, 573
- **bind_textdomain_codeset** — message handling functions, 241
- **bindtextdomain** — message handling functions, 241
- **bit and byte operations, find first set bit** — **ffs**, 120
- **bsd_signal** — simplified signal facilities, 32
- **bsdmalloc** — memory allocator, 30
- **bsearch** — binary search a sorted table, 33
- **bsprintf** — single-byte to wide-character conversion, 36
- **buffering, assign to stream** — **setbuffer**, 450
  - **setlinebuf**, 450
- **byte swap — swab**, 530
- **bzero** — operates on variable length strings of bytes, 35

### C

**C Compilation**

- close a shared object — **dlclose**, 78
- create new file from dynamic object component — **dldump**, 79
- get address of symbol in shared object or executable — **dlsym**, 94
- get diagnostic information — **dlerror**, 85
- open a shared object — **dlopen**, 90
- **catclose** — close a message catalog, 38
- **catgets** — read a program message, 37
- **catopen** — open a message catalog, 38
- **cfgetispeed** — get input baud rate, 41
- **cfgetospeed** — get output baud rate, 41
- **cfsetispeed** — set input baud rate, 42
- **cfsetospeed** — set output baud rate, 42
- **ctime** — convert date and time to string, 501

**character handling**

- **ctype**, 62
  - **isalnum**, 62
  - **isalpha**, 62
  - **iscntrl**, 62
  - **isdigit**, 62
  - **isgraph**, 62
  - **islower**, 62
  - **isprint**, 62
  - **ispunct**, 62
  - **isspace**, 62
  - **isupper**, 62
  - **isxdigit**, 62

- **clock** — report CPU time used, 43
- close or iterate over open file descriptors — **closefrom**, 45
- close or iterate over open file descriptors — **fdwalk**, 45
- close a directory stream — **closedir**, 44
- close a shared object — **dlclose**, 78
- close a stream — **fclose**, 111
- **closedir** — close a directory stream, 44
- closefrom — close or iterate over open file descriptors, 45
- **closelog** — control system log, 540
- code conversion allocation function — **iconv_open**, 279
- code conversion deallocation function — **iconv_close**, 278
code conversion for Process Code and File Code
- strtows, 527
- wstosstr, 527
code conversion function — iconv, 273
column positions of a wide-character code —
wcwidth, 646
column positions of a wide-character string —
wcswidth, 639
command options, get option letter from
argument vector — getopt, 216
command suboptions, parse suboptions from a
string — getsubopt, 238
compare wide-characters in memory —
wmemcmp, 648
compile and execute regular expressions
- re_comp, 423
- re_exec, 423
confstr — get configurable variables, 47
control system log
- closelog, 540
- openlog, 540
- setlogmask, 540
- syslog, 540
convert date and time to string — strftime, 501
convert formatted input — fscanf, 436
convert formatted input — scanf, 436
convert formatted input — vscanf, 436
convert formatted input — vsscanf, 436
convert formatted input — vsscanf, 436
convert formatted wide-character input —
fwscanf, 182
convert formatted wide-character input —
wscanf, 182
convert formatted wide-character input —
sscanf, 182
convert formatted wide-character input —
vscanf, 182
convert formatted wide-character input —
vwscanf, 182
convert formatted wide-character input —
vsscanf, 182
convert formatted wide-character input —
vwcscanf, 182
convert string to unsigned long — strtoul, 525
convert wide character string to
double-precision number — watof,
watof, 627
convert wide character string to
double-precision number — wstod,
wstod, 627
convert a character string to a wide-character
string — mbstowcs, 333
convert a character string to a wide-character
string (restartable) — mbsrtowcs, 331
convert a character to a wide-character code —
mbtowc, 334
convert a character to a wide-character code
(restartable) — mbtowc, 328
convert a wide-character code to a character —
wctomb, 643
convert a wide-character code to a character
(restartable) — wcrtomb, 620
convert a wide-character string to a character
string — wcstombs, 631
convert a wide-character string to a character
string (restartable) — wcsrtombs, 624
convert date and time to wide character string
- wcsftime, 623
convert floating-point number to string
- ecvt, 101
- fcvt, 101
- gcvt, 101
convert monetary value to string
- strfmon, 497
convert numbers to strings
- econvert, 99
- fconvert, 99
- fprintf, 385
- gconvert, 99
- printf, 385
- qconvert, 99
- qfconvert, 99
- qgconvert, 99
- sprintf, 385
- vfprintf, 385
- vprintf, 385
convert string to double-precision number
- atof, 519
- strtod, 519
convert string to double-precision number —
wcstod, 627
convert date and time to string — strftime
asctime, 501
cftime, 501
convert to wchar_t strings, wsprintf, 656
convert wide character string to unsigned long
— wcstoul, 632
copy wide-characters in memory — wmemcpy, 649
copy wide-characters in memory with overlapping areas — wmemmove, 650
CPU time, report for calling process — clock, 43
CPU-use, prepare execution profile — monitor, 351
create a temporary file — tmpfile, 559
create new file from dynamic object component — dldump, 79
crypt — string encoding function, 51
cset — get information on EUC codesets, 55
csetcol — get information on EUC codesets, 55
csetlen — get information on EUC codesets, 55
csetno — get information on EUC codesets, 55
ctermid — generate path name for controlling terminal, 56
ctermid_r — generate path name for controlling terminal, 56
ctype — character handling, 62
current location of a named directory stream — telldir, 556
current working directory, get pathname — getcwd, 190
cuserid — get character-string representation of login name of user, 65

data base subroutines — dbm, 66
dbmclose, 66
dbminit, 66
delete, 66
fetch, 66
firstkey, 66
nextkey, 66
store, 66
database functions — dbm_clearerr, 358
database functions — dbm_close, 358
database functions — dbm_delete, 358
database functions — dbm_error, 358
database functions — dbm_fetch, 358
database functions — dbm_firstkey, 358
database functions — dbm_nextkey, 358
database functions — dbm_open, 358
database functions — dbm_store, 358
database functions — ndbm, 358
date and time
convert to string — asctime, 57
convert to string — asctime_r, 57
convert to string — ctime, 57
convert to string — ctime_r, 57
convert to string — gmtime, 57
convert to string — gmtime_r, 57
convert to string — localtime, 57
convert to string — localtime_r, 57
convert to string — tzset, 57
convert user format date and time — getdate, 192
— gettimeofday, 245
date and time conversion — strftime, 513
dbm — data base subroutines, 66
dbm_clearerr — database functions, 358
dbm_close — database functions, 358
dbm_delete — database functions, 358
dbm_error — database functions, 358
dbm_fetch — database functions, 358
dbm_firstkey — database functions, 358
dbm_nextkey — database functions, 358
dbm_open — database functions, 358
dbm_store — database functions, 358
dbmclose — data base subroutines, 66
dbminit — data base subroutines, 66
dcgettext — message handling functions, 241
dcngettext — message handling functions, 241
debugging features of the umem library — umem_debug, 593
debugging memory allocator
— calloc, 617
— cfree, 617
— free, 617
— mallinfo, 617
— malloc, 617
— mallopt, 617
— memalign, 617
— realloc, 617
— valloc, 617
decimal record from double-precision floating
— double_to_decimal, 127
decimal record from extended-precision floating
— extended_to_decimal, 127
decimal record from quadruple-precision floating — quadruple_to_decimal, 127
decimal record from single-precision floating — single_to_decimal, 127
decimal record to double-precision floating — decimal_to_double, 68
decimal record to extended-precision floating — decimal_to_extended, 68
decimal record to quadruple-precision floating — decimal_to_quadruple, 68
decimal record to single-precision floating — decimal_to_single, 68
decimal_to_double — decimal record to double-precision floating, 68
decimal_to_extended — decimal record to extended-precision floating, 68
decimal_to_quadruple — decimal record to quadruple-precision floating, 68
decimal_to_single — decimal record to single-precision floating, 68
decompose floating-point number
   — modf, 350
   — modff, 350
define character class — wctype, 645
define character mapping — wctrans, 644
define default catalog — setcat, 451
define the label for pfmt() and lfmt() — setlabel, 459
delete — data base subroutines, 66
detach a name from a STREAMS-based file descriptor — fdetach, 113
determine conversion object status — mbsinit, 330
determine if address is within stack boundaries — stack_inbounds, 486
determine stack boundary violation event — stackViolation, 488
device number, manage — makedev, major, minor, 316
dgettext — message handling functions, 241
dftime — computes the difference between two calendar times, 70
directio — provide advice to file system, 71
directories
   get current working directory pathname — getwd, 262
   get pathname of current working directory — getcwd, 190
directory operations
   — alphasort, 435
   — scandir, 435
dirname — report parent directory name of file path name, 73
display error message in standard format — pfmt, 369
display error message in standard format and pass to logging and monitoring services — lfmt, 296
display error message in standard format and pass to logging and monitoring services — vlfmt, 601
display error message in standard format and pass to logging and monitoring services — vpfmt, 603
div — compute quotient and remainder, 75
division and remainder operations
   — div, 75
   — ldiv, 75
dldadr — translate address to symbolic information, 76
dldadr1 — translate address to symbolic information, 76
dlclose — close a shared object, 78
dldump — create new file from dynamic object component of calling process, 79
dlerror — get diagnostic information, 85
dlinfo — dynamic load information, 86
dlopen — open a shared object, 90
dlsym — get address of symbol in shared object or executable, 94
dngettext — message handling functions, 241
double_to_decimal — decimal record from double-precision floating, 127
dup2 — duplicate an open file descriptor, 98
duplicate an open file descriptor — dup2, 98
dynamic linking
   close a shared object — dlclose, 78
   create new file from dynamic object component — dldump, 79
   get address of symbol in shared object or executable — dlsym, 94
   get diagnostic information — dlerror, 85
   open a shared object — dlopen, 90
dynamic load information — dlinfo, 86
E

econvert — convert number to ASCII, 99
data — last location in program, 104
dend — last location in program, 104
dwendgrent — group database entry
  functions, 201
dwendpwent — get password entry from user
databaset, 226
dwendspent — get shadow password database
  entry, 234
dwendusershell() — function, 250
dwendutent — user accounting database
  functions, 251
dwendutxent — user accounting database
  functions, 254
evironment name, return value — getenviron, 199
evironment variables, change or add value —
  putenv, 395
error messages, get string — strerror, 496
error messages, system, print — perror, 368
etext — last location in program, 104
EUC character bytes, — euclen, 105
EUC characters, convert a string of Process
  Code characters to EUC characters and put it
  on a stream — putws, 399
EUC codeset, get information, — geteuccol, 263
EUC codesets, get information
  — cset, 55
  — csetcol, 55
  — csetlen, 55
  — csetno, 55
  — wcsetno, 55
EUC display width
  — euccol, 105
  — eucscol, 105
EUC display width
  — euccol, 105
  — eucscol, 105
euccol — get EUC character display width, 105
  euclen — get EUC byte length, 105
  eucscol — get EUC string display width, 105
  exit — terminate process, 106
express an intention to extend the stack —
  _stack_grow, 485
extended_to_decimal — decimal record from
  extended-precision floating, 127
Extended Unix Code, See EUC
extract mantissa and exponent from double
  precision number — frexp, 157

F

fast, scalable memory allocation —
  umem_alloc, 580
fast, scalable memory allocation —
  umem_free, 580
fast, scalable memory allocation —
  umem_nofail_callback, 580
fast, scalable memory allocation —
  umem_zalloc, 580
fattach — attach a STREAMS-based file
  descriptor to an object in the file system
  name space, 107
fclose — close a stream, 111
fconvert — convert number to ASCII, 99
FD_CLR — synchronous I/O multiplexing, 444
FD_ISSET — synchronous I/O
  multiplexing, 444
FD_SET — synchronous I/O multiplexing, 444
FD_ZERO — synchronous I/O
  multiplexing, 444
fdetach — detach a name from a
  STREAMS-based file descriptor, 113
fdopen — associate a stream with a file
  descriptor, 115
fdopendir — open directory, 366
fdwalk — close or iterate over open file
  descriptors, 45
fetch — data base subroutines, 66
fflush — flush a stream, 118
fss — find first set bit, 120
fgetc — get a byte from a stream, 121
fgetgrent — group database entry
  functions, 201
fgetgrent_r — group database entry
  functions, 201
fgetpos — get current file position
  information, 124
fgetpwent — get password entry from a
  file, 226
fgetpwent_r — get password entry from a
  file, 226
fgetspent — get shadow password database
  entry, 234
fgetspent_r — get shadow password database
  entry(reentrant), 234
fgetwc — get a wide-character code from a
  stream, 125
fgetws — get a wide-character string from a stream, 264
file descriptor
duplicate an open one — dup2, 98
STREAMS-based, attach to an object in file system name space — fattach, 107
test for a STREAMS file — isastream, 285
file descriptors, apply or remove advisory lock on open file — flock, 129
file name, make a unique one — mktemp, 342
make a unique file name — mkstemp, 341
file pointer in a stream, reposition — fsetpos, fgetpos, 161
file_to_decimal — decimal record from character stream, 510
file tree
recursively descend — ftw, 168
recursively descend — nftw, 168
files
allows sections of file to be locked — lockf, 304
optimizing usage of files — directio, 71
— remove, 432
report parent directory of file path name — dirname, 73
set a file to a specified length — truncate, 570
find a wide-character in memory — wmemchr, 647
find a wide-character substring — wcsstr, 626
find pathname of a terminal
— ttyname, 576
— ttyname_r, 576
firstkey — data base subroutines, 66
floating-point number, convert to string — ecvt, 101
floating-point number, determine type
— finite, 287
— fpclass, 287
— isnan, 287
— isnand, 287
— isnanf, 287
— unordered, 287
flock — apply or remove an advisory lock on an open file, 129
flockfile — acquire and release stream lock, 131
flush a stream — fflush, 118
flush non-transmitted output data, non-read input data or both — tcflush, 547
flushlbf — interfaces to stdio FILE structure, 109
fmtmsg — display a message on stderr or system console, 133
fnmatch — match filename or path name, 138
fopen — open a stream, 140
fopen — open stream, 143
formatted input conversion — wscanf, 657
formatted output conversion
— fprintf, 385
— printf, 385
— sprintf, 385
— vfprintf, 385
— vprintf, 385
— vsprintf, 385
fpgetmask — IEEE floating-point environment control, 145
fpgetround — IEEE floating-point environment control, 145
fpgetsticky — IEEE floating-point environment control, 145
fprintf — formatted output conversion, 385
fprintf — print formatted output, 376
fpsetmask — IEEE floating-point environment control, 145
fpsetround — IEEE floating-point environment control, 145
fpsetsticky — IEEE floating-point environment control, 145
fputc — put a byte on a stream, 147
fputwc — put wide-character code on a stream, 150
fputws — put wide character string on a stream, 152
fread — binary input, 153
free — memory allocator, 30
freopen — open a stream, 155
freopen — open stream, 143
frexp — extract mantissa and exponent from double precision number, 157
fscanf — convert formatted input, 436
fseek — reposition a file-position indicator in a stream, 158
fseeko — reposition a file-position indicator in a stream, 158
fsetpos — reposition a file pointer in a stream, 161
fsync — synchronize changes to a file, 162
ftell — return a file offset in a stream, 164
ftello — return a file offset in a stream, 164
ftime — get date and time, 165
ftruncate — set a file to a specified length, 570
ftw — walk a file tree, 168
func_to_decimal — decimal record from character function, 510
funlockfile — acquire and release stream lock, 131
fwide — set stream orientation, 173
fwprintf — print formatted wide-character output, 174
fwrite — binary output, 181
fwscanf — convert formatted wide-character input, 182

get a wide-character code from a stream —
  fgetwc, 125
get a wide-character code from a stream (restartable) —
  getwc, 260
get address of symbol in shared object or executable —
  dladdr, 94
generate path name for controlling terminal —
  ctermid, 56
get an identifier for the current host —
  gethostid, 205
generate path names matching a pattern —
  glob, 265
get current file position information —
  fgetpos, 124
get diagnostic information —
  dlerror, 85
generate path names matching a pattern —
  globfree, 265
generate path name for controlling terminal —
  ctermid_r, 56
get current file position information —
  getmntent, 211
get foreground process group ID —
  tcgetpgrp, 549
get input baud rate, —
  cfgetispeed, 41
get name of signal —
  strsignal, 518
get number of bytes in a character —
  mblen, 325
get number of bytes in a character (restartable) —
  mbrlen, 326
generate path names matching a pattern —
  glob, 265
get diagnostic information —
  derror, 85
get system load averages —
  getloadavg, 208
get system load averages for a processor set —
  pset_getloadavg, 389
get the parameters associated with the terminal —
  tcgetatt, 548
get wide character from a stream —
  getwc, 260
get wide character from stdin stream —
  getwchar, 261
getenv — return value for environment name, 199
get input baud rate, —
  cfgetispeed, 41
generate path name for controlling terminal —
  ctermid, 56
get process group ID for session leader for controlling terminal —
  tcgetsid, 550
get system load averages —
  getloadavg, 208
get system load averages for a processor set —
  pset_getloadavg, 389
get the parameters associated with the terminal —
  tcgetatt, 548
get wide character from a stream —
  getwc, 260
get wide character from stdin stream —
  getwchar, 261
getenv — return value for environment name, 199

G
gconvert — convert number to ASCII, 99
getenv — return value for environment name, 199
getexecname — return pathname of executable, 200
getextmntent — get mnttab file information, 211
getgrent — group database entry functions, 201
getgrent_r — group database entry functions, 201
getggid — group database entry functions, 201
getggid_r — group database entry functions, 201
getgrnam — group database entry functions, 201
getgrnam_r — group database entry functions, 201
gethomelgroup — obtain information on scheduling decisions, 189
gethostid — get an identifier for the current host, 205
gethostname — get name of current host, 206
gethrtime — get high resolution real time, 207
gethrvtime — get high resolution virtual time, 207
getloadavg — get system load averages, 208
getlogin — get login name, 209
getlogin_r — get login name, 209
getmntany — get mnttab file information, 211
getmntent — get mnttab file information, 211
getopt — get option letter from argument vector, 216
getpagesize — get system page size, 220
getpass — read a string of characters without echo, 222
getpassphrase — read a string of characters without echo, 222
getpriority — get or set process scheduling priority, 223
getpw — get passwd entry from UID, 225
getpwent — get password entry from user database, 226
getpwent_r — get password entry from user database, 226
getpwuid — get password entry from user database, 226
getpwuid_r — get password entry from user database, 226
getusage — get information about resource utilization, 230
getspent — get shadow password database entry, 234
getspent_r — get shadow password database entry (reentrant), 234
getspnam — get shadow password database entry, 234
getspnam_r — get shadow password database entry (reentrant), 234
getsubopt — parse suboptions from a string, 238
gettext — message handling functions, 241
gmtimeofday — get date and time, 245
gmtimeofday — get system’s notion of current Greenwich time, 247
gtxt — retrieve a text string, 248
getusershell() — get legal user shells, 250
getutent — user accounting database functions, 251
getutid — user accounting database functions, 251
getutline — user accounting database functions, 254
getutmp — user accounting database functions, 254
getutmpx — user accounting database functions, 254
getutxent — user accounting database functions, 254
getutxid — user accounting database functions, 254
getutxline — user accounting database functions, 254
getvfsany — get vfstab file entry, 258
getvfsent — get vfstab file entry, 258
getvfssfile — get vfstab file entry, 258
getvfsspec — get vfstab file entry, 258
getw — get a byte from a stream, 121
getwc — get wide character from a stream, 260
getwchar — get wide character from stdin stream, 261
getwd — get current working directory pathname, 262
getwidth — get codeset information, 263

Index 667
getws — get a wide-character string from a stream, 264
glob — generate path names matching a pattern, 265
globfree — generate path names matching a pattern, 265
grantpt — grant access to the slave pseudo-terminal device, 269
group database entry functions —
  endgrent, 201
  fgetgrent, 201
  fgetgrent_r, 201
  getgrent, 201
  getgrent_r, 201
  getgrentid, 201
  getgrgid, 201
  getgrgid_r, 201
  getgrnam, 201
  getgrnam_r, 201
  setgrent, 201
initgroups — initialize the supplementary group access list, 282
initstate — initialize the supplementary pseudorandom number
  functions, 405
input conversion, convert from wchar_t string —
wscanf, 657
input/output package, standard buffered I/O —
  stdio, 490
iconv — code conversion function, 273
iconv_close — code conversion deallocation function, 278
iconv_open — code conversion allocation function, 279
IEEE arithmetic, convert floating-point number to string —
cvt, 101
IEEE floating-point environment control —
  fpgetmasks, 145
  fpgetround, 145
  fpgetsticky, 145
  fpsetmask, 145
  fpsetround, 145
  fpsetsticky, 145
index — string operations, 281
initgroups — initialize the supplementary group access list, 282
input/output package, standard buffered I/O —
  stdio, 490
insque — insert element to a queue, 283
interfaces to stdio FILE structure —
  __flushlbf, 109
  __fbufsize, 109
  __fbfinancial, 109
  __freading, 109
  __fsetlocking, 109
  __fwritable, 109
  __fwriting, 109
invoke isa-specific executable —
  isaexec, 284
isaexec — invoke isa-specific executable, 284
isalnum — character handling, 62

H
  halt system processor, — reboot, 422
  hash-table search routine, — hsearch, 270
  hasmntopt — get mnttab file information, 211
  hcreate — create hash table, 270
  hdestroy — destroy hash table, 270
host name —
  get name of current host —
    gethostname, 206
  set name of current host —
    sethostname, 206
hsearch — hash-table search routine, 270

I
  I/O multiplexing, synchronous, — select, 444
isalpha — character handling, 62
isascii — character handling, 62
isatty — test for a terminal device, 286
isdigit — character handling, 62
isenglish — wide-character code classification functions, 289
isgraph — character handling, 62
isideogram — wide-character code classification functions, 289
islower — character handling, 62
isnumber — wide-character code classification functions, 289
isphonogram — wide-character code classification functions, 289
isprint — character handling, 62
ispunct — character handling, 62
isspace — character handling, 62
isspecial — wide-character code classification functions, 289
isupper — character handling, 62
iswalnum — wide-character code classification functions, 289
iswalpha — wide-character code classification functions, 289
iswascii — wide-character code classification functions, 289
iswcntrl — wide-character code classification functions, 289
iswctype — test character for specified class, 291
iswdigit — wide-character code classification functions, 289
iswgraph — wide-character code classification functions, 289
iswlower — wide-character code classification functions, 289
iswprint — wide-character code classification functions, 289
iswpunct — wide-character code classification functions, 289
iswspace — wide-character code classification functions, 289
iswupper — wide-character code classification functions, 289
iswxdigit — wide-character code classification functions, 289
isxdigit — character handling, 62

K
killpg — send signal to a process group, 293

L
labs — return absolute value of long integer, 22
language information — nl_langinfo, 364
ldexp — load exponent of a floating point number, 295
ldiv — compute quotient and remainder, 75
llint — display error message in standard format and pass to logging and monitoring services, 296
linear search and update routine
— lfind, 308
— lsearch, 308
llabs — return absolute value of long long integer, 22
lldiv — compute quotient and remainder, 75
lltostr — string conversion routines, 522
load exponent of a floating point number — ldexp, 295
locale, modify and query a program’s locale — setlocale, 460
localeconv — get numeric formatting information, 300
lock, apply or remove advisory lock on open file — flock, 129
lock address space, — mlockall, 348
lock memory pages, — mlock, 346
lockf — allows sections of file to be locked, 304
log message with a stdarg argument list — vsyslog, 607
login name
— getlogin, 209
— getlogin_r, 209
longjmp — non-local goto, 452, 455
_longjmp — non-local goto, 307, 455

M
madvise — provide advice to VM system, 310
make a FIFO special file — mkfifo, 339
make modified instructions executable — sync_instruction_memory, 531
makecontext — manipulate user contexts, 313
malloc — memory allocator, 30
mallocctl — MT hot memory allocator, 355
manipulate resource control blocks —
  rctlblk_get_enforced_value, 408
manipulate resource control blocks —
  rctlblk_get_firing_time, 408
manipulate resource control blocks —
  rctlblk_get_global_action, 408
manipulate resource control blocks —
  rctlblk_get_global_flags, 408
manipulate resource control blocks —
  rctlblk_get_local_action, 408
manipulate resource control blocks —
  rctlblk_get_local_flags, 408
manipulate resource control blocks —
  rctlblk_get_privilege, 408
manipulate resource control blocks —
  rctlblk_get_recipient_pid, 408
manipulate resource control blocks —
  rctlblk_get_value, 408
manipulate resource control blocks —
  rctlblk_set_local_action, 408
manipulate resource control blocks —
  rctlblk_set_local_flags, 408
manipulate resource control blocks —
  rctlblk_set_privilege, 408
manipulate resource control blocks —
  rctlblk_set_value, 408
manipulate sets of signals — sigsetops, 471
  sigaddset, 471
  sigdelset, 471
  sigemptyset, 471
  sigfillset, 471
  sigismember, 471
match filename or path name — fnmatch, 138
mblen — get number of bytes in a character, 325
mbrlen — get number of bytes in a character (restartable), 326
mbrtowc — convert a character to a wide-character code (restartable), 328
mbsinit — determine conversion object status, 330
mbstowcs — convert a character string to a wide-character string, 333
mbtowc — convert a character to a wide-character code, 334
mctl — memory management control, 335
memory — memory operations, 337
memory, optimizing usage of user mapped memory — madvise, 310
memory allocator — bsdmalloc, 30
memory allocator —
  alloca, 317
  calloc, 317, 320, 323
  malloc, 317, 320, 323
  free, 30
memory allocator —
  free, 317, 320, 323
  mallinfo, 320
memory allocator — bsdmalloc
  malloc, 30
memory allocator —
  malloc, 317, 320, 323
  mallocpt, 320
  memalign, 317
memory allocator — bsdmalloc
  realloc, 30
memory allocator —
  realloc, 317, 320, 323
  valloc, 317
memory lock or unlock, calling process — plock, 372
memory management — mctl, 335
memory management —
  get system page size — getpagesize, 220
lock address space — mlockall, 348
lock pages in memory — mlock, 346
synchronize memory with physical storage — msync, 353
unlock address space — munlockall, 348
unlock pages in memory — munlock, 346
memory operations —
  memccpy, 337
  memchr, 337
  memcmp, 337
  memcpy, 337
  memmove, 337
  memory, 337
  memset, 337
message handling functions —
bindtextdomain, 241
message handling functions —
bind_textdomain_codeset, 241
message handling functions —
dcgettext, 241
message handling functions —
dcngettext, 241
message handling functions —
dgettext, 241
message handling functions —
 dngettext, 241
message handling functions —
ggettext, 241
message handling functions —
ngettext, 241
message handling functions —
textdomain, 241
message catalog
open/catalog —
catopen, catclose, 38
read a program message —
catgets, 37
messages
display a message on stderr or system console —
fmtmsg, 133
print system error messages —
 perror, 368
system signal messages —
psignal, 390
mkfifo — make a FIFO special file, 339
mkstemp — make a unique file name, 341
mktemp — make a unique file name, 342
mtime — converts a tm structure to a calendar time, 343
modf — decompose floating-point number, 350
modff — decompose floating-point number, 350
monitor — prepare process execution profile, 351
msync — synchronize memory with physical storage, 353
MT hot memory allocator — mallocctl, 355
MT hot memory allocator — mtmalloc, 355
mtmalloc — MT hot memory allocator, 355

N
ndbm — database functions, 358
network group entry
— endnetgrent, 213
— getnetgrent, 213
— getnetgrent_r, 213
— innetgr, 213
— setnetgrent, 213
nextkey — database subroutines, 66
nftw — walk a file tree, 168
ngettext — message handling functions, 241
nice — change priority of a process, 362
nl_langinfo — language information, 364
nlist — get entries from symbol table, 363
non-local goto —
_setjmp, 452, 455
_longjmp, 455
longjmp, 452, 455
_sigsetjmp, 452
non-local goto
— _longjmp, 307
— _setjmp, 307
numbers, convert to strings —
econvert, 99
obtain information on scheduling decisions —
getcpuid, 189
obtain information on scheduling decisions —
gethomelgroup, 189
offsetof — offset of structure member, 365
open a shared object —
dlopen, 90
open directory —
fdopendir, 366
open directory —
opendir, 366
open a file —
attropen, 28
open a stream —
fopen, 140
open a stream —
freopen, 155
opendir —
open directory, 366
openlog —
control system log, 540
output conversion, wprintf —
convert to wchar_t string, 656
output conversion, formatted
— fprintf, 385
— printf, 385
— sprintf, 385
— vfprintf, 385
— vprintf, 385
— vsprintf, 385

P
page size, system, get —
getpagesize, 220
password databases
lock the lock file —
lckpwdf, 294
unlock the lock file — ulckpwdf, 294
Index 671
passwords
get password entry from a file —
   fgetpwent, 226
get password entry from a file —
   fgetpwent_r, 226
get password entry in user database —
   endpwent, 226
get password entry in user database —
   getpwent, 226
get password entry in user database —
   getpwent_r, 226
get password entry in user database —
   getpwuid, 226
get password entry in user database —
   getpwuid_r, 226
get password entry in user database —
   getpwnam, 226
get password entry in user database —
   getpwnam_r, 226
get password entry in user database —
   getpwuid, 226
get password entry in user database —
   getpwuid_r, 226
get password entry in user database —
   getpwnam, 226
get password entry in user database —
   getpwnam_r, 226
get password entry in user database —
   setpwent, 226
get passwd entry from UID — getpw, 225
write password file entry — putpwent, 396

passwords, shadow
get shadow password database entry —
   endspent, 234
get shadow password database entry —
   fgetspent, 234
get shadow password database entry —
   getspent, 234
get shadow password database entry —
   getspnam, 234
get shadow password database entry —
   setspent, 234
get shadow password database entry
   (reentrant) — fgetspent_r, 234
get shadow password database entry
   (reentrant) — getspent_r, 234
get shadow password database entry
   (reentrant) — getspnam_r, 234
write shadow password file entry —
   putspent, 398

path name, return last element — path
   name, 29

close — initiate pipe to/from a process, 373
perform word expansions
   — wordexp, 652
   — wordfree, 652

close — print system error messages, 368
pfmt — display error message in standard
   format, 369
pipes
   — printf, 376
   — printf, 376
   — snprintf, 376
   — sprintf, 376
print formatted output
   — fprintf, 174
   — printf, 174
   — wprintf, 174
print formatted wide-character output
   — fprintf, 174
   — wprintf, 174
   — wprintw, 174
print formatted output conversion, 385
print — print formatted output, 376
printstack — walk stack pointed to by
   ucontext, 615
Process Code string operations — wstring, 658
Process Code string operations — wscasecmp,
   wcsicmp, 658
Process Code string operations — wcsncmp,
   wcsncmp, 658
Process Code string operations — wcscol,
   wcscol, 658
Process Code string operations — wsdup,
   wsdup, 658
Process Code string operations — wscopy,
   wscopy, 658
process statistics, prepare execution profile —
   monitor, 351
processes
   change priority — nice, 362
duplicate an open file descriptor — dup2, 98
get character-string representation —
   cuserid, 65
generate path name for controlling terminal
   — ctermid, ctermid_r, 56
get character-string representation —
   cuserid, 65
initiate pipe to/from a process — popen,
   close, 373
manipulate user contexts — makecontext, swapcontext, 313
memory lock or unlock — plock, 372
prepare execution profile — monitor, 351
report CPU time used — clock, 43
send signal to a process group — killpg, 293
send signal to program — raise, 402
suspend execution for interval — sleep, 482
terminate process — exit, 106
terminate the process abnormally — abort, 21
wait for process to terminate or stop — WIFEXITED, 611
wait for process to terminate or stop — WIFSIGNALED, 611
wait for process to terminate or stop — WIFSTOPPED, 611
wait for process to terminate or stop — wait, 611
profiling utilities, prepare process execution profile — monitor, 351
program assertion, verify — assert, 26
program messages
  open/close a message catalog — catopen, catclose, 38
  read — catgets, 37
programs, last locations — end, etext, edata, 104
pset_getloadavg — get system load averages for a processor set, 389
pseudo-terminal device
  get name of the slave pseudo-terminal device — ptsname, 394
  grant access to the slave pseudo-terminal device — grantpt, 269
pseudorandom number functions
  initstate, 405
  random, 405
  setstate, 405
  srandom, 405
putenv — change or add value to environment, 395
putmntent — get mnttab file information, 211
putpwent — write password file entry, 396
putspent — write shadow password file entry, 398
pututline — user accounting database functions, 251
putuxline — user accounting database functions, 254
putw — put a byte on a stream, 147
putwc — put wide-character code on a stream, 147
putwchar — put wide-character code on a stream, 150
putwchar_unlocked — put wide-character code on a stream, 150
putw — put a byte on a stream, 147
putwc — put wide-character code on a stream, 147
putwchar — put wide-character code on a stream, 150
putwchar_unlocked — put wide-character code on a stream, 150
putw — convert a string of Process Code characters to EUC characters and put it on a stream, 399
q
qecovnert — convert number to ASCII, 99
qfconvert — convert number to ASCII, 99
qgconvert — convert number to ASCII, 99
qsort — quick sort, 400
quadruple_to_decimal — decimal record from quadruple-precision floating, 127
queues, insert/remove element from a queue — insque, remque, 283

R
raise — send signal to program, 402
rand — simple random number generator, 404
rand — simple random-number generator, 403
random — pseudorandom number functions, 405
random number generator
— drand48, 96
— erand48, 96
— jrand48, 96
— iconv48, 96
— irand48, 96
— mrand48, 96
— nrand48, 96
— rand, 404
— seed48, 96
— srand48, 96
random number generator, simple
— rand, 403
— srand, 403
rctl_walk — visit registered rctls on current system, 412
rctlblk_get_enforced_value — manipulate resource control blocks, 408
rctlblk_get_firing_time — manipulate resource control blocks, 408
rctlblk_get_global_action — manipulate resource control blocks, 408
rctlblk_get_global_flags — manipulate resource control blocks, 408
rctlblk_get_local_action — manipulate resource control blocks, 408
rctlblk_get_local_flags — manipulate resource control blocks, 408
rctlblk_set_privilege — manipulate resource control blocks, 408
rctlblk_set_value — manipulate resource control blocks, 408
rctlblk_size — manipulate resource control blocks, 408
re_comp — compile and execute regular expressions, 423
re_exec — compile and execute regular expressions, 423
read a string of characters without echo — getpass, 222
read a directory entry — readdir, 418
read a string of characters without echo, — getpassphrase, 222
read directory
— readdir, 414
— readdir_r, 414
readdir — read a directory entry, 418
readdir — read directory, 414
POSIX, 414
readdir_r — read directory, 414
realloc — memory allocator, 30
realpath — resolve pathname, 420
reboot — reboot system or halt processor, 422
regcmp — compile regular expression, 424
regcomp — regular expression matching, 426
regerror — regular expression matching, 426
regex — execute regular expression, 424
regexec — regular expression matching, 426
regfree — regular expression matching, 426
register a function to run at process termination or object unloading — atexit, 27
register fork handlers — pthread_atfork, 392
regular expression matching
— regcomp, 426
— regerror, 426
— regexec, 426
— regfree, 426
regular expressions, compile and execute — regcmp, regex, 424
remove — remove file, 432
remque — remove element from a queue, 283
reposition a file-position indicator in a stream
  — fseek, 158
  — fseeko, 158
reset file position indicator in a stream —
  rewind, 433
reset position of directory stream to the
  beginning of a directory — rewinddir, 434
resetmnttab — get mnttab file information, 211
resolve path name — realpath, 420
resource utilization, get information —
  getusage, 230
retrieve stack boundaries —
  stack_getbounds, 484
return a file offset for a file descriptor
  — tell, 555
return a file offset in a stream
  — ftell, 164
  — ftello, 164
return pathname of executable —
  getexecname, 200
rewind — reset file position indicator in a
  stream, 433
rewinddir — reset position of directory stream
  to the beginning of a directory, 434
rindex — string operations, 281

S
scan a directory
  — alphasort, 435
  — scandir, 435
scandir — scan a directory, 435
scanf — convert formatted input, 436
Conversion Characters, 438
Conversion Specifications, 437
scheduling priority, change priority of a process
  — nice, 362
search functions
  binary search a sorted table — bsearch, 33
  linear search and update routine — lsearch,
    lfind, 308
  manage hash search tables — hsearch, 270
  seconvert — convert number to ASCII, 99
seekdir — set position of directory stream, 443
select — synchronous I/O multiplexing, 444
send a “break” for a specific duration —
  tcsendbreak, 551
set and/or get alternate signal stack context —
  sigstack, 473
set and/or get signal stack context —
  sigstack, 475
set encoding key — setkey, 458
set foreground process group ID —
  tcsetpgid, 554
set input baud rate, — cfsetispeed, 42
set output baud rate, — cfsetospeed, 42
set position of directory stream — seekdir, 443
set stream orientation — fwide, 173
set the parameters associated with the terminal
  — tcsetattr, 552
set wide-characters in memory —
  wmemset, 651
setcat — define default catalog, 451
setgrent — group database entry functions, 201
sethostname — set name of current host, 206
setjmp — non-local goto, 452, 455
  _setjmp — non-local goto, 307, 455
setkey — set encoding key, 458
setlocale — define the label for pfmt() and
  fmt(), 459
setlogmask — control system log, 540
setpriority — get or set process scheduling
  priority, 223
setpwnam — get password entry from user
  database, 226
setspent — get shadow password database
  entry, 234
setstate — pseudorandom number
  functions, 405
settimeofday — set date and time, 245
settimeofday — set system’s notion of current
  Greenwich time, 247
setusershell() — function, 250
setutent — user accounting database
  functions, 251
setuxent — user accounting database
  functions, 254
severity levels, applications, build a list for use
  with fntmsg — addseverity, 24
sfconvert — convert number to ASCII, 99
sgconvert — convert number to ASCII, 99
shared object
  close — dlclose, 78
shared object (Continued)
    get address of symbol — dlsym, 94
    get diagnostic information — dlerror, 85
    open — dlopen, 90
shell command, issue one — system, 544
sig2str — translation between signal name and signal number, 494
sigaddset — manipulate sets of signals, 471
sigdelset — manipulate sets of signals, 471
sigemptyset — manipulate sets of signals, 471
sigfdset — manipulate sets of signals, 471
signal — modify signal disposition, 467
signal — simplified software signal facilities, 469
signal
    schedule after interval in microseconds — ualarm, 579
    suspend execution for interval in microseconds — usleep, 599
sigismember — manipulate sets of signals, 471
siglongjmp — non-local goto, 452
signal — modify signal disposition, 467
signal — simplified software signal facilities, 469
siglongjmp
    non-local goto, 452
signal
    schedule after interval in microseconds — ualarm, 579
    suspend execution for interval in microseconds — usleep, 599
sigismember — manipulate sets of signals, 471
signal management, simplified, for application processes — signal, 467
signal messages, system
    — psignal, 390, 391
signals, block
    — sigblock, 463
    — sigmask, 463
    — sigpause, 463
    — sigsetmask, 463
signals, software
    — gsignal, 483
    — ssignal, 483
signal
    schedule after interval in microseconds — ualarm, 579
    suspend execution for interval in microseconds — usleep, 599
sigismember — manipulate sets of signals, 471
signal — modify signal disposition, 467
signal — simplified software signal facilities, 469
siglongjmp
    non-local goto, 452
signal
    schedule after interval in microseconds — ualarm, 579
    suspend execution for interval in microseconds — usleep, 599
sigismember — manipulate sets of signals, 471
signal — modify signal disposition, 467
signal — simplified software signal facilities, 469
siglongjmp
    non-local goto, 452
sigsetops — manipulate sets of signals, 471
sigstack — set and/or get alternate signal stack context, 473
sigstack — set and/or get signal stack context, 475
sigvec — software signal facilities, 476
single-byte to wide-character conversion — btowc, 36
single-to-decimal — decimal record from single-precision floating, 127
sleep — suspend execution for interval, 482
sleep, suspend execution for interval in microseconds — usleep, 599
snprintf — print formatted output, 376
software signals
    — gsignal, 483
    — ssignal, 483
sort, quick — qsort, 400
sprintf — formatted output conversion, 385
sprintf — print formatted output, 376
srand — reset simple random number generator, 404
srandom — pseudorandom number functions, 405
sscanf — convert formatted input, 436
stack_getbounds — retrieve stack boundaries, 484
_stack_grow — express an intention to extend the stack, 485
stack_inbounds — determine if address is within stack boundaries, 486
stack_setbounds — update stack boundaries, 487
stackViolation — determine stack boundary violation event, 488
stdio — standard buffered input/output package, 490
string collation, — strcoll, 495
store — data base subroutines, 66
str2sig — translation between signal name and signal number, 494
str casecmp — string operations, 507
strcat — string operations, 507
strchr — string operations, 507
strcmp — string operations, 507
strcspn — string operations, 507
strcpy — string operations, 507
strdup — string operations, 507
stream
  convert a string of Process Code characters to
  EUC characters and put it on a stream —
  putw, 399
open — fopen, 143
stream, assign buffering
  — setbuf, 448
  — setvbuf, 448
stream, get string
  — fgets, 233
  — gets, 233
stream, put a string
  — fputs, 397
  — puts, 397
stream status inquiries
  — clearerr, 117
  — feof, 117
  — ferror, 117
  — fileno, 117
STREAMS
  attach a STREAMS-based file descriptor to an
  object in the file system name space —
  fattach, 107
test file descriptor for a STREAMS file —
  isastream, 285
strfmon — convert monetary value to
  string, 497
strftime — convert date and time to string, 501
string — string operations, 507
string operations — strcasecmp, 507
string operations — strcmp, 507
string operations — strcpy, 507
string operations — strcspn, 507
string operations — strdup, 507
string operations — strlen, 507
string operations — strlcat, 507
string operations — strlcpy, 507
string operations — strlens, 507
string operations — strncasecmp, 507
string operations — strncat, 507
string operations — strncmp, 507
string operations — strncpy, 507
string operations — strpbrk, 507
string operations — strrchr, 507
string operations — strspn, 507
string operations — strstr, 507
stream conversion routines
  — atoi, 522
  — atol, 522
  — atoll, 522
  — lltostr, 522
  — strtol, 522
  — strtoll, 522
  — ultostr, 522
string encoding function — crypt, 51
string operation, get error message string —
  strerror, 496
string operations
  bit and byte — bstring, 35
  — index, 281
  — rindex, 281
string_to_decimal — decimal record from
  character string, 510
string transformation — strxfrm, 528
strings, convert from numbers — econvert, 99
strlcat — string operations, 507
strlcpy — string operations, 507
strlen — string operations, 507
strncasecmp — string operations, 507
strncat — string operations, 507
strncpy — string operations, 507
strpbrk — string operations, 507
strptime — date and time conversion, 513
strrcr — string operations, 507
strsignal — get name of signal, 518
strspn — string operations, 507
strstr — string operations, 507
strtod — convert string to double-precision
  number, 519
strtok — string operations, 507
strtok — string operations, 507
strtol — string conversion routines, 522
strtotll — string conversion routines, 522
strtofl — convert string to unsigned long, 525
strtows — code conversion for Process Code
  and File Code, 527
strxfrm — string transformation, 528
suspend or restart the transmission or reception
  of data — tcflow, 546
swab — swap bytes, 530
swap bytes — swab, 530
swapcontext — manipulate user contexts, 313
swprintf — print formatted wide-character output, 174
swscanf — convert formatted wide-character input, 182
symbol address, get address in shared object or executable — dlsym, 94
symbol table, get entries — nlist, 363
sync_instruction_memory — make modified instructions executable, 531
synchronize changes to a file — fsync, 162
synchronous I/O multiplexing
— FD_CLR, 444
— FD_ISSET, 444
— FD_SET, 444
— select, 444
sys_siglist — system signal messages list, 391
syscall — indirect system call, 532
sysconf — get configurable system variables, 533
syslog — control system log, 540
system — issue shell command, 544
system error messages, print — perror, 368
system signal messages, — psignal, 390
system variables, get configurable ones — sysconf, 533
tell — return a file offset for a file descriptor, 555
telldir — current location of a named directory stream, 556
tempnam — create a name for a temporary file, 560
terminal, find the slot of the current user in the user accounting database — ttyslot, 578
terminal device, slave pseudo
get name — ptsname, 394
grant access — grantpt, 269
termios — general terminal interface, 557
test character for specified class — iswctype, 291
test for a terminal device — isatty, 286
text processing utilities
compile and execute regular expressions — regcmp, regex, 424
quick sort — qsort, 400
text string, — gettxt, 248
textdomain — message handling functions, 241
find — manage binary search trees, 573
time, computes the difference between two calendar times — difftime, 70
time, calendar, convert from a tm structure — mktime, 343
time accounting, for current process — times, 558
time and date
convert to string — asctime, 57
convert to string — ctime, 57
convert to string — gmtime, 57
convert to string — localtime, 57
convert to string — tzset, 57
convert user format date and time — getdate, 192
get — ftime, 165
— settimeofday, 245
time of day, get and set — gettimeofday, 247
times — get process times, 558
tmpfile — create a temporary file, 559
tmpnam — create a name for a temporary file, 560
toascii — translate integer to a 7-bit ASCII character, 562

T
tcdrain — wait for transmission of output, 545
tcflow — suspend or restart the transmission or reception of data, 546
tcflush — flush non-transmitted output data, non-read input data or both, 547
tcgetattr — get the parameters associated with the terminal, 548
tcgetpgrp — get foreground process group ID, 549
tcgetsid — get process group ID for session leader for controlling terminal, 550
tcsendbreak — send a “break” for a specific duration, 551
tcsetattr — set the parameters associated with the terminal, 552
tcgetpgrp — set foreground process group ID, 554
tdelete — manage binary search trees, 573
tolower — transliterate upper-case characters to lower-case, 564
_tolower — transliterate upper-case characters to lower-case, 563
toupper — transliterate lower-case characters to upper-case, 566
_toupper — transliterate lower-case characters to upper-case, 565
towctrans — wide-character mapping, 567
towlower — transliterate upper-case wide-character code to lower-case, 568
towupper — transliterate lower-case wide-character code to upper-case, 569
translate address to symbolic information — dladdr1, 76
translate address to symbolic information — dladdr, 76
translate integer to a 7-bit ASCII character — toascii, 562
translation between signal name and signal number — str2sig, 494
sig2str, 494
transliterate lower-case characters to upper-case — toupper, 566
transliterate lower-case characters to upper-case — _toupper, 565
transliterate lower-case wide-character code to upper-case — towupper, 569
transliterate upper-case characters to lower-case — tolower, 564
transliterate upper-case characters to lower-case — _tolower, 563
transliterate upper-case wide-character code to lower-case — towlower, 568
truncate — set a file to a specified length, 570
tsearch — manage binary search trees, 573
ttyname — find path name of a terminal, 576
POSIX, 576
ttyname_r — find path name of a terminal, 576
ttyslot — find the slot of the current user in the user accounting database, 578
twalk — manage binary search trees, 573

U
ualarm — schedule signal after interval in microseconds, 579
ulltostr — string conversion routines, 522
umem_alloc — fast, scalable memory allocation, 580
umem_cache_alloc — allocation cache manipulation, 585
umem_cache_create — allocation cache manipulation, 585
umem_cache_destroy — allocation cache manipulation, 585
umem_cache_free — allocation cache manipulation, 585
umem_debug — debugging features of the umem library, 593
umem_free — fast, scalable memory allocation, 580
umem_nofail_callback — fast, scalable memory allocation, 580
umem_zalloc — fast, scalable memory allocation, 580
ungetc — push byte back into input stream, 596
ungetwc — push wide-character code back into input stream, 597
unlock a pseudo-terminal master/slave pair — unlockpt, 598
unlock address space, — munlockall, 348
unlock memory pages, — munlock, 346
unlockpt — unlock a pseudo-terminal master/slave pair, 598
update stack boundaries — stack_setbounds, 487
updwtmp — user accounting database functions, 254
updwtmpx — user accounting database functions, 254
user accounting database functions — endutent, 251
user accounting database functions — endutxent, 254
user accounting database functions — getutent, 251
user accounting database functions — getutid, 251
user accounting database functions — getutline, 251
user accounting database functions — getutmp, 254
user accounting database functions —
  getutmpx, 254
user accounting database functions —
  getutxent, 254
user accounting database functions —
  getutxid, 254
user accounting database functions —
  getutxline, 254
user accounting database functions —
  pututline, 251
user accounting database functions —
  pututxline, 254
user accounting database functions —
  setutent, 251
user accounting database functions —
  setutxent, 254
user accounting database functions —
  updwtmp, 254
user accounting database functions —
  updwtmpx, 254
user accounting database functions —
  utmpname, 251
user accounting database functions —
  utmpxname, 254
user context
  — makecontext, 313
  — swapcontext, 313
user IDs, get character-string representation —
  cuserid, 65
usleep — suspend execution for interval in microseconds, 599
utmpname — user accounting database functions, 251
utmpx file, find the slot of current user —
  ttyslot, 578
utmpxname — user accounting database functions, 254

V
vfprintf — formatted output conversion, 385
vscanf — convert formatted input, 436
vstab file, — getvfsent, 258
vfwprintf — wide-character formatted output of a stdarg argument list, 600
vfwscanf — convert formatted wide-character input, 182
virtual memory, optimizing usage of user mapped memory — madvise, 310
visit registered rctls on current system —
  rctl_walk, 412
vfmt — display error message in standard format and pass to logging and monitoring services, 601
vpfmt — display error message in standard format and pass to logging and monitoring services, 603
vprintf — formatted output conversion, 385
vsscanf — convert formatted input, 436
vsscanf — formatted output conversion, 385
vsscanf — convert formatted input, 436
vswprintf — wide-character formatted output of a stdarg argument list, 600
vswscanf — convert formatted wide-character input, 182
vsyslog — log message with a stdarg argument list, 607
vwprintf — wide-character formatted output of a stdarg argument list, 600
vwscanf — convert formatted wide-character input, 182

W
wait — wait for process to terminate or stop, 611
waitpid — wait for process to terminate or stop, 611
wait for process to terminate or stop — wait3, 608
  — wait4, 608
wait for transmission of output — tcdrain, 545
wait3 — wait for process to terminate or stop, 608, 611
wait4 — wait for process to terminate or stop, 608, 611
walk stack pointed to by ucontext —
  printstack, 615
walk stack pointed to by ucontext —
  walkcontext, 615
walkcontext — walk stack pointed to by ucontext, 615
watof — convert wide character string to double-precision number, 627
watoi — convert wide character string to long integer, 629
watol — convert wide character string to long integer, 629
watoll — convert wide character string to long integer, 629
wchar_t string, number conversion — wscanf, 657
wcrtomb — convert a wide-character code to a character (restartable), 620
wcschr — wide-character string operations, 635
wcsmp — wide-character string operations, 635
wcscoll — wide character string comparison using collating information, 622
wcsncpy — wide-character string operations, 636
wcsncpy — wide-character string operations, 636
wcsstr — find a wide-character substring, 626
wcstod — convert wide character string to double-precision number, 627
wcstok — wide-character string operations, 637
wcstol — convert wide character string to long integer, 629
wcstombs — convert a wide-character string to a character string, 631
wcstoul — convert wide character string to unsigned long, 632
wcstring — wide-character string operations, 635
wcswcs — wide-character string operations, 637
wcswidth — number of column positions of a wide-character string, 639
wcsxfm — wide character string transformation, 640
wctob — wide-character to single-byte conversion, 642
wcswtomb — convert a wide-character code to a character, 643
wcstrans — define character mapping, 644
wctype — define character class, 645
wcwidth — number of column positions of a wide-character code, 646
wide character string to long integer, convert
— watoi, 629
— watol, 629
— watoll, 629
— wcstol, 629
— wstol, 629
wide-character code classification functions
— isenglish, 289
— isideogram, 289
— isnumber, 289
— isphonogram, 289
— issspecial, 289
— iswalnum, 289
— iswalpha, 289
— iswascii, 289
— iswcntrl, 289
— iswdigit, 289
— iswgraph, 289
— iswlower, 289
— iswprint, 289
— iswpunct, 289
— iswspace, 289
— iswupper, 289
— iswdxigit, 289
wide-character formatted output of a stdarg argument list
— vfwprintf, 600
— vswprintf, 600
— vwprintf, 600
wide-character mapping — towctrans, 567
wide character string comparison using collating information
— wcsoll, 622
— wscoll, 622
wide-character string operations
— wcscat, 635
— wcshr, 636
— wcscmp, 635
— wcscpy, 636
— wcsncmp, 636
— wcsncmp, 636
— wcsncpy, 636
— wcsncat, 635
— wcsncmp, 636
— wcsnmem, 636
— wcspbrk, 637
— wcsrchr, 636
— wcsppn, 637
— wcstok, 637
— wcstring, 635
— wcswcs, 637
— windex, 636
— wrindex, 636
wide character string transformation
— wcxfrm, 640
— wssxfrm, 640
wide-character to single-byte conversion —
— wctob, 642
windex — wide-character string operations, 636
wmemchr — find a wide-character in memory, 647
wmemcmp — compare wide-characters in memory, 648
wmemcpy — copy wide-characters in memory, 649
wmemmove — copy wide-characters in memory with overlapping areas, 650
wmemset — set wide-characters in memory, 651
wordexp — perform word expansions, 652
wordfree — perform word expansions, 652
working directory, get pathname — getwd, 262
wprintf — print formatted wide-character output, 174
wrindex — wide-character string operations, 636
wscanf — convert formatted wide-character input, 182

wscasecmp — Process Code string operations, 658
wscol — Process Code string operations, 658
wscoll — wide character string comparison using collating information, 622
wsdup — Process Code string operations, 658
wsncmp — Process Code string operations, 658
wsprintf — formatted output conversion, 656
wsscanf — formatted input conversion, 657
wstod — convert wide character string to double-precision number, 627
wstol — convert wide character string to long integer, 629
wstostr — code conversion for Process Code and File Code, 527
wsxfrm — wide character string transformation, 640