ChorusOS man pages section 3STDC: Standard C Library Functions

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 806-3335
December 10, 1999

Contents

PREFACE

Overview

A man page is provided for both the naive user, and sophisticated user who is familiar with the ChorusOS™ operating system and is in need of on-line information. 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.

The following is a list of sections in the ChorusOS man pages and the information it references:

- *Section 1CC: User Utilities; Host and Target Utilities*
- *Section 1M: System Management Utilities*
- *Section 2DL: System Calls; Data Link Services*
- *Section 2K: System Calls; Kernel Services*
- *Section 2MON: System Calls; Monitoring Services*
- *Section 2POSIX: System Calls; POSIX System Calls*
- *Section 2RESTART: System Calls; Hot Restart and Persistent Memory*
- *Section 2SEG: System Calls; Virtual Memory Segment Services*
- *Section 3FTPD: Libraries; FTP Daemon*
- *Section 3M: Libraries; Mathematical Libraries*
- *Section 3POSIX: Libraries; POSIX Library Functions*
- *Section 3RPC: Libraries; RPC Services*
- *Section 3STDC: Libraries; Standard C Library Functions*
- *Section 3TELD: Libraries; Telnet Services*
- *Section 4CC: Files*

- *Section 5FEA: ChorusOS Features and APIs*
- *Section 7P: Protocols*
- *Section 7S: Services*
- *Section 9DDI: Device Driver Interfaces*
- *Section 9DKI: Driver to Kernel Interface*
- *Section 9DRV: Driver Implementations*

ChorusOS man pages are grouped in Reference Manuals, with one reference manual per section.

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 pathname 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:

[ ]	The option or argument enclosed in these brackets is optional. If the brackets are omitted, the argument must be specified.
. . .	Ellipses. Several values may 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 time.
{ }	Braces. The options and/or arguments enclosed within braces are

interdependent, such that everything enclosed must be treated as a unit.

FEATURES	This section provides the list of features which offer an interface. An API may be associated with one or more system features. The interface will be available if one of the associated features has been configured.
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, functions and such, are described under USAGE.
OPTIONS	This 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 is provided as a guidance on use. This section lists special rules, features and commands that require in-depth explanations. The subsections listed below 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 filenames 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.
SEE ALSO	This section lists references to other man pages, in-house documentation and outside publications.
DIAGNOSTICS	This section lists diagnostic messages with a brief explanation of the condition causing the error.
WARNINGS	This section lists warnings about special conditions which could seriously affect your working conditions. This is not a list of diagnostics.
NOTES	This section lists additional information that does not belong anywhere else on the page. It takes the form of an aside to the user, covering points of special interest. Critical information is never covered here.

BUGS

This section describes known bugs and wherever possible, suggests workarounds.

CHAPTER

Standard C Library Functions

NAME

Intro, intro – introduction to functions and libraries

DESCRIPTION

This section describes threadsafe C library functions. Function prototypes can be obtained from the `#include` files indicated on each page.

References of the form *name* (2K), *name* (2POSIX), *name* (3POSIX) and *name* (3STDC) refer to pages in this section of this document.

DEFINITIONS

A character is any bit pattern able to fit into a byte on the machine. The null character is a character with value 0, conventionally represented in the C language as \0. A character array is a sequence of characters. A null-terminated character array (a *string*) is a sequence of characters, the last of which is the null character. The null string is a character array containing only the terminating null character. A NULL pointer is the value that is obtained by casting 0 into a pointer. C guarantees that this value will not match any legitimate pointer, so many functions that return pointers return NULL to indicate an error. The macro NULL is defined in `stdio.h` .

NOTES

Routines from (2POSIX), (3POSIX), (3STDC) are suitable for being linked and invoked in any actor, whether it is an embedded user or supervisor actor, or a c_actor. Routines from (3STDC) provide the traditional UNIX level 3 IO service. These routines assume the existence of a subset of the UNIX IO level 2 interface.

STANDARDS

All (2POSIX), (3POSIX) and (3STDC) routines that have a definition in `POSIX.1c`, `POSIX.1b`, or `ANSI-C`, conform to that definition, in this decreasing order of priority. In particular, almost all routines are reentrant. Those routines that are not reentrent are signaled in the corresponding manual page, and the `POSIX.1c` reentrent replacement is provided.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

Name	Description
`_assert`(3STDC)	See `assert`(3STDC)
`_ldexp`(3STDC)	See `ldexp`(3STDC)
`_stdc_assert`(3STDC)	See `assert`(3STDC)
`abort`(3STDC)	cause abnormal program termination
`abs`(3STDC)	integer absolute value function
`alphasort`(3STDC)	See `scandir`(3STDC)
`asctime`(3STDC)	See `ctime`(3STDC)

`asctime_r`(3STDC)	See `ctime_r`(3STDC)
`assert`(3STDC)	expression verification macro
`atexit`(3STDC)	add program termination routines
`atof`(3STDC)	See `strtod`(3STDC)
`atoi`(3STDC)	See `strtol`(3STDC)
`atol`(3STDC)	See `strtol`(3STDC)
`bcmp`(3STDC)	See `bstring`(3STDC)
`bcopy`(3STDC)	See `bstring`(3STDC)
`bsearch`(3STDC)	perform a binary search on a sorted table
`bstring`(3STDC)	bit and byte string operations
`byteorder`(3STDC)	convert values between host and network byte order
`bzero`(3STDC)	See `bstring`(3STDC)
`calloc`(3STDC)	See `malloc`(3STDC)
`clearerr`(3STDC)	See `ferror`(3STDC)
`ctime`(3STDC)	transform binary date and time value to ASCII
`ctime_r`(3STDC)	Transform binary date and time value to ASCII; Reentrent version
`ctype`(3STDC)	classify characters
`difftime`(3STDC)	See `ctime`(3STDC)
`div`(3STDC)	return quotient and remainder from division
`errno`(3STDC)	See `perror`(3STDC)
`exit`(3STDC)	terminate an actor
`fabs`(3STDC)	floating-point absolute value function
`fclose`(3STDC)	close or flush a stream
`fdopen`(3STDC)	See `fopen`(3STDC)
`feof`(3STDC)	See `ferror`(3STDC)
`ferror`(3STDC)	stream status inquiries
`fflush`(3STDC)	See `fclose`(3STDC)

`ffs`(3STDC)	See `bstring`(3STDC)
`fgetc`(3STDC)	See `getc`(3STDC)
`fgetpos`(3STDC)	See `fseek`(3STDC)
`fgets`(3STDC)	See `gets`(3STDC)
`fileno`(3STDC)	See `ferror`(3STDC)
`flockfile`(3STDC)	stream lock management
`fopen`(3STDC)	open a stream
`fprintf`(3STDC)	print formatted output
`fputc`(3STDC)	See `putc`(3STDC)
`fputs`(3STDC)	See `puts`(3STDC)
`fread`(3STDC)	binary input/output
`free`(3STDC)	See `malloc`(3STDC)
`freopen`(3STDC)	See `fopen`(3STDC)
`fscanf`(3STDC)	convert formatted input
`fseek`(3STDC)	reposition a file pointer in a stream
`fsetpos`(3STDC)	See `fseek`(3STDC)
`ftell`(3STDC)	See `fseek`(3STDC)
`ftrylockfile`(3STDC)	See `flockfile`(3STDC)
`funlockfile`(3STDC)	See `flockfile`(3STDC)
`fwrite`(3STDC)	See `fread`(3STDC)
`getc`(3STDC)	get character from a stream
`getc_unlocked`(3STDC)	See `unlocked`(3STDC)
`getchar`(3STDC)	get character from the standard input channel
`getchar_unlocked`(3STDC)	See `unlocked`(3STDC)
`getenv`(3STDC)	fetch and set environment variables
`gethostbyaddr`(3STDC)	get network host entry
`gethostbyname`(3STDC)	See `gethostbyaddr`(3STDC)

`getopt`(3STDC)	get an option letter from command line argument list
`gets`(3STDC)	get a string from a stream
`getsitebyaddr`(3STDC)	See `getsitebyname`(3STDC)
`getsitebyname`(3STDC)	get ChorusOS site information
`getsubopt`(3STDC)	get sub options from an argument
`getw`(3STDC)	See `getc`(3STDC)
`gmtime`(3STDC)	See `ctime`(3STDC)
`gmtime_r`(3STDC)	See `ctime_r`(3STDC)
`htonl`(3STDC)	See `byteorder`(3STDC)
`htons`(3STDC)	See `byteorder`(3STDC)
`index`(3STDC)	locate character in string
`inet`(3STDC)	Internet address manipulation routines
`inet_addr`(3STDC)	See `inet`(3STDC)
`inet_aton`(3STDC)	See `inet`(3STDC)
`inet_lnaof`(3STDC)	See `inet`(3STDC)
`inet_makeaddr`(3STDC)	See `inet`(3STDC)
`inet_netof`(3STDC)	See `inet`(3STDC)
`inet_network`(3STDC)	See `inet`(3STDC)
`inet_ntoa`(3STDC)	See `inet`(3STDC)
`initstate`(3STDC)	See `random`(3STDC)
`isalnum`(3STDC)	See `ctype`(3STDC)
`isalpha`(3STDC)	See `ctype`(3STDC)
`isascii`(3STDC)	test for ASCII character
`isatty`(3STDC)	check if a file descriptor is associated with a terminal
`iscntrl`(3STDC)	See `ctype`(3STDC)
`isdigit`(3STDC)	See `ctype`(3STDC)
`isgraph`(3STDC)	See `ctype`(3STDC)

`isinf`(3STDC)	test for infinity or not-a-number
`islower`(3STDC)	See `ctype`(3STDC)
`isnan`(3STDC)	See `isinf`(3STDC)
`isprint`(3STDC)	See `ctype`(3STDC)
`ispunct`(3STDC)	See `ctype`(3STDC)
`isspace`(3STDC)	See `ctype`(3STDC)
`isupper`(3STDC)	See `ctype`(3STDC)
`isxdigit`(3STDC)	See `ctype`(3STDC)
`labs`(3STDC)	return the absolute value of a long integer
`ldexp`(3STDC)	multiply floating-point number by integral power of 2
`ldiv`(3STDC)	return quotient and remainder from division
`localtime`(3STDC)	See `ctime`(3STDC)
`localtime_r`(3STDC)	See `ctime_r`(3STDC)
`longjmp`(3STDC)	See `setjmp`(3STDC)
`malloc`(3STDC)	main memory allocator
`memccpy`(3STDC)	See `memory`(3STDC)
`memchr`(3STDC)	See `memory`(3STDC)
`memcmp`(3STDC)	See `memory`(3STDC)
`memcpy`(3STDC)	See `memory`(3STDC)
`memmove`(3STDC)	See `memory`(3STDC)
`memory`(3STDC)	memory operations
`memset`(3STDC)	See `memory`(3STDC)
`mkstemp`(3STDC)	See `mktemp`(3STDC)
`mktemp`(3STDC)	make temporary file name (unique)
`mktime`(3STDC)	See `ctime`(3STDC)
`modf`(3STDC)	extract signed integral and fractional values from floating-point number
`ntohl`(3STDC)	See `byteorder`(3STDC)

`ntohs`(3STDC)	See `byteorder`(3STDC)
`perror`(3STDC)	system error messages
`printerr`(3STDC)	See `printf`(3STDC)
`printf`(3STDC)	print formatted output
`putc`(3STDC)	put character or word on a stream
`putc_unlocked`(3STDC)	See `unlocked`(3STDC)
`putchar`(3STDC)	put a character or word on the standard output channel
`putchar_unlocked`(3STDC)	See `unlocked`(3STDC)
`putenv`(3STDC)	See `getenv`(3STDC)
`puts`(3STDC)	put a string on a stream
`putw`(3STDC)	See `putc`(3STDC)
`qsort`(3STDC)	quicker sort
`rand`(3STDC)	pseudo random number generator
`rand_r`(3STDC)	thread-wise random number generator
`random`(3STDC)	better random number generator
`realloc`(3STDC)	See `malloc`(3STDC)
`regcomp`(3STDC)	See `regex`(3STDC)
`regerror`(3STDC)	See `regex`(3STDC)
`regex`(3STDC)	regular-expression library
`regexec`(3STDC)	See `regex`(3STDC)
`regfree`(3STDC)	See `regex`(3STDC)
`remove`(3STDC)	remove directory entry
`rewind`(3STDC)	See `fseek`(3STDC)
`rindex`(3STDC)	See `index`(3STDC)
`scandir`(3STDC)	scan a directory
`scanf`(3STDC)	convert formatted input
`setbuf`(3STDC)	assign buffering to a stream

`setenv`(3STDC)	See `getenv`(3STDC)
`setjmp`(3STDC)	non-local goto
`setstate`(3STDC)	See `random`(3STDC)
`setvbuf`(3STDC)	See `setbuf`(3STDC)
`snprintf`(3STDC)	See `printf`(3STDC)
`sprintf`(3STDC)	See `printf`(3STDC)
`srand`(3STDC)	See `rand`(3STDC)
`srandom`(3STDC)	See `random`(3STDC)
`sscanf`(3STDC)	See `scanf`(3STDC)
`stdarg`(3STDC)	variable argument lists
`strcasecmp`(3STDC)	See `string`(3STDC)
`strcat`(3STDC)	See `string`(3STDC)
`strchr`(3STDC)	See `string`(3STDC)
`strcmp`(3STDC)	See `string`(3STDC)
`strcoll`(3STDC)	See `string`(3STDC)
`strcpy`(3STDC)	See `string`(3STDC)
`strcspn`(3STDC)	See `string`(3STDC)
`strdup`(3STDC)	See `string`(3STDC)
`strerror`(3STDC)	system error messages
`strftime`(3STDC)	format date and time
`string`(3STDC)	string operations
`strlen`(3STDC)	See `string`(3STDC)
`strncasecmp`(3STDC)	See `string`(3STDC)
`strncat`(3STDC)	See `string`(3STDC)
`strncmp`(3STDC)	See `string`(3STDC)
`strncpy`(3STDC)	See `string`(3STDC)
`strpbrk`(3STDC)	See `string`(3STDC)
`strrchr`(3STDC)	See `string`(3STDC)
`strsep`(3STDC)	separate strings

`strspn`(3STDC)	See `string`(3STDC)
`strstr`(3STDC)	See `string`(3STDC)
`strtod`(3STDC)	convert an ASCII string to a floating-point number
`strtok`(3STDC)	string tokens
`strtok_r`(3STDC)	string tokens reentrant
`strtol`(3STDC)	convert string to integer
`strtoul`(3STDC)	convert a string to an unsigned long or uquad_t integer
`strxfrm`(3STDC)	transform a string under locale
`swab`(3STDC)	swap adjacent bytes
`sys_errlist`(3STDC)	See `perror`(3STDC)
`sys_nerr`(3STDC)	See `perror`(3STDC)
`tempnam`(3STDC)	See `tmpnam`(3STDC)
`thread_once`(3STDC)	execute an init routine once
`time`(3STDC)	get time
`tmpfile`(3STDC)	create a temporary file
`tmpnam`(3STDC)	create a name for a temporary file
`toascii`(3STDC)	convert a byte to 7-bit ASCII
`tolower`(3STDC)	See `ctype`(3STDC)
`toupper`(3STDC)	See `ctype`(3STDC)
`tzset`(3STDC)	set time conversion information
`ungetc`(3STDC)	push character back into input stream
`unlocked`(3STDC)	explicit locking functions
`unsetenv`(3STDC)	See `getenv`(3STDC)
`vfprintf`(3STDC)	print formatted output
`vprintf`(3STDC)	print formatted output
`vsnprintf`(3STDC)	See `vprintf`(3STDC)
`vsprintf`(3STDC)	See `vprintf`(3STDC)

NAME

abort – cause abnormal program termination

SYNOPSIS

```
#include <stdlib.h>
void abort(void);
```

DESCRIPTION

The *abort* function causes abnormal program termination to occur.

No open streams are closed or flushed.

In environments where signals are supported, the signal SIGABRT is first produced. The above processing takes place if and when the signal handler returns, or if the signal is ignored (default setting of this signal).

RETURN VALUES

The *abort* function never returns.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`exit`(3STDC)

NOTES

Nothing prevents concurrent invocations of *abort*. It is up to the application to deal with the possible consequences of this type of situation.

STANDARDS

The *abort* function conforms to `ANSI-C`.

NAME

abs – integer absolute value function

SYNOPSIS

```
#include <stdlib.h>
int abs(int j);
```

DESCRIPTION

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

RETURN VALUES

The *abs* function returns the absolute value.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`labs`(3STDC)

STANDARDS

The *abs* function conforms to `ANSI-C`.

RESTRICTIONS

The absolute value of the highest negative integer remains negative.

NAME

scandir, alphasort – scan a directory

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>
int **scandir**(const char * *dirname*, struct dirent *** *namelist*, int (* *select)(struct dirent *)*, int (* *compare)(const void *, const void *)*);

int **alphasort**(const void * *d1*, const char * *d2*);

DESCRIPTION

The *scandir* function reads the directory *dirname* and builds an array of pointers to directory entries using *malloc* (3STDC). It returns the number of entries in the array. A pointer to the array of directory entries is stored in the location referenced by *namelist* .

The *select* parameter is a pointer to a user supplied subroutine which is called by *scandir* to select which entries are to be included in the array. The *select* routine is passed a pointer to a directory entry and should return a non-zero value if the directory entry is to be included in the array. If *select* is null, then all the directory entries will be included.

The *compare* parameter is a pointer to a user supplied subroutine which is passed to *qsort* (3STDC) to sort the completed array. If this pointer is null, the array is not sorted.

The *alphasort* function is a routine which can be used for the *compare* parameter to sort the array alphabetically.

The memory allocated for the array can be deallocated with *free* (3STDC), by freeing each pointer in the array and then the array itself.

DIAGNOSTICS

Returns -1 if the directory cannot be opened for reading or if *malloc* (3STDC) cannot allocate enough memory to hold all the data structures.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`directory`(3POSIX) , `malloc`(3STDC) , `qsort`(3STDC)

HISTORY

The *scandir* and *alphasort* functions appeared in 4.2BSD.

NAME

ctime, asctime, difftime, gmtime, localtime, mktime – transform binary date and time value to ASCII

SYNOPSIS

#include <time.h>
struct tm * **localtime**(const time_t * *clock*);

struct tm * **gmtime**(const time_t * *clock*);

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

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

time_t **mktime**(struct tm * *tm*);

double **difftime**(time_t *time1*, time_t *time0*);

DESCRIPTION

The *ctime* , *gmtime* and *localtime* functions take as an argument a time value representing the time in seconds since the Epoch (00:00:00 UTC, January 1, 1970).

The *localtime* function converts the time value pointed to by *clock* , and returns a pointer to a *struct tm* (described below) which contains the broken-out time information for the value, after adjusting for the current time zone (and any other factors such as Daylight Saving Time). Time zone adjustments are performed as specified by the TZ environment variable (see *tzset* (3STDC). The function *localtime* uses *tzset* (3STDC) to initialize time conversion information if *tzset* (3STDC) has not already been called by the process.

The *gmtime* function also converts the time value, but without any time zone adjustment, and returns a pointer to a *tm* structure (described below).

The *ctime* function adjusts the time value for the current time zone in the same manner as *localtime* , and returns a pointer to a 26-character string of the form: Thu Nov 24 18:22:48 1986.

The *asctime* function converts the broken—down time in the structure *tm* pointed to by **tm* to the form shown in the example above.

The *mktime* function converts the broken-down time, expressed as local time, in the structure pointed to by *tm* into a time value with the same encoding as that of the values returned by the time (3STDC) function; that is, seconds from the Epoch, UTC .

The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to their normal ranges. (A positive or zero value for *tm_isdst* causes *mktime* to presume initially that summer time (for example, Daylight Saving Time) is or is not in effect for the time specified, respectively. A negative value for *tm_isdst* causes the *mktime* function to attempt to define whether summer time is in effect for the time specified.)

On successful completion, the values of the *tm_wday* and *tm_yday* components of the structure are set appropriately, and the other components are set to represent the calendar time specified, but with their values forced to their normal ranges; the final value of *tm_mday* is not set until *tm_mon* and *tm_year* are determined. The *mktime* function returns the calendar time specified; if the calendar time cannot be represented, it returns –1;

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

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

```
int tm_sec;          /* seconds (0 - 60) */
int tm_min;          /* minutes (0 - 59) */
int tm_hour;         /* hours (0 - 23) */
int tm_mday;         /* day of month (1 - 31) */
int tm_mon;          /* month of year (0 - 11) */
int tm_year;         /* year – 1900 */
int tm_wday;         /* day of week (Sunday = 0) */
int tm_yday;         /* day of year (0 - 365) */
int tm_isdst;        /* is summer time in effect? */
char *tm_zone;       /* abbreviation of timezone name */
long tm_gmtoff;      /* offset from UTC in seconds */
```

The field *tm_isdst* is non-zero if summer time is in effect.

The field *tm_gmtoff* is the offset (in seconds) of the time represented from `UTC` , with positive values indicating east of the Prime Meridian.

NOTES

asctime(3STDC) , *ctime(3STDC)* , *localtime(3STDC)* and *gmtime(3STDC)* return their result in a global variable which make them difficult to use in a multithreaded program. *asctime_r(3STDC)* , *ctime_r(3STDC)* , *localtime_r(3STDC)* and *gmtime_r(3STDC)* should be used instead.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`asctime_r`(3STDC) , `ctime_r`(3STDC) , `getenv`(3STDC) , `gmtime_r`(3STDC) , `localtime_r`(3STDC) , `time`(3STDC) , `tzset`(3STDC)

NAME

ctime_r, asctime_r, gmtime_r, localtime_r – Transform binary date and time value to ASCII; Reentrent version

SYNOPSIS

#include <time.h>
char * **ctime_r**(const time_t * *clock*, char * *result*);

char * **asctime_r**(const struct tm * *tm*, char * *result*);

struct tm * **localtime_r**(const time_t * *clock*, struct tm * *result*);

struct tm * **gmtime_r**(const time_t * *clock*, struct tm * *result*);

DESCRIPTION

The *ctime_r, gmtime_r, asctime_r,* and *localtime_r* functions do the same thing as *ctime* (3STDC), *gmtime* (3STDC), *asctime* (3STDC), and *localtime* (3STDC), with the difference that they do not store their result in a static buffer. Instead, the necessary storage must be allocated by the caller and a pointer to it passed as the *result* argument.

For *asctime_r, result* must point to a 26 byte character array. For the others, *result* must point to a memory area large enough to hold a *struct tm*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`asctime`(3STDC) , `ctime`(3STDC) , `localtime`(3STDC) , `gmtime`(3STDC) , `tzset`(3STDC)

STANDARDS

These routines conform to POSIX.1c.

NAME

assert, _assert, _stdc_assert – expression verification macro

SYNOPSIS

#include <assert.h>
assert *expression*

_assert *expression*
void **`_stdc_assert`**(const char * *file*, int *line*, const char * *expression*);

DESCRIPTION

The `_assert(x)` macro is defined as `assert(x)`. The `assert` macro tests the given *expression* and if it is false, calls `_stdc_assert()`. The `_stdc_assert()` function writes a diagnostic message to the error channel, and calls `abort`(3STDC).

If the *expression* is true, the `assert` macro does nothing.

The `assert` macro may be rendered non-operational at compile time using the `NDEBUG` option.

DIAGNOSTICS

The following diagnostic message is written to the error channel if *expression* is false:

```
("assertion %s failed: file %s, line %d\
", expression, __FILE__, __LINE__)
```

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`abort`(3STDC)

NAME

assert, _assert, _stdc_assert – expression verification macro

SYNOPSIS

#include <assert.h>
assert *expression*

_assert *expression*
void **`_stdc_assert`**(const char * *file*, int *line*, const char * *expression*);

DESCRIPTION

The `_assert(x)` macro is defined as `assert(x)` . The `assert` macro tests the given *expression* and if it is false, calls `_stdc_assert()` . The `_stdc_assert()` function writes a diagnostic message to the error channel, and calls `abort`(3STDC) .

If the *expression* is true, the `assert` macro does nothing.

The `assert` macro may be rendered non-operational at compile time using the `NDEBUG` option.

DIAGNOSTICS

The following diagnostic message is written to the error channel if *expression* is false:

```
("assertion %s failed: file %s, line %d\
", expression, __FILE__, __LINE__)
```

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`abort`(3STDC)

NAME

atexit – add program termination routines

SYNOPSIS

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

DESCRIPTION

Calling *atexit* adds the *func* function to a list of functions to be called, without argument, on normal termination of the program. Normal termination occurs either by a call to `exit`(3STDC) or by a return from *main*.

RETURN VALUES

The *atexit* function returns 0 if the registration succeeded, non-zero if it failed.

NOTES

atexit is reentrant. The related `exit`(3STDC) processing requires special attention with regard to concurrent execution.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`exit`(3STDC), `_exit`(2K)

NAME

strtod, atof – convert an ASCII string to a floating-point number

SYNOPSIS

#include <stdlib.h>
double **strtod**(const char * *str*, char ** *ptr*);

double **atof**(const char * *str*);

DESCRIPTION

The *strtod* function returns as a double-precision floating-point number the value represented by the character string pointed to by *str* . The string is scanned up to the first unrecognized character.

The *strtod* function recognizes an optional string of white-space characters, then an optional sign, then a string of digits optionally containing a decimal point, then an optional `e` or `E` followed by an optional sign or space, followed by an integer.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr* . If a number cannot be formed, **ptr* is set to *str* , and zero is returned.

The *atof(str)* call is equivalent to *strtod(str, (char **)NULL)* .

DIAGNOSTICS

If the correct value would cause overflow, HUGE is returned (according to whether the value is positive or negative), and, in if supported, *errno* is set to ERANGE If the correct value would cause underflow, zero is returned and, if supported, *errno* is set to ERANGE.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`scanf`(3STDC)

NAME

strtol, atol, atoi – convert string to integer

SYNOPSIS

#include <stdlib.h>
long **strtol**(const char * *str*, char ** *ptr*, int *base*);

long **atol**(const char * *str*);

int **atoi**(const char * *str*);

DESCRIPTION

The *strtol* function returns the value represented by the character string pointed to by *str* as a long integer. The string is scanned up to the first character inconsistent with the base. Leading “white-space” characters (as defined by *isspace* in *ctype* (3STDC)) are ignored.

The input string is divided into three parts: an initial, possibly empty, sequence of white-space characters (as defined by *isspace* in *ctype* (3STDC)); 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. The *strtol* function attempts to convert the subject sequence to an integer and return the result.

A pointer to the final string is stored in the object pointed to by ptr, provided it is not a null pointer.

If *base* is positive, it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and “0x” or “0X” is ignored if *base* is 16.

If *base* is zero, the string itself determines the base as follows: After an optional leading sign, a leading zero indicates octal conversion, and a leading “0x” or “0X” hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can be done upon assignment, or by using an explicit cast.

atol(str) is equivalent to `strtol(str, (char **)NULL, 10)`.

atoi(str) is equivalent to `(int) strtol(str, (char **)NULL, 10)`.

RETURN VALUES

Upon successful completion *strtol* returns the converted value, if any. If no conversion could be performed, 0 is returned.

If the correct value is outside the range of representable values, LONG_MAX or LONG_MIN is returned (according to the sign of the value), and errno is set to ERANGE.

USAGE

Because LONG_MIN and LONG_MAX are returned on error and are also valid returns on success, in order to check for error situations, an application should set errno to 0, then call *strtol* , then check errno; if it is non-zero, you can assume that an error has occurred.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ctype`(3STDC) , `scanf`(3STDC) , `strtod`(3STDC)

NAME

strtol, atol, atoi – convert string to integer

SYNOPSIS

```
#include <stdlib.h>
long strtol(const char * str, char ** ptr, int base);

long atol(const char * str);

int atoi(const char * str);
```

DESCRIPTION

The *strtol* function returns the value represented by the character string pointed to by *str* as a long integer. The string is scanned up to the first character inconsistent with the base. Leading “white-space” characters (as defined by *isspace* in *ctype* (3STDC)) are ignored.

The input string is divided into three parts: an initial, possibly empty, sequence of white-space characters (as defined by *isspace* in *ctype* (3STDC)); 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. The *strtol* function attempts to convert the subject sequence to an integer and return the result.

A pointer to the final string is stored in the object pointed to by ptr, provided it is not a null pointer.

If *base* is positive, it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and “0x” or “0X” is ignored if *base* is 16.

If *base* is zero, the string itself determines the base as follows: After an optional leading sign, a leading zero indicates octal conversion, and a leading “0x” or “0X” hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can be done upon assignment, or by using an explicit cast.

atol(str) is equivalent to `strtol(str, (char **)NULL, 10)`.

atoi(str) is equivalent to `(int) strtol(str, (char **)NULL, 10)`.

RETURN VALUES

Upon successful completion *strtol* returns the converted value, if any. If no conversion could be performed, 0 is returned.

If the correct value is outside the range of representable values, LONG_MAX or LONG_MIN is returned (according to the sign of the value), and errno is set to ERANGE.

USAGE

Because LONG_MIN and LONG_MAX are returned on error and are also valid returns on success, in order to check for error situations, an application should set errno to 0, then call *strtol* , then check errno; if it is non-zero, you can assume that an error has occurred.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ctype`(3STDC) , `scanf`(3STDC) , `strtod`(3STDC)

NAME

bstring, bcopy, bcmp, bzero, ffs – bit and byte string operations

SYNOPSIS

#include <string.h>
void **bcopy**(const void * *b1*, void * *b2*, size_t *length*);

int **bcmp**(const void * *b1*, const void * *b2*, size_t *length*);

void **bzero**(void * *b*, size_t *length*);

int **ffs**(int *value*);

DESCRIPTION

The *bcopy* , *bcmp* , and *bzero* functions operate on variable length strings of bytes. They do not check for null bytes as the routines in string(3STDC) do.

The *bcopy* function copies *length* bytes from string *b1* to the string *b2* . Overlapping strings are handled correctly.

The *bcmp* function compares byte string *b1* to byte string *b2* , returning 0 if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long. A *bcmp* zero bytes long always returns 0.

The *bzero* function places a *length* of 0 bytes in the string *b* .

The *ffs* function finds the first bit set in *value* and returns the index of that bit. Bits are numbered starting at 1 from the right. A return of zero indicates that the value passed is zero.

NOTES

The *bcmp* and *bcopy* routines take parameters backwards from *strcmp* and *strcpy* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`string`(3STDC)

NAME

bstring, bcopy, bcmp, bzero, ffs – bit and byte string operations

SYNOPSIS

#include <string.h>
void **bcopy**(const void * *b1*, void * *b2*, size_t *length*);

int **bcmp**(const void * *b1*, const void * *b2*, size_t *length*);

void **bzero**(void * *b*, size_t *length*);

int **ffs**(int *value*);

DESCRIPTION

The *bcopy* , *bcmp* , and *bzero* functions operate on variable length strings of bytes. They do not check for null bytes as the routines in string(3STDC) do.

The *bcopy* function copies *length* bytes from string *b1* to the string *b2* . Overlapping strings are handled correctly.

The *bcmp* function compares byte string *b1* to byte string *b2* , returning 0 if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long. A *bcmp* zero bytes long always returns 0.

The *bzero* function places a *length* of 0 bytes in the string *b* .

The *ffs* function finds the first bit set in *value* and returns the index of that bit. Bits are numbered starting at 1 from the right. A return of zero indicates that the value passed is zero.

NOTES

The *bcmp* and *bcopy* routines take parameters backwards from *strcmp* and *strcpy* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`string`(3STDC)

NAME

bsearch – perform a binary search on a sorted table

SYNOPSIS

#include <stdlib.h>

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

DESCRIPTION

The *bsearch* function is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where an item of data may be found, or a null pointer if the item of data cannot be found. The table must be previously sorted in ascending order according to the comparison function indicated by *compar*. The *key* value points to the item of data to search for. The *base* pointer indicates the element at the base of the table, *nel* is the number of elements in the table, and `size` is the number of bytes in each element. The function pointed to by *compar* 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, depending on whether the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLES

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

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define TABSIZE 1000

struct node {   /* these are stored in the table */
 char *string;
 int length;
};
struct node table[TABSIZE]; /* table to be searched */
 .
 .
 .
{
 struct node *node_ptr, node;
 /* routine to compare 2 nodes */
 int node_compare(const void*, const void*);
 char str_space[20];   /* space to read string into */
 .
 .
 .
 node.string = str_space;
 while (scanf("%s", node.string) != EOF) {
  node_ptr = (struct node *)bsearch(&node,
      table, TABSIZE,
      sizeof(struct node), node_compare);
```

```
    if (node_ptr != NULL) {
     (void)printf("string = %20s, length = %d\n",
      node_ptr->string, node_ptr->length);
    } else {
     (void)printf("not found: %s\n", node.string);
    }
   }
  }
  /*
   This routine compares two nodes based on an
   alphabetical ordering of the string field.
  */
  int
  node_compare(const void* node1, const void* node2)
  {
   return strcmp(
    ((const struct node *)node1)->string,
    ((const struct note *)node2)->string);
  }
```

NOTES

The pointers to the key and the element at the base of the table should be of the 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.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`qsort`(3STDC)

DIAGNOSTICS

A NULL pointer is returned if the key cannot be found in the table.

NAME

bstring, bcopy, bcmp, bzero, ffs – bit and byte string operations

SYNOPSIS

#include <string.h>
void **bcopy**(const void * *b1*, void * *b2*, size_t *length*);

int **bcmp**(const void * *b1*, const void * *b2*, size_t *length*);

void **bzero**(void * *b*, size_t *length*);

int **ffs**(int *value*);

DESCRIPTION

The *bcopy* , *bcmp* , and *bzero* functions operate on variable length strings of bytes. They do not check for null bytes as the routines in string(3STDC) do.

The *bcopy* function copies *length* bytes from string *b1* to the string *b2* . Overlapping strings are handled correctly.

The *bcmp* function compares byte string *b1* to byte string *b2* , returning 0 if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long. A *bcmp* zero bytes long always returns 0.

The *bzero* function places a *length* of 0 bytes in the string *b* .

The *ffs* function finds the first bit set in *value* and returns the index of that bit. Bits are numbered starting at 1 from the right. A return of zero indicates that the value passed is zero.

NOTES

The *bcmp* and *bcopy* routines take parameters backwards from *strcmp* and *strcpy* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`string`(3STDC)

NAME

byteorder, htonl, htons, ntohl, ntohs – convert values between host and network byte order

SYNOPSIS

#include <sys/param.h>
unsigned long **`htonl`**(unsigned long *hostlong*);

unsigned short **`htons`**(unsigned short *hostshort*);

unsigned long **`ntohl`**(unsigned long *netlong*);

unsigned short **`ntohs`**(unsigned short *netshort*);

DESCRIPTION

These routines convert 16– and 32–bit quantities between network byte order and host byte order. On architectures where the host byte order and network byte order are the same, these routines are defined as no-op macros.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

bstring, bcopy, bcmp, bzero, ffs – bit and byte string operations

SYNOPSIS

#include <string.h>
void **bcopy**(const void * *b1*, void * *b2*, size_t *length*);

int **bcmp**(const void * *b1*, const void * *b2*, size_t *length*);

void **bzero**(void * *b*, size_t *length*);

int **ffs**(int *value*);

DESCRIPTION

The *bcopy* , *bcmp* , and *bzero* functions operate on variable length strings of bytes. They do not check for null bytes as the routines in string(3STDC) do.

The *bcopy* function copies *length* bytes from string *b1* to the string *b2* . Overlapping strings are handled correctly.

The *bcmp* function compares byte string *b1* to byte string *b2* , returning 0 if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long. A *bcmp* zero bytes long always returns 0.

The *bzero* function places a *length* of 0 bytes in the string *b* .

The *ffs* function finds the first bit set in *value* and returns the index of that bit. Bits are numbered starting at 1 from the right. A return of zero indicates that the value passed is zero.

NOTES

The *bcmp* and *bcopy* routines take parameters backwards from *strcmp* and *strcpy* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`string`(3STDC)

NAME

malloc, free, realloc, calloc – main memory allocator

SYNOPSIS

```
#include <stdlib.h>
void * malloc(size_t size);

void free(void * ptr);

void * realloc(void * ptr, size_t size);

void *calloc(size_t nelem, size_t elsize);
```

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. ChorusOS 4.0 offers three `malloc( )` libraries. See *EXTENDED DESCRIPTION* below for details.

The argument passed to `free( )` is a pointer to a block previously allocated by `malloc( )` ; after `free( )` is performed this space is made available for further allocation, but its contents are left undisturbed.

The `free( )` function may be called with a `NULL` pointer as parameter.

If the space assigned by `malloc( )` is overrun or if a random number is passed to `free( )` , the result is undefined.

The `malloc( )` function searches for free space from the last block allocated or freed, grouping together any adjacent free blocks. It allocates the first contiguous area of free space that is at least `size( )` bytes.

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 smaller of the new and old sizes. If no free block of *size* bytes is available in the storage area, `realloc( )` will ask `malloc( )` to enlarge the area by *size* bytes and will then move the data to the new space. If the space cannot be allocated, the object pointed to by *ptr* is unchanged. If *size* is zero and *ptr* is not a null pointer, the object it points to is freed. If *ptr* is a null pointer, the `realloc( )` function behaves like the `malloc( )` function for the specified size.

The `realloc( )` function also works if *ptr* points to a block freed since the last call to `malloc( )` , `realloc( )` , or `calloc( )` ; thus sequences of `free( )` , `malloc( )` and `realloc( )` can be used to exploit the search strategy of `malloc( )` in order to do storage compacting.

The `calloc( )` function allocates space for an array of *nelem* elements of size *elsize* . The space is initialized to zeros.

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 no memory available, or if the area has been detectably corrupted by storing outside the bounds of a block. When this happens, the block indicated by *ptr* is neither damaged nor freed.

EXTENDED DESCRIPTION

ChorusOS 4.0 offers three `malloc()` libraries. The following list describes each library:

`lib/classix/libcx.a`
: The standard `malloc()` for ChorusOS 4.0, based on the standard Solaris™ `libc` implementation, which has been extended to release freed memory to the system for use by the kernel and by other actors. However, calling `free()` does not automatically return memory to the system. `malloc()` takes memory chunks from page-aligned regions. Regions are only returned to the system once all the chunks in the region have been freed. Furthermore, `free()` buffers memory chunks so that they can be reused immediately by `malloc()` if possible. Therefore, memory may not be returned to the system until `malloc()` is called again. `malloc_trim()` can be used to release empty regions to the system explicitly.

  `alloca()`, `calloc()`, `memalign()` and `valloc()` are not available in `lib/classix/libcx.a`.

`lib/classix/libleamalloc.a`
: Doug Lea's `malloc()`, also known as the `libg++` `malloc()` implementation, adapted for ChorusOS 4.0 to allow the heap to be sparsed in several regions. This implementation is especially useful in supervisor mode, because supervisor space is shared by several actors. Freed memory may be returned to the system using `malloc_trim()`. `free()` may also call `malloc_trim()` if enough memory is free at the top of the heap.

`lib/classix/libomalloc.a`
: The BSD `malloc()` is provided for backwards compatibility with previous releases. This implementation corresponds to `bsdmalloc`(3X) in 2.6. See Solaris *man Pages(3): Library Routines* in the *Solaris 2.6 Reference Manual AnswerBook* for details.

NOTES

Performance and efficiency depend upon the way the library is used. Search time increases when many objects have been allocated; that is, if a program allocates but never frees, each successive allocation takes longer. Tests on the running program should be performed in order to determine the best balance between performance and efficient use of space to achieve optimum performance.

If the program is multi-threaded, and if the `free()` and then `realloc()` feature is used, it is up to the programmer to set up the mutual exclusion schemes needed to prevent a `malloc()` taking place between `free()` and `realloc()` calls.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ferror, feof, fileno, clearerr – stream status inquiries

SYNOPSIS

#include <stdio.h>
int **ferror**(FILE * *stream*);

int **feof**(FILE * *stream*);

int **fileno**(FILE * *stream*);

void **clearerr**(FILE * *stream*);

DESCRIPTION

When an I/O error has occurred when reading from or writing to the named *stream* , the *ferror* function returns a non-zero value. If no error has occurred, it returns 0.

When EOF has been detected when reading the named input *stream* , the *feof* function returns a non-zero value. If EOF was not detected, it returns 0.

The *clearerr* function resets the error and EOF indicators to zero on the named *stream* . Once set, the error and EOF indicators remain set until reset by *clearerr,* or the *stream* is closed.

The *fileno* function returns the integer file descriptor associated with the named *stream* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC)

NAME

ctime, asctime, difftime, gmtime, localtime, mktime – transform binary date and time value to ASCII

SYNOPSIS

#include <time.h>
struct tm * **localtime**(const time_t * *clock*);

struct tm * **gmtime**(const time_t * *clock*);

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

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

time_t **mktime**(struct tm * *tm*);

double **difftime**(time_t *time1*, time_t *time0*);

DESCRIPTION

The *ctime* , *gmtime* and *localtime* functions take as an argument a time value representing the time in seconds since the Epoch (00:00:00 UTC, January 1, 1970).

The *localtime* function converts the time value pointed to by *clock* , and returns a pointer to a *struct tm* (described below) which contains the broken-out time information for the value, after adjusting for the current time zone (and any other factors such as Daylight Saving Time). Time zone adjustments are performed as specified by the TZ environment variable (see *tzset* (3STDC). The function *localtime* uses *tzset* (3STDC) to initialize time conversion information if *tzset* (3STDC) has not already been called by the process.

The *gmtime* function also converts the time value, but without any time zone adjustment, and returns a pointer to a *tm* structure (described below).

The *ctime* function adjusts the time value for the current time zone in the same manner as *localtime* , and returns a pointer to a 26-character string of the form: Thu Nov 24 18:22:48 1986.

The *asctime* function converts the broken—down time in the structure *tm* pointed to by **tm* to the form shown in the example above.

The *mktime* function converts the broken-down time, expressed as local time, in the structure pointed to by *tm* into a time value with the same encoding as that of the values returned by the `time` (3STDC) function; that is, seconds from the Epoch, UTC .

The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to their normal ranges. (A positive or zero value for *tm_isdst* causes *mktime* to presume initially that summer time (for example, Daylight Saving Time) is or is not in effect for the time specified, respectively. A negative value for *tm_isdst* causes the *mktime* function to attempt to define whether summer time is in effect for the time specified.)

On successful completion, the values of the *tm_wday* and *tm_yday* components of the structure are set appropriately, and the other components are set to represent the calendar time specified, but with their values forced to their normal ranges; the final value of *tm_mday* is not set until *tm_mon* and *tm_year* are determined. The *mktime* function returns the calendar time specified; if the calendar time cannot be represented, it returns –1;

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

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

```
int tm_sec;          /* seconds (0 - 60) */
int tm_min;          /* minutes (0 - 59) */
int tm_hour;         /* hours (0 - 23) */
int tm_mday;         /* day of month (1 - 31) */
int tm_mon;          /* month of year (0 - 11) */
int tm_year;         /* year – 1900 */
int tm_wday;         /* day of week (Sunday = 0) */
int tm_yday;         /* day of year (0 - 365) */
int tm_isdst;        /* is summer time in effect? */
char *tm_zone;       /* abbreviation of timezone name */
long tm_gmtoff;      /* offset from UTC in seconds */
```

The field *tm_isdst* is non-zero if summer time is in effect.

The field *tm_gmtoff* is the offset (in seconds) of the time represented from `UTC` , with positive values indicating east of the Prime Meridian.

NOTES

asctime(3STDC) , *ctime(3STDC)* , *localtime(3STDC)* and *gmtime(3STDC)* return their result in a global variable which make them difficult to use in a multithreaded program. *asctime_r(3STDC)* , *ctime_r(3STDC)* , *localtime_r(3STDC)* and *gmtime_r(3STDC)* should be used instead.

ATTRIBUTES

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

ATTRIBUTE TYPE	**ATTRIBUTE VALUE**
Interface Stability	Evolving

SEE ALSO

`asctime_r`(3STDC) , `ctime_r`(3STDC) , `getenv`(3STDC) , `gmtime_r`(3STDC) , `localtime_r`(3STDC) , `time`(3STDC) , `tzset`(3STDC)

NAME

ctime_r, asctime_r, gmtime_r, localtime_r – Transform binary date and time value to ASCII; Reentrent version

SYNOPSIS

#include <time.h>
char * **ctime_r**(const time_t * *clock*, char * *result*);

char * **asctime_r**(const struct tm * *tm*, char * *result*);

struct tm * **localtime_r**(const time_t * *clock*, struct tm * *result*);

struct tm * **gmtime_r**(const time_t * *clock*, struct tm * *result*);

DESCRIPTION

The *ctime_r, gmtime_r, asctime_r,* and *localtime_r* functions do the same thing as *ctime* (3STDC), *gmtime* (3STDC), *asctime* (3STDC), and *localtime* (3STDC), with the difference that they do not store their result in a static buffer. Instead, the necessary storage must be allocated by the caller and a pointer to it passed as the *result* argument.

For *asctime_r, result* must point to a 26 byte character array. For the others, *result* must point to a memory area large enough to hold a *struct tm*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`asctime`(3STDC) , `ctime`(3STDC) , `localtime`(3STDC) , `gmtime`(3STDC) , `tzset`(3STDC)

STANDARDS

These routines conform to POSIX.1c.

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

```
#include <ctype.h>
int isalpha(int c);
```

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ctime, asctime, difftime, gmtime, localtime, mktime – transform binary date and time value to ASCII

SYNOPSIS

#include <time.h>
struct tm * **`localtime`**(const time_t * *clock*);

struct tm * **`gmtime`**(const time_t * *clock*);

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

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

time_t **`mktime`**(struct tm * *tm*);

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

DESCRIPTION

The *ctime* , *gmtime* and *localtime* functions take as an argument a time value representing the time in seconds since the Epoch (00:00:00 `UTC,` January 1, 1970).

The *localtime* function converts the time value pointed to by *clock* , and returns a pointer to a *struct tm* (described below) which contains the broken-out time information for the value, after adjusting for the current time zone (and any other factors such as Daylight Saving Time). Time zone adjustments are performed as specified by the `TZ` environment variable (see *tzset* (3STDC). The function *localtime* uses *tzset* (3STDC) to initialize time conversion information if *tzset* (3STDC) has not already been called by the process.

The *gmtime* function also converts the time value, but without any time zone adjustment, and returns a pointer to a *tm* structure (described below).

The *ctime* function adjusts the time value for the current time zone in the same manner as *localtime* , and returns a pointer to a 26-character string of the form: Thu Nov 24 18:22:48 1986.

The *asctime* function converts the broken—down time in the structure *tm* pointed to by **tm* to the form shown in the example above.

The *mktime* function converts the broken-down time, expressed as local time, in the structure pointed to by *tm* into a time value with the same encoding as that of the values returned by the `time` (3STDC) function; that is, seconds from the Epoch, `UTC` .

The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to their normal ranges. (A positive or zero value for *tm_isdst* causes *mktime* to presume initially that summer time (for example, Daylight Saving Time) is or is not in effect for the time specified, respectively. A negative value for *tm_isdst* causes the *mktime* function to attempt to define whether summer time is in effect for the time specified.)

On successful completion, the values of the *tm_wday* and *tm_yday* components of the structure are set appropriately, and the other components are set to represent the calendar time specified, but with their values forced to their normal ranges; the final value of *tm_mday* is not set until *tm_mon* and *tm_year* are determined. The *mktime* function returns the calendar time specified; if the calendar time cannot be represented, it returns –1;

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

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

```
int tm_sec;          /* seconds (0 - 60) */
int tm_min;          /* minutes (0 - 59) */
int tm_hour;         /* hours (0 - 23) */
int tm_mday;         /* day of month (1 - 31) */
int tm_mon;          /* month of year (0 - 11) */
int tm_year;         /* year – 1900 */
int tm_wday;         /* day of week (Sunday = 0) */
int tm_yday;         /* day of year (0 - 365) */
int tm_isdst;        /* is summer time in effect? */
char *tm_zone;       /* abbreviation of timezone name */
long tm_gmtoff;      /* offset from UTC in seconds */
```

The field *tm_isdst* is non-zero if summer time is in effect.

The field *tm_gmtoff* is the offset (in seconds) of the time represented from `UTC` , with positive values indicating east of the Prime Meridian.

NOTES

asctime(3STDC) , *ctime(3STDC)* , *localtime(3STDC)* and *gmtime(3STDC)* return their result in a global variable which make them difficult to use in a multithreaded program. *asctime_r(3STDC)* , *ctime_r(3STDC)* , *localtime_r(3STDC)* and *gmtime_r(3STDC)* should be used instead.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`asctime_r`(3STDC) , `ctime_r`(3STDC) , `getenv`(3STDC) , `gmtime_r`(3STDC) , `localtime_r`(3STDC) , `time`(3STDC) , `tzset`(3STDC)

NAME

div – return quotient and remainder from division

SYNOPSIS

```
#include <stdlib.h>
div_t div(int num, int denom);
```

DESCRIPTION

The *div* function computes the value *num/denom* and returns the quotient and remainder in a structure named *div_t* that contains two *int* members named *quot* and *rem*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ldiv`(3STDC)

STANDARDS

The *div* function conforms to `ANSI-C`.

NAME

perror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

```
#include <stdio.h>
void perror(const char * s);
#include <errno.h>
extern char *sys_errlist[];
extern int sys_nerr;
```

DESCRIPTION

The *perror* function produces a message on the error channel, the implementation of which is system-dependent. The message describes the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a newline character. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the per thread variable *errno*, or from a global variable *errno*, whichever is provided by the library. This variable is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new line. The *sys_nerr* parameter defines the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

exit – terminate an actor

SYNOPSIS

#include <stdlib.h>
void **exit**(int *status*);

DESCRIPTION

The *exit* function terminates the calling actor with the following consequences:

- Any functions registered using the *atexit*(3STDC) function are called in the reverse order of their registration.
- Cleanup actions are performed.
- The actor is terminated by calling *_exit*(2K).

To circumvent these actions, call *_exit*(2K) directly.

NOTES

The processing of *atexit*(3STDC) functions is protected against concurrent execution. As a result, if several threads are performing `exit` at the same time, all but one will be blocked before the *atexit*(3STDC) processing is performed. The fact that one particular thread is performing `exit` does not prevent other threads from running. It is up to the application programmer to manage any conflicts resulting from this.

The *exit* function never returns. However, if the thread performing an *atexit*(3STDC) processing longjmps back into the application, further calls to `exit` will block the caller for ever; in such a case, the only valid way to terminate the application is to call *_exit(2K)*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`atexit`(3STDC), `_exit`(2K), `intro`(3STDC)

NAME

fabs – floating-point absolute value function

SYNOPSIS

```
#include <math.h>
double fabs(double x);
```

DESCRIPTION

The *fabs* function computes the absolute value of a floating-point number *x*.

RETURN VALUES

The *fabs* function returns the absolute value of *x*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`abs`(3STDC),

STANDARDS

The *fabs* function conforms to `ANSI-C`.

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

```
#include <stdio.h>
int fclose(FILE * stream);

int fflush(FILE * stream);
```

DESCRIPTION

The *fclose* function causes any buffered data for the named *stream* to be written out, and the *stream* to be closed.

When `exit` (3STDC) is called, *fclose* is called automatically for all open files.

The *fflush* function causes any buffered data for the named *stream* to be written to file. The *stream* remains open. If *stream* is NULL *fflush* flushes all open *streams* .

RETURN VALUES

These functions return 0 for success, and EOF if an error is detected (for example, when trying to write to a file that is not open for writing).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC) , `setbuf`(3STDC)

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

#include <stdio.h>
FILE * **fopen**(const char * *filename*, const char * *type*);

FILE * **freopen**(const char * *filename*, const char * *type*, FILE * *stream*);

FILE * **fdopen**(int *fildes*, const char * *type*);

DESCRIPTION

The *fopen* function opens the file named by *filename* and associates a *stream* with it. It returns a pointer to the FILE structure associated with the *stream* .

The *filename* pointer indicates a character string that contains the name of the file to be opened.

The *type* is a character string with 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 end-of-file

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 original stream is closed regardless of whether the subsequent open succeeds. The *freopen* function returns a pointer to the FILE structure associated with stream.

The *freopen* function is typically used to attach open *streams* associated with *stdin* , *stdout* , and *stderr* to other files.

When a file is opened for update, both input and output may be performed on the resulting *stream* . However, output may not be directly followed by input without an intervening *fseek* (3STDC), *rewind* (3STDC), or *fflush* (3STDC), and input may not be directly followed by output without an intervening *fseek* (3STDC), *rewind* (3STDC), *fflush* (3STDC), or an input operation which encounters end-of-file.

When a file is opened for append (that is, when `type` is `a` or `a+`), it is impossible to overwrite information already in the file. The *fseek* (3STDC) function may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is ignored. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate actors open the same file for append, each actor

may write freely to the file without fear of destroying output being written by the other. The output from the two actors will be inserted into the file in the order in which it is written.

The *fdopen* function associates a stream with the existing file descriptor, *fildes*. The *mode* of the stream must be compatible with the mode of the file descriptor.

RETURN VALUES

In case of failure, these functions return a NULL pointer and set *errno* to indicate the error condition.

NOTES

The number of streams that a process can have open at one time is OPEN_MAX.

ERRORS

The *errno* value is set to EINVAL if the *mode* provided to *fopen* , *fdopen* , or *freopen* was invalid.

The *fopen* , *fdopen* and *freopen* functions may also fail and set *errno* to any of the errors specified for the routine *malloc* (3STDC).

The *fopen* function may also fail and set *errno* to any of the errors specified for the routine *open* (2POSIX).

The *fdopen* function may also fail and set *errno* to any of the errors specified for the routine *fcntl* (2POSIX).

The *freopen* function may also fail and set *errno* for any of the errors specified for the routines *open* (2POSIX), *fclose* (3STDC), and *fflush* (3STDC).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fclose`(3STDC) , `fflush`(3STDC) , `fseek`(3STDC) , `fsetpos`(3STDC) , `fgetpos`(3STDC) , `rewind`(3STDC)

NAME

ferror, feof, fileno, clearerr – stream status inquiries

SYNOPSIS

#include <stdio.h>
int **ferror**(FILE * *stream*);

int **feof**(FILE * *stream*);

int **fileno**(FILE * *stream*);

void **clearerr**(FILE * *stream*);

DESCRIPTION

When an I/O error has occurred when reading from or writing to the named *stream* , the *ferror* function returns a non-zero value. If no error has occurred, it returns 0.

When EOF has been detected when reading the named input *stream* , the *feof* function returns a non-zero value. If EOF was not detected, it returns 0.

The *clearerr* function resets the error and EOF indicators to zero on the named *stream* . Once set, the error and EOF indicators remain set until reset by *clearerr*, or the *stream* is closed.

The *fileno* function returns the integer file descriptor associated with the named *stream* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC)

NAME

ferror, feof, fileno, clearerr – stream status inquiries

SYNOPSIS

#include <stdio.h>
int **ferror**(FILE * *stream*);

int **feof**(FILE * *stream*);

int **fileno**(FILE * *stream*);

void **clearerr**(FILE * *stream*);

DESCRIPTION

When an I/O error has occurred when reading from or writing to the named *stream* , the *ferror* function returns a non-zero value. If no error has occurred, it returns 0.

When EOF has been detected when reading the named input *stream* , the *feof* function returns a non-zero value. If EOF was not detected, it returns 0.

The *clearerr* function resets the error and EOF indicators to zero on the named *stream* . Once set, the error and EOF indicators remain set until reset by *clearerr,* or the *stream* is closed.

The *fileno* function returns the integer file descriptor associated with the named *stream* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC)

NAME

fclose, fflush – close or flush a stream

SYNOPSIS

#include <stdio.h>
int **fclose**(FILE * *stream*);

int **fflush**(FILE * *stream*);

DESCRIPTION

The *fclose* function causes any buffered data for the named *stream* to be written out, and the *stream* to be closed.

When `exit` (3STDC) is called, *fclose* is called automatically for all open files.

The *fflush* function causes any buffered data for the named *stream* to be written to file. The *stream* remains open. If *stream* is NULL *fflush* flushes all open *streams* .

RETURN VALUES

These functions return 0 for success, and EOF if an error is detected (for example, when trying to write to a file that is not open for writing).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC) , `setbuf`(3STDC)

NAME

bstring, bcopy, bcmp, bzero, ffs – bit and byte string operations

SYNOPSIS

#include <string.h>
void **bcopy**(const void * *b1*, void * *b2*, size_t *length*);

int **bcmp**(const void * *b1*, const void * *b2*, size_t *length*);

void **bzero**(void * *b*, size_t *length*);

int **ffs**(int *value*);

DESCRIPTION

The *bcopy* , *bcmp* , and *bzero* functions operate on variable length strings of bytes. They do not check for null bytes as the routines in string(3STDC) do.

The *bcopy* function copies *length* bytes from string *b1* to the string *b2* . Overlapping strings are handled correctly.

The *bcmp* function compares byte string *b1* to byte string *b2* , returning 0 if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long. A *bcmp* zero bytes long always returns 0.

The *bzero* function places a *length* of 0 bytes in the string *b* .

The *ffs* function finds the first bit set in *value* and returns the index of that bit. Bits are numbered starting at 1 from the right. A return of zero indicates that the value passed is zero.

NOTES

The *bcmp* and *bcopy* routines take parameters backwards from *strcmp* and *strcpy* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`string`(3STDC)

NAME

getc, fgetc, getw – get character from a stream

SYNOPSIS

#include <stdio.h>
int **getc**(FILE * *stream*);

int **fgetc**(FILE * *stream*);

int **getw**(FILE * *stream*);

DESCRIPTION

The *getc* and *fgetc* functions return the next character (byte) from the input *stream* specified, as an integer. The *getw* function obtains the next *int* (if present) from the stream pointed to by *stream*. These functions move the file pointer, if one is defined, ahead one character in *stream* .

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 *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 *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.

RETURN VALUES

These functions return the constant EOF at end-of-file or upon detecting an error.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fclose`(3STDC) , `ferror`(3STDC) , `fopen`(3STDC) , `fread`(3STDC) , `getchar`(3STDC) , `puts`(3STDC) , `scanf`(3STDC) , `setbuf`(3STDC)

NAME

fseek, rewind, ftell, fgetpos, fsetpos – reposition a file pointer in a stream

SYNOPSIS

#include <stdio.h>
int **fseek**(FILE * *stream*, long *offset*, int *ptrname*);

void **rewind**(FILE * *stream*);

long **ftell**(const FILE * *stream*);

int **fgetpos**(const FILE * *stream*, fpos_t * *pos*);

int **fsetpos**(FILE * *stream*, const fpos_t * *pos*);

DESCRIPTION

The *fseek* function sets the position of the next input or output operation on the *stream* . The new position, measured in bytes from the beginning of the file, is obtained by adding offset to the position specified by *ptrname* , 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

The *rewind* (*stream*) function is equivalent to *fseek* (*stream* , 0L, 0), except that no value is returned.

The *fseek* and *rewind* functions undo any effects of *ungetc* (3STDC).

After performing *fseek* or *rewind* , the next operation on a file opened for update may be either input or output.

The *ftell* function returns the offset of the current byte relative to the beginning of the file associated with the *stream* specified.

The *fgetpos* and *fsetpos* functions are alternate interfaces equivalent to *ftell* and *fseek* (with *ptrname* set to SEEK_SET), setting and storing the current value of the file offset into or from the object referenced by *pos* . On some systems an *fpos_t* object may be a complex object, and these routines may be the only way to reposition a text stream portably. This is not the case on UNIX systems.

RETURN VALUES

The *fseek* function returns 0 on success; otherwise (for example, an *fseek* done on a file that was not opened using *fopen* (3STDC)), it returns -1 and sets *errno* to indicate the error.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `fopen`(3STDC) , `ungetc`(3STDC)

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 characters from the standard input stream, *stdin,* into the array pointed to by *s* , until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

The *fgets* function reads characters from *stream* into the array pointed to by *s* , until *n* –1 characters are read, or a new-line character is read and transferred to *s* , or an end-of-file condition is encountered. The string is then terminated with a null character.

RETURN VALUES

If end-of-file is reached and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs (for eample, if you are using these functions on a file that has not been opened for reading) , a NULL pointer is returned. Otherwise *s* is returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ferror`(3STDC) , `fopen`(3STDC) , `fread`(3STDC) , `getc`(3STDC) , `scanf`(3STDC)

NAME

ferror, feof, fileno, clearerr – stream status inquiries

SYNOPSIS

#include <stdio.h>
int **ferror**(FILE * *stream*);

int **feof**(FILE * *stream*);

int **fileno**(FILE * *stream*);

void **clearerr**(FILE * *stream*);

DESCRIPTION

When an I/O error has occurred when reading from or writing to the named *stream* , the *ferror* function returns a non-zero value. If no error has occurred, it returns 0.

When EOF has been detected when reading the named input *stream* , the *feof* function returns a non-zero value. If EOF was not detected, it returns 0.

The *clearerr* function resets the error and EOF indicators to zero on the named *stream* . Once set, the error and EOF indicators remain set until reset by *clearerr,* or the *stream* is closed.

The *fileno* function returns the integer file descriptor associated with the named *stream* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC)

NAME

flockfile, ftrylockfile, funlockfile – stream lock management

SYNOPSIS

#include <stdio.h>
void **flockfile**(FILE * *file*);

int **ftrylockfile**(FILE * *file*);

void **funlockfile**(FILE * *file*);

DESCRIPTION

The *flockfile* , *ftrylockfile* and *funlockfile* functions provide for explicit application-level locking of stdio (FILE *) objects.

The *flockfile* function is used to acquire ownership of a (FILE *) object.

The *ftrylockfile* function is used to acquire ownership of a (FILE *) object if the object is available; *ftrylockfile* is a non-blocking version of *flockfile* .

The *funlockfile* function is used to relinquish the ownership of a (FILE *) object.

RETURN VALUES

The *ftrylockfile* function returns 0 on success or 1 to indicate that the lock cannot be acquired.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getc_unlocked`(3STDC)

STANDARDS

These routines conform to the POSIX.1c standards.

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

#include <stdio.h>
FILE * **fopen**(const char * *filename*, const char * *type*);

FILE * **freopen**(const char * *filename*, const char * *type*, FILE * *stream*);

FILE * **fdopen**(int *fildes*, const char * *type*);

DESCRIPTION

The *fopen* function opens the file named by *filename* and associates a *stream* with it. It returns a pointer to the FILE structure associated with the *stream* .

The *filename* pointer indicates a character string that contains the name of the file to be opened.

The *type* is a character string with 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 end-of-file

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 original stream is closed regardless of whether the subsequent open succeeds. The *freopen* function returns a pointer to the FILE structure associated with stream.

The *freopen* function is typically used to attach open *streams* associated with *stdin* , *stdout* , and *stderr* to other files.

When a file is opened for update, both input and output may be performed on the resulting *stream* . However, output may not be directly followed by input without an intervening *fseek* (3STDC), *rewind* (3STDC), or *fflush* (3STDC), and input may not be directly followed by output without an intervening *fseek* (3STDC), *rewind* (3STDC), *fflush* (3STDC), or an input operation which encounters end-of-file.

When a file is opened for append (that is, when `type` is `a` or `a+`), it is impossible to overwrite information already in the file. The *fseek* (3STDC) function may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is ignored. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate actors open the same file for append, each actor

may write freely to the file without fear of destroying output being written by the other. The output from the two actors will be inserted into the file in the order in which it is written.

The *fdopen* function associates a stream with the existing file descriptor, *fildes*. The *mode* of the stream must be compatible with the mode of the file descriptor.

RETURN VALUES

In case of failure, these functions return a NULL pointer and set *errno* to indicate the error condition.

NOTES

The number of streams that a process can have open at one time is OPEN_MAX.

ERRORS

The *errno* value is set to EINVAL if the *mode* provided to *fopen* , *fdopen* , or *freopen* was invalid.

The *fopen* , *fdopen* and *freopen* functions may also fail and set *errno* to any of the errors specified for the routine *malloc* (3STDC).

The *fopen* function may also fail and set *errno* to any of the errors specified for the routine *open* (2POSIX).

The *fdopen* function may also fail and set *errno* to any of the errors specified for the routine *fcntl* (2POSIX).

The *freopen* function may also fail and set *errno* for any of the errors specified for the routines *open* (2POSIX), *fclose* (3STDC), and *fflush* (3STDC).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fclose`(3STDC) , `fflush`(3STDC) , `fseek`(3STDC) , `fsetpos`(3STDC) , `fgetpos`(3STDC) , `rewind`(3STDC)

NAME

fprintf – print formatted output

SYNOPSIS

#include <stdio.h>
int **fprintf**(FILE **stream*, const char **format*, ... /* args */);

DESCRIPTION

The *fprintf* function places output on the output *stream* specified. The function returns the number of characters transmitted or a negative value if an output error was encountered.

This functions converts, formats, and prints its *args* in the same way as the `printf`(3STDC) function does. Characters generated by *fprintf* are printed as if *putc*(3STDC) had been called.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`putc`(3STDC), `scanf`(3STDC)

NAME

putc, fputc, putw – put character or word on a stream

SYNOPSIS

#include <stdio.h>
int **putc**(int *c*, FILE * *stream*);

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

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

DESCRIPTION

The *putc* and *fputc* functions writes the byte specified by *c* (converted to an unsigned char) to the output stream (at the position where the file pointer, if defined, is pointing).

The *putw* function writes the specified int to the defined output stream.

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.

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* (3STDC)) will change 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 it is written. When it is buffered, a number characters are saved 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* (3STDC) or *setvbuf* (3STDC) function may be used to change the stream's buffering strategy.

RETURN VALUES

Upon successful completion, these functions each return the value they have written. If unsuccessful, they return the constant EOF. This will occur if the file *stream* is not open for writing or if the output file cannot be extended.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fclose`(3STDC) , `ferror`(3STDC) , `fopen`(3STDC) , `fread`(3STDC) , `printf`(3STDC) , `putchar`(3STDC) , `puts`(3STDC) , `setbuf`(3STDC)

NAME

puts, fputs – put a string on a stream

SYNOPSIS

#include <stdio.h>
int **puts**(const char * *s*);

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

DESCRIPTION

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

The *fputs* function writes the null-terminated string pointed to by *s* to the named output *stream* .

Neither function writes the terminating null character.

RETURN VALUES

Both routines return EOF on error. This will happen if the routines try to write to a file that has not been opened for writing.

NOTES

The *puts* appends a new-line character while *fputs* does not.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ferror`(3STDC) , `fopen`(3STDC) , `fread`(3STDC) , `printf`(3STDC) , `putc`(3STDC)

NAME

fread, fwrite – binary input/output

SYNOPSIS

```
#include <stdio.h>
int fread(void * ptr, size_t size, size_t nitems, FILE * stream);

int fwrite(const void * ptr, size_t size, size_t nitems, FILE * stream);
```

DESCRIPTION

The *fread* function copies, into an array pointed to by *ptr* , *nitems* items of data from the named input *stream* , where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of the length `size` . The *fread* function stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or when *nitems* items have been read. The *fread* function leaves the file pointer in *stream* , if defined, pointing to the byte following the last byte read. It does not change the contents of *stream* .

The *fwrite* function appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream* . The *fwrite* function stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream* . It does not change the contents of the array pointed to by *ptr* .

The `size` argument is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr* .

RETURN VALUES

The *fread* and *fwrite* functions return the number of items read or written. If *nitems* is negative, no characters are read or written and 0 is returned by both *fread* and *fwrite* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC) , `getc`(3STDC) , `gets`(3STDC) , `putc`(3STDC) , `puts`(3STDC)

NAME

malloc, free, realloc, calloc – main memory allocator

SYNOPSIS

#include <stdlib.h>
void * **malloc**(size_t *size*);

void **free**(void * *ptr*);

void * **realloc**(void * *ptr*, size_t *size*);

void ***calloc**(size_t *nelem*, size_t *elsize*);

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. ChorusOS 4.0 offers three `malloc( )` libraries. See *EXTENDED DESCRIPTION* below for details.

The argument passed to `free( )` is a pointer to a block previously allocated by `malloc( )` ; after `free( )` is performed this space is made available for further allocation, but its contents are left undisturbed.

The `free( )` function may be called with a `NULL` pointer as parameter.

If the space assigned by `malloc( )` is overrun or if a random number is passed to `free( )` , the result is undefined.

The `malloc( )` function searches for free space from the last block allocated or freed, grouping together any adjacent free blocks. It allocates the first contiguous area of free space that is at least `size( )` bytes.

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 smaller of the new and old sizes. If no free block of *size* bytes is available in the storage area, `realloc( )` will ask `malloc( )` to enlarge the area by *size* bytes and will then move the data to the new space. If the space cannot be allocated, the object pointed to by *ptr* is unchanged. If *size* is zero and *ptr* is not a null pointer, the object it points to is freed. If *ptr* is a null pointer, the `realloc( )` function behaves like the `malloc( )` function for the specified size.

The `realloc( )` function also works if *ptr* points to a block freed since the last call to `malloc( )` , `realloc( )` , or `calloc( )` ; thus sequences of `free( )` , `malloc( )` and `realloc( )` can be used to exploit the search strategy of `malloc( )` in order to do storage compacting.

The `calloc( )` function allocates space for an array of *nelem* elements of size *elsize* . The space is initialized to zeros.

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 no memory available, or if the area has been detectably corrupted by storing outside the bounds of a block. When this happens, the block indicated by *ptr* is neither damaged nor freed.

EXTENDED DESCRIPTION

ChorusOS 4.0 offers three `malloc()` libraries. The following list describes each library:

`lib/classix/libcx.a`
: The standard `malloc()` for ChorusOS 4.0, based on the standard Solaris™ `libc` implementation, which has been extended to release freed memory to the system for use by the kernel and by other actors. However, calling `free()` does not automatically return memory to the system. `malloc()` takes memory chunks from page-aligned regions. Regions are only returned to the system once all the chunks in the region have been freed. Furthermore, `free()` buffers memory chunks so that they can be reused immediately by `malloc()` if possible. Therefore, memory may not be returned to the system until `malloc()` is called again. `malloc_trim()` can be used to release empty regions to the system explicitly.

  `alloca()`, `calloc()`, `memalign()` and `valloc()` are not available in `lib/classix/libcx.a`.

`lib/classix/libleamalloc.a`
: Doug Lea's `malloc()`, also known as the `libg++` `malloc()` implementation, adapted for ChorusOS 4.0 to allow the heap to be sparsed in several regions. This implementation is especially useful in supervisor mode, because supervisor space is shared by several actors. Freed memory may be returned to the system using `malloc_trim()`. `free()` may also call `malloc_trim()` if enough memory is free at the top of the heap.

`lib/classix/libomalloc.a`
: The BSD `malloc()` is provided for backwards compatibility with previous releases. This implementation corresponds to `bsdmalloc`(3X) in 2.6. See Solaris *man Pages(3): Library Routines* in the *Solaris 2.6 Reference Manual AnswerBook* for details.

NOTES

Performance and efficiency depend upon the way the library is used. Search time increases when many objects have been allocated; that is, if a program allocates but never frees, each successive allocation takes longer. Tests on the running program should be performed in order to determine the best balance between performance and efficient use of space to achieve optimum performance.

If the program is multi-threaded, and if the `free()` and then `realloc()` feature is used, it is up to the programmer to set up the mutual exclusion schemes needed to prevent a `malloc()` taking place between `free()` and `realloc()` calls.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

fopen, freopen, fdopen – open a stream

SYNOPSIS

#include <stdio.h>
FILE * **fopen**(const char * *filename*, const char * *type*);

FILE * **freopen**(const char * *filename*, const char * *type*, FILE * *stream*);

FILE * **fdopen**(int *fildes*, const char * *type*);

DESCRIPTION

The *fopen* function opens the file named by *filename* and associates a *stream* with it. It returns a pointer to the FILE structure associated with the *stream* .

The *filename* pointer indicates a character string that contains the name of the file to be opened.

The *type* is a character string with 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 end-of-file

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 original stream is closed regardless of whether the subsequent open succeeds. The *freopen* function returns a pointer to the FILE structure associated with stream.

The *freopen* function is typically used to attach open *streams* associated with *stdin* , *stdout* , and *stderr* to other files.

When a file is opened for update, both input and output may be performed on the resulting *stream* . However, output may not be directly followed by input without an intervening *fseek* (3STDC), *rewind* (3STDC), or *fflush* (3STDC), and input may not be directly followed by output without an intervening *fseek* (3STDC), *rewind* (3STDC), *fflush* (3STDC), or an input operation which encounters end-of-file.

When a file is opened for append (that is, when `type` is `a` or `a+`), it is impossible to overwrite information already in the file. The *fseek* (3STDC) function may be used to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is ignored. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output. If two separate actors open the same file for append, each actor

may write freely to the file without fear of destroying output being written by the other. The output from the two actors will be inserted into the file in the order in which it is written.

The *fdopen* function associates a stream with the existing file descriptor, *fildes*. The *mode* of the stream must be compatible with the mode of the file descriptor.

RETURN VALUES

In case of failure, these functions return a NULL pointer and set *errno* to indicate the error condition.

NOTES

The number of streams that a process can have open at one time is OPEN_MAX.

ERRORS

The *errno* value is set to EINVAL if the *mode* provided to *fopen* , *fdopen* , or *freopen* was invalid.

The *fopen* , *fdopen* and *freopen* functions may also fail and set *errno* to any of the errors specified for the routine *malloc* (3STDC).

The *fopen* function may also fail and set *errno* to any of the errors specified for the routine *open* (2POSIX).

The *fdopen* function may also fail and set *errno* to any of the errors specified for the routine *fcntl* (2POSIX).

The *freopen* function may also fail and set *errno* for any of the errors specified for the routines *open* (2POSIX), *fclose* (3STDC), and *fflush* (3STDC).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fclose`(3STDC) , `fflush`(3STDC) , `fseek`(3STDC) , `fsetpos`(3STDC) , `fgetpos`(3STDC) , `rewind`(3STDC)

NAME

fscanf – convert formatted input

SYNOPSIS

#include <stdio.h>

int **fscanf**(FILE **stream*, const char **format*, ...);

DESCRIPTION

The *fscanf* function reads from the input *stream* specified. This function reads characters and interprets them in the same way that *scanf*(3STDC) does.

NOTE

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

RETURN VALUES

This function returns EOF on end of input and a short count for missing or illegal data items.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`scanf`(3STDC)

NAME

fseek, rewind, ftell, fgetpos, fsetpos – reposition a file pointer in a stream

SYNOPSIS

#include <stdio.h>
int **fseek**(FILE * *stream*, long *offset*, int *ptrname*);

void **rewind**(FILE * *stream*);

long **ftell**(const FILE * *stream*);

int **fgetpos**(const FILE * *stream*, fpos_t * *pos*);

int **fsetpos**(FILE * *stream*, const fpos_t * *pos*);

DESCRIPTION

The *fseek* function sets the position of the next input or output operation on the *stream* . The new position, measured in bytes from the beginning of the file, is obtained by adding offset to the position specified by *ptrname* , 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

The *rewind* (*stream*) function is equivalent to *fseek* (*stream* , 0L, 0), except that no value is returned.

The *fseek* and *rewind* functions undo any effects of *ungetc* (3STDC).

After performing *fseek* or *rewind* , the next operation on a file opened for update may be either input or output.

The *ftell* function returns the offset of the current byte relative to the beginning of the file associated with the *stream* specified.

The *fgetpos* and *fsetpos* functions are alternate interfaces equivalent to *ftell* and *fseek* (with *ptrname* set to SEEK_SET), setting and storing the current value of the file offset into or from the object referenced by *pos* . On some systems an *fpos_t* object may be a complex object, and these routines may be the only way to reposition a text stream portably. This is not the case on UNIX systems.

RETURN VALUES

The *fseek* function returns 0 on success; otherwise (for example, an *fseek* done on a file that was not opened using *fopen* (3STDC)), it returns -1 and sets *errno* to indicate the error.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC) , `ungetc`(3STDC)

NAME

fseek, rewind, ftell, fgetpos, fsetpos – reposition a file pointer in a stream

SYNOPSIS

#include <stdio.h>
int **fseek**(FILE * *stream*, long *offset*, int *ptrname*);

void **rewind**(FILE * *stream*);

long **ftell**(const FILE * *stream*);

int **fgetpos**(const FILE * *stream*, fpos_t * *pos*);

int **fsetpos**(FILE * *stream*, const fpos_t * *pos*);

DESCRIPTION

The *fseek* function sets the position of the next input or output operation on the *stream* . The new position, measured in bytes from the beginning of the file, is obtained by adding offset to the position specified by *ptrname* , 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

The *rewind* (*stream*) function is equivalent to *fseek* (*stream* , 0L, 0), except that no value is returned.

The *fseek* and *rewind* functions undo any effects of *ungetc* (3STDC).

After performing *fseek* or *rewind* , the next operation on a file opened for update may be either input or output.

The *ftell* function returns the offset of the current byte relative to the beginning of the file associated with the *stream* specified.

The *fgetpos* and *fsetpos* functions are alternate interfaces equivalent to *ftell* and *fseek* (with *ptrname* set to SEEK_SET), setting and storing the current value of the file offset into or from the object referenced by *pos* . On some systems an *fpos_t* object may be a complex object, and these routines may be the only way to reposition a text stream portably. This is not the case on UNIX systems.

RETURN VALUES

The *fseek* function returns 0 on success; otherwise (for example, an *fseek* done on a file that was not opened using *fopen* (3STDC)), it returns -1 and sets *errno* to indicate the error.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC) , `ungetc`(3STDC)

NAME

fseek, rewind, ftell, fgetpos, fsetpos – reposition a file pointer in a stream

SYNOPSIS

#include <stdio.h>
int **fseek**(FILE * *stream*, long *offset*, int *ptrname*);

void **rewind**(FILE * *stream*);

long **ftell**(const FILE * *stream*);

int **fgetpos**(const FILE * *stream*, fpos_t * *pos*);

int **fsetpos**(FILE * *stream*, const fpos_t * *pos*);

DESCRIPTION

The *fseek* function sets the position of the next input or output operation on the *stream* . The new position, measured in bytes from the beginning of the file, is obtained by adding offset to the position specified by *ptrname* , 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

The *rewind* (*stream*) function is equivalent to *fseek* (*stream* , 0L, 0), except that no value is returned.

The *fseek* and *rewind* functions undo any effects of *ungetc* (3STDC).

After performing *fseek* or *rewind* , the next operation on a file opened for update may be either input or output.

The *ftell* function returns the offset of the current byte relative to the beginning of the file associated with the *stream* specified.

The *fgetpos* and *fsetpos* functions are alternate interfaces equivalent to *ftell* and *fseek* (with *ptrname* set to SEEK_SET), setting and storing the current value of the file offset into or from the object referenced by *pos* . On some systems an *fpos_t* object may be a complex object, and these routines may be the only way to reposition a text stream portably. This is not the case on UNIX systems.

RETURN VALUES

The *fseek* function returns 0 on success; otherwise (for example, an *fseek* done on a file that was not opened using *fopen* (3STDC)), it returns -1 and sets *errno* to indicate the error.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `fopen`(3STDC) , `ungetc`(3STDC)

NAME

flockfile, ftrylockfile, funlockfile – stream lock management

SYNOPSIS

```
#include <stdio.h>
void flockfile(FILE * file);

int ftrylockfile(FILE * file);

void funlockfile(FILE * file);
```

DESCRIPTION

The *flockfile* , *ftrylockfile* and *funlockfile* functions provide for explicit application-level locking of stdio (FILE *) objects.

The *flockfile* function is used to acquire ownership of a (FILE *) object.

The *ftrylockfile* function is used to acquire ownership of a (FILE *) object if the object is available; *ftrylockfile* is a non-blocking version of *flockfile* .

The *funlockfile* function is used to relinquish the ownership of a (FILE *) object.

RETURN VALUES

The *ftrylockfile* function returns 0 on success or 1 to indicate that the lock cannot be acquired.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getc_unlocked`(3STDC)

STANDARDS

These routines conform to the POSIX.1c standards.

NAME

flockfile, ftrylockfile, funlockfile – stream lock management

SYNOPSIS

#include <stdio.h>
void **flockfile**(FILE * *file*);

int **ftrylockfile**(FILE * *file*);

void **funlockfile**(FILE * *file*);

DESCRIPTION

The *flockfile* , *ftrylockfile* and *funlockfile* functions provide for explicit application-level locking of stdio (FILE *) objects.

The *flockfile* function is used to acquire ownership of a (FILE *) object.

The *ftrylockfile* function is used to acquire ownership of a (FILE *) object if the object is available; *ftrylockfile* is a non-blocking version of *flockfile* .

The *funlockfile* function is used to relinquish the ownership of a (FILE *) object.

RETURN VALUES

The *ftrylockfile* function returns 0 on success or 1 to indicate that the lock cannot be acquired.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getc_unlocked`(3STDC)

STANDARDS

These routines conform to the POSIX.1c standards.

NAME

fread, fwrite – binary input/output

SYNOPSIS

#include <stdio.h>
int **`fread`**(void * *ptr*, size_t *size*, size_t *nitems*, FILE * *stream*);

int **`fwrite`**(const void * *ptr*, size_t *size*, size_t *nitems*, FILE * *stream*);

DESCRIPTION

The *fread* function copies, into an array pointed to by *ptr* , *nitems* items of data from the named input *stream* , where an item of data is a sequence of bytes (not necessarily terminated by a null byte) of the length `size` . The *fread* function stops appending bytes if an end-of-file or error condition is encountered while reading *stream*, or when *nitems* items have been read. The *fread* function leaves the file pointer in *stream* , if defined, pointing to the byte following the last byte read. It does not change the contents of *stream* .

The *fwrite* function appends at most *nitems* items of data from the array pointed to by *ptr* to the named output *stream* . The *fwrite* function stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream* . It does not change the contents of the array pointed to by *ptr* .

The `size` argument is typically *sizeof(*ptr)* where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr* .

RETURN VALUES

The *fread* and *fwrite* functions return the number of items read or written. If *nitems* is negative, no characters are read or written and 0 is returned by both *fread* and *fwrite* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC) , `getc`(3STDC) , `gets`(3STDC) , `putc`(3STDC) , `puts`(3STDC)

NAME

getc, fgetc, getw – get character from a stream

SYNOPSIS

#include <stdio.h>
int **getc**(FILE * *stream*);

int **fgetc**(FILE * *stream*);

int **getw**(FILE * *stream*);

DESCRIPTION

The *getc* and *fgetc* functions return the next character (byte) from the input *stream* specified, as an integer. The *getw* function obtains the next *int* (if present) from the stream pointed to by *stream*. These functions move the file pointer, if one is defined, ahead one character in *stream* .

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 *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 *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.

RETURN VALUES

These functions return the constant EOF at end-of-file or upon detecting an error.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fclose`(3STDC) , `ferror`(3STDC) , `fopen`(3STDC) , `fread`(3STDC) , `getchar`(3STDC) , `puts`(3STDC) , `scanf`(3STDC) , `setbuf`(3STDC)

NAME

getchar – get character from the standard input channel

SYNOPSIS

```
#include <stdio.h>
int getchar(void);
```

DESCRIPTION

The *getchar* function returns the next character (byte) from the standard input channel, which is operating-system dependent. On systems where *stdin,* has a meaning, *getc* (3STDC) is part of the library, and *getchar* is a macro defined as *getc(stdin)*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getc`(3STDC)

RETURN VALUES

This function returns the constant EOF at end-of-input (if the system supports this abstraction) or upon detecting an error.

NAME

unlocked, getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked – explicit locking functions

SYNOPSIS

#include <stdio.h>
int **getc_unlocked**(FILE * *stream*);

int **getchar_unlocked**(void);

int **putc_unlocked**(int *c*, FILE * *stream*);

int **putchar_unlocked**(int *c*);

DESCRIPTION

The *getc_unlocked* , *getchar_unlocked* , *putc_unlocked* and *putchar_unlocked* are functionally identical to *getc* , *getchar* , *putc* and *putchar* functions with the exception that they are not re-entrant.

getc_unlocked , *getchar_unlocked* , and *putchar_unlocked* routines are implemented as macros.

They may only safely be used within a scope protected by *flockfile* (or *ftrylockfile*) and *funlockedfile* .

STANDARDS

These routines conform to the POSIX.1c standards.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getc`(3STDC) , `getchar`(3STDC) , `putc`(3STDC) , `putchar`(3STDC) , `flockfile`(3STDC)

NAME

unlocked, getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked – explicit locking functions

SYNOPSIS

#include <stdio.h>
int **getc_unlocked**(FILE * *stream*);

int **getchar_unlocked**(void);

int **putc_unlocked**(int *c*, FILE * *stream*);

int **putchar_unlocked**(int *c*);

DESCRIPTION

The *getc_unlocked* , *getchar_unlocked* , *putc_unlocked* and *putchar_unlocked* are functionally identical to *getc* , *getchar* , *putc* and *putchar* functions with the exception that they are not re-entrant.

getc_unlocked , *getchar_unlocked* , and *putchar_unlocked* routines are implemented as macros.

They may only safely be used within a scope protected by *flockfile* (or *ftrylockfile*) and *funlockedfile* .

STANDARDS

These routines conform to the POSIX.1c standards.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getc`(3STDC) , `getchar`(3STDC) , `putc`(3STDC) , `putchar`(3STDC) , `flockfile`(3STDC)

NAME

getenv, putenv, setenv, unsetenv – fetch and set environment variables

SYNOPSIS

#include <stdlib.h>
char * **getenv**(const char * *name*);

int **setenv**(const char * *name*, const char * *value*, int *overwrite*);

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

void **unsetenv**(const char * *name*);

DESCRIPTION

These functions set, unset and fetch environment variables from the host *environment* list. For compatibility with differing environment conventions, the *name* and *value* arguments given may be appended and prepended, respectively, with an equal sign. The *getenv* function obtains the current value of the environment variable, *name*. If the variable *name* is not in the current environment, a null pointer is returned.

The `setenv` function inserts or resets the environment variable *name* in the current environment list. If the variable *name* does not exist in the list, it is inserted with the given *value*. If the variable does exist, the *overwrite* argument is tested; if *overwrite* is zero, the variable is not reset, otherwise it is reset to the given *value*.

The *putenv* function takes an argument of the form `name=value` and is equivalent to: `setenv(name, value, 1)` .

The `unsetenv` function deletes all instances of the variable name pointed to by *name* from the list.

RETURN VALUES

The `setenv` and *putenv* functions return zero if successful; otherwise –1 is returned. The `setenv` or *putenv* functions fail if they were unable to allocate memory for the environment.

STANDARDS

The *getenv* function conforms to `ANSI-C` .

NOTE

These functions are reentrant, but the environment is global to the actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

gethostbyaddr, gethostbyname – get network host entry

SYNOPSIS

#include <netdb.h>

struct hostent * **gethostbyname**(const char * *name*);

structhostent ***gethostbyaddr**(const char * *addr*, int *len*, int *type*);

DESCRIPTION

The *gethostbyname()* and *gethostbyaddr()* functions each return a pointer to an object containing the broken-out fields of a line in the network host data base. The object has the following structure:

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

#define h_addr  h_addr_list[0]   /* address, for backward compatiblity */
};
```

The members of this structure are:

h_name	Official name of the host.
h_aliases	A zero terminated array of alternate names for the host.
h_addrtype	The type of address being returned; currently always AF_INET.
h_length	The length, in bytes, of the address.
h_addr_list	A pointer to a list of network addresses for the named host. Host addresses are returned in network byte order.

In the case of *gethostbyaddr()* , *addr* is a pointer to the binary format address (supplied in network order) of length *len* (not a character string) and `type` is the type of the address.

To obtain this information, an Internet Name Server daemon must be running.

RETURN VALUES

A NULL pointer is returned on error.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`inetNS`(1M)

RESTRICTIONS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

NAME

gethostbyaddr, gethostbyname – get network host entry

SYNOPSIS

#include <netdb.h>

struct hostent * **gethostbyname**(const char * *name*);

structhostent ***gethostbyaddr**(const char * *addr*, int *len*, int *type*);

DESCRIPTION

The *gethostbyname()* and *gethostbyaddr()* functions each return a pointer to an object containing the broken-out fields of a line in the network host data base. The object has the following structure:

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

#define h_addr  h_addr_list[0]   /* address, for backward compatiblity */
};
```

The members of this structure are:

h_name	Official name of the host.
h_aliases	A zero terminated array of alternate names for the host.
h_addrtype	The type of address being returned; currently always AF_INET.
h_length	The length, in bytes, of the address.
h_addr_list	A pointer to a list of network addresses for the named host. Host addresses are returned in network byte order.

In the case of *gethostbyaddr()* , *addr* is a pointer to the binary format address (supplied in network order) of length *len* (not a character string) and `type` is the type of the address.

To obtain this information, an Internet Name Server daemon must be running.

RETURN VALUES

A NULL pointer is returned on error.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`inetNS`(1M)

RESTRICTIONS

All information is contained in a static area so it must be copied if it is to be saved. Only the Internet address format is currently understood.

NAME

getopt – get an option letter from command line argument list

SYNOPSIS

#include <stdlib.h>

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

```
extern char *optarg;
extern int optind;
extern int optopt;
extern int opterr;
extern int optreset;
```

DESCRIPTION

The `getopt` function incrementally parses the command line argument list *argv* and returns the next *known* option letter. An option letter is *known* if it has been specified in the string of accepted option letters, *optstring*.

The option string *optstring* can contain individual characters and characters followed by a colon indicating that an option argument follows. For example, an option string *x* indicates an option *-x*, and an option string *x:* indicates an option that has an argument, *-x argument*. It does not matter whether an argument has leading white space., that is **`–xarg`** and **`–x arg`** are interpreted as being the same.

On return from `getopt`, *optarg* points to an option argument, if one is expected, and the variable *optind* contains the index to the next *argv* argument for a subsequent call to `getopt`. The variable *optopt* saves the last *known* option letter returned by `getopt`.

The variables *opterr* and *optind* are both initialized to 1. The *optind* variable may be set to another value before a set of calls to `getopt` in order to access any given argv entry. In other words, you do not have to process the argv entries in order.

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

The `getopt` function returns an EOF when the argument list is exhausted, or a non-recognized option is encountered. The interpretation of options in the argument list may be cancelled by the option "–" (double dash) which causes `getopt` to signal the end of argument processing and return an EOF. When all options have been processed (up to the first non-option argument), `getopt` returns EOF.

RETURN VALUES

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

Option arguments are allowed to begin with "-", which reduces the amount of error checking possible.

EXAMPLES

```
#include <stdlib.h>
#include <stdio.h>
main (int argc, char **argv)
{
    int bflag= 0;
    char* fname;
    int ch, fd;

    while ((ch = getopt(argc, argv, "bf:")) != EOF) {
         switch(ch) {
             case ’b’:
                 bflag = 1;
                 break;
             case ’f’:
                 fname = optarg;
                 break;
             case ’?’:
             default:
                      fprintf(stderr, "usage: cmd [-b] [-f <arg>] \n");
                      exit(1);
         }
     }
}
argc -= optind;
argv += optind;
```

RESTRICTIONS

The `getopt` function is not thread safe.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getsubopt`(3STDC)

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 characters from the standard input stream, *stdin*, into the array pointed to by *s* , until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

The *fgets* function reads characters from *stream* into the array pointed to by *s* , until *n* –1 characters are read, or a new-line character is read and transferred to *s* , or an end-of-file condition is encountered. The string is then terminated with a null character.

RETURN VALUES

If end-of-file is reached and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs (for eample, if you are using these functions on a file that has not been opened for reading) , a NULL pointer is returned. Otherwise *s* is returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ferror`(3STDC) , `fopen`(3STDC) , `fread`(3STDC) , `getc`(3STDC) , `scanf`(3STDC)

NAME

getsitebyname, getsitebyaddr – get ChorusOS site information

SYNOPSIS

#include <chorusdb.h>
int **getsitebyname**(const char * *name*, int * *site*);

int **getsitebyaddr**(const int *site*, char * *name*, int * *length*);

DESCRIPTION

The *getsitebyname* function returns, in the object pointed to by *site* , the ChorusOS site number of the ChorusOS site whose symbolic name is *name* .

The *getsitebyaddr* function returns, in the character array *name* , the symbolic name of the ChorusOS site whose site number is *site* . If the real length of the symbolic name is greater than *length* , it is truncated to *length* bytes.

TIn order to obtain this information, a ChorusOS Name Server daemon must be running.

RETURN VALUES

The *getsitebyname* and *getsitebyaddr* functions return 0 in case of success. Otherwise they return -1 and set errno to indicate the error condition. The *getsitebyaddr* function returns the real *name* string length in *length* (including the NULL character).

ERRORS

Error code:

[ENOENT]	No such ChorusOS site is known.
[ETIMEDOUT]	The ChorusOS Name Server cannot be reached.
[EINVAL]	Invalid *length* (must be >= 0).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chorusNS`(1M)

NAME

getsitebyname, getsitebyaddr – get ChorusOS site information

SYNOPSIS

#include <chorusdb.h>
int **getsitebyname**(const char * *name*, int * *site*);

int **getsitebyaddr**(const int *site*, char * *name*, int * *length*);

DESCRIPTION

The *getsitebyname* function returns, in the object pointed to by *site* , the ChorusOS site number of the ChorusOS site whose symbolic name is *name* .

The *getsitebyaddr* function returns, in the character array *name* , the symbolic name of the ChorusOS site whose site number is *site* . If the real length of the symbolic name is greater than *length* , it is truncated to *length* bytes.

TIn order to obtain this information, a ChorusOS Name Server daemon must be running.

RETURN VALUES

The *getsitebyname* and *getsitebyaddr* functions return 0 in case of success. Otherwise they return -1 and set errno to indicate the error condition. The *getsitebyaddr* function returns the real *name* string length in *length* (including the NULL character).

ERRORS

Error code:

[ENOENT]	No such ChorusOS site is known.
[ETIMEDOUT]	The ChorusOS Name Server cannot be reached.
[EINVAL]	Invalid *length* (must be >= 0).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chorusNS`(1M)

NAME

getsubopt – get sub options from an argument

SYNOPSIS

#include <stdlib.h>

int **getsubopt**(char ***optionp*, char *const **tokens*, char ***valuep*);

```
extern char *suboptarg;
```

DESCRIPTION

The *getsubopt* function parses a string containing tokens delimited by one or more tab, space or comma (",") characters. It is intended for use in parsing groups of option arguments provided as part of a utility command line.

The *optionp* argument is a pointer to a pointer to the string. The *tokens* argument is a pointer to a NULL-terminated array of pointers to strings.

The *getsubopt* function returns the zero-based offset of the pointer in the *tokens* array, referencing a string which matches the first token in the string, or –1 if there is no match.

If the token is of the form "name=value", the location referenced by *valuep* will be set to point to the start of the "value" portion of the token.

On return from *getsubopt*, *optionp* will be set to point to the start of the next token in the string, or the NULL at the end of the string if no more tokens are present. The external variable *suboptarg* will be set to point to the start of the current token, or NULL if no tokens were present. The argument *valuep* will be set to point to the "value" portion of the token, or NULL if no "value" portion was present.

EXAMPLES

```
   char *tokens[] = {
        #define  ONE  0
                "one",
        #define  TWO  1
                "two",
        NULL
    };

    ...

    char *options, *value;

    while ((ch = getopt(argc, argv, "ab:")) != –1) {
        switch(ch) {
        case ’a’:
                /* process "a" option */
                break;
        case ’b’:
                options = optarg;
                while (*options) {
                        switch(getsubopt(&options, tokens, &value)) {
                        case ONE:
                                /* process "one" sub option */
                                break;
                        case TWO:
                                /* process "two" sub option */
                                if (!value) {
                                   printerr("no value for two");
                                }
                                else {
                                i = atoi(value);
                                }
                                break;
                        case -1:
                                if (suboptarg) {
                                    printerr("illegal sub option %s", suboptarg);
                                } else {
                                    printerr("missing sub option");
                                }
                                break;
                }
                break;
        }
```

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getopt`(3STDC)

RESTRICTIONS

The *getsubopt* function is not thread—safe.

NAME

getc, fgetc, getw – get character from a stream

SYNOPSIS

```
#include <stdio.h>
int getc(FILE * stream);

int fgetc(FILE * stream);

int getw(FILE * stream);
```

DESCRIPTION

The *getc* and *fgetc* functions return the next character (byte) from the input *stream* specified, as an integer. The *getw* function obtains the next *int* (if present) from the stream pointed to by *stream*. These functions move the file pointer, if one is defined, ahead one character in *stream* .

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 *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 *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.

RETURN VALUES

These functions return the constant EOF at end-of-file or upon detecting an error.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fclose`(3STDC) , `ferror`(3STDC) , `fopen`(3STDC) , `fread`(3STDC) , `getchar`(3STDC) , `puts`(3STDC) , `scanf`(3STDC) , `setbuf`(3STDC)

NAME

ctime, asctime, difftime, gmtime, localtime, mktime – transform binary date and time value to ASCII

SYNOPSIS

#include <time.h>
struct tm * **localtime**(const time_t * *clock*);

struct tm * **gmtime**(const time_t * *clock*);

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

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

time_t **mktime**(struct tm * *tm*);

double **difftime**(time_t *time1*, time_t *time0*);

DESCRIPTION

The *ctime* , *gmtime* and *localtime* functions take as an argument a time value representing the time in seconds since the Epoch (00:00:00 UTC, January 1, 1970).

The *localtime* function converts the time value pointed to by *clock* , and returns a pointer to a *struct tm* (described below) which contains the broken-out time information for the value, after adjusting for the current time zone (and any other factors such as Daylight Saving Time). Time zone adjustments are performed as specified by the TZ environment variable (see *tzset* (3STDC). The function *localtime* uses *tzset* (3STDC) to initialize time conversion information if *tzset* (3STDC) has not already been called by the process.

The *gmtime* function also converts the time value, but without any time zone adjustment, and returns a pointer to a *tm* structure (described below).

The *ctime* function adjusts the time value for the current time zone in the same manner as *localtime* , and returns a pointer to a 26-character string of the form: Thu Nov 24 18:22:48 1986.

The *asctime* function converts the broken—down time in the structure *tm* pointed to by **tm* to the form shown in the example above.

The *mktime* function converts the broken-down time, expressed as local time, in the structure pointed to by *tm* into a time value with the same encoding as that of the values returned by the `time` (3STDC) function; that is, seconds from the Epoch, UTC .

The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to their normal ranges. (A positive or zero value for *tm_isdst* causes *mktime* to presume initially that summer time (for example, Daylight Saving Time) is or is not in effect for the time specified, respectively. A negative value for *tm_isdst* causes the *mktime* function to attempt to define whether summer time is in effect for the time specified.)

On successful completion, the values of the *tm_wday* and *tm_yday* components of the structure are set appropriately, and the other components are set to represent the calendar time specified, but with their values forced to their normal ranges; the final value of *tm_mday* is not set until *tm_mon* and *tm_year* are determined. The *mktime* function returns the calendar time specified; if the calendar time cannot be represented, it returns –1;

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

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

```
int tm_sec;          /* seconds (0 - 60) */
int tm_min;          /* minutes (0 - 59) */
int tm_hour;         /* hours (0 - 23) */
int tm_mday;         /* day of month (1 - 31) */
int tm_mon;          /* month of year (0 - 11) */
int tm_year;         /* year – 1900 */
int tm_wday;         /* day of week (Sunday = 0) */
int tm_yday;         /* day of year (0 - 365) */
int tm_isdst;        /* is summer time in effect? */
char *tm_zone;       /* abbreviation of timezone name */
long tm_gmtoff;      /* offset from UTC in seconds */
```

The field *tm_isdst* is non-zero if summer time is in effect.

The field *tm_gmtoff* is the offset (in seconds) of the time represented from `UTC` , with positive values indicating east of the Prime Meridian.

NOTES

asctime(3STDC) , *ctime(3STDC)* , *localtime(3STDC)* and *gmtime(3STDC)* return their result in a global variable which make them difficult to use in a multithreaded program. *asctime_r(3STDC)* , *ctime_r(3STDC)* , *localtime_r(3STDC)* and *gmtime_r(3STDC)* should be used instead.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`asctime_r`(3STDC) , `ctime_r`(3STDC) , `getenv`(3STDC) , `gmtime_r`(3STDC) , `localtime_r`(3STDC) , `time`(3STDC) , `tzset`(3STDC)

NAME

ctime_r, asctime_r, gmtime_r, localtime_r – Transform binary date and time value to ASCII; Reentrent version

SYNOPSIS

#include <time.h>
char * **ctime_r**(const time_t * *clock*, char * *result*);

char * **asctime_r**(const struct tm * *tm*, char * *result*);

struct tm * **localtime_r**(const time_t * *clock*, struct tm * *result*);

struct tm * **gmtime_r**(const time_t * *clock*, struct tm * *result*);

DESCRIPTION

The *ctime_r, gmtime_r, asctime_r,* and *localtime_r* functions do the same thing as *ctime* (3STDC), *gmtime* (3STDC), *asctime* (3STDC), and *localtime* (3STDC), with the difference that they do not store their result in a static buffer. Instead, the necessary storage must be allocated by the caller and a pointer to it passed as the *result* argument.

For *asctime_r, result* must point to a 26 byte character array. For the others, *result* must point to a memory area large enough to hold a *struct tm*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`asctime`(3STDC) , `ctime`(3STDC) , `localtime`(3STDC) , `gmtime`(3STDC) , `tzset`(3STDC)

STANDARDS

These routines conform to POSIX.1c.

NAME

byteorder, htonl, htons, ntohl, ntohs – convert values between host and network byte order

SYNOPSIS

#include <sys/param.h>
unsigned long **htonl**(unsigned long *hostlong*);

unsigned short **htons**(unsigned short *hostshort*);

unsigned long **ntohl**(unsigned long *netlong*);

unsigned short **ntohs**(unsigned short *netshort*);

DESCRIPTION

These routines convert 16– and 32–bit quantities between network byte order and host byte order. On architectures where the host byte order and network byte order are the same, these routines are defined as no-op macros.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

byteorder, htonl, htons, ntohl, ntohs – convert values between host and network byte order

SYNOPSIS

#include <sys/param.h>
unsigned long **htonl**(unsigned long *hostlong*);

unsigned short **htons**(unsigned short *hostshort*);

unsigned long **ntohl**(unsigned long *netlong*);

unsigned short **ntohs**(unsigned short *netshort*);

DESCRIPTION

These routines convert 16– and 32–bit quantities between network byte order and host byte order. On architectures where the host byte order and network byte order are the same, these routines are defined as no-op macros.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

index, rindex – locate character in string

SYNOPSIS

#include <string.h>
char * **index**(const char * *s*, int *c*);

char ***rindex**(const char * *s*, int *c*);

DESCRIPTION

The *index* function locates the first character matching *c* (converted to a *char*) in the null-terminated string *s* .

The *rindex* function locates the last character matching *c* (converted to a *char*) in the null-terminated string *s* .

RETURN VALUES

A pointer to the character is returned if found; otherwise NULL is returned.
If *c* is 0 , *rindex* or *index* locates the terminating 0.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`memchr`(3STDC) , `string`(3STDC) , `strsep`(3STDC) , `strtok`(3STDC)

NAME

inet, inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int **inet_aton**(const char * *cp*, struct in_addr * *pin*);

unsigned **longinet_addr**(const char * *cp*);

unsigned **longinet_network**(const char * *cp*);

char ***inet_ntoa**(struct in_addr *in*);

struct **in_addrinet_makeaddr**(u_long *net*, u_long *lna*);

unsigned **longinet_lnaof**(struct in_addr *in*);

unsigned **longinet_netof**(struct in_addr *in*);

DESCRIPTION

The *inet_aton* , *inet_addr* and *inet_network* routines interpret character strings representing numbers expressed in the Internet standard notation. The *inet_aton* routine interprets the specified character string as an Internet address, placing the address in the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid. The *inet_addr* and *inet_network* functions return numbers suitable for use as Internet addresses and Internet network numbers, respectively. The *inet_ntoa* routine takes an Internet address and returns an ASCII string representing the address in Internet notation. The *inet_makeaddr* routine takes an Internet network number and a local network address and constructs an Internet address from it. The *inet_netof* and *inet_lnaof* routines break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the Internet notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This

makes the three part address format convenient for specifying Class B network addresses such as: `128.net.host` .

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right—most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses such as: `net.host` .

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as *parts* in Internet notation may be decimal, octal, or hexadecimal, as specified in the C language (a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

DIAGNOSTICS

The constant INADDR_NONE is returned by *inet_addr* and *inet_network* for malformed requests.

RESTRICTIONS

The value INADDR_NONE (0xffffffff) is a valid broadcast address, but *inet_addr* cannot return that value without indicating failure. The newer *inet_aton* function does not share this problem. The problem of host byte ordering versus network byte ordering is confusing. The string returned by *inet_ntoa* resides in a static memory area, which means that this routine is not reentrant.

inet_addr should return a *struct in_addr.*

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

inet, inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int **inet_aton**(const char * *cp*, struct in_addr * *pin*);

unsigned **longinet_addr**(const char * *cp*);

unsigned **longinet_network**(const char * *cp*);

char ***inet_ntoa**(struct in_addr *in*);

struct **in_addrinet_makeaddr**(u_long *net*, u_long *lna*);

unsigned **longinet_lnaof**(struct in_addr *in*);

unsigned **longinet_netof**(struct in_addr *in*);

DESCRIPTION

The *inet_aton* , *inet_addr* and *inet_network* routines interpret character strings representing numbers expressed in the Internet standard notation. The *inet_aton* routine interprets the specified character string as an Internet address, placing the address in the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid. The *inet_addr* and *inet_network* functions return numbers suitable for use as Internet addresses and Internet network numbers, respectively. The *inet_ntoa* routine takes an Internet address and returns an ASCII string representing the address in Internet notation. The *inet_makeaddr* routine takes an Internet network number and a local network address and constructs an Internet address from it. The *inet_netof* and *inet_lnaof* routines break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the Internet notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This

makes the three part address format convenient for specifying Class B network addresses such as: `128.net.host` .

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right—most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses such as: `net.host` .

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as *parts* in Internet notation may be decimal, octal, or hexadecimal, as specified in the C language (a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

DIAGNOSTICS

The constant INADDR_NONE is returned by *inet_addr* and *inet_network* for malformed requests.

RESTRICTIONS

The value INADDR_NONE (0xffffffff) is a valid broadcast address, but *inet_addr* cannot return that value without indicating failure. The newer *inet_aton* function does not share this problem. The problem of host byte ordering versus network byte ordering is confusing. The string returned by *inet_ntoa* resides in a static memory area, which means that this routine is not reentrant.

inet_addr should return a *struct in_addr.*

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

inet, inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int **inet_aton**(const char * *cp*, struct in_addr * *pin*);

unsigned **longinet_addr**(const char * *cp*);

unsigned **longinet_network**(const char * *cp*);

char ***inet_ntoa**(struct in_addr *in*);

struct **in_addrinet_makeaddr**(u_long *net*, u_long *lna*);

unsigned **longinet_lnaof**(struct in_addr *in*);

unsigned **longinet_netof**(struct in_addr *in*);

DESCRIPTION

The *inet_aton* , *inet_addr* and *inet_network* routines interpret character strings representing numbers expressed in the Internet standard notation. The *inet_aton* routine interprets the specified character string as an Internet address, placing the address in the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid. The *inet_addr* and *inet_network* functions return numbers suitable for use as Internet addresses and Internet network numbers, respectively. The *inet_ntoa* routine takes an Internet address and returns an ASCII string representing the address in Internet notation. The *inet_makeaddr* routine takes an Internet network number and a local network address and constructs an Internet address from it. The *inet_netof* and *inet_lnaof* routines break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the Internet notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This

makes the three part address format convenient for specifying Class B network addresses such as: `128.net.host` .

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right—most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses such as: `net.host` .

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as *parts* in Internet notation may be decimal, octal, or hexadecimal, as specified in the C language (a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

DIAGNOSTICS

The constant INADDR_NONE is returned by *inet_addr* and *inet_network* for malformed requests.

RESTRICTIONS

The value INADDR_NONE (0xffffffff) is a valid broadcast address, but *inet_addr* cannot return that value without indicating failure. The newer *inet_aton* function does not share this problem. The problem of host byte ordering versus network byte ordering is confusing. The string returned by *inet_ntoa* resides in a static memory area, which means that this routine is not reentrant.

inet_addr should return a *struct in_addr.*

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

inet, inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int **inet_aton**(const char * *cp*, struct in_addr * *pin*);

unsigned **longinet_addr**(const char * *cp*);

unsigned **longinet_network**(const char * *cp*);

char ***inet_ntoa**(struct in_addr *in*);

struct **in_addrinet_makeaddr**(u_long *net*, u_long *lna*);

unsigned **longinet_lnaof**(struct in_addr *in*);

unsigned **longinet_netof**(struct in_addr *in*);

DESCRIPTION

The *inet_aton* , *inet_addr* and *inet_network* routines interpret character strings representing numbers expressed in the Internet standard notation. The *inet_aton* routine interprets the specified character string as an Internet address, placing the address in the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid. The *inet_addr* and *inet_network* functions return numbers suitable for use as Internet addresses and Internet network numbers, respectively. The *inet_ntoa* routine takes an Internet address and returns an ASCII string representing the address in Internet notation. The *inet_makeaddr* routine takes an Internet network number and a local network address and constructs an Internet address from it. The *inet_netof* and *inet_lnaof* routines break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the Internet notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This

makes the three part address format convenient for specifying Class B network addresses such as: `128.net.host` .

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right—most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses such as: `net.host` .

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as *parts* in Internet notation may be decimal, octal, or hexadecimal, as specified in the C language (a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

DIAGNOSTICS

The constant INADDR_NONE is returned by *inet_addr* and *inet_network* for malformed requests.

RESTRICTIONS

The value INADDR_NONE (0xffffffff) is a valid broadcast address, but *inet_addr* cannot return that value without indicating failure. The newer *inet_aton* function does not share this problem. The problem of host byte ordering versus network byte ordering is confusing. The string returned by *inet_ntoa* resides in a static memory area, which means that this routine is not reentrant.

inet_addr should return a *struct in_addr.*

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

inet, inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int **inet_aton**(const char * *cp*, struct in_addr * *pin*);

unsigned **longinet_addr**(const char * *cp*);

unsigned **longinet_network**(const char * *cp*);

char ***inet_ntoa**(struct in_addr *in*);

struct **in_addrinet_makeaddr**(u_long *net*, u_long *lna*);

unsigned **longinet_lnaof**(struct in_addr *in*);

unsigned **longinet_netof**(struct in_addr *in*);

DESCRIPTION

The *inet_aton* , *inet_addr* and *inet_network* routines interpret character strings representing numbers expressed in the Internet standard notation. The *inet_aton* routine interprets the specified character string as an Internet address, placing the address in the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid. The *inet_addr* and *inet_network* functions return numbers suitable for use as Internet addresses and Internet network numbers, respectively. The *inet_ntoa* routine takes an Internet address and returns an ASCII string representing the address in Internet notation. The *inet_makeaddr* routine takes an Internet network number and a local network address and constructs an Internet address from it. The *inet_netof* and *inet_lnaof* routines break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the Internet notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This

makes the three part address format convenient for specifying Class B network addresses such as: `128.net.host` .

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right—most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses such as: `net.host` .

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as *parts* in Internet notation may be decimal, octal, or hexadecimal, as specified in the C language (a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

DIAGNOSTICS

The constant INADDR_NONE is returned by *inet_addr* and *inet_network* for malformed requests.

RESTRICTIONS

The value INADDR_NONE (0xffffffff) is a valid broadcast address, but *inet_addr* cannot return that value without indicating failure. The newer *inet_aton* function does not share this problem. The problem of host byte ordering versus network byte ordering is confusing. The string returned by *inet_ntoa* resides in a static memory area, which means that this routine is not reentrant.

inet_addr should return a *struct in_addr.*

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

inet, inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int **inet_aton**(const char * *cp*, struct in_addr * *pin*);

unsigned **longinet_addr**(const char * *cp*);

unsigned **longinet_network**(const char * *cp*);

char ***inet_ntoa**(struct in_addr *in*);

struct **in_addrinet_makeaddr**(u_long *net*, u_long *lna*);

unsigned **longinet_lnaof**(struct in_addr *in*);

unsigned **longinet_netof**(struct in_addr *in*);

DESCRIPTION

The *inet_aton* , *inet_addr* and *inet_network* routines interpret character strings representing numbers expressed in the Internet standard notation. The *inet_aton* routine interprets the specified character string as an Internet address, placing the address in the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid. The *inet_addr* and *inet_network* functions return numbers suitable for use as Internet addresses and Internet network numbers, respectively. The *inet_ntoa* routine takes an Internet address and returns an ASCII string representing the address in Internet notation. The *inet_makeaddr* routine takes an Internet network number and a local network address and constructs an Internet address from it. The *inet_netof* and *inet_lnaof* routines break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the Internet notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This

makes the three part address format convenient for specifying Class B network addresses such as: `128.net.host` .

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right—most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses such as: `net.host` .

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as *parts* in Internet notation may be decimal, octal, or hexadecimal, as specified in the C language (a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

DIAGNOSTICS

The constant INADDR_NONE is returned by *inet_addr* and *inet_network* for malformed requests.

RESTRICTIONS

The value INADDR_NONE (0xffffffff) is a valid broadcast address, but *inet_addr* cannot return that value without indicating failure. The newer *inet_aton* function does not share this problem. The problem of host byte ordering versus network byte ordering is confusing. The string returned by *inet_ntoa* resides in a static memory area, which means that this routine is not reentrant.

inet_addr should return a *struct in_addr.*

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

inet, inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int **inet_aton**(const char * *cp*, struct in_addr * *pin*);

unsigned **longinet_addr**(const char * *cp*);

unsigned **longinet_network**(const char * *cp*);

char ***inet_ntoa**(struct in_addr *in*);

struct **in_addrinet_makeaddr**(u_long *net*, u_long *lna*);

unsigned **longinet_lnaof**(struct in_addr *in*);

unsigned **longinet_netof**(struct in_addr *in*);

DESCRIPTION

The *inet_aton* , *inet_addr* and *inet_network* routines interpret character strings representing numbers expressed in the Internet standard notation. The *inet_aton* routine interprets the specified character string as an Internet address, placing the address in the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid. The *inet_addr* and *inet_network* functions return numbers suitable for use as Internet addresses and Internet network numbers, respectively. The *inet_ntoa* routine takes an Internet address and returns an ASCII string representing the address in Internet notation. The *inet_makeaddr* routine takes an Internet network number and a local network address and constructs an Internet address from it. The *inet_netof* and *inet_lnaof* routines break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the Internet notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This

makes the three part address format convenient for specifying Class B network addresses such as: `128.net.host` .

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right—most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses such as: `net.host` .

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as *parts* in Internet notation may be decimal, octal, or hexadecimal, as specified in the C language (a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

DIAGNOSTICS

The constant INADDR_NONE is returned by *inet_addr* and *inet_network* for malformed requests.

RESTRICTIONS

The value INADDR_NONE (0xffffffff) is a valid broadcast address, but *inet_addr* cannot return that value without indicating failure. The newer *inet_aton* function does not share this problem. The problem of host byte ordering versus network byte ordering is confusing. The string returned by *inet_ntoa* resides in a static memory area, which means that this routine is not reentrant.

inet_addr should return a *struct in_addr.*

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

inet, inet_aton, inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof – Internet address manipulation routines

SYNOPSIS

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int **inet_aton**(const char * *cp*, struct in_addr * *pin*);

unsigned **longinet_addr**(const char * *cp*);

unsigned **longinet_network**(const char * *cp*);

char ***inet_ntoa**(struct in_addr *in*);

struct **in_addrinet_makeaddr**(u_long *net*, u_long *lna*);

unsigned **longinet_lnaof**(struct in_addr *in*);

unsigned **longinet_netof**(struct in_addr *in*);

DESCRIPTION

The *inet_aton* , *inet_addr* and *inet_network* routines interpret character strings representing numbers expressed in the Internet standard notation. The *inet_aton* routine interprets the specified character string as an Internet address, placing the address in the structure provided. It returns 1 if the string was successfully interpreted, or 0 if the string is invalid. The *inet_addr* and *inet_network* functions return numbers suitable for use as Internet addresses and Internet network numbers, respectively. The *inet_ntoa* routine takes an Internet address and returns an ASCII string representing the address in Internet notation. The *inet_makeaddr* routine takes an Internet network number and a local network address and constructs an Internet address from it. The *inet_netof* and *inet_lnaof* routines break apart Internet host addresses, returning the network number and local network address part, respectively.

All Internet addresses are returned in network order (bytes ordered from left to right). All network numbers and local address parts are returned as machine format integer values.

INTERNET ADDRESSES

Values specified using the Internet notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet address.

When a three part address is specified, the last part is interpreted as a 16-bit quantity and placed in the right-most two bytes of the network address. This

makes the three part address format convenient for specifying Class B network addresses such as: `128.net.host` .

When a two part address is supplied, the last part is interpreted as a 24-bit quantity and placed in the right—most three bytes of the network address. This makes the two part address format convenient for specifying Class A network addresses such as: `net.host` .

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as *parts* in Internet notation may be decimal, octal, or hexadecimal, as specified in the C language (a leading 0x or 0X implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

DIAGNOSTICS

The constant INADDR_NONE is returned by *inet_addr* and *inet_network* for malformed requests.

RESTRICTIONS

The value INADDR_NONE (0xffffffff) is a valid broadcast address, but *inet_addr* cannot return that value without indicating failure. The newer *inet_aton* function does not share this problem. The problem of host byte ordering versus network byte ordering is confusing. The string returned by *inet_ntoa* resides in a static memory area, which means that this routine is not reentrant.

inet_addr should return a *struct in_addr.*

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

random, srandom, initstate, setstate – better random number generator

SYNOPSIS

```
#include <stdlib.h>
long random(void);

void srandom(unsigned seed);

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

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

DESCRIPTION

The *random* function uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from `0` to $2^{31} - 1$. The period of this random number generator is very large, approximately $16 \times (2^{31} - 1)$.

The *random/srandom* functions have (almost) the same calling sequence and initialization properties as *rand/srand* (3STDC) The difference is that *rand* produces a much less random sequence — in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by *random* are usable. For example, *random* `&01` will produce a random binary value.

Unlike *srand* , *srandom* does not return the old seed; the reason being that the amount of state information used is much more than a single word (two other routines are provided to deal with restarting / changing random number generators). Like *rand* , however, *random* will by default produce a sequence of numbers that can be duplicated by calling *srandom* with 1 as the seed.

The *initstate* routine allows a state array, passed as an argument, to be initialized for future use. The size of the state array (in bytes) is used by *initstate* to decide how sophisticated a random number generator it should use — the bigger the state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error.) The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. The *initstate* function returns a pointer to the previous state information array.

Once a state has been initialized, the *setstate* routine provides for rapid switching between states. The *setstate* function returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to *initstate* or *setstate* .

Once a state array has been initialized, it may be restarted at a different point either by calling *initstate* (with the desired seed, the state array, and its size) or by calling both *setstate* (with the state array) and *srandom* (with the desired seed).

The advantage of calling both *setstate* and *srandom* is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{690}, which should be sufficient for most purposes.

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` , it returns `NULL` and *random* uses a simple linear congruential random number generator.

DIAGNOSTICS

If *initstate* is called with less than 8 bytes of state information, or if *setstate* detects that the state information has been garbled, error messages are printed to the standard error output.

NOTE

Though these functions are reentrant, the state information is global to the actor. Therefore, repeatability of a given suite of number will not be experienced by several threads in parallel. For a reentrent repeatability of suites, see *rand_r(3STDC)* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rand`(3STDC) , `rand_r`(3STDC)

RESTRICTIONS

random operates at about 2/3 the speed of *rand* (3STDC).

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

```
#include <ctype.h>
int isalpha(int c);
```

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

```
#include <ctype.h>
int isalpha(int c);
```

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

isascii – test for ASCII character

SYNOPSIS

```
#include <ctype.h>
int isascii(int c);
```

DESCRIPTION

The*isascii* function tests for an `ASCII` character, which is any character with a value less than or equal to 0177.

The *isascii* function returns 1 if *number* is `ASCII,` otherwise 0.

The validity of the test is limited to the defaut *locale*. If another `locale` is currently in effect, the semantical correctness of the result is unspecified.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ctype`(3STDC)

STANDARDS

Due to its dubious validity when used in conjunction with *setlocale*, this function is no longer part of `ANSI-C`.

NAME

isatty – check if a file descriptor is associated with a terminal

SYNOPSIS

#include <unistd.h>
int **isatty**(int *fd*);

DESCRIPTION

The *isatty* function checks whether or not the file descriptor soecified by *fd* is associated with a terminal device.

RETURN VALUES

The *isatty*function returns 1 if the file descriptor is associated with a terminal device. It returns 0 otherwise.

NOTE

In ChorusOS, *isatty* always returns 1 on a socket file descriptor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ioctl`(2POSIX)

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

```
#include <ctype.h>
int isalpha(int c);
```

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

```
#include <ctype.h>
int isalpha(int c);
```

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

```
#include <ctype.h>
int isalpha(int c);
```

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

isinf, isnan – test for infinity or not-a-number

SYNOPSIS

#include <math.h>
int **isinf**(double *number*);

int **isnan**(double *number*);

DESCRIPTION

The *isinf* function returns 1 if *number* is “infinite”, otherwise 0.

The *isnan* function returns 1 if *number* is “not-a-number”, otherwise 0.

ATTRIBUTES

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

ATTRIBUTE TYPE	**ATTRIBUTE VALUE**
Interface Stability	Evolving

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

```
#include <ctype.h>
int isalpha(int c);
```

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

isinf, isnan – test for infinity or not-a-number

SYNOPSIS

#include <math.h>
int **isinf**(double *number*);

int **isnan**(double *number*);

DESCRIPTION

The *isinf* function returns 1 if *number* is “infinite”, otherwise 0.

The *isnan* function returns 1 if *number* is “not-a-number”, otherwise 0.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

#include <ctype.h>
int **isalpha**(int *c*);

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

```
#include <ctype.h>
int isalpha(int c);
```

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

#include <ctype.h>
int **isalpha**(int *c*);

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

```
#include <ctype.h>
int isalpha(int c);
```

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

```
#include <ctype.h>
int isalpha(int c);
```

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

labs – return the absolute value of a long integer

SYNOPSIS

```
#include <stdlib.h>
long labs(long j);
```

DESCRIPTION

The *labs* function returns the absolute value of the long integer *j*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`abs`(3STDC)

STANDARDS

The *labs* function conforms to `ANSI-C`.

RESTRICTIONS

The absolute value of the highest negative integer remains negative.

NAME

ldexp, _ldexp – multiply floating-point number by integral power of 2

SYNOPSIS

```
#include <math.h>
double ldexp(double x, int exp);

double _ldexp(double x, int exp);
```

DESCRIPTION

The *ldexp* function multiplies a floating-point number by an integral power of 2. The *_ldexp* function implements the real floating-point calculation; *ldexp* performs the range checking and calls *_ldexp*. It is therefore faster to call *_ldexp* if the arguments are known to be within the function's domain.

RETURN VALUES

The *ldexp* function returns the value of *x* times 2 raised to the power *exp* :

$x * 2^{\mathrm{exp}}$

If the resultant value would cause an overflow, the global variable *errno* is set to ERANGE and the value HUGE is returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`modf`(3STDC)

STANDARDS

The *ldexp* function conforms to `ANSI-C` .

NAME

ldexp, _ldexp – multiply floating-point number by integral power of 2

SYNOPSIS

```
#include <math.h>
double ldexp(double x, int exp);

double _ldexp(double x, int exp);
```

DESCRIPTION

The *ldexp* function multiplies a floating-point number by an integral power of 2. The *_ldexp* function implements the real floating-point calculation; *ldexp* performs the range checking and calls *_ldexp*. It is therefore faster to call *_ldexp* if the arguments are known to be within the function's domain.

RETURN VALUES

The *ldexp* function returns the value of *x* times 2 raised to the power *exp* :

$x * 2^{\text{exp}}$

If the resultant value would cause an overflow, the global variable *errno* is set to ERANGE and the value HUGE is returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`modf`(3STDC)

STANDARDS

The *ldexp* function conforms to `ANSI-C` .

NAME

ldiv – return quotient and remainder from division

SYNOPSIS

#include <stdlib.h>
ldiv_t **ldiv**(long *num*, long *denom*);

DESCRIPTION

The *ldiv* function computes the value *num/denom* and returns the quotient and remainder in a structure named *ldiv_t* which contains two *long integer* members named `quot` and *rem*.

When an input of zero is applied to the *denom* parameter, the behavior of the function is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`div`(3STDC)

STANDARDS

The *ldiv* function conforms to `ANSI-C`.

NAME

ctime, asctime, difftime, gmtime, localtime, mktime – transform binary date and time value to ASCII

SYNOPSIS

#include <time.h>
struct tm * **localtime**(const time_t * *clock*);

struct tm * **gmtime**(const time_t * *clock*);

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

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

time_t **mktime**(struct tm * *tm*);

double **difftime**(time_t *time1*, time_t *time0*);

DESCRIPTION

The *ctime* , *gmtime* and *localtime* functions take as an argument a time value representing the time in seconds since the Epoch (00:00:00 `UTC,` January 1, 1970).

The *localtime* function converts the time value pointed to by *clock* , and returns a pointer to a *struct tm* (described below) which contains the broken-out time information for the value, after adjusting for the current time zone (and any other factors such as Daylight Saving Time). Time zone adjustments are performed as specified by the `TZ` environment variable (see *tzset* (3STDC). The function *localtime* uses *tzset* (3STDC) to initialize time conversion information if *tzset* (3STDC) has not already been called by the process.

The *gmtime* function also converts the time value, but without any time zone adjustment, and returns a pointer to a *tm* structure (described below).

The *ctime* function adjusts the time value for the current time zone in the same manner as *localtime* , and returns a pointer to a 26-character string of the form: Thu Nov 24 18:22:48 1986.

The *asctime* function converts the broken—down time in the structure *tm* pointed to by **tm* to the form shown in the example above.

The *mktime* function converts the broken-down time, expressed as local time, in the structure pointed to by *tm* into a time value with the same encoding as that of the values returned by the `time` (3STDC) function; that is, seconds from the Epoch, `UTC` .

The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to their normal ranges. (A positive or zero value for *tm_isdst* causes *mktime* to presume initially that summer time (for example, Daylight Saving Time) is or is not in effect for the time specified, respectively. A negative value for *tm_isdst* causes the *mktime* function to attempt to define whether summer time is in effect for the time specified.)

On successful completion, the values of the *tm_wday* and *tm_yday* components of the structure are set appropriately, and the other components are set to represent the calendar time specified, but with their values forced to their normal ranges; the final value of *tm_mday* is not set until *tm_mon* and *tm_year* are determined. The *mktime* function returns the calendar time specified; if the calendar time cannot be represented, it returns –1;

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

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

```
int tm_sec;          /* seconds (0 - 60) */
int tm_min;          /* minutes (0 - 59) */
int tm_hour;         /* hours (0 - 23) */
int tm_mday;         /* day of month (1 - 31) */
int tm_mon;          /* month of year (0 - 11) */
int tm_year;         /* year – 1900 */
int tm_wday;         /* day of week (Sunday = 0) */
int tm_yday;         /* day of year (0 - 365) */
int tm_isdst;        /* is summer time in effect? */
char *tm_zone;       /* abbreviation of timezone name */
long tm_gmtoff;      /* offset from UTC in seconds */
```

The field *tm_isdst* is non-zero if summer time is in effect.

The field *tm_gmtoff* is the offset (in seconds) of the time represented from `UTC` , with positive values indicating east of the Prime Meridian.

NOTES

asctime(3STDC) , *ctime(3STDC)* , *localtime(3STDC)* and *gmtime(3STDC)* return their result in a global variable which make them difficult to use in a multithreaded program. *asctime_r(3STDC)* , *ctime_r(3STDC)* , *localtime_r(3STDC)* and *gmtime_r(3STDC)* should be used instead.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`asctime_r`(3STDC) , `ctime_r`(3STDC) , `getenv`(3STDC) , `gmtime_r`(3STDC) , `localtime_r`(3STDC) , `time`(3STDC) , `tzset`(3STDC)

NAME

ctime_r, asctime_r, gmtime_r, localtime_r – Transform binary date and time value to ASCII; Reentrent version

SYNOPSIS

#include <time.h>
char * **ctime_r**(const time_t * *clock*, char * *result*);

char * **asctime_r**(const struct tm * *tm*, char * *result*);

struct tm * **localtime_r**(const time_t * *clock*, struct tm * *result*);

struct tm * **gmtime_r**(const time_t * *clock*, struct tm * *result*);

DESCRIPTION

The *ctime_r, gmtime_r, asctime_r,* and *localtime_r* functions do the same thing as *ctime* (3STDC), *gmtime* (3STDC), *asctime* (3STDC), and *localtime* (3STDC), with the difference that they do not store their result in a static buffer. Instead, the necessary storage must be allocated by the caller and a pointer to it passed as the *result* argument.

For *asctime_r, result* must point to a 26 byte character array. For the others, *result* must point to a memory area large enough to hold a *struct tm*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`asctime`(3STDC) , `ctime`(3STDC) , `localtime`(3STDC) , `gmtime`(3STDC) , `tzset`(3STDC)

STANDARDS

These routines conform to POSIX.1c.

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

```
#include <setjmp.h>
int setjmp(jmp_buf env);

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

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in low-level subroutines of a program.

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

The *longjmp* function restores the environment saved by the last call of *setjmp* with the corresponding `env` argument. After *longjmp* has completed, program execution continues as if the corresponding call of *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. All accessible data will have the values stored at the time *longjmp* was called.

WARNING

If *longjmp* is called without first priming `env` using a calll to *setjmp* , or if the last such call was performed by another thread, or if the last such call was in a function that has since returned, this will cause severe disruption to the system.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

malloc, free, realloc, calloc – main memory allocator

SYNOPSIS

#include <stdlib.h>
void * **malloc**(size_t *size*);

void **free**(void * *ptr*);

void * **realloc**(void * *ptr*, size_t *size*);

void ***calloc**(size_t *nelem*, size_t *elsize*);

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. ChorusOS 4.0 offers three `malloc( )` libraries. See *EXTENDED DESCRIPTION* below for details.

The argument passed to `free( )` is a pointer to a block previously allocated by `malloc( )` ; after `free( )` is performed this space is made available for further allocation, but its contents are left undisturbed.

The `free( )` function may be called with a `NULL` pointer as parameter.

If the space assigned by `malloc( )` is overrun or if a random number is passed to `free( )` , the result is undefined.

The `malloc( )` function searches for free space from the last block allocated or freed, grouping together any adjacent free blocks. It allocates the first contiguous area of free space that is at least `size( )` bytes.

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 smaller of the new and old sizes. If no free block of *size* bytes is available in the storage area, `realloc( )` will ask `malloc( )` to enlarge the area by *size* bytes and will then move the data to the new space. If the space cannot be allocated, the object pointed to by *ptr* is unchanged. If *size* is zero and *ptr* is not a null pointer, the object it points to is freed. If *ptr* is a null pointer, the `realloc( )` function behaves like the `malloc( )` function for the specified size.

The `realloc( )` function also works if *ptr* points to a block freed since the last call to `malloc( )` , `realloc( )` , or `calloc( )` ; thus sequences of `free( )` , `malloc( )` and `realloc( )` can be used to exploit the search strategy of `malloc( )` in order to do storage compacting.

The `calloc( )` function allocates space for an array of *nelem* elements of size *elsize* . The space is initialized to zeros.

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 no memory available, or if the area has been detectably corrupted by storing outside the bounds of a block. When this happens, the block indicated by *ptr* is neither damaged nor freed.

EXTENDED DESCRIPTION

ChorusOS 4.0 offers three `malloc( )` libraries. The following list describes each library:

`lib/classix/libcx.a`

The standard `malloc( )` for ChorusOS 4.0, based on the standard Solaris™ `libc` implementation, which has been extended to release freed memory to the system for use by the kernel and by other actors. However, calling `free( )` does not automatically return memory to the system. `malloc( )` takes memory chunks from page-aligned regions. Regions are only returned to the system once all the chunks in the region have been freed. Furthermore, `free( )` buffers memory chunks so that they can be reused immediately by `malloc( )` if possible. Therefore, memory may not be returned to the system until `malloc( )` is called again. `malloc_trim( )` can be used to release empty regions to the system explicitly.

`alloca( )` , `calloc( )` , `memalign( )` and `valloc( )` are not available in `lib/classix/libcx.a` .

`lib/classix/libleamalloc.a`

Doug Lea’s `malloc( )` , also known as the `libg++` `malloc( )` implementation, adapted for ChorusOS 4.0 to allow the heap to be sparsed in several regions. This implementation is especially useful in supervisor mode, because supervisor space is shared by several actors. Freed memory may be returned to the system using `malloc_trim( )` . `free( )` may also call `malloc_trim( )` if enough memory is free at the top of the heap.

`lib/classix/libomalloc.a`

The BSD `malloc( )` is provided for backwards compatibility with previous releases. This implementation corresponds to `bsdmalloc`(3X) in 2.6. See Solaris *man Pages(3): Library Routines* in the *Solaris 2.6 Reference Manual AnswerBook* for details.

NOTES

Performance and efficiency depend upon the way the library is used. Search time increases when many objects have been allocated; that is, if a program allocates but never frees, each successive allocation takes longer. Tests on the running program should be performed in order to determine the best balance between performance and efficient use of space to achieve optimum performance.

If the program is multi-threaded, and if the `free( )` and then `realloc( )` feature is used, it is up to the programmer to set up the mutual exclusion schemes needed to prevent a `malloc( )` taking place between `free( )` and `realloc( )` calls.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

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*);

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters 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* is not found.

The *memcmp* function compares its arguments, looking at the first `n` bytes (each interpreted as an unsigned only, and returns an integer less than, equal to, or greater than 0, depending on whether *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 area *s2* to *s1* . Copying between objects that overlap will take place correctly. It returns *s1* .

The *memset* function sets the first *n* characters in memory area *s* to the value of character *c.* It returns *s.*

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

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*);

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters 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* is not found.

The *memcmp* function compares its arguments, looking at the first `n` bytes (each interpreted as an unsigned only, and returns an integer less than, equal to, or greater than 0, depending on whether *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 area *s2* to *s1* . Copying between objects that overlap will take place correctly. It returns *s1* .

The *memset* function sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

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*);

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters 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* is not found.

The *memcmp* function compares its arguments, looking at the first n bytes (each interpreted as an unsigned only, and returns an integer less than, equal to, or greater than 0, depending on whether *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 area *s2* to *s1* . Copying between objects that overlap will take place correctly. It returns *s1* .

The *memset* function sets the first *n* characters in memory area *s* to the value of character *c.* It returns *s.*

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

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*);

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters 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* is not found.

The *memcmp* function compares its arguments, looking at the first `n` bytes (each interpreted as an unsigned only, and returns an integer less than, equal to, or greater than 0, depending on whether *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 area *s2* to *s1* . Copying between objects that overlap will take place correctly. It returns *s1* .

The *memset* function sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

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*);

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters 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* is not found.

The *memcmp* function compares its arguments, looking at the first n bytes (each interpreted as an unsigned only, and returns an integer less than, equal to, or greater than 0, depending on whether *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 area *s2* to *s1* . Copying between objects that overlap will take place correctly. It returns *s1* .

The *memset* function sets the first *n* characters in memory area *s* to the value of character *c.* It returns *s.*

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

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*);

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters 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* is not found.

The *memcmp* function compares its arguments, looking at the first n bytes (each interpreted as an unsigned only, and returns an integer less than, equal to, or greater than 0, depending on whether *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 area *s2* to *s1* . Copying between objects that overlap will take place correctly. It returns *s1* .

The *memset* function sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

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*);

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters 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* is not found.

The *memcmp* function compares its arguments, looking at the first `n` bytes (each interpreted as an unsigned only, and returns an integer less than, equal to, or greater than 0, depending on whether *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 area *s2* to *s1* . Copying between objects that overlap will take place correctly. It returns *s1* .

The *memset* function sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

mktemp, mkstemp – make temporary file name (unique)

SYNOPSIS

#include <stdlib.h>
char * **mktemp**(char * *template*);

int **mkstemp**(char * *template*);

DESCRIPTION

The *mktemp* function takes the given file name template and overwrites a portion of it to create a file name. This file name is unique and suitable for use by the application. The template may be any file name with a number of `X` 's appended to it, for example: `/tmp/temp.XXXX` The trailing `X` 's are replaced with the current process number and / or a unique letter combination. The number of unique file names *mktemp* can return depends on the number of `X` 's provided; six `X` 's will result in *mktemp* testing roughly 26 ** 6 combinations.

The *mkstemp* function makes the same replacement to the template and creates the template file, mode 0600, returning a file descriptor opened for reading and writing. This avoids conflict between testing for a file's existence and opening it for use.

RETURN VALUES

The *mktemp* function returns a pointer to the template on success and NULL on failure. The *mkstemp* function returns –1 if no suitable file could be created. If either call fails, the global variable *errno* is set to indicate one of the following error condiitons.

ERRORS

The *mktemp* and *mkstemp* functions will set *errno* to ENOTDIR if the pathname portion of the template is not an existing directory.

The *mktemp* and *mkstemp* functions can also set *errno* to any value specified by the *stat* (2POSIX) function.

The *mkstemp* function can also set *errno* to any value specified by the *open* (2POSIX) function.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chmod`(2POSIX) , `agetId`(2K) , `open`(2POSIX) , `stat`(2POSIX)

NAME

mktemp, mkstemp – make temporary file name (unique)

SYNOPSIS

```
#include <stdlib.h>
char * mktemp(char * template);

int mkstemp(char * template);
```

DESCRIPTION

The *mktemp* function takes the given file name template and overwrites a portion of it to create a file name. This file name is unique and suitable for use by the application. The template may be any file name with a number of `X` 's appended to it, for example: `/tmp/temp.XXXX` The trailing `X` 's are replaced with the current process number and / or a unique letter combination. The number of unique file names *mktemp* can return depends on the number of `X` 's provided; six `X` 's will result in *mktemp* testing roughly 26 ** 6 combinations.

The *mkstemp* function makes the same replacement to the template and creates the template file, mode 0600, returning a file descriptor opened for reading and writing. This avoids conflict between testing for a file's existence and opening it for use.

RETURN VALUES

The *mktemp* function returns a pointer to the template on success and NULL on failure. The *mkstemp* function returns –1 if no suitable file could be created. If either call fails, the global variable *errno* is set to indicate one of the following error condiitons.

ERRORS

The *mktemp* and *mkstemp* functions will set *errno* to ENOTDIR if the pathname portion of the template is not an existing directory.

The *mktemp* and *mkstemp* functions can also set *errno* to any value specified by the *stat* (2POSIX) function.

The *mkstemp* function can also set *errno* to any value specified by the *open* (2POSIX) function.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`chmod`(2POSIX) , `agetId`(2K) , `open`(2POSIX) , `stat`(2POSIX)

NAME

ctime, asctime, difftime, gmtime, localtime, mktime – transform binary date and time value to ASCII

SYNOPSIS

#include <time.h>
struct tm * **localtime**(const time_t * *clock*);

struct tm * **gmtime**(const time_t * *clock*);

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

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

time_t **mktime**(struct tm * *tm*);

double **difftime**(time_t *time1*, time_t *time0*);

DESCRIPTION

The *ctime* , *gmtime* and *localtime* functions take as an argument a time value representing the time in seconds since the Epoch (00:00:00 UTC, January 1, 1970).

The *localtime* function converts the time value pointed to by *clock* , and returns a pointer to a *struct tm* (described below) which contains the broken-out time information for the value, after adjusting for the current time zone (and any other factors such as Daylight Saving Time). Time zone adjustments are performed as specified by the TZ environment variable (see *tzset* (3STDC). The function *localtime* uses *tzset* (3STDC) to initialize time conversion information if *tzset* (3STDC) has not already been called by the process.

The *gmtime* function also converts the time value, but without any time zone adjustment, and returns a pointer to a *tm* structure (described below).

The *ctime* function adjusts the time value for the current time zone in the same manner as *localtime* , and returns a pointer to a 26-character string of the form: Thu Nov 24 18:22:48 1986.

The *asctime* function converts the broken—down time in the structure *tm* pointed to by **tm* to the form shown in the example above.

The *mktime* function converts the broken-down time, expressed as local time, in the structure pointed to by *tm* into a time value with the same encoding as that of the values returned by the `time` (3STDC) function; that is, seconds from the Epoch, UTC .

The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to their normal ranges. (A positive or zero value for *tm_isdst* causes *mktime* to presume initially that summer time (for example, Daylight Saving Time) is or is not in effect for the time specified, respectively. A negative value for *tm_isdst* causes the *mktime* function to attempt to define whether summer time is in effect for the time specified.)

On successful completion, the values of the *tm_wday* and *tm_yday* components of the structure are set appropriately, and the other components are set to represent the calendar time specified, but with their values forced to their normal ranges; the final value of *tm_mday* is not set until *tm_mon* and *tm_year* are determined. The *mktime* function returns the calendar time specified; if the calendar time cannot be represented, it returns –1;

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

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

```
int tm_sec;          /* seconds (0 - 60) */
int tm_min;          /* minutes (0 - 59) */
int tm_hour;         /* hours (0 - 23) */
int tm_mday;         /* day of month (1 - 31) */
int tm_mon;          /* month of year (0 - 11) */
int tm_year;         /* year – 1900 */
int tm_wday;         /* day of week (Sunday = 0) */
int tm_yday;         /* day of year (0 - 365) */
int tm_isdst;        /* is summer time in effect? */
char *tm_zone;       /* abbreviation of timezone name */
long tm_gmtoff;      /* offset from UTC in seconds */
```

The field *tm_isdst* is non-zero if summer time is in effect.

The field *tm_gmtoff* is the offset (in seconds) of the time represented from `UTC` , with positive values indicating east of the Prime Meridian.

NOTES

asctime(3STDC) , *ctime(3STDC)* , *localtime(3STDC)* and *gmtime(3STDC)* return their result in a global variable which make them difficult to use in a multithreaded program. *asctime_r(3STDC)* , *ctime_r(3STDC)* , *localtime_r(3STDC)* and *gmtime_r(3STDC)* should be used instead.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`asctime_r`(3STDC) , `ctime_r`(3STDC) , `getenv`(3STDC) , `gmtime_r`(3STDC) , `localtime_r`(3STDC) , `time`(3STDC) , `tzset`(3STDC)

NAME

modf – extract signed integral and fractional values from floating-point number

SYNOPSIS

```
#include <math.h>
double modf(double value, double *iptr);
```

DESCRIPTION

The *modf* function breaks the argument *value* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a *double* in the object pointed to by *iptr*.

RETURN VALUES

The *modf* function returns the signed fractional part of *value*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ldexp`(3STDC)

STANDARDS

The *modf* function conforms to `ANSI-C`.

NAME

byteorder, htonl, htons, ntohl, ntohs – convert values between host and network byte order

SYNOPSIS

```
#include <sys/param.h>
unsigned long htonl(unsigned long hostlong);

unsigned short htons(unsigned short hostshort);

unsigned long ntohl(unsigned long netlong);

unsigned short ntohs(unsigned short netshort);
```

DESCRIPTION

These routines convert 16– and 32–bit quantities between network byte order and host byte order. On architectures where the host byte order and network byte order are the same, these routines are defined as no-op macros.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

byteorder, htonl, htons, ntohl, ntohs – convert values between host and network byte order

SYNOPSIS

#include <sys/param.h>
unsigned long **`htonl`**(unsigned long *hostlong*);

unsigned short **`htons`**(unsigned short *hostshort*);

unsigned long **`ntohl`**(unsigned long *netlong*);

unsigned short **`ntohs`**(unsigned short *netshort*);

DESCRIPTION

These routines convert 16– and 32–bit quantities between network byte order and host byte order. On architectures where the host byte order and network byte order are the same, these routines are defined as no-op macros.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

perror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

#include <stdio.h>
void **perror**(const char * *s*);

```
#include <errno.h>
extern char *sys_errlist[];
extern int sys_nerr;
```

DESCRIPTION

The *perror* function produces a message on the error channel, the implementation of which is system-dependent. The message describes the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a newline character. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the per thread variable *errno*, or from a global variable *errno*, whichever is provided by the library. This variable is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new line. The *sys_nerr* parameter defines the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

printf, sprintf, snprintf, printerr – print formatted output

SYNOPSIS

#include <stdio.h>
int **printf**(const char * *format*, ... /* args */);

int **sprintf**(char * *s*, const char * *format*, ... /* args */);

int **snprintf**(char * *s*, size_t *size*, const char * *format*, ... /* args */);

int **printerr**(const char * *format*, ... /* args */);

DESCRIPTION

The printf function sends output to the standard output channel, which is system defined. The `printerr()` function sends output to on the standard error channel, which is system defined. The `sprintf()` function sends 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. Each function returns the number of characters transmitted (not including the `\0` in the case of *sprintf*), or a negative value if an output error was encountered.

The `snprintf()` function writes at most *size-1* of the characters printed to the output string (the *size* character then gets the terminating zero). If the return value is greater than or equal to the `size` argument, the string was too short and some of the printed characters were discarded.

Each of these functions converts, formats, and prints its *arg* s under control of the `format` . The `format` is a character string that contains two types of objects: plain characters, which are simply copied to the output channel, and conversion specifications, each of which results in obtaining zero or more *arg* s. The results are undefined if there are insufficient *arg* s for the format. If the format is exhausted while *arg* s remain, the excess *arg* s are simply ignored.

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 set) to the field width. If the field width for an s conversion is preceded by a 0, the string is right adjusted with zero-padding on the left.

A *precision* that gives the minimum number of digits to appear for the `d` , `o` , `u` , `x` , or `X` conversions, the number of digits to appear after the decimal point for the `e` and `f` conversions, the maximum number of significant digits for the `g` conversion, or the maximum number of characters to be printed from a string in an `s` conversion. The precision takes the form of a dot (`.`) followed by a decimal digit string; a null digit string is treated as zero.

An optional `l` (ell) specifying that a following `d` , `o` , `u` , `x` , or `X` conversion character applies to a long integer *arg* . A `l` before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision 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 *arg* s specifying field width or precision must appear *before* the *arg* (if any) to be converted.

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` , `s` , and `u` conversions, the flag has no effect. For `o` conversion, it increases the precision to force the first digit of the result to be a zero. For `x` or `X` conversion, a non-zero result will have `0x` or `0X` prefixed to it. For `e` , `E` , `f` , `g` , and `G` conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For `g` and `G` conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

`d` , `i` , `o` , `u` , `x` , `X`	The integer *arg* is converted to signed decimal (`d` or i `),` unsigned octal (`o`), decimal (`u`), or hexadecimal notation (`x` and `X`), respectively. The letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
`f`	The float or double *arg* is converted to decimal notation in the style "[`–`]ddd `.` ddd," where the number of digits after the decimal point is equal to the precision specification. If the precision is not specified, six digits are output; if the precision is explicitly 0, no decimal point appears.
`e` , `E`	The float or double *arg* is converted in the style "[`–`]d `.` ddd `e±` dd," where there is one digit before the decimal point and the number of digits after it is equal to the precision. If the precision is not specified, six digits are produced; if the precision is explicitly 0, no decimal point appears. The `E` format code will produce a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits.
`g` , `G`	The float or double *arg* is printed in style `f` or `e` (or in style `E` in the case of a `G` format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style `e` 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.
`c`	The character *arg* is printed.
`s`	The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (`\0`) is encountered, or until the number of characters indicated by the precision specification is reached. If the precision is not specified, it is assumed to be infinite and all

characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.

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

A non-existent or small field width will never 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` are printed in the same way as if *putchar* (3STDC) had been called.

EXAMPLES

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 %d, %d:%.2d", weekday, month, day, hour, min);
```

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`putchar`(3STDC) , `scanf`(3STDC)

NAME

printf, sprintf, snprintf, printerr – print formatted output

SYNOPSIS

```
#include <stdio.h>
int printf(const char * format, ...  /* args */);

int sprintf(char * s, const char * format, ...  /* args */);

int snprintf(char * s, size_t size, const char * format, ...  /* args */);

int printerr(const char * format, ...  /* args */);
```

DESCRIPTION

The printf function sends output to the standard output channel, which is system defined. The `printerr()` function sends output to on the standard error channel, which is system defined. The `sprintf()` function sends 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. Each function returns the number of characters transmitted (not including the `\0` in the case of *sprintf*), or a negative value if an output error was encountered.

The `snprintf()` function writes at most *size-1* of the characters printed to the output string (the *size* character then gets the terminating zero). If the return value is greater than or equal to the `size` argument, the string was too short and some of the printed characters were discarded.

Each of these functions converts, formats, and prints its *arg* s under control of the `format` . The `format` is a character string that contains two types of objects: plain characters, which are simply copied to the output channel, and conversion specifications, each of which results in obtaining zero or more *arg* s. The results are undefined if there are insufficient *arg* s for the format. If the format is exhausted while *arg* s remain, the excess *arg* s are simply ignored.

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 set) to the field width. If the field width for an s conversion is preceded by a 0, the string is right adjusted with zero-padding on the left.

A *precision* that gives the minimum number of digits to appear for the `d` , `o` , `u` , `x` , or `X` conversions, the number of digits to appear after the decimal point for the `e` and `f` conversions, the maximum number of significant digits for the `g` conversion, or the maximum number of characters to be printed from a string in an `s` conversion. The precision takes the form of a dot (`.`) followed by a decimal digit string; a null digit string is treated as zero.

An optional `l` (ell) specifying that a following `d` , `o` , `u` , `x` , or `X` conversion character applies to a long integer *arg* . A `l` before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision 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 *arg* s specifying field width or precision must appear *before* the *arg* (if any) to be converted.

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` , `s` , and `u` conversions, the flag has no effect. For `o` conversion, it increases the precision to force the first digit of the result to be a zero. For `x` or `X` conversion, a non-zero result will have `0x` or `0X` prefixed to it. For `e` , `E` , `f` , `g` , and `G` conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For `g` and `G` conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

`d` `,` `i` `,` `o` `,` `u` `,` `x` `,` `X`	The integer *arg* is converted to signed decimal (`d` or `i` `),` unsigned octal (`o`), decimal (`u`), or hexadecimal notation (`x` and `X`), respectively. The letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
`f`	The float or double *arg* is converted to decimal notation in the style "[`–`]ddd `.` ddd," where the number of digits after the decimal point is equal to the precision specification. If the precision is not specified, six digits are output; if the precision is explicitly 0, no decimal point appears.
`e` `,` `E`	The float or double *arg* is converted in the style "[`–`]d `.` ddd `e±` dd," where there is one digit before the decimal point and the number of digits after it is equal to the precision. If the precision is not specified, six digits are produced; if the precision is explicitly 0, no decimal point appears. The `E` format code will produce a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits.
`g` `,` `G`	The float or double *arg* is printed in style `f` or `e` (or in style `E` in the case of a `G` format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style `e` 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.
`c`	The character *arg* is printed.
`s`	The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (`\0`) is encountered, or until the number of characters indicated by the precision specification is reached. If the precision is not specified, it is assumed to be infinite and all

characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.

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

A non-existent or small field width will never 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` are printed in the same way as if *putchar* (3STDC) had been called.

EXAMPLES

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 %d, %d:%.2d", weekday, month, day, hour, min);
```

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`putchar`(3STDC) , `scanf`(3STDC)

NAME

putc, fputc, putw – put character or word on a stream

SYNOPSIS

#include <stdio.h>
int **putc**(int *c*, FILE * *stream*);

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

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

DESCRIPTION

The *putc* and *fputc* functions writes the byte specified by *c* (converted to an unsigned char) to the output stream (at the position where the file pointer, if defined, is pointing).

The *putw* function writes the specified int to the defined output stream.

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.

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* (3STDC)) will change 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 it is written. When it is buffered, a number characters are saved 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* (3STDC) or *setvbuf* (3STDC) function may be used to change the stream's buffering strategy.

RETURN VALUES

Upon successful completion, these functions each return the value they have written. If unsuccessful, they return the constant EOF. This will occur if the file *stream* is not open for writing or if the output file cannot be extended.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fclose`(3STDC) , `ferror`(3STDC) , `fopen`(3STDC) , `fread`(3STDC) , `printf`(3STDC) , `putchar`(3STDC) , `puts`(3STDC) , `setbuf`(3STDC)

NAME

putchar – put a character or word on the standard output channel

SYNOPSIS

#include <stdio.h>
int **putchar**(int *c*);

DESCRIPTION

The *putchar(c)* function writes the character *c* to the standard output channel, which is operating-system dependent. The effect of this operation outside of the program is operating-system defined. On systems where *stdout* has a meaning, *putc*(3STDC) is part of the C library, and *putchar*(*c*) is defined as *putc(c, stdout).*

DIAGNOSTICS

Upon successful completion, this function returns the value it has written. If unsuccessful, it returns the constant EOF. This will occur if the output channel can no longer be written to; possible reasons for this are operating-system dependent.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`putc`(3STDC)

NAME

unlocked, getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked – explicit locking functions

SYNOPSIS

#include <stdio.h>
int **getc_unlocked**(FILE * *stream*);

int **getchar_unlocked**(void);

int **putc_unlocked**(int *c*, FILE * *stream*);

int **putchar_unlocked**(int *c*);

DESCRIPTION

The *getc_unlocked* , *getchar_unlocked* , *putc_unlocked* and *putchar_unlocked* are functionally identical to *getc* , *getchar* , *putc* and *putchar* functions with the exception that they are not re-entrant.

getc_unlocked , *getchar_unlocked* , and *putchar_unlocked* routines are implemented as macros.

They may only safely be used within a scope protected by *flockfile* (or *ftrylockfile*) and *funlockedfile* .

STANDARDS

These routines conform to the POSIX.1c standards.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getc`(3STDC) , `getchar`(3STDC) , `putc`(3STDC) , `putchar`(3STDC) , `flockfile`(3STDC)

NAME

unlocked, getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked – explicit locking functions

SYNOPSIS

#include <stdio.h>
int **getc_unlocked**(FILE * *stream*);

int **getchar_unlocked**(void);

int **putc_unlocked**(int *c*, FILE * *stream*);

int **putchar_unlocked**(int *c*);

DESCRIPTION

The *getc_unlocked* , *getchar_unlocked* , *putc_unlocked* and *putchar_unlocked* are functionally identical to *getc* , *getchar* , *putc* and *putchar* functions with the exception that they are not re-entrant.

getc_unlocked , *getchar_unlocked* , and *putchar_unlocked* routines are implemented as macros.

They may only safely be used within a scope protected by *flockfile* (or *ftrylockfile*) and *funlockedfile* .

STANDARDS

These routines conform to the POSIX.1c standards.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getc`(3STDC) , `getchar`(3STDC) , `putc`(3STDC) , `putchar`(3STDC) , `flockfile`(3STDC)

NAME

getenv, putenv, setenv, unsetenv – fetch and set environment variables

SYNOPSIS

#include <stdlib.h>
char * **getenv**(const char * *name*);

int **setenv**(const char * *name*, const char * *value*, int *overwrite*);

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

void **unsetenv**(const char * *name*);

DESCRIPTION

These functions set, unset and fetch environment variables from the host *environment* list. For compatibility with differing environment conventions, the *name* and *value* arguments given may be appended and prepended, respectively, with an equal sign. The *getenv* function obtains the current value of the environment variable, *name*. If the variable *name* is not in the current environment, a null pointer is returned.

The `setenv` function inserts or resets the environment variable *name* in the current environment list. If the variable *name* does not exist in the list, it is inserted with the given *value*. If the variable does exist, the *overwrite* argument is tested; if *overwrite* is zero, the variable is not reset, otherwise it is reset to the given *value*.

The *putenv* function takes an argument of the form `name=value` and is equivalent to: `setenv(name, value, 1)`.

The `unsetenv` function deletes all instances of the variable name pointed to by *name* from the list.

RETURN VALUES

The `setenv` and *putenv* functions return zero if successful; otherwise –1 is returned. The `setenv` or *putenv* functions fail if they were unable to allocate memory for the environment.

STANDARDS

The *getenv* function conforms to `ANSI-C`.

NOTE

These functions are reentrant, but the environment is global to the actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

puts, fputs – put a string on a stream

SYNOPSIS

```
#include <stdio.h>
int puts(const char * s);

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

DESCRIPTION

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

The *fputs* function writes the null-terminated string pointed to by *s* to the named output *stream* .

Neither function writes the terminating null character.

RETURN VALUES

Both routines return EOF on error. This will happen if the routines try to write to a file that has not been opened for writing.

NOTES

The *puts* appends a new-line character while *fputs* does not.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ferror`(3STDC) , `fopen`(3STDC) , `fread`(3STDC) , `printf`(3STDC) , `putc`(3STDC)

NAME

putc, fputc, putw – put character or word on a stream

SYNOPSIS

#include <stdio.h>
int **putc**(int *c*, FILE * *stream*);

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

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

DESCRIPTION

The *putc* and *fputc* functions writes the byte specified by *c* (converted to an unsigned char) to the output stream (at the position where the file pointer, if defined, is pointing).

The *putw* function writes the specified int to the defined output stream.

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.

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* (3STDC)) will change 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 it is written. When it is buffered, a number characters are saved 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* (3STDC) or *setvbuf* (3STDC) function may be used to change the stream's buffering strategy.

RETURN VALUES

Upon successful completion, these functions each return the value they have written. If unsuccessful, they return the constant EOF. This will occur if the file *stream* is not open for writing or if the output file cannot be extended.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fclose`(3STDC) , `ferror`(3STDC) , `fopen`(3STDC) , `fread`(3STDC) , `printf`(3STDC) , `putchar`(3STDC) , `puts`(3STDC) , `setbuf`(3STDC)

NAME

qsort – quicker 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 quicker-sort algorithm; it sorts a table of data. The contents of the table are sorted in ascending order according to a user-supplied comparison function.

The *base* pointer indicates the element at the base of the table. *nel* is the number of elements in the table., and *width* specifies the size of each element in bytes. The name of the comparison function, *compar*, 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.

NOTES

The comparison function need not compare every byte. Arbitrary data may be contained in the elements in addition to the values being compared. If two items compare as equal, the order of output is unpredictable.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`bsearch`(3STDC), `string`(3STDC)

NAME

rand, srand – pseudo random number generator

SYNOPSIS

```
#include <stdlib.h>
void srand(unsigned seed);

int rand(void);
```

DESCRIPTION

The *rand* function computes a sequence of pseudo-random integers in the range of 0 to RAND_MAX (as defined by the header file stdlib.h).

The *srand* function sets its argument as the seed for a new sequence of pseudo-random numbers to be returned by *rand* . These sequences are repeatable by calling *srand* with the same seed value.

If no seed value is provided, the functions are automatically seeded with a value of 1.

NOTE

Though these functions are reentrant, the state information is global to the actor. Therefore, repeatability of a given suite of numbers will not be experienced by several threads in parallel. For a reentrant repeatability of suites, see *rand_r(3STDC)* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`random`(3STDC) , `rand_r`(3STDC)

STANDARDS

The *rand* and *srand* functions conform to ANSI-C.

NAME

random, srandom, initstate, setstate – better random number generator

SYNOPSIS

#include <stdlib.h>
long **random**(void);

void **srandom**(unsigned *seed*);

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

char ***setstate**(char * *state*);

DESCRIPTION

The *random* function uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31} - 1$. The period of this random number generator is very large, approximately $16 \times (2^{31} - 1)$.

The *random/srandom* functions have (almost) the same calling sequence and initialization properties as *rand/srand* (3STDC) The difference is that *rand* produces a much less random sequence — in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by *random* are usable. For example, *random* `&01` will produce a random binary value.

Unlike *srand* , *srandom* does not return the old seed; the reason being that the amount of state information used is much more than a single word (two other routines are provided to deal with restarting / changing random number generators). Like *rand* , however, *random* will by default produce a sequence of numbers that can be duplicated by calling *srandom* with 1 as the seed.

The *initstate* routine allows a state array, passed as an argument, to be initialized for future use. The size of the state array (in bytes) is used by *initstate* to decide how sophisticated a random number generator it should use — the bigger the state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error.) The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. The *initstate* function returns a pointer to the previous state information array.

Once a state has been initialized, the *setstate* routine provides for rapid switching between states. The *setstate* function returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to *initstate* or *setstate* .

Once a state array has been initialized, it may be restarted at a different point either by calling *initstate* (with the desired seed, the state array, and its size) or by calling both *setstate* (with the state array) and *srandom* (with the desired seed).

The advantage of calling both *setstate* and *srandom* is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{690}, which should be sufficient for most purposes.

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` , it returns `NULL` and *random* uses a simple linear congruential random number generator.

DIAGNOSTICS

If *initstate* is called with less than 8 bytes of state information, or if *setstate* detects that the state information has been garbled, error messages are printed to the standard error output.

NOTE

Though these functions are reentrant, the state information is global to the actor. Therefore, repeatability of a given suite of number will not be experienced by several threads in parallel. For a reentrent repeatability of suites, see *rand_r(3STDC)* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rand`(3STDC) , `rand_r`(3STDC)

RESTRICTIONS

random operates at about 2/3 the speed of *rand* (3STDC).

NAME

rand_r – thread-wise random number generator

SYNOPSIS

#include <stdlib.h>
int **rand_r**(unsigned int **seed*);

DESCRIPTION

The *rand_r* function computes a sequence of pseudo-random integers in the range of 0 to RAND_MAX (as defined by the header file stdlib.h) .

The status is stored in the application space, and its address is given to *rand_r* via the *seed* parameter. It is not mandatory to initialize **seed* but it can be reset to an arbitrary value at any time. Each particular value will lead to a particular suite of *rand_r* results. This suite is the same as that produced by calling *srand(3STDC)* once with the initial **seed* value, and then calling *rand(3STDC)* repeatedly. By allocating **seed* in its stack, each thread can have its own repeatable suite of numbers.

STANDARDS

The *rand_r* function conforms to `POSIX.1c` standards.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`random`(3STDC), `rand`(3STDC)

NAME

malloc, free, realloc, calloc – main memory allocator

SYNOPSIS

#include <stdlib.h>
void * **malloc**(size_t *size*);

void **free**(void * *ptr*);

void * **realloc**(void * *ptr*, size_t *size*);

void ***calloc**(size_t *nelem*, size_t *elsize*);

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. ChorusOS 4.0 offers three `malloc( )` libraries. See *EXTENDED DESCRIPTION* below for details.

The argument passed to `free( )` is a pointer to a block previously allocated by `malloc( )` ; after `free( )` is performed this space is made available for further allocation, but its contents are left undisturbed.

The `free( )` function may be called with a `NULL` pointer as parameter.

If the space assigned by `malloc( )` is overrun or if a random number is passed to `free( )` , the result is undefined.

The `malloc( )` function searches for free space from the last block allocated or freed, grouping together any adjacent free blocks. It allocates the first contiguous area of free space that is at least `size( )` bytes.

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 smaller of the new and old sizes. If no free block of *size* bytes is available in the storage area, `realloc( )` will ask `malloc( )` to enlarge the area by *size* bytes and will then move the data to the new space. If the space cannot be allocated, the object pointed to by *ptr* is unchanged. If *size* is zero and *ptr* is not a null pointer, the object it points to is freed. If *ptr* is a null pointer, the `realloc( )` function behaves like the `malloc( )` function for the specified size.

The `realloc( )` function also works if *ptr* points to a block freed since the last call to `malloc( )` , `realloc( )` , or `calloc( )` ; thus sequences of `free( )` , `malloc( )` and `realloc( )` can be used to exploit the search strategy of `malloc( )` in order to do storage compacting.

The `calloc( )` function allocates space for an array of *nelem* elements of size *elsize* . The space is initialized to zeros.

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 no memory available, or if the area has been detectably corrupted by storing outside the bounds of a block. When this happens, the block indicated by *ptr* is neither damaged nor freed.

EXTENDED DESCRIPTION

ChorusOS 4.0 offers three `malloc()` libraries. The following list describes each library:

`lib/classix/libcx.a`
: The standard `malloc()` for ChorusOS 4.0, based on the standard Solaris™ `libc` implementation, which has been extended to release freed memory to the system for use by the kernel and by other actors. However, calling `free()` does not automatically return memory to the system. `malloc()` takes memory chunks from page-aligned regions. Regions are only returned to the system once all the chunks in the region have been freed. Furthermore, `free()` buffers memory chunks so that they can be reused immediately by `malloc()` if possible. Therefore, memory may not be returned to the system until `malloc()` is called again. `malloc_trim()` can be used to release empty regions to the system explicitly.

  `alloca()`, `calloc()`, `memalign()` and `valloc()` are not available in `lib/classix/libcx.a`.

`lib/classix/libleamalloc.a`
: Doug Lea's `malloc()`, also known as the `libg++` `malloc()` implementation, adapted for ChorusOS 4.0 to allow the heap to be sparsed in several regions. This implementation is especially useful in supervisor mode, because supervisor space is shared by several actors. Freed memory may be returned to the system using `malloc_trim()`. `free()` may also call `malloc_trim()` if enough memory is free at the top of the heap.

`lib/classix/libomalloc.a`
: The BSD `malloc()` is provided for backwards compatibility with previous releases. This implementation corresponds to `bsdmalloc`(3X) in 2.6. See Solaris *man Pages(3): Library Routines* in the *Solaris 2.6 Reference Manual AnswerBook* for details.

NOTES

Performance and efficiency depend upon the way the library is used. Search time increases when many objects have been allocated; that is, if a program allocates but never frees, each successive allocation takes longer. Tests on the running program should be performed in order to determine the best balance between performance and efficient use of space to achieve optimum performance.

If the program is multi-threaded, and if the `free()` and then `realloc()` feature is used, it is up to the programmer to set up the mutual exclusion schemes needed to prevent a `malloc()` taking place between `free()` and `realloc()` calls.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

regex, regcomp, regexec, regerror, regfree – regular-expression library

SYNOPSIS

#include <sys/types.h>
#include <regex.h>
int **regcomp**(regex_t * *preg*, const char * *pattern*, int *cflags*);

int **regexec**(constregex_t * *preg*, constchar * *string*, size_t *nmatch*, regmatch_tp *match* [], int *eflags*);

size_t **regerror**(int *errcode*, constregex_t * *preg*, char * *errbuf*, size_t *errbuf_size*);

void **regfree**(regex_t * *preg*);

FEATURES

STDC

DESCRIPTION

These routines implement POSIX 1003.2 regular expressions ("RE"s); see the SEE ALSO section below. The *regcomp* function compiles an RE written as a string into an internal form, *regexec* matches that internal form against a string and reports results, *regerror* transforms error codes from either into human-readable messages, and *regfree* frees any dynamically-allocated storage used by the internal form of an RE.

The header <*regex.h*> declares two structure types, *regex_t* and *regmatch_t* , the former for compiled internal forms and the latter for match reporting. It also declares the four functions, a type *regoff_t* , and a number of constants with names starting with "REG_".

The *regcomp* function compiles the regular expression contained in the *pattern* string, subject to the flags in *cflags* , and places the results in the *regex_t* structure pointed to by *preg* . The *cflags* parameter is the bitwise OR of zero or one or more of the following flags:

REG_EXTENDED	Compile modern ("extended") REs, rather than the obsolete ("basic") REs that are the default.
REG_BASIC	This is a synonym for 0, provided as a counterpart to REG_EXTENDED to improve readability.
REG_NOSPEC	Compile with recognition of all special characters turned off. All characters are thus considered ordinary, so the "RE" is a literal string. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. REG_EXTENDED and REG_NOSPEC may not be used in the same call to *regcomp* .

REG_ICASE	Compile for matching that ignores upper/lower case distinctions. See the citation in the SEE ALSO section below
REG_NOSUB	Compile for matching that need only report success or failure, not what was matched.
REG_NEWLINE	Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, '[^' bracket expressions and '.' never match newline, a '^' anchor matches the null string after any newline in the string in addition to its normal function, and the '$' anchor matches the null string before any newline in the string in addition to its normal function.
REG_PEND	The regular expression ends, not at the first NULL, but just before the character pointed to by the *re_endp* member of the structure pointed to by *preg* . The *re_endp* member is of the type *const char* * . This flag permits inclusion of NULs in the RE; they are considered ordinary characters. This is an extension, compatible with, but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

When successful, *regcomp* returns 0 and fills in the structure pointed to by *preg* . One member of that structure (other than *re_endp*) is published: *re_nsub* , of type *size_t* , contains the number of parenthesized subexpressions within the RE (except that the value of this member is undefined if the REG_NOSUB flag was used). If *regcomp* fails, it returns a non-zero error code; see DIAGNOSTICS.

The *regexec* function matches the compiled RE pointed to by *preg* against the *string* , subject to the flags in *eflags* , and reports results using *nmatch* , *pmatch* , and the returned value. The RE must have been compiled using a previous invocation of *regcomp* . The compiled form is not altered during execution of *regexec* , a single compiled RE can therefore be used simultaneously by multiple threads.

By default, the NUL-terminated string pointed to by *string* is considered to be the text of an entire line, minus any terminating newline. The *eflags* argument is the bitwise OR of zero or one or more of the following flags:

REG_NOTBOL	The first character of the string is not the beginning of a line, so the '^' anchor should not

	match before it. This does not affect the behavior of newlines under REG_NEWLINE.
REG_NOTEOL	The NUL terminating the string does not end a line, so the '$' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.
REG_STARTEND	The string is considered to start at *string* + *pmatch* [0]. *rm_so* and to have a terminating NUL located at *string* + *pmatch* [0]. *rm_eo* (there need not actually be a NUL at that location), regardless of the value of *nmatch* . See below for the definition of *pmatch* and *nmatch* . This is an extension, compatible with, but not specified by, POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Note that a non-zero *rm_so* does not imply REG_NOTBOL; REG_STARTEND affects only the location of the string, not how it is matched.

See the citation in the SEE ALSO SECTION for an explanation of what is matched in situations where an RE or a portion thereof could match any of several substrings of *string* .

Normally, *regexec* returns 0 for success and the non-zero code REG_NOMATCH for failure. Other non-zero error codes may be returned in exceptional situations; see DIAGNOSTICS.

If REG_NOSUB was specified in the compilation of the RE, or if *nmatch* is 0, *regexec* ignores the *pmatch* argument (see below for the case where REG_STARTEND is specified). Otherwise, *pmatch* points to an array of *nmatch* structures of the type *regmatch_t* . This a structure has at least the members *rm_so* and *rm_eo* , both of type *regoff_t* (a signed arithmetic type at least as large as an *off_t* and a *ssize_t*), containing respectively the offset of the first character of a substring and the offset of the first character after the end of the substring. Offsets are measured from the beginning of the *string* argument given to *regexec* . An empty substring is denoted by equal offsets, both indicating the character following the empty substring.

The 0th member of the *pmatch* array is filled in to indicate what substring of *string* was matched by the entire RE. Remaining members report what substring was matched by parenthesized subexpressions within the RE; member *i* reports subexpression *i* , with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. Unused entries in the array—corresponding either to subexpressions that did not participate in the match at all, or to subexpressions that do not exist in the RE (that is, *i* > *preg* –>

re_nsub)—have both *rm_so* and *rm_eo* set to –1. If a subexpression participated in the match several times, the substring reported is the last one it matched. (Note that, when the RE '(b*)+' matches 'bbb', the parenthesized subexpression matches each of the three 'b's and then an infinite number of empty strings following the last 'b', the substring reported is therefore empty.)

If REG_STARTEND is specified, *pmatch* must point to at least one *regmatch_t* (even if *nmatch* is 0 or REG_NOSUB was specified), to hold the input offsets for REG_STARTEND. Use for output is still entirely controlled by *nmatch* ; if *nmatch* is 0 or REG_NOSUB was specified, the value of *pmatch* [0] will not be changed by a successful *regexec* .

The *regerror* function maps a non-zero *errcode* from either *regcomp* or *regexec* to a human-readable, printable message. If *preg* is non-NULL, the error code should have arisen from use of the *regex_t* pointed to by *preg* , and if the error code came from *regcomp* , it should have been the result of the most recent *regcomp* using that *regex_t* . (*Regerror* may be able to supply a more detailed message using information from the *regex_t* .) The *regerror* function places the NUL-terminated message into the buffer pointed to by *errbuf* , limiting the length (including the NUL) to at most *errbuf_size* bytes. If the whole message won't fit, as much of it as will fit before the terminating NUL is supplied. The value returned is the size of buffer needed to hold the whole message (including the terminating NULL). If *errbuf_size* is 0, *errbuf* is ignored but the return value is still correct.

If the *errcode* given to *regerror* is first ORed with REG_ITOA, the "message" that results is the printable name of the error code, for example, "REG_NOMATCH", rather than an explanation of it If *errcode* is REG_ATOI, *preg* will be non-NULL and the *re_endp* member of the structure it points to must point to the printable name of an error code; in this case, the result in *errbuf* is the decimal digits of the numeric value of the error code (0 if the name is not recognized). The REG_ITOA and REG_ATOI functions are intended primarily as debugging facilities; they are extensions, compatible with, but not specified by, POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Be warned also that they are considered experimental and changes are possible.

The *regfree* function frees any dynamically-allocated storage associated with the compiled RE pointed to by *preg* . The remaining *regex_t* is no longer a valid compiled RE and the effect of supplying it to *regexec* or *regerror* is undefined.

None of these functions references global variables except for tables of constants; all are safe for use by multiple threads if the arguments are safe.

IMPLEMENTATION CHOICES

There are a number of decisions that 1003.2 leaves up to the implementor, either by explicitly saying "undefined" or by virtue of them being forbidden by the RE grammar. This implementation treats them as follows.

See the citation in the SEE ALSO section for an explanation of the definition of case-independent matching.

There is no particular limit to the length of REs, apart from memory limitations. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See BUGS for one short RE using them that will run almost any system out of memory.

A backslashed character other than one specifically given a magic meaning by 1003.2 (such magic meanings occur only in obsolete [“basic”] REs) is taken as an ordinary character.

Any unmatched [is a REG_EBRACK error.

Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.

The RE_DUP_MAX option defines the limit on repetition counts in bounded repetitions, the maximum is 255.

A repetition operator (?, *, +, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow ‘^’ or ‘|’.

The pipe symbol. (‘|’) cannot appear first or last in a (sub)expression or after another ‘|’, in other words, an operand of ‘|’ cannot be an empty subexpression. An empty parenthesized subexpression, ‘()’, is legal and matches an empty (sub)string. An empty string is not a legal RE.

A brace (“{”) followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A ‘{’ *not* followed by a digit is considered an ordinary character.

A circumflex (’”‘^”) and dollar sign (“$”) beginning and ending subexpressions in obsolete (“basic”) REs are anchors, not ordinary characters.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

grep(1UNIX), re_format(7UNIX)

POSIX 1003.2, sections 2.8 (Regular Expression Notation) and B.5 (C Binding for Regular Expression Matching).

DIAGNOSTICS

Non-zero error codes from *regcomp* and *regexec* include the following:

```
REG_NOMATCH regexec() failed to match
REG_BADPAT invalid regular expression
```

```
REG_ECOLLATE invalid collating element
REG_ECTYPE invalid character class
REG_EESCAPE \ applied to unescapable character
REG_ESUBREG invalid backreference number
REG_EBRACK brackets [ ] not balanced
REG_EPAREN parentheses ( ) not balanced
REG_EBRACE braces { } not balanced
REG_BADBR invalid repetition count(s) in { }
REG_ERANGE invalid character range in [ ]
REG_ESPACE ran out of memory
REG_BADRPT ?, *, or + operand invalid
REG_EMPTY empty (sub)expression
REG_ASSERT ‘‘can’t happen’’—you found a bug
REG_INVARG invalid argument, for example,  negative-length string
```

HISTORY

Originally written by Henry Spencer. Altered for inclusion in the 4.4BSD distribution.

BUGS

This is an alpha release with known defects. Please report problems.

There is one known functionality bug. The implementation of internationalization is incomplete: the locale is always assumed to be the default one of 1003.2, therefore, only information pertaining to that locale is available.

The back-reference code is subtle and there are doubts about its correctness in complex cases.

The *regexec* function’s performance is poor. This will improve with later releases. An *nmatch* exceeding 0 is expensive; *nmatch* exceeding 1 is worse. The *regexec* function is largely insensitive to RE complexity *except* that back references are extremely expensive. RE length does matter; in particular, there is an apprecialbe speed bonus for keeping RE length under approximately 30 characters, most special characters are worth roughly double.

The *regcomp* function implements bounded repetitions using expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like the following, ((((a{1,100}){1,100}){1,100}’ will (eventually) run almost any existing machine out of swap space.

There are suspected problems with responses to obscure error conditions. Notably, certain kinds of internal overflow, produced only by extremely large REs or by multiply—nested bounded repetitions, are probably not handled well.

Due to a mistake in 1003.2, things like ‘a)b’ are legal REs because ‘)’ is a special character only in the presence of a previous unmatched ‘(’. This can’t be fixed until the spec is fixed.

The standard’s definition of back references is vague. For example, does ‘a\(\(b\)*\2\)*d’ match ‘abbbd’? Until the standard is clarified, behavior in such cases should not be relied on.

The implementation of word-boundary matching is imprecise, and bugs may lurk in combinations of word-boundary matching and anchoring.

RESTRICTIONS

As part of the BSD library, this function is not *thread-safe* .

NAME

regex, regcomp, regexec, regerror, regfree – regular-expression library

SYNOPSIS

#include <sys/types.h>
#include <regex.h>
int **regcomp**(regex_t * *preg*, const char * *pattern*, int *cflags*);

int **regexec**(constregex_t * *preg*, constchar * *string*, size_t *nmatch*, regmatch_tp *match* [], int *eflags*);

size_t **regerror**(int *errcode*, constregex_t * *preg*, char * *errbuf*, size_t *errbuf_size*);

void **regfree**(regex_t * *preg*);

FEATURES

STDC

DESCRIPTION

These routines implement POSIX 1003.2 regular expressions (“RE”s); see the SEE ALSO section below. The *regcomp* function compiles an RE written as a string into an internal form, *regexec* matches that internal form against a string and reports results, *regerror* transforms error codes from either into human-readable messages, and *regfree* frees any dynamically-allocated storage used by the internal form of an RE.

The header <*regex.h*> declares two structure types, *regex_t* and *regmatch_t* , the former for compiled internal forms and the latter for match reporting. It also declares the four functions, a type *regoff_t* , and a number of constants with names starting with “REG_”.

The *regcomp* function compiles the regular expression contained in the *pattern* string, subject to the flags in *cflags* , and places the results in the *regex_t* structure pointed to by *preg* . The *cflags* parameter is the bitwise OR of zero or one or more of the following flags:

REG_EXTENDED	Compile modern (“extended”) REs, rather than the obsolete (“basic”) REs that are the default.
REG_BASIC	This is a synonym for 0, provided as a counterpart to REG_EXTENDED to improve readability.
REG_NOSPEC	Compile with recognition of all special characters turned off. All characters are thus considered ordinary, so the “RE” is a literal string. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. REG_EXTENDED and REG_NOSPEC may not be used in the same call to *regcomp* .

REG_ICASE	Compile for matching that ignores upper/lower case distinctions. See the citation in the SEE ALSO section below
REG_NOSUB	Compile for matching that need only report success or failure, not what was matched.
REG_NEWLINE	Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, '[^' bracket expressions and '.' never match newline, a '^' anchor matches the null string after any newline in the string in addition to its normal function, and the '$' anchor matches the null string before any newline in the string in addition to its normal function.
REG_PEND	The regular expression ends, not at the first NULL, but just before the character pointed to by the *re_endp* member of the structure pointed to by *preg* . The *re_endp* member is of the type *const char ** . This flag permits inclusion of NULs in the RE; they are considered ordinary characters. This is an extension, compatible with, but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

When successful, *regcomp* returns 0 and fills in the structure pointed to by *preg* . One member of that structure (other than *re_endp*) is published: *re_nsub* , of type *size_t* , contains the number of parenthesized subexpressions within the RE (except that the value of this member is undefined if the REG_NOSUB flag was used). If *regcomp* fails, it returns a non-zero error code; see DIAGNOSTICS.

The *regexec* function matches the compiled RE pointed to by *preg* against the *string* , subject to the flags in *eflags* , and reports results using *nmatch* , *pmatch* , and the returned value. The RE must have been compiled using a previous invocation of *regcomp* . The compiled form is not altered during execution of *regexec* , a single compiled RE can therefore be used simultaneously by multiple threads.

By default, the NUL-terminated string pointed to by *string* is considered to be the text of an entire line, minus any terminating newline. The *eflags* argument is the bitwise OR of zero or one or more of the following flags:

REG_NOTBOL	The first character of the string is not the beginning of a line, so the '^' anchor should not

	match before it. This does not affect the behavior of newlines under REG_NEWLINE.
REG_NOTEOL	The NUL terminating the string does not end a line, so the '$' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.
REG_STARTEND	The string is considered to start at *string* + *pmatch* [0]. *rm_so* and to have a terminating NUL located at *string* + *pmatch* [0]. *rm_eo* (there need not actually be a NUL at that location), regardless of the value of *nmatch* . See below for the definition of *pmatch* and *nmatch* . This is an extension, compatible with, but not specified by, POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Note that a non-zero *rm_so* does not imply REG_NOTBOL; REG_STARTEND affects only the location of the string, not how it is matched.

See the citation in the SEE ALSO SECTION for an explanation of what is matched in situations where an RE or a portion thereof could match any of several substrings of *string* .

Normally, *regexec* returns 0 for success and the non-zero code REG_NOMATCH for failure. Other non-zero error codes may be returned in exceptional situations; see DIAGNOSTICS.

If REG_NOSUB was specified in the compilation of the RE, or if *nmatch* is 0, *regexec* ignores the *pmatch* argument (see below for the case where REG_STARTEND is specified). Otherwise, *pmatch* points to an array of *nmatch* structures of the type *regmatch_t* . This a structure has at least the members *rm_so* and *rm_eo* , both of type *regoff_t* (a signed arithmetic type at least as large as an *off_t* and a *ssize_t*), containing respectively the offset of the first character of a substring and the offset of the first character after the end of the substring. Offsets are measured from the beginning of the *string* argument given to *regexec* . An empty substring is denoted by equal offsets, both indicating the character following the empty substring.

The 0th member of the *pmatch* array is filled in to indicate what substring of *string* was matched by the entire RE. Remaining members report what substring was matched by parenthesized subexpressions within the RE; member *i* reports subexpression *i* , with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. Unused entries in the array—corresponding either to subexpressions that did not participate in the match at all, or to subexpressions that do not exist in the RE (that is, *i* > *preg* –>

re_nsub)—have both *rm_so* and *rm_eo* set to –1. If a subexpression participated in the match several times, the substring reported is the last one it matched. (Note that, when the RE '(b*)+' matches 'bbb', the parenthesized subexpression matches each of the three 'b's and then an infinite number of empty strings following the last 'b', the substring reported is therefore empty.)

If REG_STARTEND is specified, *pmatch* must point to at least one *regmatch_t* (even if *nmatch* is 0 or REG_NOSUB was specified), to hold the input offsets for REG_STARTEND. Use for output is still entirely controlled by *nmatch* ; if *nmatch* is 0 or REG_NOSUB was specified, the value of *pmatch* [0] will not be changed by a successful *regexec* .

The *regerror* function maps a non-zero *errcode* from either *regcomp* or *regexec* to a human-readable, printable message. If *preg* is non-NULL, the error code should have arisen from use of the *regex_t* pointed to by *preg* , and if the error code came from *regcomp* , it should have been the result of the most recent *regcomp* using that *regex_t* . (*Regerror* may be able to supply a more detailed message using information from the *regex_t* .) The *regerror* function places the NUL-terminated message into the buffer pointed to by *errbuf* , limiting the length (including the NUL) to at most *errbuf_size* bytes. If the whole message won't fit, as much of it as will fit before the terminating NUL is supplied. The value returned is the size of buffer needed to hold the whole message (including the terminating NULL). If *errbuf_size* is 0, *errbuf* is ignored but the return value is still correct.

If the *errcode* given to *regerror* is first ORed with REG_ITOA, the "message" that results is the printable name of the error code, for example, "REG_NOMATCH", rather than an explanation of it If *errcode* is REG_ATOI, *preg* will be non-NULL and the *re_endp* member of the structure it points to must point to the printable name of an error code; in this case, the result in *errbuf* is the decimal digits of the numeric value of the error code (0 if the name is not recognized). The REG_ITOA and REG_ATOI functions are intended primarily as debugging facilities; they are extensions, compatible with, but not specified by, POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Be warned also that they are considered experimental and changes are possible.

The *regfree* function frees any dynamically-allocated storage associated with the compiled RE pointed to by *preg* . The remaining *regex_t* is no longer a valid compiled RE and the effect of supplying it to *regexec* or *regerror* is undefined.

None of these functions references global variables except for tables of constants; all are safe for use by multiple threads if the arguments are safe.

IMPLEMENTATION CHOICES

There are a number of decisions that 1003.2 leaves up to the implementor, either by explicitly saying "undefined" or by virtue of them being forbidden by the RE grammar. This implementation treats them as follows.

See the citation in the SEE ALSO section for an explanation of the definition of case-independent matching.

There is no particular limit to the length of REs, apart from memory limitations. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See BUGS for one short RE using them that will run almost any system out of memory.

A backslashed character other than one specifically given a magic meaning by 1003.2 (such magic meanings occur only in obsolete [“basic”] REs) is taken as an ordinary character.

Any unmatched [is a REG_EBRACK error.

Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.

The RE_DUP_MAX option defines the limit on repetition counts in bounded repetitions, the maximum is 255.

A repetition operator (?, *, +, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow ‘^’ or ‘|’.

The pipe symbol. (‘|’) cannot appear first or last in a (sub)expression or after another ‘|’, in other words, an operand of ‘|’ cannot be an empty subexpression. An empty parenthesized subexpression, ‘()’, is legal and matches an empty (sub)string. An empty string is not a legal RE.

A brace (“{”) followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A ‘{’ *not* followed by a digit is considered an ordinary character.

A circumflex (’”‘^”) and dollar sign (“$”) beginning and ending subexpressions in obsolete (“basic”) REs are anchors, not ordinary characters.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

grep(1UNIX), re_format(7UNIX)

POSIX 1003.2, sections 2.8 (Regular Expression Notation) and B.5 (C Binding for Regular Expression Matching).

DIAGNOSTICS

Non-zero error codes from *regcomp* and *regexec* include the following:

```
REG_NOMATCH regexec() failed to match
REG_BADPAT invalid regular expression
```

```
REG_ECOLLATE invalid collating element
REG_ECTYPE invalid character class
REG_EESCAPE \ applied to unescapable character
REG_ESUBREG invalid backreference number
REG_EBRACK brackets [ ] not balanced
REG_EPAREN parentheses ( ) not balanced
REG_EBRACE braces { } not balanced
REG_BADBR invalid repetition count(s) in { }
REG_ERANGE invalid character range in [ ]
REG_ESPACE ran out of memory
REG_BADRPT ?, *, or + operand invalid
REG_EMPTY empty (sub)expression
REG_ASSERT ''can't happen''—you found a bug
REG_INVARG invalid argument, for example,  negative-length string
```

HISTORY

Originally written by Henry Spencer. Altered for inclusion in the 4.4BSD distribution.

BUGS

This is an alpha release with known defects. Please report problems.

There is one known functionality bug. The implementation of internationalization is incomplete: the locale is always assumed to be the default one of 1003.2, therefore, only information pertaining to that locale is available.

The back-reference code is subtle and there are doubts about its correctness in complex cases.

The *regexec* function's performance is poor. This will improve with later releases. An *nmatch* exceeding 0 is expensive; *nmatch* exceeding 1 is worse. The *regexec* function is largely insensitive to RE complexity *except* that back references are extremely expensive. RE length does matter; in particular, there is an apprecialbe speed bonus for keeping RE length under approximately 30 characters, most special characters are worth roughly double.

The *regcomp* function implements bounded repetitions using expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like the following, ((((a{1,100}){1,100}){1,100}' will (eventually) run almost any existing machine out of swap space.

There are suspected problems with responses to obscure error conditions. Notably, certain kinds of internal overflow, produced only by extremely large REs or by multiply—nested bounded repetitions, are probably not handled well.

Due to a mistake in 1003.2, things like 'a)b' are legal REs because ')' is a special character only in the presence of a previous unmatched '('. This can't be fixed until the spec is fixed.

The standard's definition of back references is vague. For example, does 'a\(\(b\)*\2\)*d' match 'abbbd'? Until the standard is clarified, behavior in such cases should not be relied on.

The implementation of word-boundary matching is imprecise, and bugs may lurk in combinations of word-boundary matching and anchoring.

RESTRICTIONS

As part of the BSD library, this function is not *thread-safe* .

NAME

regex, regcomp, regexec, regerror, regfree – regular-expression library

SYNOPSIS

#include <sys/types.h>
#include <regex.h>
int **regcomp**(regex_t * *preg*, const char * *pattern*, int *cflags*);

int **regexec**(constregex_t * *preg*, constchar * *string*, size_t *nmatch*, regmatch_tp *match* [], int *eflags*);

size_t **regerror**(int *errcode*, constregex_t * *preg*, char * *errbuf*, size_t *errbuf_size*);

void **regfree**(regex_t * *preg*);

FEATURES

STDC

DESCRIPTION

These routines implement POSIX 1003.2 regular expressions (“RE”s); see the SEE ALSO section below. The *regcomp* function compiles an RE written as a string into an internal form, *regexec* matches that internal form against a string and reports results, *regerror* transforms error codes from either into human-readable messages, and *regfree* frees any dynamically-allocated storage used by the internal form of an RE.

The header <*regex.h*> declares two structure types, *regex_t* and *regmatch_t* , the former for compiled internal forms and the latter for match reporting. It also declares the four functions, a type *regoff_t* , and a number of constants with names starting with “REG_”.

The *regcomp* function compiles the regular expression contained in the *pattern* string, subject to the flags in *cflags* , and places the results in the *regex_t* structure pointed to by *preg* . The *cflags* parameter is the bitwise OR of zero or one or more of the following flags:

REG_EXTENDED	Compile modern (“extended”) REs, rather than the obsolete (“basic”) REs that are the default.
REG_BASIC	This is a synonym for 0, provided as a counterpart to REG_EXTENDED to improve readability.
REG_NOSPEC	Compile with recognition of all special characters turned off. All characters are thus considered ordinary, so the “RE” is a literal string. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. REG_EXTENDED and REG_NOSPEC may not be used in the same call to *regcomp* .

REG_ICASE	Compile for matching that ignores upper/lower case distinctions. See the citation in the SEE ALSO section below
REG_NOSUB	Compile for matching that need only report success or failure, not what was matched.
REG_NEWLINE	Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, '[^' bracket expressions and '.' never match newline, a '^' anchor matches the null string after any newline in the string in addition to its normal function, and the '$' anchor matches the null string before any newline in the string in addition to its normal function.
REG_PEND	The regular expression ends, not at the first NULL, but just before the character pointed to by the *re_endp* member of the structure pointed to by *preg* . The *re_endp* member is of the type *const char ** . This flag permits inclusion of NULs in the RE; they are considered ordinary characters. This is an extension, compatible with, but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

When successful, *regcomp* returns 0 and fills in the structure pointed to by *preg* . One member of that structure (other than *re_endp*) is published: *re_nsub* , of type *size_t* , contains the number of parenthesized subexpressions within the RE (except that the value of this member is undefined if the REG_NOSUB flag was used). If *regcomp* fails, it returns a non-zero error code; see DIAGNOSTICS.

The *regexec* function matches the compiled RE pointed to by *preg* against the *string* , subject to the flags in *eflags* , and reports results using *nmatch* , *pmatch* , and the returned value. The RE must have been compiled using a previous invocation of *regcomp* . The compiled form is not altered during execution of *regexec* , a single compiled RE can therefore be used simultaneously by multiple threads.

By default, the NUL-terminated string pointed to by *string* is considered to be the text of an entire line, minus any terminating newline. The *eflags* argument is the bitwise OR of zero or one or more of the following flags:

REG_NOTBOL	The first character of the string is not the beginning of a line, so the '^' anchor should not

	match before it. This does not affect the behavior of newlines under REG_NEWLINE.
REG_NOTEOL	The NUL terminating the string does not end a line, so the '$' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.
REG_STARTEND	The string is considered to start at *string* + *pmatch* [0]. *rm_so* and to have a terminating NUL located at *string* + *pmatch* [0]. *rm_eo* (there need not actually be a NUL at that location), regardless of the value of *nmatch* . See below for the definition of *pmatch* and *nmatch* . This is an extension, compatible with, but not specified by, POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Note that a non-zero *rm_so* does not imply REG_NOTBOL; REG_STARTEND affects only the location of the string, not how it is matched.

See the citation in the SEE ALSO SECTION for an explanation of what is matched in situations where an RE or a portion thereof could match any of several substrings of *string* .

Normally, *regexec* returns 0 for success and the non-zero code REG_NOMATCH for failure. Other non-zero error codes may be returned in exceptional situations; see DIAGNOSTICS.

If REG_NOSUB was specified in the compilation of the RE, or if *nmatch* is 0, *regexec* ignores the *pmatch* argument (see below for the case where REG_STARTEND is specified). Otherwise, *pmatch* points to an array of *nmatch* structures of the type *regmatch_t* . This a structure has at least the members *rm_so* and *rm_eo* , both of type *regoff_t* (a signed arithmetic type at least as large as an *off_t* and a *ssize_t*), containing respectively the offset of the first character of a substring and the offset of the first character after the end of the substring. Offsets are measured from the beginning of the *string* argument given to *regexec* . An empty substring is denoted by equal offsets, both indicating the character following the empty substring.

The 0th member of the *pmatch* array is filled in to indicate what substring of *string* was matched by the entire RE. Remaining members report what substring was matched by parenthesized subexpressions within the RE; member *i* reports subexpression *i* , with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. Unused entries in the array—corresponding either to subexpressions that did not participate in the match at all, or to subexpressions that do not exist in the RE (that is, *i* > *preg* –>

re_nsub)—have both *rm_so* and *rm_eo* set to –1. If a subexpression participated in the match several times, the substring reported is the last one it matched. (Note that, when the RE '(b*)+' matches 'bbb', the parenthesized subexpression matches each of the three 'b's and then an infinite number of empty strings following the last 'b', the substring reported is therefore empty.)

If REG_STARTEND is specified, *pmatch* must point to at least one *regmatch_t* (even if *nmatch* is 0 or REG_NOSUB was specified), to hold the input offsets for REG_STARTEND. Use for output is still entirely controlled by *nmatch* ; if *nmatch* is 0 or REG_NOSUB was specified, the value of *pmatch* [0] will not be changed by a successful *regexec* .

The *regerror* function maps a non-zero *errcode* from either *regcomp* or *regexec* to a human-readable, printable message. If *preg* is non-NULL, the error code should have arisen from use of the *regex_t* pointed to by *preg* , and if the error code came from *regcomp* , it should have been the result of the most recent *regcomp* using that *regex_t* . (*Regerror* may be able to supply a more detailed message using information from the *regex_t* .) The *regerror* function places the NUL-terminated message into the buffer pointed to by *errbuf* , limiting the length (including the NUL) to at most *errbuf_size* bytes. If the whole message won't fit, as much of it as will fit before the terminating NUL is supplied. The value returned is the size of buffer needed to hold the whole message (including the terminating NULL). If *errbuf_size* is 0, *errbuf* is ignored but the return value is still correct.

If the *errcode* given to *regerror* is first ORed with REG_ITOA, the "message" that results is the printable name of the error code, for example, "REG_NOMATCH", rather than an explanation of it If *errcode* is REG_ATOI, *preg* will be non-NULL and the *re_endp* member of the structure it points to must point to the printable name of an error code; in this case, the result in *errbuf* is the decimal digits of the numeric value of the error code (0 if the name is not recognized). The REG_ITOA and REG_ATOI functions are intended primarily as debugging facilities; they are extensions, compatible with, but not specified by, POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Be warned also that they are considered experimental and changes are possible.

The *regfree* function frees any dynamically-allocated storage associated with the compiled RE pointed to by *preg* . The remaining *regex_t* is no longer a valid compiled RE and the effect of supplying it to *regexec* or *regerror* is undefined.

None of these functions references global variables except for tables of constants; all are safe for use by multiple threads if the arguments are safe.

IMPLEMENTATION CHOICES

There are a number of decisions that 1003.2 leaves up to the implementor, either by explicitly saying "undefined" or by virtue of them being forbidden by the RE grammar. This implementation treats them as follows.

See the citation in the SEE ALSO section for an explanation of the definition of case-independent matching.

There is no particular limit to the length of REs, apart from memory limitations. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See BUGS for one short RE using them that will run almost any system out of memory.

A backslashed character other than one specifically given a magic meaning by 1003.2 (such magic meanings occur only in obsolete [“basic”] REs) is taken as an ordinary character.

Any unmatched [is a REG_EBRACK error.

Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.

The RE_DUP_MAX option defines the limit on repetition counts in bounded repetitions, the maximum is 255.

A repetition operator (?, *, +, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow ‘^’ or ‘|’.

The pipe symbol. (‘|’) cannot appear first or last in a (sub)expression or after another ‘|’, in other words, an operand of ‘|’ cannot be an empty subexpression. An empty parenthesized subexpression, ‘()’, is legal and matches an empty (sub)string. An empty string is not a legal RE.

A brace (“{”) followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A ‘{’ *not* followed by a digit is considered an ordinary character.

A circumflex (’”‘^”) and dollar sign (“$”) beginning and ending subexpressions in obsolete (“basic”) REs are anchors, not ordinary characters.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

grep(1UNIX), re_format(7UNIX)

POSIX 1003.2, sections 2.8 (Regular Expression Notation) and B.5 (C Binding for Regular Expression Matching).

DIAGNOSTICS

Non-zero error codes from *regcomp* and *regexec* include the following:

```
REG_NOMATCH regexec() failed to match
REG_BADPAT invalid regular expression
```

```
REG_ECOLLATE invalid collating element
REG_ECTYPE invalid character class
REG_EESCAPE \ applied to unescapable character
REG_ESUBREG invalid backreference number
REG_EBRACK brackets [ ] not balanced
REG_EPAREN parentheses ( ) not balanced
REG_EBRACE braces { } not balanced
REG_BADBR invalid repetition count(s) in { }
REG_ERANGE invalid character range in [ ]
REG_ESPACE ran out of memory
REG_BADRPT ?, *, or + operand invalid
REG_EMPTY empty (sub)expression
REG_ASSERT ‘‘can’t happen’’—you found a bug
REG_INVARG invalid argument, for example,  negative-length string
```

HISTORY

Originally written by Henry Spencer. Altered for inclusion in the 4.4BSD distribution.

BUGS

This is an alpha release with known defects. Please report problems.

There is one known functionality bug. The implementation of internationalization is incomplete: the locale is always assumed to be the default one of 1003.2, therefore, only information pertaining to that locale is available.

The back-reference code is subtle and there are doubts about its correctness in complex cases.

The *regexec* function’s performance is poor. This will improve with later releases. An *nmatch* exceeding 0 is expensive; *nmatch* exceeding 1 is worse. The *regexec* function is largely insensitive to RE complexity *except* that back references are extremely expensive. RE length does matter; in particular, there is an appreciable speed bonus for keeping RE length under approximately 30 characters, most special characters are worth roughly double.

The *regcomp* function implements bounded repetitions using expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like the following, ((((a{1,100}){1,100}){1,100}’ will (eventually) run almost any existing machine out of swap space.

There are suspected problems with responses to obscure error conditions. Notably, certain kinds of internal overflow, produced only by extremely large REs or by multiply—nested bounded repetitions, are probably not handled well.

Due to a mistake in 1003.2, things like ‘a)b’ are legal REs because ‘)’ is a special character only in the presence of a previous unmatched ‘(’. This can’t be fixed until the spec is fixed.

The standard’s definition of back references is vague. For example, does ‘a\(\(b\)*\2\)*d’ match ‘abbbd’? Until the standard is clarified, behavior in such cases should not be relied on.

The implementation of word-boundary matching is imprecise, and bugs may lurk in combinations of word-boundary matching and anchoring.

RESTRICTIONS

As part of the BSD library, this function is not *thread-safe* .

NAME

regex, regcomp, regexec, regerror, regfree – regular-expression library

SYNOPSIS

#include <sys/types.h>
#include <regex.h>
int **regcomp**(regex_t * *preg*, const char * *pattern*, int *cflags*);

int **regexec**(constregex_t * *preg*, constchar * *string*, size_t *nmatch*, regmatch_tp *match* [], int *eflags*);

size_t **regerror**(int *errcode*, constregex_t * *preg*, char * *errbuf*, size_t *errbuf_size*);

void **regfree**(regex_t * *preg*);

FEATURES

STDC

DESCRIPTION

These routines implement POSIX 1003.2 regular expressions (“RE”s); see the SEE ALSO section below. The *regcomp* function compiles an RE written as a string into an internal form, *regexec* matches that internal form against a string and reports results, *regerror* transforms error codes from either into human-readable messages, and *regfree* frees any dynamically-allocated storage used by the internal form of an RE.

The header <*regex.h*> declares two structure types, *regex_t* and *regmatch_t* , the former for compiled internal forms and the latter for match reporting. It also declares the four functions, a type *regoff_t* , and a number of constants with names starting with “REG_”.

The *regcomp* function compiles the regular expression contained in the *pattern* string, subject to the flags in *cflags* , and places the results in the *regex_t* structure pointed to by *preg* . The *cflags* parameter is the bitwise OR of zero or one or more of the following flags:

REG_EXTENDED	Compile modern (“extended”) REs, rather than the obsolete (“basic”) REs that are the default.
REG_BASIC	This is a synonym for 0, provided as a counterpart to REG_EXTENDED to improve readability.
REG_NOSPEC	Compile with recognition of all special characters turned off. All characters are thus considered ordinary, so the “RE” is a literal string. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. REG_EXTENDED and REG_NOSPEC may not be used in the same call to *regcomp* .

REG_ICASE	Compile for matching that ignores upper/lower case distinctions. See the citation in the SEE ALSO section below
REG_NOSUB	Compile for matching that need only report success or failure, not what was matched.
REG_NEWLINE	Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, '[^' bracket expressions and '.' never match newline, a '^' anchor matches the null string after any newline in the string in addition to its normal function, and the '$' anchor matches the null string before any newline in the string in addition to its normal function.
REG_PEND	The regular expression ends, not at the first NULL, but just before the character pointed to by the *re_endp* member of the structure pointed to by *preg* . The *re_endp* member is of the type *const char* * . This flag permits inclusion of NULs in the RE; they are considered ordinary characters. This is an extension, compatible with, but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

When successful, *regcomp* returns 0 and fills in the structure pointed to by *preg* . One member of that structure (other than *re_endp*) is published: *re_nsub* , of type *size_t* , contains the number of parenthesized subexpressions within the RE (except that the value of this member is undefined if the REG_NOSUB flag was used). If *regcomp* fails, it returns a non-zero error code; see DIAGNOSTICS.

The *regexec* function matches the compiled RE pointed to by *preg* against the *string* , subject to the flags in *eflags* , and reports results using *nmatch* , *pmatch* , and the returned value. The RE must have been compiled using a previous invocation of *regcomp* . The compiled form is not altered during execution of *regexec* , a single compiled RE can therefore be used simultaneously by multiple threads.

By default, the NUL-terminated string pointed to by *string* is considered to be the text of an entire line, minus any terminating newline. The *eflags* argument is the bitwise OR of zero or one or more of the following flags:

REG_NOTBOL	The first character of the string is not the beginning of a line, so the '^' anchor should not

	match before it. This does not affect the behavior of newlines under REG_NEWLINE.
REG_NOTEOL	The NUL terminating the string does not end a line, so the '$' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.
REG_STARTEND	The string is considered to start at *string* + *pmatch* [0]. *rm_so* and to have a terminating NUL located at *string* + *pmatch* [0]. *rm_eo* (there need not actually be a NUL at that location), regardless of the value of *nmatch* . See below for the definition of *pmatch* and *nmatch* . This is an extension, compatible with, but not specified by, POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Note that a non-zero *rm_so* does not imply REG_NOTBOL; REG_STARTEND affects only the location of the string, not how it is matched.

See the citation in the SEE ALSO SECTION for an explanation of what is matched in situations where an RE or a portion thereof could match any of several substrings of *string* .

Normally, *regexec* returns 0 for success and the non-zero code REG_NOMATCH for failure. Other non-zero error codes may be returned in exceptional situations; see DIAGNOSTICS.

If REG_NOSUB was specified in the compilation of the RE, or if *nmatch* is 0, *regexec* ignores the *pmatch* argument (see below for the case where REG_STARTEND is specified). Otherwise, *pmatch* points to an array of *nmatch* structures of the type *regmatch_t* . This a structure has at least the members *rm_so* and *rm_eo* , both of type *regoff_t* (a signed arithmetic type at least as large as an *off_t* and a *ssize_t*), containing respectively the offset of the first character of a substring and the offset of the first character after the end of the substring. Offsets are measured from the beginning of the *string* argument given to *regexec* . An empty substring is denoted by equal offsets, both indicating the character following the empty substring.

The 0th member of the *pmatch* array is filled in to indicate what substring of *string* was matched by the entire RE. Remaining members report what substring was matched by parenthesized subexpressions within the RE; member *i* reports subexpression *i* , with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. Unused entries in the array—corresponding either to subexpressions that did not participate in the match at all, or to subexpressions that do not exist in the RE (that is, *i* > *preg* –>

re_nsub)—have both *rm_so* and *rm_eo* set to –1. If a subexpression participated in the match several times, the substring reported is the last one it matched. (Note that, when the RE '(b*)+' matches 'bbb', the parenthesized subexpression matches each of the three 'b's and then an infinite number of empty strings following the last 'b', the substring reported is therefore empty.)

If REG_STARTEND is specified, *pmatch* must point to at least one *regmatch_t* (even if *nmatch* is 0 or REG_NOSUB was specified), to hold the input offsets for REG_STARTEND. Use for output is still entirely controlled by *nmatch* ; if *nmatch* is 0 or REG_NOSUB was specified, the value of *pmatch* [0] will not be changed by a successful *regexec* .

The *regerror* function maps a non-zero *errcode* from either *regcomp* or *regexec* to a human-readable, printable message. If *preg* is non-NULL, the error code should have arisen from use of the *regex_t* pointed to by *preg* , and if the error code came from *regcomp* , it should have been the result of the most recent *regcomp* using that *regex_t* . (*Regerror* may be able to supply a more detailed message using information from the *regex_t* .) The *regerror* function places the NUL-terminated message into the buffer pointed to by *errbuf* , limiting the length (including the NUL) to at most *errbuf_size* bytes. If the whole message won't fit, as much of it as will fit before the terminating NUL is supplied. The value returned is the size of buffer needed to hold the whole message (including the terminating NULL). If *errbuf_size* is 0, *errbuf* is ignored but the return value is still correct.

If the *errcode* given to *regerror* is first ORed with REG_ITOA, the "message" that results is the printable name of the error code, for example, "REG_NOMATCH", rather than an explanation of it If *errcode* is REG_ATOI, *preg* will be non-NULL and the *re_endp* member of the structure it points to must point to the printable name of an error code; in this case, the result in *errbuf* is the decimal digits of the numeric value of the error code (0 if the name is not recognized). The REG_ITOA and REG_ATOI functions are intended primarily as debugging facilities; they are extensions, compatible with, but not specified by, POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Be warned also that they are considered experimental and changes are possible.

The *regfree* function frees any dynamically-allocated storage associated with the compiled RE pointed to by *preg* . The remaining *regex_t* is no longer a valid compiled RE and the effect of supplying it to *regexec* or *regerror* is undefined.

None of these functions references global variables except for tables of constants; all are safe for use by multiple threads if the arguments are safe.

IMPLEMENTATION CHOICES

There are a number of decisions that 1003.2 leaves up to the implementor, either by explicitly saying "undefined" or by virtue of them being forbidden by the RE grammar. This implementation treats them as follows.

See the citation in the SEE ALSO section for an explanation of the definition of case-independent matching.

There is no particular limit to the length of REs, apart from memory limitations. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See BUGS for one short RE using them that will run almost any system out of memory.

A backslashed character other than one specifically given a magic meaning by 1003.2 (such magic meanings occur only in obsolete [“basic”] REs) is taken as an ordinary character.

Any unmatched [is a REG_EBRACK error.

Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.

The RE_DUP_MAX option defines the limit on repetition counts in bounded repetitions, the maximum is 255.

A repetition operator (?, *, +, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow ‘^’ or ‘|’.

The pipe symbol. (‘|’) cannot appear first or last in a (sub)expression or after another ‘|’, in other words, an operand of ‘|’ cannot be an empty subexpression. An empty parenthesized subexpression, ‘()’, is legal and matches an empty (sub)string. An empty string is not a legal RE.

A brace (“{”) followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A ‘{’ *not* followed by a digit is considered an ordinary character.

A circumflex (’”‘^”) and dollar sign (“$”) beginning and ending subexpressions in obsolete (“basic”) REs are anchors, not ordinary characters.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

grep(1UNIX), re_format(7UNIX)

POSIX 1003.2, sections 2.8 (Regular Expression Notation) and B.5 (C Binding for Regular Expression Matching).

DIAGNOSTICS

Non-zero error codes from *regcomp* and *regexec* include the following:

```
REG_NOMATCH regexec() failed to match
REG_BADPAT invalid regular expression
```

```
REG_ECOLLATE invalid collating element
REG_ECTYPE invalid character class
REG_EESCAPE \ applied to unescapable character
REG_ESUBREG invalid backreference number
REG_EBRACK brackets [ ] not balanced
REG_EPAREN parentheses ( ) not balanced
REG_EBRACE braces { } not balanced
REG_BADBR invalid repetition count(s) in { }
REG_ERANGE invalid character range in [ ]
REG_ESPACE ran out of memory
REG_BADRPT ?, *, or + operand invalid
REG_EMPTY empty (sub)expression
REG_ASSERT ‘‘can’t happen’’—you found a bug
REG_INVARG invalid argument, for example,  negative-length string
```

HISTORY

Originally written by Henry Spencer. Altered for inclusion in the 4.4BSD distribution.

BUGS

This is an alpha release with known defects. Please report problems.

There is one known functionality bug. The implementation of internationalization is incomplete: the locale is always assumed to be the default one of 1003.2, therefore, only information pertaining to that locale is available.

The back-reference code is subtle and there are doubts about its correctness in complex cases.

The *regexec* function’s performance is poor. This will improve with later releases. An *nmatch* exceeding 0 is expensive; *nmatch* exceeding 1 is worse. The *regexec* function is largely insensitive to RE complexity *except* that back references are extremely expensive. RE length does matter; in particular, there is an appreciablbe speed bonus for keeping RE length under approximately 30 characters, most special characters are worth roughly double.

The *regcomp* function implements bounded repetitions using expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like the following, ((((a{1,100}){1,100}){1,100}’ will (eventually) run almost any existing machine out of swap space.

There are suspected problems with responses to obscure error conditions. Notably, certain kinds of internal overflow, produced only by extremely large REs or by multiply—nested bounded repetitions, are probably not handled well.

Due to a mistake in 1003.2, things like ‘a)b’ are legal REs because ‘)’ is a special character only in the presence of a previous unmatched ‘(’. This can’t be fixed until the spec is fixed.

The standard’s definition of back references is vague. For example, does ‘a\(\(b\)*\2\)*d’ match ‘abbbd’? Until the standard is clarified, behavior in such cases should not be relied on.

The implementation of word-boundary matching is imprecise, and bugs may lurk in combinations of word-boundary matching and anchoring.

RESTRICTIONS

As part of the BSD library, this function is not *thread-safe* .

NAME

regex, regcomp, regexec, regerror, regfree – regular-expression library

SYNOPSIS

#include <sys/types.h>
#include <regex.h>
int **regcomp**(regex_t * *preg*, const char * *pattern*, int *cflags*);

int **regexec**(constregex_t * *preg*, constchar * *string*, size_t *nmatch*, regmatch_tp *match* [], int *eflags*);

size_t **regerror**(int *errcode*, constregex_t * *preg*, char * *errbuf*, size_t *errbuf_size*);

void **regfree**(regex_t * *preg*);

FEATURES

STDC

DESCRIPTION

These routines implement POSIX 1003.2 regular expressions (“RE”s); see the SEE ALSO section below. The *regcomp* function compiles an RE written as a string into an internal form, *regexec* matches that internal form against a string and reports results, *regerror* transforms error codes from either into human-readable messages, and *regfree* frees any dynamically-allocated storage used by the internal form of an RE.

The header <*regex.h*> declares two structure types, *regex_t* and *regmatch_t* , the former for compiled internal forms and the latter for match reporting. It also declares the four functions, a type *regoff_t* , and a number of constants with names starting with “REG_”.

The *regcomp* function compiles the regular expression contained in the *pattern* string, subject to the flags in *cflags* , and places the results in the *regex_t* structure pointed to by *preg* . The *cflags* parameter is the bitwise OR of zero or one or more of the following flags:

REG_EXTENDED	Compile modern (“extended”) REs, rather than the obsolete (“basic”) REs that are the default.
REG_BASIC	This is a synonym for 0, provided as a counterpart to REG_EXTENDED to improve readability.
REG_NOSPEC	Compile with recognition of all special characters turned off. All characters are thus considered ordinary, so the “RE” is a literal string. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. REG_EXTENDED and REG_NOSPEC may not be used in the same call to *regcomp* .

REG_ICASE	Compile for matching that ignores upper/lower case distinctions. See the citation in the SEE ALSO section below
REG_NOSUB	Compile for matching that need only report success or failure, not what was matched.
REG_NEWLINE	Compile for newline-sensitive matching. By default, newline is a completely ordinary character with no special meaning in either REs or strings. With this flag, '[^' bracket expressions and '.' never match newline, a '^' anchor matches the null string after any newline in the string in addition to its normal function, and the '$' anchor matches the null string before any newline in the string in addition to its normal function.
REG_PEND	The regular expression ends, not at the first NULL, but just before the character pointed to by the *re_endp* member of the structure pointed to by *preg* . The *re_endp* member is of the type *const char* * . This flag permits inclusion of NULs in the RE; they are considered ordinary characters. This is an extension, compatible with, but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems.

When successful, *regcomp* returns 0 and fills in the structure pointed to by *preg* . One member of that structure (other than *re_endp*) is published: *re_nsub* , of type *size_t* , contains the number of parenthesized subexpressions within the RE (except that the value of this member is undefined if the REG_NOSUB flag was used). If *regcomp* fails, it returns a non-zero error code; see DIAGNOSTICS.

The *regexec* function matches the compiled RE pointed to by *preg* against the *string* , subject to the flags in *eflags* , and reports results using *nmatch* , *pmatch* , and the returned value. The RE must have been compiled using a previous invocation of *regcomp* . The compiled form is not altered during execution of *regexec* , a single compiled RE can therefore be used simultaneously by multiple threads.

By default, the NUL-terminated string pointed to by *string* is considered to be the text of an entire line, minus any terminating newline. The *eflags* argument is the bitwise OR of zero or one or more of the following flags:

REG_NOTBOL	The first character of the string is not the beginning of a line, so the '^' anchor should not

	match before it. This does not affect the behavior of newlines under REG_NEWLINE.
REG_NOTEOL	The NUL terminating the string does not end a line, so the '$' anchor should not match before it. This does not affect the behavior of newlines under REG_NEWLINE.
REG_STARTEND	The string is considered to start at *string* + *pmatch* [0]. *rm_so* and to have a terminating NUL located at *string* + *pmatch* [0]. *rm_eo* (there need not actually be a NUL at that location), regardless of the value of *nmatch* . See below for the definition of *pmatch* and *nmatch* . This is an extension, compatible with, but not specified by, POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Note that a non-zero *rm_so* does not imply REG_NOTBOL; REG_STARTEND affects only the location of the string, not how it is matched.

See the citation in the SEE ALSO SECTION for an explanation of what is matched in situations where an RE or a portion thereof could match any of several substrings of *string* .

Normally, *regexec* returns 0 for success and the non-zero code REG_NOMATCH for failure. Other non-zero error codes may be returned in exceptional situations; see DIAGNOSTICS.

If REG_NOSUB was specified in the compilation of the RE, or if *nmatch* is 0, *regexec* ignores the *pmatch* argument (see below for the case where REG_STARTEND is specified). Otherwise, *pmatch* points to an array of *nmatch* structures of the type *regmatch_t* . This a structure has at least the members *rm_so* and *rm_eo* , both of type *regoff_t* (a signed arithmetic type at least as large as an *off_t* and a *ssize_t*), containing respectively the offset of the first character of a substring and the offset of the first character after the end of the substring. Offsets are measured from the beginning of the *string* argument given to *regexec* . An empty substring is denoted by equal offsets, both indicating the character following the empty substring.

The 0th member of the *pmatch* array is filled in to indicate what substring of *string* was matched by the entire RE. Remaining members report what substring was matched by parenthesized subexpressions within the RE; member *i* reports subexpression *i* , with subexpressions counted (starting at 1) by the order of their opening parentheses in the RE, left to right. Unused entries in the array—corresponding either to subexpressions that did not participate in the match at all, or to subexpressions that do not exist in the RE (that is, *i* > *preg* –>

re_nsub)—have both *rm_so* and *rm_eo* set to –1. If a subexpression participated in the match several times, the substring reported is the last one it matched. (Note that, when the RE '(b*)+' matches 'bbb', the parenthesized subexpression matches each of the three 'b's and then an infinite number of empty strings following the last 'b', the substring reported is therefore empty.)

If REG_STARTEND is specified, *pmatch* must point to at least one *regmatch_t* (even if *nmatch* is 0 or REG_NOSUB was specified), to hold the input offsets for REG_STARTEND. Use for output is still entirely controlled by *nmatch* ; if *nmatch* is 0 or REG_NOSUB was specified, the value of *pmatch* [0] will not be changed by a successful *regexec* .

The *regerror* function maps a non-zero *errcode* from either *regcomp* or *regexec* to a human-readable, printable message. If *preg* is non-NULL, the error code should have arisen from use of the *regex_t* pointed to by *preg* , and if the error code came from *regcomp* , it should have been the result of the most recent *regcomp* using that *regex_t* . (*Regerror* may be able to supply a more detailed message using information from the *regex_t* .) The *regerror* function places the NUL-terminated message into the buffer pointed to by *errbuf* , limiting the length (including the NUL) to at most *errbuf_size* bytes. If the whole message won't fit, as much of it as will fit before the terminating NUL is supplied. The value returned is the size of buffer needed to hold the whole message (including the terminating NULL). If *errbuf_size* is 0, *errbuf* is ignored but the return value is still correct.

If the *errcode* given to *regerror* is first ORed with REG_ITOA, the "message" that results is the printable name of the error code, for example, "REG_NOMATCH", rather than an explanation of it If *errcode* is REG_ATOI, *preg* will be non-NULL and the *re_endp* member of the structure it points to must point to the printable name of an error code; in this case, the result in *errbuf* is the decimal digits of the numeric value of the error code (0 if the name is not recognized). The REG_ITOA and REG_ATOI functions are intended primarily as debugging facilities; they are extensions, compatible with, but not specified by, POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. Be warned also that they are considered experimental and changes are possible.

The *regfree* function frees any dynamically-allocated storage associated with the compiled RE pointed to by *preg* . The remaining *regex_t* is no longer a valid compiled RE and the effect of supplying it to *regexec* or *regerror* is undefined.

None of these functions references global variables except for tables of constants; all are safe for use by multiple threads if the arguments are safe.

IMPLEMENTATION CHOICES

There are a number of decisions that 1003.2 leaves up to the implementor, either by explicitly saying "undefined" or by virtue of them being forbidden by the RE grammar. This implementation treats them as follows.

See the citation in the SEE ALSO section for an explanation of the definition of case-independent matching.

There is no particular limit to the length of REs, apart from memory limitations. Memory usage is approximately linear in RE size, and largely insensitive to RE complexity, except for bounded repetitions. See BUGS for one short RE using them that will run almost any system out of memory.

A backslashed character other than one specifically given a magic meaning by 1003.2 (such magic meanings occur only in obsolete [“basic”] REs) is taken as an ordinary character.

Any unmatched [is a REG_EBRACK error.

Equivalence classes cannot begin or end bracket-expression ranges. The endpoint of one range cannot begin another.

The RE_DUP_MAX option defines the limit on repetition counts in bounded repetitions, the maximum is 255.

A repetition operator (?, *, +, or bounds) cannot follow another repetition operator. A repetition operator cannot begin an expression or subexpression or follow ‘^’ or ‘|’.

The pipe symbol. (‘|’) cannot appear first or last in a (sub)expression or after another ‘|’, in other words, an operand of ‘|’ cannot be an empty subexpression. An empty parenthesized subexpression, ‘()’, is legal and matches an empty (sub)string. An empty string is not a legal RE.

A brace (“{”) followed by a digit is considered the beginning of bounds for a bounded repetition, which must then follow the syntax for bounds. A ‘{’ *not* followed by a digit is considered an ordinary character.

A circumflex (’”‘^”) and dollar sign (“$”) beginning and ending subexpressions in obsolete (“basic”) REs are anchors, not ordinary characters.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

grep(1UNIX), re_format(7UNIX)

POSIX 1003.2, sections 2.8 (Regular Expression Notation) and B.5 (C Binding for Regular Expression Matching).

DIAGNOSTICS

Non-zero error codes from *regcomp* and *regexec* include the following:

```
REG_NOMATCH regexec() failed to match
REG_BADPAT invalid regular expression
```

```
REG_ECOLLATE invalid collating element
REG_ECTYPE invalid character class
REG_EESCAPE \ applied to unescapable character
REG_ESUBREG invalid backreference number
REG_EBRACK brackets [ ] not balanced
REG_EPAREN parentheses ( ) not balanced
REG_EBRACE braces { } not balanced
REG_BADBR invalid repetition count(s) in { }
REG_ERANGE invalid character range in [ ]
REG_ESPACE ran out of memory
REG_BADRPT ?, *, or + operand invalid
REG_EMPTY empty (sub)expression
REG_ASSERT ‘‘can’t happen’’—you found a bug
REG_INVARG invalid argument, for example,  negative-length string
```

HISTORY

Originally written by Henry Spencer. Altered for inclusion in the 4.4BSD distribution.

BUGS

This is an alpha release with known defects. Please report problems.

There is one known functionality bug. The implementation of internationalization is incomplete: the locale is always assumed to be the default one of 1003.2, therefore, only information pertaining to that locale is available.

The back-reference code is subtle and there are doubts about its correctness in complex cases.

The *regexec* function’s performance is poor. This will improve with later releases. An *nmatch* exceeding 0 is expensive; *nmatch* exceeding 1 is worse. The *regexec* function is largely insensitive to RE complexity *except* that back references are extremely expensive. RE length does matter; in particular, there is an appreciable speed bonus for keeping RE length under approximately 30 characters, most special characters are worth roughly double.

The *regcomp* function implements bounded repetitions using expansion, which is costly in time and space if counts are large or bounded repetitions are nested. An RE like the following, ((((a{1,100}){1,100}){1,100}’ will (eventually) run almost any existing machine out of swap space.

There are suspected problems with responses to obscure error conditions. Notably, certain kinds of internal overflow, produced only by extremely large REs or by multiply—nested bounded repetitions, are probably not handled well.

Due to a mistake in 1003.2, things like ‘a)b’ are legal REs because ‘)’ is a special character only in the presence of a previous unmatched ‘(’. This can’t be fixed until the spec is fixed.

The standard’s definition of back references is vague. For example, does ‘a\(\(b\)*\2\)*d’ match ‘abbbd’? Until the standard is clarified, behavior in such cases should not be relied on.

The implementation of word-boundary matching is imprecise, and bugs may lurk in combinations of word-boundary matching and anchoring.

RESTRICTIONS

As part of the BSD library, this function is not *thread-safe* .

NAME

remove – remove directory entry

SYNOPSIS

```
#include <stdio.h>
int remove(const char *path);
```

DESCRIPTION

The *remove* function is an alias for the `unlink`(2POSIX) system call. It deletes the file referenced by *path*.

RETURN VALUES

Upon successful completion, *remove* returns 0. Otherwise, –1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The *remove* function may fail and set *errno* for any of the errors specified for the routine *unlink*(2POSIX).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`unlink`(2POSIX)

STANDARDS

The *remove* function conforms to `ANSI-C`.

NAME

fseek, rewind, ftell, fgetpos, fsetpos – reposition a file pointer in a stream

SYNOPSIS

#include <stdio.h>
int **fseek**(FILE * *stream*, long *offset*, int *ptrname*);

void **rewind**(FILE * *stream*);

long **ftell**(const FILE * *stream*);

int **fgetpos**(const FILE * *stream*, fpos_t * *pos*);

int **fsetpos**(FILE * *stream*, const fpos_t * *pos*);

DESCRIPTION

The *fseek* function sets the position of the next input or output operation on the *stream* . The new position, measured in bytes from the beginning of the file, is obtained by adding offset to the position specified by *ptrname* , 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

The *rewind* (*stream*) function is equivalent to *fseek* (*stream* , 0L, 0), except that no value is returned.

The *fseek* and *rewind* functions undo any effects of *ungetc* (3STDC).

After performing *fseek* or *rewind* , the next operation on a file opened for update may be either input or output.

The *ftell* function returns the offset of the current byte relative to the beginning of the file associated with the *stream* specified.

The *fgetpos* and *fsetpos* functions are alternate interfaces equivalent to *ftell* and *fseek* (with *ptrname* set to SEEK_SET), setting and storing the current value of the file offset into or from the object referenced by *pos* . On some systems an *fpos_t* object may be a complex object, and these routines may be the only way to reposition a text stream portably. This is not the case on UNIX systems.

RETURN VALUES

The *fseek* function returns 0 on success; otherwise (for example, an *fseek* done on a file that was not opened using *fopen* (3STDC)), it returns -1 and sets *errno* to indicate the error.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO `fopen`(3STDC) , `ungetc`(3STDC)

NAME

index, rindex – locate character in string

SYNOPSIS

```
#include <string.h>
char * index(const char * s, int c);

char *rindex(const char * s, int c);
```

DESCRIPTION

The *index* function locates the first character matching *c* (converted to a *char*) in the null-terminated string *s* .

The *rindex* function locates the last character matching *c* (converted to a *char*) in the null-terminated string *s* .

RETURN VALUES

A pointer to the character is returned if found; otherwise NULL is returned. If *c* is 0 , *rindex* or *index* locates the terminating 0.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`memchr`(3STDC) , `string`(3STDC) , `strsep`(3STDC) , `strtok`(3STDC)

NAME

scandir, alphasort – scan a directory

SYNOPSIS

#include <sys/types.h>
#include <dirent.h>
int **scandir**(const char * *dirname*, struct dirent *** *namelist*, int (* *select)(struct dirent *)*, int (* *compare)(const void *, const void *)*);

int **alphasort**(const void * *d1*, const char * *d2*);

DESCRIPTION

The *scandir* function reads the directory *dirname* and builds an array of pointers to directory entries using *malloc* (3STDC). It returns the number of entries in the array. A pointer to the array of directory entries is stored in the location referenced by *namelist* .

The *select* parameter is a pointer to a user supplied subroutine which is called by *scandir* to select which entries are to be included in the array. The *select* routine is passed a pointer to a directory entry and should return a non-zero value if the directory entry is to be included in the array. If *select* is null, then all the directory entries will be included.

The *compare* parameter is a pointer to a user supplied subroutine which is passed to *qsort* (3STDC) to sort the completed array. If this pointer is null, the array is not sorted.

The *alphasort* function is a routine which can be used for the *compare* parameter to sort the array alphabetically.

The memory allocated for the array can be deallocated with *free* (3STDC), by freeing each pointer in the array and then the array itself.

DIAGNOSTICS

Returns -1 if the directory cannot be opened for reading or if *malloc* (3STDC) cannot allocate enough memory to hold all the data structures.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

directory(3POSIX) , malloc(3STDC) , qsort(3STDC)

HISTORY

The *scandir* and *alphasort* functions appeared in 4.2BSD.

NAME

scanf, sscanf – convert formatted input

SYNOPSIS

#include <stdio.h>
int **scanf**(const char * *format*, ...);

int **sscanf**(const char * *s*, const char * *format*, ...);

DESCRIPTION

The `scanf( )` function reads from the standard input channel, which is operating system dependent. The `sscanf( )` function reads from the character string *s* . Each function reads characters, 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 control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not `%`), which must match the next character of the input channel.
3. Conversion specifications, consisting of the character `%` , an optional assignment suppressing character `*` , an optional numerical maximum field width, an optional `l` (ell) or `h` indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated using `*` . The suppression of assignment allows you to define an input filed to be ignlored. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except “[” and “c”, white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

`%`	a single `%` is expected in the input at this point; no assignment is done.
`d`	a decimal integer is expected; the corresponding argument should be an integer pointer.
`u`	an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

`o` an octal integer is expected; the corresponding argument should be an integer pointer.

`x` a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

`i` an integer is expected; the corresponding argument should be an integer pointer. It will store the value of the next input item interpreted according to C conventions: a leading "0" implies octal; a leading "0x" implies hexadecimal; otherwise, decimal.

`n` stores in an integer argument the total number of characters (including white space) that have been scanned so far since the function call. No input is consumed.

`e` , `f` , `g` a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float* . The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an `E` or an `e` , followed by an optional +, –, or space, followed by an integer.

`s` a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating `\0` , which will be added automatically. The input field is terminated by a white-space character.

`c` a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use `%1s` . If a field width is given, the corresponding argument should refer to a character array; the number of characters indicated is read.

`[` indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, called the *scanset,* and a right bracket; the input field is the maximaum sequence of input characters consisting entirely of characters in the scanset. The circumflex (`^`), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first–last* , thus [0123456789] may be expressed [0–9]. Using this convention, *first* must be lexically less than or equal to `last` , otherwise the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating `\0` , which will be added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters `d` , `u` , `o` , and `x` may be preceded by `l` or `h` to indicate that a pointer to `long` or to `short` rather than to `int` is in the argument list. Similarly, the conversion characters `e` , `f` , and `g` may be preceded by `l` to indicate that a pointer to `double` rather than to `float` is in the argument list. The `l` or `h` modifier is ignored for other conversion characters.

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

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

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E−1 thompson
```

will assign to *n* the value `3` , to *i* the value `25` , to *x* the value `5.432` , and *name* will contain `thompson\0` .

Or:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0−9]", &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* (3STDC) will return `a` .

NOTE

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

DIAGNOSTICS

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getchar`(3STDC) , `printf`(3STDC) , `strtod`(3STDC) , `strtol`(3STDC)

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*, int *size*);

DESCRIPTION

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

A constant BUFSIZ , defined in the `<stdio.h>` header file, defines how big an array is needed:

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

The *setvbuf* function can be used after a stream has been opened but before it is read from or written to. The *type* parameter determines how *stream* will be buffered. Legal values for `type` (defined in stdio.h) are:

_IOFBF	Causes input and output to be fully buffered.
_IOLBF	Causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.
_IONBF	Causes input and output to be completely unbuffered.

If *buf* is not a NULL pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. The *size* parameter specifies the size of the buffer to be used. The BUFSIZ constant in `<stdio.h>` is a recommended buffer size. If input and output are unbuffered, *buf* and `size` are ignored.

Output streams directed to terminals are always line-buffered (unless they are unbuffered).

NOTE

A common source of error is allocating buffer space as an “automatic” variable in a code block, and then failing to close the stream in the same block.

RETURN VALUES

If an illegal value for type is provided, *setvbuf* returns a non-zero value. Otherwise, it returns 0.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC) , `getc`(3STDC) , `malloc`(3STDC) , `putc`(3STDC)

NAME

getenv, putenv, setenv, unsetenv – fetch and set environment variables

SYNOPSIS

#include <stdlib.h>
char * **getenv**(const char * *name*);

int **setenv**(const char * *name*, const char * *value*, int *overwrite*);

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

void **unsetenv**(const char * *name*);

DESCRIPTION

These functions set, unset and fetch environment variables from the host *environment* list. For compatibility with differing environment conventions, the *name* and *value* arguments given may be appended and prepended, respectively, with an equal sign. The *getenv* function obtains the current value of the environment variable, *name*. If the variable *name* is not in the current environment, a null pointer is returned.

The `setenv` function inserts or resets the environment variable *name* in the current environment list. If the variable *name* does not exist in the list, it is inserted with the given *value*. If the variable does exist, the *overwrite* argument is tested; if *overwrite* is zero, the variable is not reset, otherwise it is reset to the given *value*.

The *putenv* function takes an argument of the form `name=value` and is equivalent to: `setenv(name, value, 1)` .

The `unsetenv` function deletes all instances of the variable name pointed to by *name* from the list.

RETURN VALUES

The `setenv` and *putenv* functions return zero if successful; otherwise –1 is returned. The `setenv` or *putenv* functions fail if they were unable to allocate memory for the environment.

STANDARDS

The *getenv* function conforms to `ANSI-C` .

NOTE

These functions are reentrant, but the environment is global to the actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

setjmp, longjmp – non-local goto

SYNOPSIS

#include <setjmp.h>
int **setjmp**(jmp_buf *env*);

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

DESCRIPTION

These functions are useful for dealing with errors and interrupts encountered in low-level subroutines of a program.

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

The *longjmp* function restores the environment saved by the last call of *setjmp* with the corresponding `env` argument. After *longjmp* has completed, program execution continues as if the corresponding call of *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. All accessible data will have the values stored at the time *longjmp* was called.

WARNING

If *longjmp* is called without first priming `env` using a calll to *setjmp* , or if the last such call was performed by another thread, or if the last such call was in a function that has since returned, this will cause severe disruption to the system.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

random, srandom, initstate, setstate – better random number generator

SYNOPSIS

#include <stdlib.h>
long **random**(void);

void **srandom**(unsigned *seed*);

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

char ***setstate**(char * *state*);

DESCRIPTION

The *random* function uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31} -1$. The period of this random number generator is very large, approximately $16\times(2^{31} -1)$.

The *random/srandom* functions have (almost) the same calling sequence and initialization properties as *rand/srand* (3STDC) The difference is that *rand* produces a much less random sequence — in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by *random* are usable. For example, *random* `&01` will produce a random binary value.

Unlike *srand* , *srandom* does not return the old seed; the reason being that the amount of state information used is much more than a single word (two other routines are provided to deal with restarting / changing random number generators). Like *rand* , however, *random* will by default produce a sequence of numbers that can be duplicated by calling *srandom* with 1 as the seed.

The *initstate* routine allows a state array, passed as an argument, to be initialized for future use. The size of the state array (in bytes) is used by *initstate* to decide how sophisticated a random number generator it should use — the bigger the state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error.) The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. The *initstate* function returns a pointer to the previous state information array.

Once a state has been initialized, the *setstate* routine provides for rapid switching between states. The *setstate* function returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to *initstate* or *setstate* .

Once a state array has been initialized, it may be restarted at a different point either by calling *initstate* (with the desired seed, the state array, and its size) or by calling both *setstate* (with the state array) and *srandom* (with the desired seed).

The advantage of calling both *setstate* and *srandom* is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than `2` ^{690} , which should be sufficient for most purposes.

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` , it returns `NULL` and *random* uses a simple linear congruential random number generator.

DIAGNOSTICS

If *initstate* is called with less than 8 bytes of state information, or if *setstate* detects that the state information has been garbled, error messages are printed to the standard error output.

NOTE

Though these functions are reentrant, the state information is global to the actor. Therefore, repeatability of a given suite of number will not be experienced by several threads in parallel. For a reentrent repeatability of suites, see *rand_r(3STDC)* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rand`(3STDC) , `rand_r`(3STDC)

RESTRICTIONS

random operates at about 2 / 3 the speed of *rand* (3STDC).

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, int size);
```

DESCRIPTION

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

A constant BUFSIZ , defined in the `<stdio.h>` header file, defines how big an array is needed:

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

The *setvbuf* function can be used after a stream has been opened but before it is read from or written to. The *type* parameter determines how *stream* will be buffered. Legal values for `type` (defined in stdio.h) are:

_IOFBF	Causes input and output to be fully buffered.
_IOLBF	Causes output to be line buffered; the buffer will be flushed when a newline is written, the buffer is full, or input is requested.
_IONBF	Causes input and output to be completely unbuffered.

If *buf* is not a NULL pointer, the array it points to will be used for buffering, instead of an automatically allocated buffer. The *size* parameter specifies the size of the buffer to be used. The BUFSIZ constant in `<stdio.h>` is a recommended buffer size. If input and output are unbuffered, *buf* and `size` are ignored.

Output streams directed to terminals are always line-buffered (unless they are unbuffered).

NOTE

A common source of error is allocating buffer space as an “automatic” variable in a code block, and then failing to close the stream in the same block.

RETURN VALUES

If an illegal value for type is provided, *setvbuf* returns a non-zero value. Otherwise, it returns 0.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC) , `getc`(3STDC) , `malloc`(3STDC) , `putc`(3STDC)

NAME

printf, sprintf, snprintf, printerr – print formatted output

SYNOPSIS

#include <stdio.h>
int **printf**(const char * *format*, ... /* args */);

int **sprintf**(char * *s*, const char * *format*, ... /* args */);

int **snprintf**(char * *s*, size_t *size*, const char * *format*, ... /* args */);

int **printerr**(const char * *format*, ... /* args */);

DESCRIPTION

The printf function sends output to the standard output channel, which is system defined. The `printerr()` function sends output to on the standard error channel, which is system defined. The `sprintf()` function sends 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. Each function returns the number of characters transmitted (not including the `\0` in the case of *sprintf*), or a negative value if an output error was encountered.

The `snprintf()` function writes at most *size-1* of the characters printed to the output string (the *size* character then gets the terminating zero). If the return value is greater than or equal to the `size` argument, the string was too short and some of the printed characters were discarded.

Each of these functions converts, formats, and prints its *arg* s under control of the `format` . The `format` is a character string that contains two types of objects: plain characters, which are simply copied to the output channel, and conversion specifications, each of which results in obtaining zero or more *arg* s. The results are undefined if there are insufficient *arg* s for the format. If the format is exhausted while *arg* s remain, the excess *arg* s are simply ignored.

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 set) to the field width. If the field width for an s conversion is preceded by a 0, the string is right adjusted with zero-padding on the left.

A *precision* that gives the minimum number of digits to appear for the `d` , `o` , `u` , `x` , or `X` conversions, the number of digits to appear after the decimal point for the `e` and `f` conversions, the maximum number of significant digits for the `g` conversion, or the maximum number of characters to be printed from a string in an `s` conversion. The precision takes the form of a dot (`.`) followed by a decimal digit string; a null digit string is treated as zero.

An optional `l` (ell) specifying that a following `d` , `o` , `u` , `x` , or `X` conversion character applies to a long integer *arg* . A `l` before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision 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 *arg* s specifying field width or precision must appear *before* the *arg* (if any) to be converted.

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` , `s` , and `u` conversions, the flag has no effect. For `o` conversion, it increases the precision to force the first digit of the result to be a zero. For `x` or `X` conversion, a non-zero result will have `0x` or `0X` prefixed to it. For `e` , `E` , `f` , `g` , and `G` conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For `g` and `G` conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

`d`, `i`, `o`, `u`, `x`, `X`	The integer *arg* is converted to signed decimal (`d` or `i`), unsigned octal (`o`), decimal (`u`), or hexadecimal notation (`x` and `X`), respectively. The letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
`f`	The float or double *arg* is converted to decimal notation in the style "[–]ddd `.` ddd," where the number of digits after the decimal point is equal to the precision specification. If the precision is not specified, six digits are output; if the precision is explicitly 0, no decimal point appears.
`e`, `E`	The float or double *arg* is converted in the style "[–]d `.` ddd `e±` dd," where there is one digit before the decimal point and the number of digits after it is equal to the precision. If the precision is not specified, six digits are produced; if the precision is explicitly 0, no decimal point appears. The `E` format code will produce a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits.
`g`, `G`	The float or double *arg* is printed in style `f` or `e` (or in style `E` in the case of a `G` format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style `e` 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.
`c`	The character *arg* is printed.
`s`	The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (`\0`) is encountered, or until the number of characters indicated by the precision specification is reached. If the precision is not specified, it is assumed to be infinite and all

characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.

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

A non-existent or small field width will never 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` are printed in the same way as if *putchar* (3STDC) had been called.

EXAMPLES

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 %d, %d:%.2d", weekday, month, day, hour, min);
```

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`putchar`(3STDC) , `scanf`(3STDC)

NAME

printf, sprintf, snprintf, printerr – print formatted output

SYNOPSIS

#include <stdio.h>
int **printf**(const char * *format*, ... /* args */);

int **sprintf**(char * *s*, const char * *format*, ... /* args */);

int **snprintf**(char * *s*, size_t *size*, const char * *format*, ... /* args */);

int **printerr**(const char * *format*, ... /* args */);

DESCRIPTION

The printf function sends output to the standard output channel, which is system defined. The `printerr()` function sends output to on the standard error channel, which is system defined. The `sprintf()` function sends 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. Each function returns the number of characters transmitted (not including the `\0` in the case of *sprintf*), or a negative value if an output error was encountered.

The `snprintf()` function writes at most *size-1* of the characters printed to the output string (the *size* character then gets the terminating zero). If the return value is greater than or equal to the `size` argument, the string was too short and some of the printed characters were discarded.

Each of these functions converts, formats, and prints its *arg* s under control of the `format` . The `format` is a character string that contains two types of objects: plain characters, which are simply copied to the output channel, and conversion specifications, each of which results in obtaining zero or more *arg* s. The results are undefined if there are insufficient *arg* s for the format. If the format is exhausted while *arg* s remain, the excess *arg* s are simply ignored.

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 set) to the field width. If the field width for an s conversion is preceded by a 0, the string is right adjusted with zero-padding on the left.

A *precision* that gives the minimum number of digits to appear for the `d` , `o` , `u` , `x` , or `X` conversions, the number of digits to appear after the decimal point for the `e` and `f` conversions, the maximum number of significant digits for the `g` conversion, or the maximum number of characters to be printed from a string in an `s` conversion. The precision takes the form of a dot (`.`) followed by a decimal digit string; a null digit string is treated as zero.

An optional `l` (ell) specifying that a following `d` , `o` , `u` , `x` , or `X` conversion character applies to a long integer *arg* . A `l` before any other conversion character is ignored.

A character that indicates the type of conversion to be applied.

A field width or precision 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 *arg* s specifying field width or precision must appear *before* the *arg* (if any) to be converted.

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` , `s` , and `u` conversions, the flag has no effect. For `o` conversion, it increases the precision to force the first digit of the result to be a zero. For `x` or `X` conversion, a non-zero result will have `0x` or `0X` prefixed to it. For `e` , `E` , `f` , `g` , and `G` conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For `g` and `G` conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d , i , o , u , x , X	The integer *arg* is converted to signed decimal (d or i), unsigned octal (o), decimal (u), or hexadecimal notation (x and X), respectively. The letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. (For compatibility with older versions, padding with leading zeroes may alternatively be specified by prepending a zero to the field width. This does not imply an octal value for the field width.) The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
f	The float or double *arg* is converted to decimal notation in the style "[–]ddd . ddd," where the number of digits after the decimal point is equal to the precision specification. If the precision is not specified, six digits are output; if the precision is explicitly 0, no decimal point appears.
e , E	The float or double *arg* is converted in the style "[–]d . ddd e± dd," where there is one digit before the decimal point and the number of digits after it is equal to the precision. If the precision is not specified, six digits are produced; if the precision is explicitly 0, no decimal point appears. The E format code will produce a number with E instead of e introducing the exponent. The exponent always contains at least two digits.
g , G	The float or double *arg* is printed in style f or e (or in style E in the case of a G format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style e 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.
c	The character *arg* is printed.
s	The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (\0) is encountered, or until the number of characters indicated by the precision specification is reached. If the precision is not specified, it is assumed to be infinite and all

characters up to the first null character are printed. A NULL value for *arg* will yield undefined results.

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

A non-existent or small field width will never 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` are printed in the same way as if *putchar* (3STDC) had been called.

EXAMPLES

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 %d, %d:%.2d", weekday, month, day, hour, min);
```

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`putchar`(3STDC) , `scanf`(3STDC)

NAME

rand, srand – pseudo random number generator

SYNOPSIS

```
#include <stdlib.h>
void srand(unsigned seed);

int rand(void);
```

DESCRIPTION

The *rand* function computes a sequence of pseudo-random integers in the range of 0 to RAND_MAX (as defined by the header file stdlib.h).

The *srand* function sets its argument as the seed for a new sequence of pseudo-random numbers to be returned by *rand* . These sequences are repeatable by calling *srand* with the same seed value.

If no seed value is provided, the functions are automatically seeded with a value of 1.

NOTE

Though these functions are reentrant, the state information is global to the actor. Therefore, repeatability of a given suite of numbers will not be experienced by several threads in parallel. For a reentrant repeatability of suites, see *rand_r(3STDC)* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`random`(3STDC) , `rand_r`(3STDC)

STANDARDS

The *rand* and *srand* functions conform to ANSI-C.

NAME

random, srandom, initstate, setstate – better random number generator

SYNOPSIS

#include <stdlib.h>
long **random**(void);

void **srandom**(unsigned *seed*);

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

char ***setstate**(char * *state*);

DESCRIPTION

The *random* function uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to $2^{31} - 1$. The period of this random number generator is very large, approximately $16 \times (2^{31} - 1)$.

The *random/srandom* functions have (almost) the same calling sequence and initialization properties as *rand/srand* (3STDC) The difference is that *rand* produces a much less random sequence — in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by *random* are usable. For example, *random* &01 will produce a random binary value.

Unlike *srand* , *srandom* does not return the old seed; the reason being that the amount of state information used is much more than a single word (two other routines are provided to deal with restarting / changing random number generators). Like *rand* , however, *random* will by default produce a sequence of numbers that can be duplicated by calling *srandom* with 1 as the seed.

The *initstate* routine allows a state array, passed as an argument, to be initialized for future use. The size of the state array (in bytes) is used by *initstate* to decide how sophisticated a random number generator it should use — the bigger the state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error.) The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. The *initstate* function returns a pointer to the previous state information array.

Once a state has been initialized, the *setstate* routine provides for rapid switching between states. The *setstate* function returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to *initstate* or *setstate* .

Once a state array has been initialized, it may be restarted at a different point either by calling *initstate* (with the desired seed, the state array, and its size) or by calling both *setstate* (with the state array) and *srandom* (with the desired seed).

The advantage of calling both *setstate* and *srandom* is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{690}, which should be sufficient for most purposes.

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` , it returns `NULL` and *random* uses a simple linear congruential random number generator.

DIAGNOSTICS

If *initstate* is called with less than 8 bytes of state information, or if *setstate* detects that the state information has been garbled, error messages are printed to the standard error output.

NOTE

Though these functions are reentrant, the state information is global to the actor. Therefore, repeatability of a given suite of number will not be experienced by several threads in parallel. For a reentrent repeatability of suites, see *rand_r(3STDC)* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`rand`(3STDC) , `rand_r`(3STDC)

RESTRICTIONS

random operates at about 2/3 the speed of *rand* (3STDC).

NAME

scanf, sscanf – convert formatted input

SYNOPSIS

#include <stdio.h>
int **scanf**(const char * *format*, ...);

int **sscanf**(const char * *s*, const char * *format*, ...);

DESCRIPTION

The `scanf( )` function reads from the standard input channel, which is operating system dependent. The `sscanf( )` function reads from the character string *s* . Each function reads characters, 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 control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. White-space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described below, cause input to be read up to the next non-white-space character.
2. An ordinary character (not `%`), which must match the next character of the input channel.
3. Conversion specifications, consisting of the character `%` , an optional assignment suppressing character `*` , an optional numerical maximum field width, an optional `l` (ell) or `h` indicating the size of the receiving variable, and a conversion code.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated using `*` . The suppression of assignment allows you to define an input filed to be ignlored. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted. For all descriptors except “[” and “c”, white space leading an input field is ignored.

The conversion code indicates the interpretation of the input field; the corresponding pointer argument must be of a restricted type. For a suppressed field, no pointer argument is given. The following conversion codes are legal:

`%`	a single `%` is expected in the input at this point; no assignment is done.
`d`	a decimal integer is expected; the corresponding argument should be an integer pointer.
`u`	an unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

`o`	an octal integer is expected; the corresponding argument should be an integer pointer.
`x`	a hexadecimal integer is expected; the corresponding argument should be an integer pointer.
`i`	an integer is expected; the corresponding argument should be an integer pointer. It will store the value of the next input item interpreted according to C conventions: a leading "0" implies octal; a leading "0x" implies hexadecimal; otherwise, decimal.
`n`	stores in an integer argument the total number of characters (including white space) that have been scanned so far since the function call. No input is consumed.
`e` , `f` , `g`	a floating point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float* . The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an `E` or an `e` , followed by an optional +, –, or space, followed by an integer.
`s`	a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating `\0` , which will be added automatically. The input field is terminated by a white-space character.
`c`	a character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-space character, use `%1s` . If a field width is given, the corresponding argument should refer to a character array; the number of characters indicated is read.
`[`	indicates string data and the normal skip over leading white space is suppressed. The left bracket is followed by a set of characters, called the *scanset*, and a right bracket; the input field is the maximaum sequence of input characters consisting entirely of characters in the scanset. The circumflex (`^`), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters *not* contained in the remainder of the scanset string. There are some conventions used in the construction of the scanset. A range of characters may be represented by the construct *first–last* , thus [0123456789] may be expressed [0–9]. Using this convention, *first* must be lexically less than or equal to `last` , otherwise the dash will stand for itself. The dash will also stand for itself whenever it is the first or the last character in the scanset. To

include the right square bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset, and in this case it will not be syntactically interpreted as the closing bracket. The corresponding argument must point to a character array large enough to hold the data field and the terminating `\0` , which will be added automatically. At least one character must match for this conversion to be considered successful.

The conversion characters `d` , `u` , `o` , and `x` may be preceded by `l` or `h` to indicate that a pointer to `long` or to `short` rather than to `int` is in the argument list. Similarly, the conversion characters `e` , `f` , and `g` may be preceded by `l` to indicate that a pointer to `double` rather than to `float` is in the argument list. The `l` or `h` modifier is ignored for other conversion characters.

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

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

EXAMPLES

The call:

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E−1 thompson
```

will assign to *n* the value `3` , to *i* the value `25` , to *x* the value `5.432` , and *name* will contain `thompson\0` .

Or:

```
int i; float x; char name[50];
(void) scanf("%2d%f%*d %[0−9]", &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* (3STDC) will return `a` .

NOTE

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

DIAGNOSTICS

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

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getchar`(3STDC) , `printf`(3STDC) , `strtod`(3STDC) , `strtol`(3STDC)

NAME

stdarg – variable argument lists

SYNOPSIS

#include <stdarg.h>
void **va_start**(va_list *ap*, *last*);

type **va_arg**(va_list *ap*, *type*);

void **va_end**(va_list *ap*);

DESCRIPTION

A function may be called with a varying number of arguments of a number of types. The include file declares a type *va_list* and defines three macros for stepping through a list of arguments whose number and types are not known to the called function.

The called function must declare an object of type *va_list* which is used by the macros *va_start*, *va_arg*, and *va_end*.

The *va_start* macro initializes *ap* for subsequent use by *va_arg* and *va_end*, and must be called first.

The parameter `last` is the name of the last parameter before the variable argument list, in other words, the last parameter of which the calling function knows the type.

Because the address of this parameter is used in the *va_start* macro, it should not be declared as a register variable, or as a function or an array type.

The *va_start* macro does not return a value.

The *va_arg* macro expands to an expression that has the type and value of the next argument in the call. The parameter *ap* is the *va_list* initialized by *va_start*. Each call to *va_arg* modifies *ap* so that the next call returns the next argument. The parameter `type` is a type—name specified to allow the type of pointer to an object of the specified type can be obtained simply by adding a * to *type*.

If there is no next argument, or if `type` is not compatible with the actual type of the next argument (as promoted according to the default argument promotions), random errors will occur.

The first use of the *va_arg* macro after that of the *va_start6* macro returns the argument after *last*. Successive invocations return the values of the remaining arguments.

The *va_end* macro handles a normal return from the function whose variable argument list was initialized using *va_start*.

The *va_end* macro does not return a value.

EXAMPLES

```
void foo(char *fmt, ...)
{
        va_list ap;
        int d;
        char c, *p, *s;
        va_start(ap, fmt);
        while (*fmt)
                switch(*fmt++) {
                case 's':                       /* string */
                        s = va_arg(ap, char *);
                        printf("string %s\n", s);
                        break;
                case 'd':                       /* int */
                        d = va_arg(ap, int);
                        printf("int %d\n", d);
                        break;
                case 'c':                       /* char */
                        c = va_arg(ap, char);
                        printf("char %c\n", c);
                        break;
                }
        va_end(ap);
}
```

STANDARDS

The *va_start*, *va_arg*, and *va_end* macros conform to `ANSI-C.`

COMPATIBILITY

These macros are `not` compatible with the macros they replace. A backward compatible version can be found in the include file *varargs.h.*

RESTRICTIONS

Unlike the *varargs* macros, the *stdarg* macros do not permit programmers to code a function with no fixed arguments.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

assert, _assert, _stdc_assert – expression verification macro

SYNOPSIS

#include <assert.h>
assert *expression*

_assert *expression*
void **`_stdc_assert`**(const char * *file*, int *line*, const char * *expression*);

DESCRIPTION

The `_assert(x)` macro is defined as `assert(x)` . The `assert` macro tests the given *expression* and if it is false, calls `_stdc_assert()` . The `_stdc_assert()` function writes a diagnostic message to the error channel, and calls `abort`(3STDC) .

If the *expression* is true, the `assert` macro does nothing.

The `assert` macro may be rendered non-operational at compile time using the `NDEBUG` option.

DIAGNOSTICS

The following diagnostic message is written to the error channel if *expression* is false:

```
("assertion %s failed: file %s, line %d\
", expression, __FILE__, __LINE__)
```

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`abort`(3STDC)

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of `n` characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of `n` characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

strerror – system error messages

SYNOPSIS

#include <string.h>
char ***strerror**(int *errnum*);

DESCRIPTION

The *strerror* function look up the error message string corresponding to an error number. It accepts an error number argument *errnum* and returns a pointer to the corresponding message string.

If *errnum* is not a recognized error number, the error message string will contain "Unknown error:" followed by the error number in decimal notation.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`perror`(3STDC)

RESTRICTIONS

For unknown error numbers, *strerror* returns its result to a static buffer which could be overwritten by subsequent or concurrent calls.

NAME

strftime – format date and time

SYNOPSIS

#include <time.h>
size_t **strftime**(char **buf*, size_t *maxsize*, const char **format*, const struct tm **timeptr*);

DESCRIPTION

The *strftime* function formats the information from *timeptr* into the buffer *buf* according to the string pointed to by `format`.

The `format` string consists of zero or more conversion specifications and ordinary characters. All ordinary characters are copied directly into the buffer. A conversion specification consists of a percent sign (“%”) and one other character.

No more than *maxsize* characters will be placed into the array. If the total number of resulting characters, including the terminating null character, is not more than *maxsize*, *strftime* returns the number of characters in the array, not counting the terminating null. Otherwise, zero is returned.

Each conversion specification is replaced by the following characters which are then copied to the buffer.

`%A`	is replaced by the full weekday name.
`%a`	is replaced by the abbreviated weekday name, where the abbreviation is the first three characters.
`%B`	is replaced by the full month name.
`%b` or `%h`	is replaced by the abbreviated month name, where the abbreviation is the first three characters.
`%C`	is equivalent to "%a %b %e %H:%M:%S %Y" (the format produced by *asctime*(3STDC).
`%c`	is equivalent to "%m / %d / %y %H:%M:%S".
`%D`	is replaced by the date in the format "mm / dd / yy".
`%d`	is replaced by the day of the month as a decimal number (01-31).
`%e`	is replaced by the day of the month as a decimal number (1-31); single digits are preceded by a blank.
`%H`	is replaced by the hour (24-hour clock) as a decimal number (00-23).
`%I`	is replaced by the hour (12-hour clock) as a decimal number (01-12).
`%j`	is replaced by the day of the year as a decimal number (001-366).

`%k`	is replaced by the hour (24-hour clock) as a decimal number (0-23); single digits are preceded by a blank.
`%l`	is replaced by the hour (12-hour clock) as a decimal number (1-12); single digits are preceded by a blank.
`%M`	is replaced by the minute as a decimal number (00-59).
`%m`	is replaced by the month as a decimal number (01-12).
`%n`	is replaced by a newline.
`%p`	is replaced by either "AM" or "PM", as appropriate.
`%R`	is equivalent to "%H:%M".
`%r`	is equivalent to "%I:%M:%S %p".
`%t`	is replaced by a tab.
`%S`	is replaced by the second as a decimal number (00-60).
`%s`	is replaced by the number of seconds since the Epoch, UCT (see *mktime*(3STDC)).
`%T` or `%X`	is equivalent to "%H:%M:%S".
`%U`	is replaced by the week number of the year (Sunday as the first day of the week) as a decimal number (00-53).
`%W`	is replaced by the week number of the year (Monday as the first day of the week) as a decimal number (00-53).
`%w`	is replaced by the weekday (Sunday as the first day of the week) as a decimal number (0-6).
`%x`	is equivalent to "%m / %d / %y".
`%Y`	is replaced by the year with century as a decimal number.
`%y`	is replaced by the year without century as a decimal number (00-99).
`%Z`	is replaced by the time zone name.
`%%`	is replaced by ‘%’.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ctime`(3STDC), `printf`(3STDC)

STANDARDS

The *strftime* function conforms to `ANSI-C.` The '%s' conversion specification is an extension.

RESTRICTIONS

There is no conversion specification for the phase of the moon.

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of `n` characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of `n` characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>
int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

strsep – separate strings

SYNOPSIS

```
#include <string.h>
char *strsep(char **sp, const char *d);
```

DESCRIPTION

The *strsep* function locates, in the string referenced by **sp* , the first occurrence of any character in the string *d* (or the terminating null character) and replaces it with a 0 . The location of the next character after the delimiter character (or NULL , if the end of the string was reached) is stored in **sp*. The original value of **sp* is returned.

An “empty” field caused by two adjacent delimiter characters, can be detected by comparing the location referenced by the pointer returned in **sp* to 0 .

If **sp* is initially NULL , *strsep* returns NULL.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>

int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

string, strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcoll, strcpy, strdup, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strstr, strcspn – string operations

SYNOPSIS

#include <string.h>

int * **strcasecmp**(const char * *s1*, const char * *s2*);

int * **strncasecmp**(const char * *s1*, const char * *s2*, size_t *n*);

char * **strcat**(char * *s1*, const char * *s2*);

char * **strncat**(char * *s1*, const char * *s2*, size_t *n*);

int **strcmp**(const char * *s1*, const char * *s2*);

int **strcoll**(const char * *s1*, const char * *s2*);

char ***strdup**(const char * *s*);

int **strncmp**(const char * *s1*, const char * *s2*, size_t *n*);

char ***strcpy**(char * *s1*, const char * *s2*);

char ***strncpy**(char * *s1*, const char * *s2*, size_t *n*);

size_t **strlen**(const char * *s*);

char ***strchr**(const char * *s*, int *c*);

char ***strrchr**(const char * *s*, int *c*);

char ***strpbrk**(const char * *s1*, const char * *s2*);

size_t **strspn**(const char * *s1*, const char * *s2*);

char ***strstr**(const char * *s1*, const char * *s2*);

size_t **strcspn**(const char * *s1*, const char * *s2*);

DESCRIPTION

The *s1* , *s2* and *s* arguments point to strings (arrays of characters terminated by a null character). The *strcat* , *strncat* , *strcpy* and *strncpy* functions all alter *s1*. These functions do not check for overflow of the array pointed to by *s1* .

The *strcasecmp* and *strncasecmp* functions compare the null-terminated strings *s1* and *s2* and return an integer greater than, equal to, or less than 0, according to whether *s1* is lexicographically greater than, equal to, or less than *s2* (after translation of each corresponding character to lower-case). The strings themselves are not modified. The comparison is done using unsigned characters, meaning that 200 is greater than 0 .

The *strncasecmp* compares a maximum of n characters.

The *strcat* and *strncat* functions append a copy of string *s2* to the end of string *s1* . The *strncat* function copies only the first *n* bytes of *s2* . Each returns a pointer to the null-terminated result.

The *strcmp* function compares its arguments and returns an integer less than, equal to, or greater than 0, according to whether *s1* is lexicographically less than, equal to, or greater than *s2* .

If insufficient memory is available, NULL is returned. The *strncmp* function makes the same comparison, but looks at a maximum of *n* characters.

The *strcoll* function lexicographically compares the null-terminated strings *s1* and *s2* according to the current locale collation and returns an integer greater than, equal to, or less than 0, according to whether *s1* is greater than, equal to, or less than *s2* .

The *strdup* function allocates sufficient memory for a copy of the string *s* , does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free(3STDC).

The *strcpy* function copies string *s2* to *s1* , stopping after the null character has been copied. *strncpy* copies exactly *n* characters, 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 *strlen* function returns the number of characters in *s* , not including the terminating null character.

The *strchr* and *strrchr* functions return a pointer to the first or last)cccurrence of character *c* in string *s* , respectively,. If *c* does not occur in the string, a NULL pointer is returned. The null character terminating a string is considered to be part of the string.

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 *strspn* and *strcspn* functions return the length of the initial segment of string *s1* which consists entirely of characters from or to string *s2* , respectively.

The *strstr* function locates the first occurence of the null-terminated string *s2* in the null-terminated string *s1* . If *s2* is the empty string, *strstr* returns *s1* ; if *s2* occurs nowhere in *s1* , *strstr* returns NULL , otherwise *strstr* returns a pointer to the first character of the first occurrence of *s2* .

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

strtod, atof – convert an ASCII string to a floating-point number

SYNOPSIS

#include <stdlib.h>
double **strtod**(const char * *str*, char ** *ptr*);

double **atof**(const char * *str*);

DESCRIPTION

The *strtod* function returns as a double-precision floating-point number the value represented by the character string pointed to by *str* . The string is scanned up to the first unrecognized character.

The *strtod* function recognizes an optional string of white-space characters, then an optional sign, then a string of digits optionally containing a decimal point, then an optional `e` or `E` followed by an optional sign or space, followed by an integer.

If the value of *ptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr* . If a number cannot be formed, **ptr* is set to *str* , and zero is returned.

The *atof(str)* call is equivalent to *strtod(str, (char **)NULL)* .

DIAGNOSTICS

If the correct value would cause overflow, HUGE is returned (according to whether the value is positive or negative), and, in if supported, *errno* is set to ERANGE If the correct value would cause underflow, zero is returned and, if supported, *errno* is set to ERANGE.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`scanf`(3STDC)

NAME

strtok – string tokens

SYNOPSIS

```
#include <string.h>
char *strtok(char *str, const char *sep);
```

DESCRIPTION

The *strtok* function is used to isolate sequential tokens in a null-terminated string, *str*. These tokens are separated in the string by at least one of the characters in *sep*. The first time *strtok* is called, *str* should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass the NULL pointer instead. The separator string, *sep*, must be supplied each time, and may change between calls.

The *strtok* function returns a pointer to the beginning of each subsequent token in the string, after replacing the token itself with a null-character. When no more tokens remain, the NULL pointer is returned.

NOTES

The interface is inappropriate to a thread-safe implementation. Therefore this function is not reentrant. For a reentrant equivalent, use *strtok_r* (3STDC), which conforms to `POSIX.1c`.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`index`(3STDC), `memchr`(3STDC), `rindex`(3STDC), `string`(3STDC), `strcspn`(3STDC), `strsep`(3STDC), `strtok_r`(3STDC)

STANDARDS

The *strtok* function conforms to `ANSI-C`.

RESTRICTIONS

It is not possible to get tokens from multiple strings simultaneously.

The System V *strtok*, if handed a string containing only delimiter characters, will not alter the next starting point, so that a call to *strtok* with a different (or empty) delimiter string may return a non- NULL value. As this implementation always alters the next starting point, this sequence of calls will always return NULL.

NAME

strtok_r – string tokens reentrant

SYNOPSIS

#include <string.h>
char ***strtok_r**(char **str*, const char **sep*, char ***last*);

DESCRIPTION

The *strtok_r* function is used to isolate sequential tokens in a null-terminated string, *str*. These tokens are separated in the string by at least one of the characters in *sep*. The *strtok_r* function performs the same task as *strtok* (3STDC) , except the current position in the string is recorded in **last*. This parameter is used the following way; if *str* is null, **last* is used as the starting point. Otherwise, the value of **last* is unimportant. This routine can therefore be used exactly like *strtok* (3STDC) , except that the extra parameter `last` must point at proper storage for a character pointer. It can be useful for setting *str* to NULL and initialising **last* the first time, thus making all invocations look the same. It can also be useful to modify the information returned in **last* or to use it to compute the next value for the *str* parameter, or a combination of these methods.

When no more tokens remain, a null pointer is returned. and **last* is set to NULL.

NOTE

This function is fully reentrant. It is the application's responsability to protect `last` and *str* against concurrent manipulations, if necessary. The invoking thread's stack is the best place to store **last*.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`index`(3STDC), `memchr`(3STDC), `rindex`(3STDC), `string`(3STDC), `strsep`(3STDC), `strtok`(3STDC)

STANDARDS

The *strtok_r* function conforms to `POSIX.1c.`

RESTRICTIONS

It is not always possible to get tokens from multiple strings simultaneously.

NAME

strtol, atol, atoi – convert string to integer

SYNOPSIS

#include <stdlib.h>
long **strtol**(const char * *str*, char ** *ptr*, int *base*);

long **atol**(const char * *str*);

int **atoi**(const char * *str*);

DESCRIPTION

The *strtol* function returns the value represented by the character string pointed to by *str* as a long integer. The string is scanned up to the first character inconsistent with the base. Leading "white-space" characters (as defined by *isspace* in *ctype* (3STDC)) are ignored.

The input string is divided into three parts: an initial, possibly empty, sequence of white-space characters (as defined by *isspace* in *ctype* (3STDC)); 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. The *strtol* function attempts to convert the subject sequence to an integer and return the result.

A pointer to the final string is stored in the object pointed to by ptr, provided it is not a null pointer.

If *base* is positive, it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and "0x" or "0X" is ignored if *base* is 16.

If *base* is zero, the string itself determines the base as follows: After an optional leading sign, a leading zero indicates octal conversion, and a leading "0x" or "0X" hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from long to int can be done upon assignment, or by using an explicit cast.

atol(str) is equivalent to `strtol(str, (char **)NULL, 10)`.

atoi(str) is equivalent to `(int) strtol(str, (char **)NULL, 10)`.

RETURN VALUES

Upon successful completion *strtol* returns the converted value, if any. If no conversion could be performed, 0 is returned.

If the correct value is outside the range of representable values, LONG_MAX or LONG_MIN is returned (according to the sign of the value), and errno is set to ERANGE.

USAGE

Because LONG_MIN and LONG_MAX are returned on error and are also valid returns on success, in order to check for error situations, an application should set errno to 0, then call *strtol* , then check errno; if it is non-zero, you can assume that an error has occurred.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ctype`(3STDC) , `scanf`(3STDC) , `strtod`(3STDC)

NAME

strtoul – convert a string to an unsigned long or uquad_t integer

SYNOPSIS

```
#include <stdlib.h>
#include <limits.h>
unsigned long strtoul(const char *nptr, char **endptr, int base);
```

DESCRIPTION

The *strtoul* function converts the string in *nptr* to an *unsigned long* value. The conversion is done according to the *base* given, which must be between 2 and 36 inclusive, or be the special value of 0.

The string may begin with an arbitrary amount of white space (as determined by *isspace* (3STDC)) followed by a single optional + or – sign. If *base* is zero or 16, the string may then include a *0x* prefix, and the number will be read in base 16; otherwise, a zero *base* is taken as 10 (decimal) unless the next character is 0, in which case it is taken as 8 (octal).

The remainder of the string is converted to an *unsigned long* value stopping at the end of the string or at the first character that does not produce a valid digit in the base given. (In bases above 10, the letter *A* in either upper or lowercase represents 10, *B* represents 11, and so forth, with *Z* representing 35.)

If *endptr* is non nil, *strtoul* stores the address of the first invalid character in **endptr*. If there were no digits at all, however, *strtoul* stores the original value of *nptr* in **endptr*. (Thus, if **nptr* is not 0 but ***endptr* is 0 on return, the entire string will have been valid.)

RETURN VALUES

The *strtoul* function returns either the result of the conversion or, if there was a leading minus sign, the negation of the result of the conversion, unless the original (non-negated) value would overflow. In that case, *strtoul* returns ULONG_MAX and, in contexts where it is supported, sets the global variable *errno* to ERANGE.

ERRORS

The string given was out of range; the converted value has been clamped.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`strtol`(3STDC)

STANDARDS

The *strtoul* function conforms to `ANSI-C.`

NAME

strxfrm – transform a string under locale

SYNOPSIS

#include <string.h>
size_t **strxfrm**(char **dst*, const char **src*, size_t *n*);

DESCRIPTION

strxfrm does something horrible (see `ANSI` standard). In this implementation it just copies.

STANDARDS

The *strxfrm* function conforms to `ANSI-C,` given that the *setlocale* function is not supported.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

swab – swap adjacent bytes

SYNOPSIS

#include <string.h>
void **swab**(const void **src*, void **dst*, size_t *len*);

DESCRIPTION

The *swab* function copies *nbytes* bytes, which are pointed to by *src*, to the object pointed to by *dst*, exchanging adjacent bytes. The *len* argument should be even. If *len* is odd, *swab* copies and exchanges *len*`-1` bytes and the disposition of the last byte is unspecified. Copying between objects that overlap can lead to unpredictable results. If *len* is negative, *swab* does nothing.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`bzero`(3STDC), `memset`(3STDC)

NAME

perror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

```
#include <stdio.h>
void perror(const char * s);
```

```
#include <errno.h>
extern char *sys_errlist[];
extern int sys_nerr;
```

DESCRIPTION

The *perror* function produces a message on the error channel, the implementation of which is system-dependent. The message describes the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a newline character. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the per thread variable *errno*, or from a global variable *errno*, whichever is provided by the library. This variable is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new line. The *sys_nerr* parameter defines the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

perror, errno, sys_errlist, sys_nerr – system error messages

SYNOPSIS

#include <stdio.h>
void **perror**(const char * *s*);

```
#include <errno.h>
extern char *sys_errlist[];
extern int sys_nerr;
```

DESCRIPTION

The *perror* function produces a message on the error channel, the implementation of which is system-dependent. The message describes the last error encountered during a call to a system or library function. The argument string *s* is printed first, then a colon and a blank, then the message and a newline character. To be of most use, the argument string should include the name of the program that incurred the error. The error number is taken from the per thread variable *errno*, or from a global variable *errno*, whichever is provided by the library. This variable is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the array of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new line. The *sys_nerr* parameter defines the largest message number provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

tmpnam, tempnam – create a name for a temporary file

SYNOPSIS

#include <stdio.h>
char * **tmpnam**(char * *s*);

char ***tempnam**(const char * *tmpdir*, const char * *prefix*);

DESCRIPTION

This function generates file names that can be used safely for a temporary file.

The *tmpnam* function always generates a file name using the path-prefix defined as *P_tmpdir* in the `<stdio.h>` header file. If *s* is NULL, *tmpnam* leaves its result in an internal per thread area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least *L_tmpnam* bytes, where *L_tmpnam* is a constant defined in `<stdio.h>` ; *tmpnam* places its result in that array and returns *s* . The *tempnam* function is similar to *tmpnam* , but provides the ability to specify the directory which will contain the temporary file and the file name prefix.

The environment variable TMPDIR (if set), the argument *tmpdir* (if not NULL) , the directory *P_tmpdir* , and the directory `/tmp` are tried, in the order listed, as directories in which to store the temporary file.

The argument *prefix,* if not NULL , is used to specify a file name prefix, which will be the first part of the created file name.

The *tempnam* function allocates memory in which to store the file name. The pointer returned may be used as a subsequent argument to *free(3STDC)* .

NOTES

This function generates a different file name each time it is called.

Files created using this function and *fopen* (3STDC) are temporary only in the sense that they reside in a directory intended for temporary use, and that their names are unique.

For e thread safety, *tmpnam* allocates a per-thread buffer. For this buffer to be freed upon thread deletion, the *ptdThreadDelete* (2K) function must be called.

RESTRICTIONS

If called more than 17,576 times in a single actor, this function will start recycling previously used names. Between the time a file is created and it is opened, it is possible for another actor to create a file with the same name. This can never happen, however, if the other actor is using this function and file names are chosen in order to render duplication by other means unlikely.

ERRORS

The *tmpnam* function may fail and set errno for any of the errors specified for the library function *mktemp* (3STDC).

The *tempnam* function may fail and set errno for any of the errors specfied for the library functions *malloc* (3STDC) or *mktemp* (3STDC).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC) , `malloc`(3STDC) , `tmpfile`(3STDC)

NAME

thread_once – execute an init routine once

SYNOPSIS

#include <stdlib.h>
int **thread_once**(thread_once_t **once_control*, void (**init_routine*)(void));

DESCRIPTION

The first call to *thread_once* by any thread in an actor, with a given *once_control*, will call the *init_routine* with no arguments. Subsequent calls to *thread_once* with the same *once_control* will not call the *init_routine*. On return from *thread_once*, it is guaranteed that *init_routine* has completed. The *once_control* parameter is used to determine whether the associated initialization routine has been called.

The behaviour of *thread_once* is undefined if *once_control* has an automatic storage duration or is not initialized by zero.

DIAGNOSTICS

Upon completion, *thread_once* returns zero.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

time – get time

SYNOPSIS

```
#include <time.h>
time_t time(time_t *tloc);
```

DESCRIPTION

The `time` function returns the value of time in seconds since Epoch (00:00:00 UTC, January 1, 1970).

The *tloc* argument points to an area where the return value is also stored. If *tloc* is a NULL pointer, no value is stored.

The `time` function relies on the *univTime(2K)* call to retrieve the current time. If the time was not properly set at system initialization (see *univTimeSet(2K)*), `time` returns a value of -1, otherwise the value of time is returned.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`univTime`(2K), `univTimeSet`(2K)

NAME

tmpfile – create a temporary file

SYNOPSIS

```
#include <stdio.h>
FILE *tmpfile(void);
```

DESCRIPTION

The *tmpfile* function creates a temporary file using a name generated by *tmpnam*(3STDC), and returns a corresponding FILE pointer. If the file cannot be opened, an error message is printed using *perror*(3STDC), and a NULL pointer is returned. The file will automatically be deleted when the process using it terminates. The file is opened for update ("w+").

RESTRICTIONS

If a thread is deleted while performing tmpfile, it is possible that the temporary file will not be deleted when the program terminates.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC), `perror`(3STDC), `tmpnam`(3STDC)

NAME

tmpnam, tempnam – create a name for a temporary file

SYNOPSIS

#include <stdio.h>
char * **tmpnam**(char * *s*);

char ***tempnam**(const char * *tmpdir*, const char * *prefix*);

DESCRIPTION

This function generates file names that can be used safely for a temporary file.

The *tmpnam* function always generates a file name using the path-prefix defined as *P_tmpdir* in the `<stdio.h>` header file. If *s* is NULL, *tmpnam* leaves its result in an internal per thread area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If *s* is not NULL, it is assumed to be the address of an array of at least *L_tmpnam* bytes, where *L_tmpnam* is a constant defined in `<stdio.h>` ; *tmpnam* places its result in that array and returns *s* . The *tempnam* function is similar to *tmpnam* , but provides the ability to specify the directory which will contain the temporary file and the file name prefix.

The environment variable TMPDIR (if set), the argument *tmpdir* (if not NULL) , the directory *P_tmpdir* , and the directory `/tmp` are tried, in the order listed, as directories in which to store the temporary file.

The argument *prefix,* if not NULL , is used to specify a file name prefix, which will be the first part of the created file name.

The *tempnam* function allocates memory in which to store the file name. The pointer returned may be used as a subsequent argument to *free(3STDC)* .

NOTES

This function generates a different file name each time it is called.

Files created using this function and *fopen* (3STDC) are temporary only in the sense that they reside in a directory intended for temporary use, and that their names are unique.

For e thread safety, *tmpnam* allocates a per-thread buffer. For this buffer to be freed upon thread deletion, the *ptdThreadDelete* (2K) function must be called.

RESTRICTIONS

If called more than 17,576 times in a single actor, this function will start recycling previously used names. Between the time a file is created and it is opened, it is possible for another actor to create a file with the same name. This can never happen, however, if the other actor is using this function and file names are chosen in order to render duplication by other means unlikely.

ERRORS

The *tmpnam* function may fail and set errno for any of the errors specified for the library function *mktemp* (3STDC).

The *tempnam* function may fail and set errno for any of the errors specfied for the library functions *malloc* (3STDC) or *mktemp* (3STDC).

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fopen`(3STDC) , `malloc`(3STDC) , `tmpfile`(3STDC)

NAME

toascii – convert a byte to 7-bit ASCII

SYNOPSIS

```
#include <ctype.h>
int toascii(int c);
```

DESCRIPTION

The *toascii* function strips all but the low 7 bits from a letter, including parity or other marker bits.

RETURN VALUES

The *toascii* function returns a valid `ASCII` character. This character is `ASCII` only according to the default *locale*. If another `locale` is currently in effect, the semantical correctness of the result is unspecified.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`isascii`(3STDC), `ctype`(3STDC), `isalnum`(3STDC), `isalpha`(3STDC), `iscntrl`(3STDC), `isdigit`(3STDC), `isgraph`(3STDC), `islower`(3STDC), `isprint`(3STDC), `ispunct`(3STDC), `isspace`(3STDC), `isupper`(3STDC), `isxdigit`(3STDC), `tolower`(3STDC), `toupper`(3STDC)

NOTES

This macro is only available in sources which have not used _POSIX_SOURCE or _ANSI_SOURCE flags when being compiled.

STANDARDS

Due to its dubious validity when used in conjunction with *setlocale*, this function is no longer a part of `ANSI-C.`

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

```
#include <ctype.h>
int isalpha(int c);
```

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, tolower, toupper – classify characters

SYNOPSIS

All functions described in this page have the same syntax.

#include <ctype.h>
int **isalpha**(int *c*);

DESCRIPTION

These macros classify character-coded integer values by looking them up in a table. Each is a predicate returning nonzero for true, or zero for false.

isalpha	*c* is a letter.
isupper	*c* is an upper-case letter.
islower	*c* is a lower-case letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F] or [a-f].
isalnum	*c* is an alphanumeric (letter or digit).
isspace	*c* is a space, tab, carriage return, new-line, vertical tab, or form-feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) to 0176 (tilde).
isgraph	*c* is a printing character, like *isprint* except for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).

The conversion functions and macros translate a character from lowercase (uppercase) to uppercase (lowercase).

tolower	If *c* is a character for which *isupper* is true and there is a corresponding lowercase character, *tolower* returns the corresponding lowercase character. Otherwise, the character is returned unchanged.

toupper	If *c* is a character for which *islower* is true and there is a corresponding uppercase character, *toupper* returns the corresponding uppercase character. Otherwise, the character is returned unchanged.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

tzset – set time conversion information

SYNOPSIS

```
#include <time.h>
void tzset(void);
extern char *tzname[2];
```

DESCRIPTION

The *tzset* function uses the value of the environment variable TZ to set time conversion information used by *localtime(3STDC)*, *ctime(3STDC)*, *strftime(3STDC)*, and *mktime(3STDC)*.

When *tzset* is called, the time zone names contained in the external variable *tzname* are set according to the contents of TZ.

The format of TZ is:

```
stdoffset[dst[offset][,start/[time],end[/time]]]
```

where:

std and dst	Indicate no less than three and no more than TZNAME_MAX bytes, which designate the standard (*std*) and daylight saving time (*dst*) time zones. Only *std* is required,; if *dst* is not specifed, daylight saving time does not apply in this area. Upper- and lowercase letters are allowed. Any characters except a leading colon (:), digits, a comma (,), a minus (-), a plus (+) or a null character are allowed.
offset	Indicates the value to be added to the local time to arrive at Coordinated Universal Time. The *offset* has the form: *hh[:mm[:ss]]*. The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst* , daylight saving time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between 0 and 24, and the minutes and seconds (if present) between 0 and 59. Entering a value that is out of range may produce unpredictable results. If preceded by a "-", the time zone is east of the Prime Meridian; otherwise it is west (which may be indicated by an optional preceding "+" sign).
start/time,end/time	Indicates when to change to and back from daylight saving time, where *start/time* describes when the change from standard time to daylight

saving time happens, and *end/time* describes when the change back happens. Each time field describes when, in current local time, the change to the other time is made. The formats of *start* and *end* are one of the following:

Jn The Julian day *n* (1 <= *n* <= 365). Leap days are not counted. That is, in all years, February 28 is day 59 and March 1 is day 60. It is impossible to refer to February 29.

n The zero-based Julian day (0 <= *n* <= 365). Leap days are counted, and it is possible to refer to February 29.

Mm.n.d The *d*th day, (0 <= *d* <= 6) of week *n* of month *m* of the year (1 <= *n* <= 5, 1 <= *m* <= 12), where week 5 means "the last *d*-day in month *m*" which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*th day occurs. Day zero is Sunday.

Implementation-specific defaults are used for *start* and *end* if these optional fields are not given. The `time` has the same format as *offset* except that no leading sign ("-" or "+") is allowed. The default, if `time` is not given is 02:00:00.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`ctime`(3STDC), `getenv`(3STDC), `localtime`(3STDC), `mktime`(3STDC), `setenv`(3STDC), `strftime`(3STDC)

NAME

ungetc – push character back into input stream

SYNOPSIS

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

DESCRIPTION

The *ungetc* function inserts the character *c* into the buffer associated with an input *stream*. The *c* character will be returned by the next *getc* (3STDC) call on that *stream*. The *ungetc* function returns *c*, and leaves the *stream* file unchanged.

One character of pushback is guaranteed, provided something has already been read from the stream and the stream is actually buffered.

If *c* equals EOF, *ungetc* does nothing to the buffer and returns EOF.

Using *fseek*(3STDC) erases all memory of inserted characters.

RETURN VALUES

The *ungetc* function returns EOF if it cannot insert the character.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fseek`(3STDC), `getc`(3STDC), `setbuf`(3STDC)

NAME

unlocked, getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked – explicit locking functions

SYNOPSIS

#include <stdio.h>
int **getc_unlocked**(FILE * *stream*);

int **getchar_unlocked**(void);

int **putc_unlocked**(int *c*, FILE * *stream*);

int **putchar_unlocked**(int *c*);

DESCRIPTION

The *getc_unlocked* , *getchar_unlocked* , *putc_unlocked* and *putchar_unlocked* are functionally identical to *getc* , *getchar* , *putc* and *putchar* functions with the exception that they are not re-entrant.

getc_unlocked , *getchar_unlocked* , and *putchar_unlocked* routines are implemented as macros.

They may only safely be used within a scope protected by *flockfile* (or *ftrylockfile*) and *funlockedfile* .

STANDARDS

These routines conform to the POSIX.1c standards.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`getc`(3STDC) , `getchar`(3STDC) , `putc`(3STDC) , `putchar`(3STDC) , `flockfile`(3STDC)

NAME

getenv, putenv, setenv, unsetenv – fetch and set environment variables

SYNOPSIS

#include <stdlib.h>
char * **getenv**(const char * *name*);

int **setenv**(const char * *name*, const char * *value*, int *overwrite*);

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

void **unsetenv**(const char * *name*);

DESCRIPTION

These functions set, unset and fetch environment variables from the host *environment* list. For compatibility with differing environment conventions, the *name* and *value* arguments given may be appended and prepended, respectively, with an equal sign. The *getenv* function obtains the current value of the environment variable, *name*. If the variable *name* is not in the current environment, a null pointer is returned.

The `setenv` function inserts or resets the environment variable *name* in the current environment list. If the variable *name* does not exist in the list, it is inserted with the given *value*. If the variable does exist, the *overwrite* argument is tested; if *overwrite* is zero, the variable is not reset, otherwise it is reset to the given *value*.

The *putenv* function takes an argument of the form `name=value` and is equivalent to: `setenv(name, value, 1)`.

The `unsetenv` function deletes all instances of the variable name pointed to by *name* from the list.

RETURN VALUES

The `setenv` and *putenv* functions return zero if successful; otherwise –1 is returned. The `setenv` or *putenv* functions fail if they were unable to allocate memory for the environment.

STANDARDS

The *getenv* function conforms to `ANSI-C`.

NOTE

These functions are reentrant, but the environment is global to the actor.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

NAME

vfprintf – print formatted output

SYNOPSIS

#include <stdio.h>
#include <varargs.h>
int **vfprintf**(FILE **stream*, const char **format*, va_list *ap*);

DESCRIPTION

The *vfprintf* function is the same as *fprintf* (3STDC) with the exception that it is called with an argument list as defined by the `<varargs.h>` header file.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`fprintf`(3STDC), `stdarg`(3STDC)

NAME

vprintf, vsprintf, vsnprintf – print formatted output

SYNOPSIS

#include <stdio.h>
#include <varargs.h>
int **vprintf**(const char * *format*, va_list *ap*);

int **vsprintf**(char * *s*, const char * *format*, va_list *ap*);

int **vsnprintf**(char * *s*, size_t *size*, const char * *format*, va_list *ap*);

DESCRIPTION

The *vprintf*, *vsprintf*, and *vsnprintf* functions are the same as *printf* (3STDC), *sprintf* (3STDC), and *snprintf* (3STDC) functions respectively, with the exception that they are called with an argument list as defined by the `<varargs.h>` header file.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`printf`(3STDC)

NAME

vprintf, vsprintf, vsnprintf – print formatted output

SYNOPSIS

#include <stdio.h>
#include <varargs.h>
int **vprintf**(const char * *format*, va_list *ap*);

int **vsprintf**(char * *s*, const char * *format*, va_list *ap*);

int **vsnprintf**(char * *s*, size_t *size*, const char * *format*, va_list *ap*);

DESCRIPTION

The *vprintf*, *vsprintf*, and *vsnprintf* functions are the same as *printf* (3STDC), *sprintf* (3STDC), and *snprintf* (3STDC) functions respectively, with the exception that they are called with an argument list as defined by the `<varargs.h>` header file.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`printf`(3STDC)

NAME

vprintf, vsprintf, vsnprintf – print formatted output

SYNOPSIS

#include <stdio.h>
#include <varargs.h>
int **`vprintf`**(const char * *format*, va_list *ap*);

int **`vsprintf`**(char * *s*, const char * *format*, va_list *ap*);

int **`vsnprintf`**(char * *s*, size_t *size*, const char * *format*, va_list *ap*);

DESCRIPTION

The *vprintf*, *vsprintf*, and *vsnprintf* functions are the same as *printf* (3STDC), *sprintf* (3STDC), and *snprintf* (3STDC) functions respectively, with the exception that they are called with an argument list as defined by the `<varargs.h>` header file.

ATTRIBUTES

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

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Evolving

SEE ALSO

`printf`(3STDC)

Index