The Red Hat newlib C Library

Full Configuration

`libc` 2.5.0
December 2016

Steve Chamberlain
Roland Pesch
Red Hat Support
Jeff Johnston

sac@cygnus.com, pesch@cygnus.com, jjohnstn@redhat.com *The Red Hat newlib C Library*
Copyright © 1992, 1993, 1994-2004 Red Hat Inc.

'`libc`' includes software developed by the University of California, Berkeley and its contributors.

'`libc`' includes software developed by Martin Jackson, Graham Haley and Steve Chamberlain of Tadpole Technology and released to Cygnus.

'`libc`' uses floating-point conversion software developed at AT&T, which includes this copyright information:

The author of this software is David M. Gay.

Copyright (c) 1991 by AT&T.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR AT&T MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, subject to the terms of the GNU General Public License, which includes the provision that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

1 Introduction

This reference manual describes the functions provided by the Red Hat "newlib" version of the standard ANSI C library. This document is not intended as an overview or a tutorial for the C library. Each library function is listed with a synopsis of its use, a brief description, return values (including error handling), and portability issues.

Some of the library functions depend on support from the underlying operating system and may not be available on every platform. For embedded systems in particular, many of these underlying operating system services may not be available or may not be fully functional. The specific operating system subroutines required for a particular library function are listed in the "Portability" section of the function description. See Chapter 13 [Syscalls], page 319, for a description of the relevant operating system calls.

2 Standard Utility Functions (‘stdlib.h’)

This chapter groups utility functions useful in a variety of programs. The corresponding declarations are in the header file ‘stdlib.h’.

2.1 _Exit—end program execution with no cleanup processing

Synopsis

```
#include <stdlib.h>
void _Exit(int code);
```

Description

Use `_Exit` to return control from a program to the host operating environment. Use the argument *code* to pass an exit status to the operating environment: two particular values, `EXIT_SUCCESS` and `EXIT_FAILURE`, are defined in '`stdlib.h`' to indicate success or failure in a portable fashion.

`_Exit` differs from `exit` in that it does not run any application-defined cleanup functions registered with `atexit` and it does not clean up files and streams. It is identical to `_exit`.

Returns

`_Exit` does not return to its caller.

Portability

`_Exit` is defined by the C99 standard.

Supporting OS subroutines required: `_exit`.

2.2 a64l, l64a—convert between radix-64 ASCII string and long

Synopsis

```
#include <stdlib.h>
long a64l(const char *input);
char *l64a(long input);
```

Description

Conversion is performed between long and radix-64 characters. The `l64a` routine transforms up to 32 bits of input value starting from least significant bits to the most significant bits. The input value is split up into a maximum of 5 groups of 6 bits and possibly one group of 2 bits (bits 31 and 30).

Each group of 6 bits forms a value from 0–63 which is translated into a character as follows:

- 0 = ’.’
- 1 = ’/’
- 2–11 = ’0’ to ’9’
- 12–37 = ’A’ to ’Z’
- 38–63 = ’a’ to ’z’

When the remaining bits are zero or all bits have been translated, a null terminator is appended to the string. An input value of 0 results in the empty string.

The `a64l` function performs the reverse translation. Each character is used to generate a 6-bit value for up to 30 bits and then a 2-bit value to complete a 32-bit result. The null terminator means that the remaining digits are 0. An empty input string or NULL string results in 0L. An invalid string results in undefined behavior. If the size of a long is greater than 32 bits, the result is sign-extended.

Returns

`l64a` returns a null-terminated string of 0 to 6 characters. `a64l` returns the 32-bit translated value from the input character string.

Portability

`l64a` and `a64l` are non-ANSI and are defined by the Single Unix Specification.

Supporting OS subroutines required: None.

2.3 `abort`—abnormal termination of a program

Synopsis

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

Description

Use `abort` to signal that your program has detected a condition it cannot deal with. Normally, `abort` ends your program's execution.

Before terminating your program, `abort` raises the exception `SIGABRT` (using '`raise(SIGABRT)`'). If you have used `signal` to register an exception handler for this condition, that handler has the opportunity to retain control, thereby avoiding program termination.

In this implementation, `abort` does not perform any stream- or file-related cleanup (the host environment may do so; if not, you can arrange for your program to do its own cleanup with a `SIGABRT` exception handler).

Returns

`abort` does not return to its caller.

Portability

ANSI C requires `abort`.

Supporting OS subroutines required: `_exit` and optionally, `write`.

2.4 abs—integer absolute value (magnitude)

Synopsis

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

Description

`abs` returns $|x|$, the absolute value of *i* (also called the magnitude of *i*). That is, if *i* is negative, the result is the opposite of *i*, but if *i* is nonnegative the result is *i*.

The similar function `labs` uses and returns `long` rather than `int` values.

Returns

The result is a nonnegative integer.

Portability

`abs` is ANSI.

No supporting OS subroutines are required.

2.5 `assert`—macro for debugging diagnostics

Synopsis

```
#include <assert.h>
void assert(int expression);
```

Description

Use this macro to embed debuggging diagnostic statements in your programs. The argument *expression* should be an expression which evaluates to true (nonzero) when your program is working as you intended.

When *expression* evaluates to false (zero), `assert` calls `abort`, after first printing a message showing what failed and where:

```
Assertion failed: expression, file filename, line lineno, function: func
```

If the name of the current function is not known (for example, when using a C89 compiler that does not understand __func__), the function location is omitted.

The macro is defined to permit you to turn off all uses of `assert` at compile time by defining `NDEBUG` as a preprocessor variable. If you do this, the `assert` macro expands to

```
(void(0))
```

Returns

`assert` does not return a value.

Portability

The `assert` macro is required by ANSI, as is the behavior when `NDEBUG` is defined.

Supporting OS subroutines required (only if enabled): `close`, `fstat`, `getpid`, `isatty`, `kill`, `lseek`, `read`, `sbrk`, `write`.

2.6 atexit—request execution of functions at program exit

Synopsis

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

Description

You can use `atexit` to enroll functions in a list of functions that will be called when your program terminates normally. The argument is a pointer to a user-defined function (which must not require arguments and must not return a result).

The functions are kept in a LIFO stack; that is, the last function enrolled by `atexit` will be the first to execute when your program exits.

There is no built-in limit to the number of functions you can enroll in this list; however, after every group of 32 functions is enrolled, `atexit` will call `malloc` to get space for the next part of the list. The initial list of 32 functions is statically allocated, so you can always count on at least that many slots available.

Returns

`atexit` returns `0` if it succeeds in enrolling your function, `-1` if it fails (possible only if no space was available for `malloc` to extend the list of functions).

Portability

`atexit` is required by the ANSI standard, which also specifies that implementations must support enrolling at least 32 functions.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

2.7 atof, atoff—string to double or float

Synopsis

```
#include <stdlib.h>
double atof(const char *s);
float atoff(const char *s);
```

Description

`atof` converts the initial portion of a string to a `double`. `atoff` converts the initial portion of a string to a `float`.

The functions parse the character string *s*, locating a substring which can be converted to a floating-point value. The substring must match the format:

```
[+|-]digits[.][digits][(e|E)[+|-]digits]
```

The substring converted is the longest initial fragment of *s* that has the expected format, beginning with the first non-whitespace character. The substring is empty if `str` is empty, consists entirely of whitespace, or if the first non-whitespace character is something other than `+`, `-`, `.`, or a digit.

`atof(s)` is implemented as `strtod(s, NULL)`. `atoff(s)` is implemented as `strtof(s, NULL)`.

Returns

`atof` returns the converted substring value, if any, as a `double`; or `0.0`, if no conversion could be performed. If the correct value is out of the range of representable values, plus or minus `HUGE_VAL` is returned, and `ERANGE` is stored in `errno`. If the correct value would cause underflow, `0.0` is returned and `ERANGE` is stored in `errno`.

`atoff` obeys the same rules as `atof`, except that it returns a `float`.

Portability

`atof` is ANSI C. `atof`, `atoi`, and `atol` are subsumed by `strod` and `strol`, but are used extensively in existing code. These functions are less reliable, but may be faster if the argument is verified to be in a valid range.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

2.8 atoi, atol—string to integer

Synopsis

```
#include <stdlib.h>
int atoi(const char *s);
long atol(const char *s);
int _atoi_r(struct _reent *ptr, const char *s);
long _atol_r(struct _reent *ptr, const char *s);
```

Description

`atoi` converts the initial portion of a string to an `int`. `atol` converts the initial portion of a string to a `long`.

`atoi(s)` is implemented as `(int)strtol(s, NULL, 10)`. `atol(s)` is implemented as `strtol(s, NULL, 10)`.

`_atoi_r` and `_atol_r` are reentrant versions of `atoi` and `atol` respectively, passing the reentrancy struct pointer.

Returns

The functions return the converted value, if any. If no conversion was made, `0` is returned.

Portability

`atoi`, `atol` are ANSI.

No supporting OS subroutines are required.

2.9 `atoll`—convert a string to a long long integer

Synopsis

```
#include <stdlib.h>
long long atoll(const char *str);
long long _atoll_r(struct _reent *ptr, const char *str);
```

Description

The function `atoll` converts the initial portion of the string pointed to by `*str` to a type `long long`. A call to atoll(str) in this implementation is equivalent to strtoll(str, (char **)NULL, 10) including behavior on error.

The alternate function `_atoll_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

The converted value.

Portability

`atoll` is ISO 9899 (C99) and POSIX 1003.1-2001 compatable.

No supporting OS subroutines are required.

2.10 bsearch—binary search

Synopsis

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
    size_t nmemb, size_t size,
    int (*compar)(const void *, const void *));
```

Description

`bsearch` searches an array beginning at *base* for any element that matches *key*, using binary search. *nmemb* is the element count of the array; *size* is the size of each element.

The array must be sorted in ascending order with respect to the comparison function *compar* (which you supply as the last argument of `bsearch`).

You must define the comparison function `(*compar)` to have two arguments; its result must be negative if the first argument is less than the second, zero if the two arguments match, and positive if the first argument is greater than the second (where “less than” and “greater than” refer to whatever arbitrary ordering is appropriate).

Returns

Returns a pointer to an element of *array* that matches *key*. If more than one matching element is available, the result may point to any of them.

Portability

`bsearch` is ANSI.

No supporting OS subroutines are required.

2.11 `calloc`—allocate space for arrays

Synopsis

```
#include <stdlib.h>
void *calloc(size_t n, size_t s);
void *_calloc_r(void *reent, size_t n, size_t s);
```

Description

Use `calloc` to request a block of memory sufficient to hold an array of *n* elements, each of which has size *s*.

The memory allocated by `calloc` comes out of the same memory pool used by `malloc`, but the memory block is initialized to all zero bytes. (To avoid the overhead of initializing the space, use `malloc` instead.)

The alternate function `_calloc_r` is reentrant. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

If successful, a pointer to the newly allocated space.

If unsuccessful, `NULL`.

Portability

`calloc` is ANSI.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

2.12 div—divide two integers

Synopsis

```
#include <stdlib.h>
div_t div(int n, int d);
```

Description

Divide n/d, returning quotient and remainder as two integers in a structure `div_t`.

Returns

The result is represented with the structure

```
typedef struct
{
 int quot;
 int rem;
} div_t;
```

where the `quot` field represents the quotient, and `rem` the remainder. For nonzero d, if ‘`r = div(n,d);`’ then n equals ‘`r.rem + d*r.quot`’.

To divide `long` rather than `int` values, use the similar function `ldiv`.

Portability

`div` is ANSI.

No supporting OS subroutines are required.

2.13 ecvt, ecvtf, fcvt, fcvtf—double or float to string

Synopsis

```
#include <stdlib.h>

char *ecvt(double val, int chars, int *decpt, int *sgn);
char *ecvtf(float val, int chars, int *decpt, int *sgn);

char *fcvt(double val, int decimals,
    int *decpt, int *sgn);
char *fcvtf(float val, int decimals,
    int *decpt, int *sgn);
```

Description

`ecvt` and `fcvt` produce (null-terminated) strings of digits representating the `double` number *val*. `ecvtf` and `fcvtf` produce the corresponding character representations of `float` numbers.

(The `stdlib` functions `ecvtbuf` and `fcvtbuf` are reentrant versions of `ecvt` and `fcvt`.)

The only difference between `ecvt` and `fcvt` is the interpretation of the second argument (*chars* or *decimals*). For `ecvt`, the second argument *chars* specifies the total number of characters to write (which is also the number of significant digits in the formatted string, since these two functions write only digits). For `fcvt`, the second argument *decimals* specifies the number of characters to write after the decimal point; all digits for the integer part of *val* are always included.

Since `ecvt` and `fcvt` write only digits in the output string, they record the location of the decimal point in `*decpt`, and the sign of the number in `*sgn`. After formatting a number, `*decpt` contains the number of digits to the left of the decimal point. `*sgn` contains `0` if the number is positive, and `1` if it is negative.

Returns

All four functions return a pointer to the new string containing a character representation of *val*.

Portability

None of these functions are ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

2.14 gcvt, gcvtf—format double or float as string

Synopsis

```
#include <stdlib.h>

char *gcvt(double val, int precision, char *buf);
char *gcvtf(float val, int precision, char *buf);
```

Description

`gcvt` writes a fully formatted number as a null-terminated string in the buffer `*buf`. `gcvtf` produces corresponding character representations of `float` numbers.

`gcvt` uses the same rules as the `printf` format ‘`%.precisiong`’—only negative values are signed (with ‘`-`’), and either exponential or ordinary decimal-fraction format is chosen depending on the number of significant digits (specified by *precision*).

Returns

The result is a pointer to the formatted representation of *val* (the same as the argument *buf*).

Portability

Neither function is ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

2.15 ecvtbuf, fcvtbuf—double or float to string

Synopsis

```
#include <stdio.h>

char *ecvtbuf(double val, int chars, int *decpt,
    int *sgn, char *buf);

char *fcvtbuf(double val, int decimals, int *decpt,
    int *sgn, char *buf);
```

Description

`ecvtbuf` and `fcvtbuf` produce (null-terminated) strings of digits representating the `double` number *val*.

The only difference between `ecvtbuf` and `fcvtbuf` is the interpretation of the second argument (*chars* or *decimals*). For `ecvtbuf`, the second argument *chars* specifies the total number of characters to write (which is also the number of significant digits in the formatted string, since these two functions write only digits). For `fcvtbuf`, the second argument *decimals* specifies the number of characters to write after the decimal point; all digits for the integer part of *val* are always included.

Since `ecvtbuf` and `fcvtbuf` write only digits in the output string, they record the location of the decimal point in `*decpt`, and the sign of the number in `*sgn`. After formatting a number, `*decpt` contains the number of digits to the left of the decimal point. `*sgn` contains `0` if the number is positive, and `1` if it is negative. For both functions, you supply a pointer *buf* to an area of memory to hold the converted string.

Returns

Both functions return a pointer to *buf*, the string containing a character representation of *val*.

Portability

Neither function is ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

2.16 __env_lock, __env_unlock—lock environ variable

Synopsis

```
#include <envlock.h>
void __env_lock (struct _reent *reent);
void __env_unlock (struct _reent *reent);
```

Description

The setenv family of routines call these functions when they need to modify the environ variable. The version of these routines supplied in the library use the lock API defined in sys/lock.h. If multiple threads of execution can call setenv, or if setenv can be called reentrantly, then you need to define your own versions of these functions in order to safely lock the memory pool during a call. If you do not, the memory pool may become corrupted.

A call to setenv may call __env_lock recursively; that is, the sequence of calls may go __env_lock, __env_lock, __env_unlock, __env_unlock. Any implementation of these routines must be careful to avoid causing a thread to wait for a lock that it already holds.

2.17 `exit`—end program execution

Synopsis

```
#include <stdlib.h>
void exit(int code);
```

Description

Use `exit` to return control from a program to the host operating environment. Use the argument *code* to pass an exit status to the operating environment: two particular values, `EXIT_SUCCESS` and `EXIT_FAILURE`, are defined in '`stdlib.h`' to indicate success or failure in a portable fashion.

`exit` does two kinds of cleanup before ending execution of your program. First, it calls all application-defined cleanup functions you have enrolled with `atexit`. Second, files and streams are cleaned up: any pending output is delivered to the host system, each open file or stream is closed, and files created by `tmpfile` are deleted.

Returns

`exit` does not return to its caller.

Portability

ANSI C requires `exit`, and specifies that `EXIT_SUCCESS` and `EXIT_FAILURE` must be defined.

Supporting OS subroutines required: `_exit`.

2.18 getenv—look up environment variable

Synopsis

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

Description

`getenv` searches the list of environment variable names and values (using the global pointer “`char **environ`”) for a variable whose name matches the string at *name*. If a variable name matches, `getenv` returns a pointer to the associated value.

Returns

A pointer to the (string) value of the environment variable, or `NULL` if there is no such environment variable.

Portability

`getenv` is ANSI, but the rules for properly forming names of environment variables vary from one system to another.

`getenv` requires a global pointer `environ`.

2.19 `itoa`—integer to string

Synopsis

```
#include <stdlib.h>
char *itoa(int value, char *str, int base);
char *__itoa(int value, char *str, int base);
```

Description

`itoa` converts the integer *value* to a null-terminated string using the specified base, which must be between 2 and 36, inclusive. If *base* is 10, *value* is treated as signed and the string will be prefixed with '-' if negative. For all other bases, *value* is treated as unsigned. *str* should be an array long enough to contain the converted value, which in the worst case is sizeof(int)*8+1 bytes.

Returns

A pointer to the string, *str*, or NULL if *base* is invalid.

Portability

`itoa` is non-ANSI.

No supporting OS subroutine calls are required.

2.20 labs—long integer absolute value

Synopsis

```
#include <stdlib.h>
long labs(long i);
```

Description

`labs` returns $|x|$, the absolute value of *i* (also called the magnitude of *i*). That is, if *i* is negative, the result is the opposite of *i*, but if *i* is nonnegative the result is *i*.

The similar function `abs` uses and returns `int` rather than `long` values.

Returns

The result is a nonnegative long integer.

Portability

`labs` is ANSI.

No supporting OS subroutine calls are required.

2.21 `ldiv`—divide two long integers

Synopsis

```
#include <stdlib.h>
ldiv_t ldiv(long n, long d);
```

Description

Divide n/d, returning quotient and remainder as two long integers in a structure `ldiv_t`.

Returns

The result is represented with the structure

```
typedef struct
{
 long quot;
 long rem;
} ldiv_t;
```

where the `quot` field represents the quotient, and `rem` the remainder. For nonzero d, if '`r = ldiv(n,d);`' then n equals '`r.rem + d*r.quot`'.

To divide `int` rather than `long` values, use the similar function `div`.

Portability

`ldiv` is ANSI.

No supporting OS subroutines are required.

2.22 llabs—compute the absolute value of an long long integer.

Synopsis

```
#include <stdlib.h>
long long llabs(long long j);
```

Description

The `llabs` function computes the absolute value of the long long integer argument *j* (also called the magnitude of *j*).

The similar function `labs` uses and returns `long` rather than `long long` values.

Returns

A nonnegative long long integer.

Portability

`llabs` is ISO 9899 (C99) compatable.

No supporting OS subroutines are required.

2.23 `lldiv`—divide two long long integers

Synopsis

```
#include <stdlib.h>
lldiv_t lldiv(long long n, long long d);
```

Description

Divide n/d, returning quotient and remainder as two long long integers in a structure `lldiv_t`.

Returns

The result is represented with the structure

```
typedef struct
{
 long long quot;
 long long rem;
} lldiv_t;
```

where the `quot` field represents the quotient, and `rem` the remainder. For nonzero d, if '`r = lldiv(n,d);`' then n equals '`r.rem + d*r.quot`'.

To divide `long` rather than `long long` values, use the similar function `ldiv`.

Portability

`lldiv` is ISO 9899 (C99) compatable.

No supporting OS subroutines are required.

2.24 malloc, realloc, free—manage memory

Synopsis

```
#include <stdlib.h>
void *malloc(size_t nbytes);
void *realloc(void *aptr, size_t nbytes);
void *reallocf(void *aptr, size_t nbytes);
void free(void *aptr);

void *memalign(size_t align, size_t nbytes);

size_t malloc_usable_size(void *aptr);

void *_malloc_r(void *reent, size_t nbytes);
void *_realloc_r(void *reent,
    void *aptr, size_t nbytes);
void *_reallocf_r(void *reent,
    void *aptr, size_t nbytes);
void _free_r(void *reent, void *aptr);

void *_memalign_r(void *reent,
    size_t align, size_t nbytes);

size_t _malloc_usable_size_r(void *reent, void *aptr);
```

Description

These functions manage a pool of system memory.

Use `malloc` to request allocation of an object with at least *nbytes* bytes of storage available. If the space is available, `malloc` returns a pointer to a newly allocated block as its result.

If you already have a block of storage allocated by `malloc`, but you no longer need all the space allocated to it, you can make it smaller by calling `realloc` with both the object pointer and the new desired size as arguments. `realloc` guarantees that the contents of the smaller object match the beginning of the original object.

Similarly, if you need more space for an object, use `realloc` to request the larger size; again, `realloc` guarantees that the beginning of the new, larger object matches the contents of the original object.

When you no longer need an object originally allocated by `malloc` or `realloc` (or the related function `calloc`), return it to the memory storage pool by calling `free` with the address of the object as the argument. You can also use `realloc` for this purpose by calling it with `0` as the *nbytes* argument.

The `reallocf` function behaves just like `realloc` except if the function is required to allocate new storage and this fails. In this case `reallocf` will free the original object passed in whereas `realloc` will not.

The `memalign` function returns a block of size *nbytes* aligned to a *align* boundary. The *align* argument must be a power of two.

The `malloc_usable_size` function takes a pointer to a block allocated by `malloc`. It returns the amount of space that is available in the block. This may or may not be more than the size requested from `malloc`, due to alignment or minimum size constraints.

The alternate functions `_malloc_r`, `_realloc_r`, `_reallocf_r`, `_free_r`, `_memalign_r`, and `_malloc_usable_size_r` are reentrant versions. The extra argument *reent* is a pointer to a reentrancy structure.

If you have multiple threads of execution which may call any of these routines, or if any of these routines may be called reentrantly, then you must provide implementations of the `__malloc_lock` and `__malloc_unlock` functions for your system. See the documentation for those functions.

These functions operate by calling the function `_sbrk_r` or `sbrk`, which allocates space. You may need to provide one of these functions for your system. `_sbrk_r` is called with a positive value to allocate more space, and with a negative value to release previously allocated space if it is no longer required. See Section 13.1 [Stubs], page 319.

Returns

`malloc` returns a pointer to the newly allocated space, if successful; otherwise it returns `NULL`. If your application needs to generate empty objects, you may use `malloc(0)` for this purpose.

`realloc` returns a pointer to the new block of memory, or `NULL` if a new block could not be allocated. `NULL` is also the result when you use '`realloc(`*`aptr`*`,0)`' (which has the same effect as '`free(`*`aptr`*`)`'). You should always check the result of `realloc`; successful reallocation is not guaranteed even when you request a smaller object.

`free` does not return a result.

`memalign` returns a pointer to the newly allocated space.

`malloc_usable_size` returns the usable size.

Portability

`malloc`, `realloc`, and `free` are specified by the ANSI C standard, but other conforming implementations of `malloc` may behave differently when *nbytes* is zero.

`memalign` is part of SVR4.

`malloc_usable_size` is not portable.

Supporting OS subroutines required: `sbrk`.

2.25 mallinfo, malloc_stats, mallopt—malloc support

Synopsis

```
#include <malloc.h>
struct mallinfo mallinfo(void);
void malloc_stats(void);
int mallopt(int parameter, value);

struct mallinfo _mallinfo_r(void *reent);
void _malloc_stats_r(void *reent);
int _mallopt_r(void *reent, int parameter, value);
```

Description

`mallinfo` returns a structure describing the current state of memory allocation. The structure is defined in malloc.h. The following fields are defined: `arena` is the total amount of space in the heap; `ordblks` is the number of chunks which are not in use; `uordblks` is the total amount of space allocated by `malloc`; `fordblks` is the total amount of space not in use; `keepcost` is the size of the top most memory block.

`malloc_stats` print some statistics about memory allocation on standard error.

`mallopt` takes a parameter and a value. The parameters are defined in malloc.h, and may be one of the following: `M_TRIM_THRESHOLD` sets the maximum amount of unused space in the top most block before releasing it back to the system in `free` (the space is released by calling `_sbrk_r` with a negative argument); `M_TOP_PAD` is the amount of padding to allocate whenever `_sbrk_r` is called to allocate more space.

The alternate functions `_mallinfo_r`, `_malloc_stats_r`, and `_mallopt_r` are reentrant versions. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`mallinfo` returns a mallinfo structure. The structure is defined in malloc.h.

`malloc_stats` does not return a result.

`mallopt` returns zero if the parameter could not be set, or non-zero if it could be set.

Portability

`mallinfo` and `mallopt` are provided by SVR4, but `mallopt` takes different parameters on different systems. `malloc_stats` is not portable.

2.26 `__malloc_lock`, `__malloc_unlock`—lock malloc pool

Synopsis

```
#include <malloc.h>
void __malloc_lock (struct _reent *reent);
void __malloc_unlock (struct _reent *reent);
```

Description

The `malloc` family of routines call these functions when they need to lock the memory pool. The version of these routines supplied in the library use the lock API defined in sys/lock.h. If multiple threads of execution can call `malloc`, or if `malloc` can be called reentrantly, then you need to define your own versions of these functions in order to safely lock the memory pool during a call. If you do not, the memory pool may become corrupted.

A call to `malloc` may call `__malloc_lock` recursively; that is, the sequence of calls may go `__malloc_lock`, `__malloc_lock`, `__malloc_unlock`, `__malloc_unlock`. Any implementation of these routines must be careful to avoid causing a thread to wait for a lock that it already holds.

2.27 mblen—minimal multibyte length function

Synopsis

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Description

When _MB_CAPABLE is not defined, this is a minimal ANSI-conforming implementation of `mblen`. In this case, the only “multi-byte character sequences” recognized are single bytes, and thus `1` is returned unless *s* is the null pointer or has a length of 0 or is the empty string.

When _MB_CAPABLE is defined, this routine calls `_mbtowc_r` to perform the conversion, passing a state variable to allow state dependent decoding. The result is based on the locale setting which may be restricted to a defined set of locales.

Returns

This implementation of `mblen` returns `0` if *s* is `NULL` or the empty string; it returns `1` if not _MB_CAPABLE or the character is a single-byte character; it returns `-1` if the multi-byte character is invalid; otherwise it returns the number of bytes in the multibyte character.

Portability

`mblen` is required in the ANSI C standard. However, the precise effects vary with the locale.

`mblen` requires no supporting OS subroutines.

2.28 mbsrtowcs, mbsnrtowcs—convert a character string to a wide-character string

Synopsis

```
#include <wchar.h>
size_t mbsrtowcs(wchar_t *__restrict dst,
    const char **__restrict src,
    size_t len,
    mbstate_t *__restrict ps);

#include <wchar.h>
size_t _mbsrtowcs_r(struct _reent *ptr, wchar_t *dst,
    const char **src, size_t len,
    mbstate_t *ps);

#include <wchar.h>
size_t mbsnrtowcs(wchar_t *__ restrict dst,
    const char **__restrict src, size_t nms,
    size_t len, mbstate_t *__restrict ps);

#include <wchar.h>
size_t _mbsnrtowcs_r(struct _reent *ptr, wchar_t *dst,
    const char **src, size_t nms,
    size_t len, mbstate_t *ps);
```

Description

The `mbsrtowcs` function converts a sequence of multibyte characters pointed to indirectly by *src* into a sequence of corresponding wide characters and stores at most *len* of them in the wchar_t array pointed to by *dst*, until it encounters a terminating null character ('\0').

If *dst* is NULL, no characters are stored.

If *dst* is not NULL, the pointer pointed to by *src* is updated to point to the character after the one that conversion stopped at. If conversion stops because a null character is encountered, **src* is set to NULL.

The mbstate_t argument, *ps*, is used to keep track of the shift state. If it is NULL, `mbsrtowcs` uses an internal, static mbstate_t object, which is initialized to the initial conversion state at program startup.

The `mbsnrtowcs` function behaves identically to `mbsrtowcs`, except that conversion stops after reading at most *nms* bytes from the buffer pointed to by *src*.

Returns

The `mbsrtowcs` and `mbsnrtowcs` functions return the number of wide characters stored in the array pointed to by *dst* if successful, otherwise it returns (size_t)-1.

Portability

`mbsrtowcs` is defined by the C99 standard. `mbsnrtowcs` is defined by the POSIX.1-2008 standard.

2.29 mbstowcs—minimal multibyte string to wide char converter

Synopsis

```
#include <stdlib.h>
int mbstowcs(wchar_t *restrict pwc, const char *restrict s, size_t n);
```

Description

When _MB_CAPABLE is not defined, this is a minimal ANSI-conforming implementation of `mbstowcs`. In this case, the only “multi-byte character sequences” recognized are single bytes, and they are “converted” to wide-char versions simply by byte extension.

When _MB_CAPABLE is defined, this routine calls `_mbstowcs_r` to perform the conversion, passing a state variable to allow state dependent decoding. The result is based on the locale setting which may be restricted to a defined set of locales.

Returns

This implementation of `mbstowcs` returns `0` if *s* is `NULL` or is the empty string; it returns `-1` if _MB_CAPABLE and one of the multi-byte characters is invalid or incomplete; otherwise it returns the minimum of: `n` or the number of multi-byte characters in `s` plus 1 (to compensate for the nul character). If the return value is -1, the state of the `pwc` string is indeterminate. If the input has a length of 0, the output string will be modified to contain a wchar_t nul terminator.

Portability

`mbstowcs` is required in the ANSI C standard. However, the precise effects vary with the locale.

`mbstowcs` requires no supporting OS subroutines.

2.30 mbtowc—minimal multibyte to wide char converter

Synopsis

```
#include <stdlib.h>
int mbtowc(wchar_t *restrict pwc, const char *restrict s, size_t n);
```

Description

When _MB_CAPABLE is not defined, this is a minimal ANSI-conforming implementation of `mbtowc`. In this case, only "multi-byte character sequences" recognized are single bytes, and they are "converted" to themselves. Each call to `mbtowc` copies one character from `*s` to `*pwc`, unless *s* is a null pointer. The argument n is ignored.

When _MB_CAPABLE is defined, this routine calls `_mbtowc_r` to perform the conversion, passing a state variable to allow state dependent decoding. The result is based on the locale setting which may be restricted to a defined set of locales.

Returns

This implementation of `mbtowc` returns `0` if *s* is `NULL` or is the empty string; it returns `1` if not _MB_CAPABLE or the character is a single-byte character; it returns `-1` if n is `0` or the multi-byte character is invalid; otherwise it returns the number of bytes in the multibyte character. If the return value is -1, no changes are made to the `pwc` output string. If the input is the empty string, a wchar_t nul is placed in the output string and 0 is returned. If the input has a length of 0, no changes are made to the `pwc` output string.

Portability

`mbtowc` is required in the ANSI C standard. However, the precise effects vary with the locale.

`mbtowc` requires no supporting OS subroutines.

2.31 on_exit—request execution of function with argument at program exit

Synopsis

```
#include <stdlib.h>
int on_exit (void (*function)(int, void *), void *arg);
```

Description

You can use on_exit to enroll functions in a list of functions that will be called when your program terminates normally. The argument is a pointer to a user-defined function which takes two arguments. The first is the status code passed to exit and the second argument is of type pointer to void. The function must not return a result. The value of *arg* is registered and passed as the argument to *function*.

The functions are kept in a LIFO stack; that is, the last function enrolled by atexit or on_exit will be the first to execute when your program exits. You can intermix functions using atexit and on_exit.

There is no built-in limit to the number of functions you can enroll in this list; however, after every group of 32 functions is enrolled, atexit/on_exit will call malloc to get space for the next part of the list. The initial list of 32 functions is statically allocated, so you can always count on at least that many slots available.

Returns

on_exit returns 0 if it succeeds in enrolling your function, -1 if it fails (possible only if no space was available for malloc to extend the list of functions).

Portability

on_exit is a non-standard glibc extension

Supporting OS subroutines required: None

2.32 qsort—sort an array

Synopsis

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *) );
```

Description

`qsort` sorts an array (beginning at *base*) of *nmemb* objects. *size* describes the size of each element of the array.

You must supply a pointer to a comparison function, using the argument shown as *compar*. (This permits sorting objects of unknown properties.) Define the comparison function to accept two arguments, each a pointer to an element of the array starting at *base*. The result of `(*compar)` must be negative if the first argument is less than the second, zero if the two arguments match, and positive if the first argument is greater than the second (where "less than" and "greater than" refer to whatever arbitrary ordering is appropriate).

The array is sorted in place; that is, when `qsort` returns, the array elements beginning at *base* have been reordered.

Returns

`qsort` does not return a result.

Portability

`qsort` is required by ANSI (without specifying the sorting algorithm).

2.33 rand, srand—pseudo-random numbers

Synopsis

```
#include <stdlib.h>
int rand(void);
void srand(unsigned int seed);
int rand_r(unsigned int *seed);
```

Description

`rand` returns a different integer each time it is called; each integer is chosen by an algorithm designed to be unpredictable, so that you can use `rand` when you require a random number. The algorithm depends on a static variable called the “random seed”; starting with a given value of the random seed always produces the same sequence of numbers in successive calls to `rand`.

You can set the random seed using `srand`; it does nothing beyond storing its argument in the static variable used by `rand`. You can exploit this to make the pseudo-random sequence less predictable, if you wish, by using some other unpredictable value (often the least significant parts of a time-varying value) as the random seed before beginning a sequence of calls to `rand`; or, if you wish to ensure (for example, while debugging) that successive runs of your program use the same “random” numbers, you can use `srand` to set the same random seed at the outset.

Returns

`rand` returns the next pseudo-random integer in sequence; it is a number between `0` and `RAND_MAX` (inclusive).

`srand` does not return a result.

Notes

`rand` and `srand` are unsafe for multi-threaded applications. `rand_r` is thread-safe and should be used instead.

Portability

`rand` is required by ANSI, but the algorithm for pseudo-random number generation is not specified; therefore, even if you use the same random seed, you cannot expect the same sequence of results on two different systems.

`rand` requires no supporting OS subroutines.

2.34 `random`, `srandom`—pseudo-random numbers

Synopsis

```
#define _XOPEN_SOURCE 500
#include <stdlib.h>
long int random(void);
void srandom(unsigned int seed);
```

Description

`random` returns a different integer each time it is called; each integer is chosen by an algorithm designed to be unpredictable, so that you can use `random` when you require a random number. The algorithm depends on a static variable called the "random seed"; starting with a given value of the random seed always produces the same sequence of numbers in successive calls to `random`.

You can set the random seed using `srandom`; it does nothing beyond storing its argument in the static variable used by `rand`. You can exploit this to make the pseudo-random sequence less predictable, if you wish, by using some other unpredictable value (often the least significant parts of a time-varying value) as the random seed before beginning a sequence of calls to `rand`; or, if you wish to ensure (for example, while debugging) that successive runs of your program use the same "random" numbers, you can use `srandom` to set the same random seed at the outset.

Returns

`random` returns the next pseudo-random integer in sequence; it is a number between `0` and `RAND_MAX` (inclusive).

`srandom` does not return a result.

Notes

`random` and `srandom` are unsafe for multi-threaded applications.

_XOPEN_SOURCE may be any value >= 500.

Portability

`random` is required by XSI. This implementation uses the same algorithm as `rand`.

`random` requires no supporting OS subroutines.

2.35 rand48, drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48—pseudo-random number generators and initialization routines

Synopsis

```
#include <stdlib.h>
double drand48(void);
double erand48(unsigned short xseed[3]);
long lrand48(void);
long nrand48(unsigned short xseed[3]);
long mrand48(void);
long jrand48(unsigned short xseed[3]);
void srand48(long seed);
unsigned short *seed48(unsigned short xseed[3]);
void lcong48(unsigned short p[7]);
```

Description

The rand48 family of functions generates pseudo-random numbers using a linear congruential algorithm working on integers 48 bits in size. The particular formula employed is r(n+1) = (a * r(n) + c) mod m where the default values are for the multiplicand a = 0xfdeece66d = 25214903917 and the addend c = 0xb = 11. The modulo is always fixed at m = 2 ** 48. r(n) is called the seed of the random number generator.

For all the six generator routines described next, the first computational step is to perform a single iteration of the algorithm.

drand48 and erand48 return values of type double. The full 48 bits of r(n+1) are loaded into the mantissa of the returned value, with the exponent set such that the values produced lie in the interval [0.0, 1.0].

lrand48 and nrand48 return values of type long in the range [0, 2**31-1]. The high-order (31) bits of r(n+1) are loaded into the lower bits of the returned value, with the topmost (sign) bit set to zero.

mrand48 and jrand48 return values of type long in the range [-2**31, 2**31-1]. The high-order (32) bits of r(n+1) are loaded into the returned value.

drand48, lrand48, and mrand48 use an internal buffer to store r(n). For these functions the initial value of r(0) = 0x1234abcd330e = 20017429951246.

On the other hand, erand48, nrand48, and jrand48 use a user-supplied buffer to store the seed r(n), which consists of an array of 3 shorts, where the zeroth member holds the least significant bits.

All functions share the same multiplicand and addend.

srand48 is used to initialize the internal buffer r(n) of drand48, lrand48, and mrand48 such that the 32 bits of the seed value are copied into the upper 32 bits of r(n), with the lower 16 bits of r(n) arbitrarily being set to 0x330e. Additionally, the constant multiplicand and addend of the algorithm are reset to the default values given above.

seed48 also initializes the internal buffer r(n) of drand48, lrand48, and mrand48, but here all 48 bits of the seed can be specified in an array of 3 shorts, where the zeroth member specifies the lowest bits. Again, the constant multiplicand and addend of the algorithm are reset to the default values given above. seed48 returns a pointer to an array of 3 shorts

which contains the old seed. This array is statically allocated, thus its contents are lost after each new call to `seed48`.

Finally, `lcong48` allows full control over the multiplicand and addend used in `drand48`, `erand48`, `lrand48`, `nrand48`, `mrand48`, and `jrand48`, and the seed used in `drand48`, `lrand48`, and `mrand48`. An array of 7 shorts is passed as parameter; the first three shorts are used to initialize the seed; the second three are used to initialize the multiplicand; and the last short is used to initialize the addend. It is thus not possible to use values greater than 0xffff as the addend.

Note that all three methods of seeding the random number generator always also set the multiplicand and addend for any of the six generator calls.

For a more powerful random number generator, see `random`.

Portability
SUS requires these functions.

No supporting OS subroutines are required.

2.36 rpmatch—determine whether response to question is affirmative or negative

Synopsis

```
#include <stdlib.h>
int rpmatch(const char *response);
```

Description

The `rpmatch` function determines whether *response* is an affirmative or negative response to a question according to the current locale.

Returns

`rpmatch` returns 1 if *response* is affirmative, 0 if negative, or -1 if not recognized as either.

Portability

`rpmatch` is a BSD extension also found in glibc.

Notes

No supporting OS subroutines are required.

2.37 strtod, strtof, strtold, strtod_l, strtof_l, strtold_l— string to double or float

Synopsis

```
#include <stdlib.h>
double strtod(const char *restrict str, char **restrict tail);
float strtof(const char *restrict str, char **restrict tail);
long double strtold(const char *restrict str,
    char **restrict tail);

#include <stdlib.h>
double strtod_l(const char *restrict str, char **restrict tail,
    locale_t locale);
float strtof_l(const char *restrict str, char **restrict tail,
    locale_t locale);
long double strtold_l(const char *restrict str,
    char **restrict tail,
    locale_t locale);

double _strtod_r(void *reent,
    const char *restrict str, char **restrict tail);
```

Description

`strtod`, `strtof`, `strtold` parse the character string *str*, producing a substring which can be converted to a double, float, or long double value, respectively. The substring converted is the longest initial subsequence of *str*, beginning with the first non-whitespace character, that has one of these formats:

```
[+|-]digits[.[digits]][(e|E)[+|-]digits]
[+|-].digits[(e|E)[+|-]digits]
[+|-](i|I)(n|N)(f|F)[(i|I)(n|N)(i|I)(t|T)(y|Y)]
[+|-](n|N)(a|A)(n|N)[<(>[hexdigits]<)>]
[+|-]0(x|X)hexdigits[.[hexdigits]][(p|P)[+|-]digits]
[+|-]0(x|X).hexdigits[(p|P)[+|-]digits]
```

The substring contains no characters if *str* is empty, consists entirely of whitespace, or if the first non-whitespace character is something other than `+`, `-`, `.`, or a digit, and cannot be parsed as infinity or NaN. If the platform does not support NaN, then NaN is treated as an empty substring. If the substring is empty, no conversion is done, and the value of *str* is stored in `*tail`. Otherwise, the substring is converted, and a pointer to the final string (which will contain at least the terminating null character of *str*) is stored in `*tail`. If you want no assignment to `*tail`, pass a null pointer as *tail*.

This implementation returns the nearest machine number to the input decimal string. Ties are broken by using the IEEE round-even rule. However, `strtof` is currently subject to double rounding errors.

`strtod_l`, `strtof_l`, `strtold_l` are like `strtod`, `strtof`, `strtold` but perform the conversion based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

The alternate function `_strtod_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns
These functions return the converted substring value, if any. If no conversion could be performed, 0 is returned. If the correct value is out of the range of representable values, plus or minus `HUGE_VAL` (`HUGE_VALF`, `HUGE_VALL`) is returned, and `ERANGE` is stored in errno. If the correct value would cause underflow, 0 is returned and `ERANGE` is stored in errno.

Portability
`strtod` is ANSI. `strtof`, `strtold` are C99. `strtod_l`, `strtof_l`, `strtold_l` are GNU extensions.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

2.38 strtol, strtol_l—string to long

Synopsis

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

#include <stdlib.h>
long strtol_l(const char *restrict s, char **restrict ptr,
    int base, locale_t locale);

long _strtol_r(void *reent, const char *restrict s,
    char **restrict ptr,int base);
```

Description

The function `strtol` converts the string `*s` to a `long`. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of characters resembling an integer in the radix specified by *base*; and a trailing portion consisting of zero or more unparseable characters, and always including the terminating null character. Then, it attempts to convert the subject string into a `long` and returns the result.

If the value of *base* is 0, the subject string is expected to look like a normal C integer constant: an optional sign, a possible '`0x`' indicating a hexadecimal base, and a number. If *base* is between 2 and 36, the expected form of the subject is a sequence of letters and digits representing an integer in the radix specified by *base*, with an optional plus or minus sign. The letters `a`–`z` (or, equivalently, `A`–`Z`) are used to signify values from 10 to 35; only letters whose ascribed values are less than *base* are permitted. If *base* is 16, a leading `0x` is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible letter or digit, the subject string is empty.

If the subject string is acceptable, and the value of *base* is zero, `strtol` attempts to determine the radix from the input string. A string with a leading `0x` is treated as a hexadecimal value; a string with a leading 0 and no `x` is treated as octal; all other strings are treated as decimal. If *base* is between 2 and 36, it is used as the conversion radix, as described above. If the subject string begins with a minus sign, the value is negated. Finally, a pointer to the first character past the converted subject string is stored in *ptr*, if *ptr* is not `NULL`.

If the subject string is empty (or not in acceptable form), no conversion is performed and the value of *s* is stored in *ptr* (if *ptr* is not `NULL`).

`strtol_l` is like `strtol` but performs the conversion based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

The alternate function `_strtol_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns
`strtol`, `strtol_l` return the converted value, if any. If no conversion was made, 0 is returned.

`strtol`, `strtol_l` return `LONG_MAX` or `LONG_MIN` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

Portability
`strtol` is ANSI. `strtol_l` is a GNU extension.

No supporting OS subroutines are required.

2.39 strtoll, strtoll_l—string to long long

Synopsis

```
#include <stdlib.h>
long long strtoll(const char *restrict s, char **restrict ptr,
    int base);

#include <stdlib.h>
long long strtoll_l(const char *restrict s,
    char **restrict ptr, int base,
    locale_t locale);

long long _strtoll_r(void *reent,
    const char *restrict s,
    char **restrict ptr, int base);
```

Description

The function `strtoll` converts the string `*s` to a `long long`. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of characters resembling an integer in the radix specified by *base*; and a trailing portion consisting of zero or more unparseable characters, and always including the terminating null character. Then, it attempts to convert the subject string into a `long long` and returns the result.

If the value of *base* is 0, the subject string is expected to look like a normal C integer constant: an optional sign, a possible '`0x`' indicating a hexadecimal base, and a number. If *base* is between 2 and 36, the expected form of the subject is a sequence of letters and digits representing an integer in the radix specified by *base*, with an optional plus or minus sign. The letters `a`–`z` (or, equivalently, `A`–`Z`) are used to signify values from 10 to 35; only letters whose ascribed values are less than *base* are permitted. If *base* is 16, a leading `0x` is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible letter or digit, the subject string is empty.

If the subject string is acceptable, and the value of *base* is zero, `strtoll` attempts to determine the radix from the input string. A string with a leading `0x` is treated as a hexadecimal value; a string with a leading 0 and no `x` is treated as octal; all other strings are treated as decimal. If *base* is between 2 and 36, it is used as the conversion radix, as described above. If the subject string begins with a minus sign, the value is negated. Finally, a pointer to the first character past the converted subject string is stored in *ptr*, if *ptr* is not `NULL`.

If the subject string is empty (or not in acceptable form), no conversion is performed and the value of *s* is stored in *ptr* (if *ptr* is not `NULL`).

`strtoll_l` is like `strtoll` but performs the conversion based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

The alternate function `_strtoll_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns
`strtoll`, `strtoll_l` return the converted value, if any. If no conversion was made, 0 is returned.

`strtoll`, `strtoll_l` return `LONG_LONG_MAX` or `LONG_LONG_MIN` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

Portability
`strtoll` is ANSI. `strtoll_l` is a GNU extension.

No supporting OS subroutines are required.

2.40 strtoul, strtoul_l—string to unsigned long

Synopsis

```
#include <stdlib.h>
unsigned long strtoul(const char *restrict s,
    char **restrict ptr, int base);

#include <stdlib.h>
unsigned long strtoul_l(const char *restrict s,
    char **restrict ptr, int base,
    locale_t locale);

unsigned long _strtoul_r(void *reent, const char *restrict s,
    char **restrict ptr, int base);
```

Description

The function `strtoul` converts the string `*s` to an `unsigned long`. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of the digits meaningful in the radix specified by *base* (for example, `0` through `7` if the value of *base* is 8); and a trailing portion consisting of one or more unparseable characters, which always includes the terminating null character. Then, it attempts to convert the subject string into an unsigned long integer, and returns the result.

If the value of *base* is zero, the subject string is expected to look like a normal C integer constant (save that no optional sign is permitted): a possible `0x` indicating hexadecimal radix, and a number. If *base* is between 2 and 36, the expected form of the subject is a sequence of digits (which may include letters, depending on the base) representing an integer in the radix specified by *base*. The letters `a`–`z` (or `A`–`Z`) are used as digits valued from 10 to 35. If *base* is 16, a leading `0x` is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible digit, the subject string is empty.

If the subject string is acceptable, and the value of *base* is zero, `strtoul` attempts to determine the radix from the input string. A string with a leading `0x` is treated as a hexadecimal value; a string with a leading `0` and no `x` is treated as octal; all other strings are treated as decimal. If *base* is between 2 and 36, it is used as the conversion radix, as described above. Finally, a pointer to the first character past the converted subject string is stored in *ptr*, if *ptr* is not `NULL`.

If the subject string is empty (that is, if `*s` does not start with a substring in acceptable form), no conversion is performed and the value of *s* is stored in *ptr* (if *ptr* is not `NULL`).

`strtoul_l` is like `strtoul` but performs the conversion based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

The alternate function `_strtoul_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns
`strtoul`, `strtoul_l` return the converted value, if any. If no conversion was made, `0` is returned.

`strtoul`, `strtoul_l` return `ULONG_MAX` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

Portability
`strtoul` is ANSI. `strtoul_l` is a GNU extension.

`strtoul` requires no supporting OS subroutines.

2.41 strtoull, strtoull_l—string to unsigned long long

Synopsis

```
#include <stdlib.h>
unsigned long long strtoull(const char *restrict s,
    char **restrict ptr, int base);

#include <stdlib.h>
unsigned long long strtoull_l(const char *restrict s,
    char **restrict ptr, int base,
    locale_t locale);

unsigned long long _strtoull_r(void *reent,
    const char *restrict s,
    char **restrict ptr, int base);
```

Description

The function `strtoull` converts the string `*s` to an `unsigned long long`. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of the digits meaningful in the radix specified by *base* (for example, `0` through `7` if the value of *base* is 8); and a trailing portion consisting of one or more unparseable characters, which always includes the terminating null character. Then, it attempts to convert the subject string into an unsigned long long integer, and returns the result.

If the value of *base* is zero, the subject string is expected to look like a normal C integer constant (save that no optional sign is permitted): a possible `0x` indicating hexadecimal radix, and a number. If *base* is between 2 and 36, the expected form of the subject is a sequence of digits (which may include letters, depending on the base) representing an integer in the radix specified by *base*. The letters `a`–`z` (or `A`–`Z`) are used as digits valued from 10 to 35. If *base* is 16, a leading `0x` is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible digit, the subject string is empty.

If the subject string is acceptable, and the value of *base* is zero, `strtoull` attempts to determine the radix from the input string. A string with a leading `0x` is treated as a hexadecimal value; a string with a leading `0` and no `x` is treated as octal; all other strings are treated as decimal. If *base* is between 2 and 36, it is used as the conversion radix, as described above. Finally, a pointer to the first character past the converted subject string is stored in *ptr*, if *ptr* is not `NULL`.

If the subject string is empty (that is, if `*s` does not start with a substring in acceptable form), no conversion is performed and the value of *s* is stored in *ptr* (if *ptr* is not `NULL`).

`strtoull_l` is like `strtoull` but performs the conversion based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

The alternate function `_strtoull_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns
`strtoull`, `strtoull_l` return the converted value, if any. If no conversion was made, `0` is returned.

`strtoull`, `strtoull_l` return `ULONG_LONG_MAX` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

Portability
`strtoull` is ANSI. `strtoull_l` is a GNU extension.

`strtoull` requires no supporting OS subroutines.

2.42 wcsrtombs, wcsnrtombs—convert a wide-character string to a character string

Synopsis

```
#include <wchar.h>
size_t wcsrtombs(char *__restrict dst,
    const wchar_t **__restrict src, size_t len,
    mbstate_t *__restrict ps);

#include <wchar.h>
size_t _wcsrtombs_r(struct _reent *ptr, char *dst,
    const wchar_t **src, size_t len,
    mbstate_t *ps);

#include <wchar.h>
size_t wcsnrtombs(char *__restrict dst,
    const wchar_t **__restrict src,
    size_t nwc, size_t len,
    mbstate_t *__restrict ps);

#include <wchar.h>
size_t _wcsnrtombs_r(struct _reent *ptr, char *dst,
    const wchar_t **src, size_t nwc,
    size_t len, mbstate_t *ps);
```

Description

The `wcsrtombs` function converts a string of wide characters indirectly pointed to by *src* to a corresponding multibyte character string stored in the array pointed to by *dst*. No more than *len* bytes are written to *dst*.

If *dst* is NULL, no characters are stored.

If *dst* is not NULL, the pointer pointed to by *src* is updated to point to the character after the one that conversion stopped at. If conversion stops because a null character is encountered, **src* is set to NULL.

The mbstate_t argument, *ps*, is used to keep track of the shift state. If it is NULL, `wcsrtombs` uses an internal, static mbstate_t object, which is initialized to the initial conversion state at program startup.

The `wcsnrtombs` function behaves identically to `wcsrtombs`, except that conversion stops after reading at most *nwc* characters from the buffer pointed to by *src*.

Returns

The `wcsrtombs` and `wcsnrtombs` functions return the number of bytes stored in the array pointed to by *dst* (not including any terminating null), if successful, otherwise it returns (size_t)-1.

Portability

`wcsrtombs` is defined by C99 standard. `wcsnrtombs` is defined by the POSIX.1-2008 standard.

2.43 wcstod, wcstof, wcstold, wcstod_l, wcstof_l, wcstold_l— wide char string to double or float

Synopsis

```
#include <stdlib.h>
double wcstod(const wchar_t *__restrict str,
    wchar_t **__restrict tail);
float wcstof(const wchar_t *__restrict str,
    wchar_t **__restrict tail);
long double wcstold(const wchar_t *__restrict str,
    wchar_t **__restrict tail);

#include <stdlib.h>
double wcstod_l(const wchar_t *__restrict str,
    wchar_t **__restrict tail, locale_t locale);
float wcstof_l(const wchar_t *__restrict str,
    wchar_t **__restrict tail, locale_t locale);
long double wcstold_l(const wchar_t *__restrict str,
    wchar_t **__restrict tail,
    locale_t locale);

double _wcstod_r(void *reent,
    const wchar_t *str, wchar_t **tail);
float _wcstof_r(void *reent,
    const wchar_t *str, wchar_t **tail);
```

Description

`wcstod`, `wcstof`, `wcstold` parse the wide-character string *str*, producing a substring which can be converted to a double, float, or long double value. The substring converted is the longest initial subsequence of *str*, beginning with the first non-whitespace character, that has one of these formats:

```
[+|-]digits[.[digits]][(e|E)[+|-]digits]
[+|-].digits[(e|E)[+|-]digits]
[+|-](i|I)(n|N)(f|F)[(i|I)(n|N)(i|I)(t|T)(y|Y)]
[+|-](n|N)(a|A)(n|N)[<(>[hexdigits]<)>]
[+|-]0(x|X)hexdigits[.[hexdigits]][(p|P)[+|-]digits]
[+|-]0(x|X).hexdigits[(p|P)[+|-]digits]
```

The substring contains no characters if *str* is empty, consists entirely of whitespace, or if the first non-whitespace character is something other than `+`, `-`, `.`, or a digit, and cannot be parsed as infinity or NaN. If the platform does not support NaN, then NaN is treated as an empty substring. If the substring is empty, no conversion is done, and the value of *str* is stored in `*tail`. Otherwise, the substring is converted, and a pointer to the final string (which will contain at least the terminating null character of *str*) is stored in `*tail`. If you want no assignment to `*tail`, pass a null pointer as *tail*.

This implementation returns the nearest machine number to the input decimal string. Ties are broken by using the IEEE round-even rule. However, `wcstof` is currently subject to double rounding errors.

`wcstod_l`, `wcstof_l`, `wcstold_l` are like `wcstod`, `wcstof`, `wcstold` but perform the conversion based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

The alternate functions `_wcstod_r` and `_wcstof_r` are reentrant versions of `wcstod` and `wcstof`, respectively. The extra argument *reent* is a pointer to a reentrancy structure.

Returns
Return the converted substring value, if any. If no conversion could be performed, 0 is returned. If the correct value is out of the range of representable values, plus or minus `HUGE_VAL` is returned, and `ERANGE` is stored in errno. If the correct value would cause underflow, 0 is returned and `ERANGE` is stored in errno.

Portability
`wcstod` is ANSI. `wcstof`, `wcstold` are C99. `wcstod_l`, `wcstof_l`, `wcstold_l` are GNU extensions.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

2.44 wcstol, wcstol_l—wide string to long

Synopsis

```
#include <wchar.h>
long wcstol(const wchar_t *__restrict s,
    wchar_t **__restrict ptr, int base);

#include <wchar.h>
long wcstol_l(const wchar_t *__restrict s,
    wchar_t **__restrict ptr, int base,
    locale_t locale);

long _wcstol_r(void *reent, const wchar_t *s,
    wchar_t **ptr, int base);
```

Description

The function `wcstol` converts the wide string `*s` to a `long`. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of characters resembling an integer in the radix specified by *base*; and a trailing portion consisting of zero or more unparseable characters, and always including the terminating null character. Then, it attempts to convert the subject string into a `long` and returns the result.

If the value of *base* is 0, the subject string is expected to look like a normal C integer constant: an optional sign, a possible ‘`0x`’ indicating a hexadecimal base, and a number. If *base* is between 2 and 36, the expected form of the subject is a sequence of letters and digits representing an integer in the radix specified by *base*, with an optional plus or minus sign. The letters `a`–`z` (or, equivalently, `A`–`Z`) are used to signify values from 10 to 35; only letters whose ascribed values are less than *base* are permitted. If *base* is 16, a leading `0x` is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible letter or digit, the subject string is empty.

If the subject string is acceptable, and the value of *base* is zero, `wcstol` attempts to determine the radix from the input string. A string with a leading `0x` is treated as a hexadecimal value; a string with a leading 0 and no `x` is treated as octal; all other strings are treated as decimal. If *base* is between 2 and 36, it is used as the conversion radix, as described above. If the subject string begins with a minus sign, the value is negated. Finally, a pointer to the first character past the converted subject string is stored in *ptr*, if *ptr* is not `NULL`.

If the subject string is empty (or not in acceptable form), no conversion is performed and the value of *s* is stored in *ptr* (if *ptr* is not `NULL`).

The alternate function `_wcstol_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

`wcstol_l` is like `wcstol` but performs the conversion based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns
`wcstol`, `wcstol_l` return the converted value, if any. If no conversion was made, 0 is returned.

`wcstol`, `wcstol_l` return `LONG_MAX` or `LONG_MIN` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

Portability
`wcstol` is ANSI. `wcstol_l` is a GNU extension.

No supporting OS subroutines are required.

2.45 wcstoll, wcstoll_l—wide string to long long

Synopsis

```
#include <wchar.h>
long long wcstoll(const wchar_t *__restrict s,
    wchar_t **__restrict ptr,int base);

#include <wchar.h>
long long wcstoll_l(const wchar_t *__restrict s,
    wchar_t **__restrict ptr, int base,
    locale_t locale);

long long _wcstoll_r(void *reent, const wchar_t *s,
    wchar_t **ptr, int base);
```

Description

The function `wcstoll` converts the wide string `*s` to a `long long`. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of characters resembling an integer in the radix specified by *base*; and a trailing portion consisting of zero or more unparseable characters, and always including the terminating null character. Then, it attempts to convert the subject string into a `long long` and returns the result.

If the value of *base* is 0, the subject string is expected to look like a normal C integer constant: an optional sign, a possible ‘`0x`’ indicating a hexadecimal base, and a number. If *base* is between 2 and 36, the expected form of the subject is a sequence of letters and digits representing an integer in the radix specified by *base*, with an optional plus or minus sign. The letters `a`–`z` (or, equivalently, `A`–`Z`) are used to signify values from 10 to 35; only letters whose ascribed values are less than *base* are permitted. If *base* is 16, a leading `0x` is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible letter or digit, the subject string is empty.

If the subject string is acceptable, and the value of *base* is zero, `wcstoll` attempts to determine the radix from the input string. A string with a leading `0x` is treated as a hexadecimal value; a string with a leading 0 and no `x` is treated as octal; all other strings are treated as decimal. If *base* is between 2 and 36, it is used as the conversion radix, as described above. If the subject string begins with a minus sign, the value is negated. Finally, a pointer to the first character past the converted subject string is stored in *ptr*, if *ptr* is not `NULL`.

If the subject string is empty (or not in acceptable form), no conversion is performed and the value of *s* is stored in *ptr* (if *ptr* is not `NULL`).

The alternate function `_wcstoll_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

`wcstoll_l` is like `wcstoll` but performs the conversion based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns
`wcstoll`, `wcstoll_l` return the converted value, if any. If no conversion was made, 0 is returned.

`wcstoll`, `wcstoll_l` return `LONG_LONG_MAX` or `LONG_LONG_MIN` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

Portability
`wcstoll` is ANSI. `wcstoll_l` is a GNU extension.

No supporting OS subroutines are required.

2.46 wcstoul, wcstoul_l—wide string to unsigned long

Synopsis

```
#include <wchar.h>
unsigned long wcstoul(const wchar_t *__restrict s,
    wchar_t **__restrict ptr, int base);

#include <wchar.h>
unsigned long wcstoul_l(const wchar_t *__restrict s,
    wchar_t **__restrict ptr, int base,
    locale_t locale);

unsigned long _wcstoul_r(void *reent, const wchar_t *s,
    wchar_t **ptr, int base);
```

Description

The function `wcstoul` converts the wide string `*s` to an `unsigned long`. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of the digits meaningful in the radix specified by *base* (for example, `0` through `7` if the value of *base* is 8); and a trailing portion consisting of one or more unparseable characters, which always includes the terminating null character. Then, it attempts to convert the subject string into an unsigned long integer, and returns the result.

If the value of *base* is zero, the subject string is expected to look like a normal C integer constant (save that no optional sign is permitted): a possible `0x` indicating hexadecimal radix, and a number. If *base* is between 2 and 36, the expected form of the subject is a sequence of digits (which may include letters, depending on the base) representing an integer in the radix specified by *base*. The letters `a`–`z` (or `A`–`Z`) are used as digits valued from 10 to 35. If *base* is 16, a leading `0x` is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible digit, the subject string is empty.

If the subject string is acceptable, and the value of *base* is zero, `wcstoul` attempts to determine the radix from the input string. A string with a leading `0x` is treated as a hexadecimal value; a string with a leading `0` and no `x` is treated as octal; all other strings are treated as decimal. If *base* is between 2 and 36, it is used as the conversion radix, as described above. Finally, a pointer to the first character past the converted subject string is stored in *ptr*, if *ptr* is not `NULL`.

If the subject string is empty (that is, if `*s` does not start with a substring in acceptable form), no conversion is performed and the value of *s* is stored in *ptr* (if *ptr* is not `NULL`).

The alternate function `_wcstoul_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

`wcstoul_l` is like `wcstoul` but performs the conversion based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns
`wcstoul`, `wcstoul_l` return the converted value, if any. If no conversion was made, `0` is returned.

`wcstoul`, `wcstoul_l` return `ULONG_MAX` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

Portability
`wcstoul` is ANSI. `wcstoul_l` is a GNU extension.

`wcstoul` requires no supporting OS subroutines.

2.47 wcstoull, wcstoull_l—wide string to unsigned long long

Synopsis

```
#include <wchar.h>
unsigned long long wcstoull(const wchar_t *__restrict s,
    wchar_t **__restrict ptr,
    int base);

#include <wchar.h>
unsigned long long wcstoull_l(const wchar_t *__restrict s,
    wchar_t **__restrict ptr,
    int base,
    locale_t locale);

unsigned long long _wcstoull_r(void *reent, const wchar_t *s,
    wchar_t **ptr, int base);
```

Description

The function `wcstoull` converts the wide string `*s` to an `unsigned long long`. First, it breaks down the string into three parts: leading whitespace, which is ignored; a subject string consisting of the digits meaningful in the radix specified by *base* (for example, `0` through `7` if the value of *base* is 8); and a trailing portion consisting of one or more unparseable characters, which always includes the terminating null character. Then, it attempts to convert the subject string into an unsigned long long integer, and returns the result.

If the value of *base* is zero, the subject string is expected to look like a normal C integer constant: an optional sign (`+` or `-`), a possible `0x` indicating hexadecimal radix or a possible `<0>` indicating octal radix, and a number. If *base* is between 2 and 36, the expected form of the subject is a sequence of digits (which may include letters, depending on the base) representing an integer in the radix specified by *base*. The letters `a`–`z` (or `A`–`Z`) are used as digits valued from 10 to 35. If *base* is 16, a leading `0x` is permitted.

The subject sequence is the longest initial sequence of the input string that has the expected form, starting with the first non-whitespace character. If the string is empty or consists entirely of whitespace, or if the first non-whitespace character is not a permissible digit, the subject string is empty.

If the subject string is acceptable, and the value of *base* is zero, `wcstoull` attempts to determine the radix from the input string. A string with a leading `0x` is treated as a hexadecimal value; a string with a leading `0` and no `x` is treated as octal; all other strings are treated as decimal. If *base* is between 2 and 36, it is used as the conversion radix, as described above. Finally, a pointer to the first character past the converted subject string is stored in *ptr*, if *ptr* is not `NULL`.

If the subject string is empty (that is, if `*s` does not start with a substring in acceptable form), no conversion is performed and the value of *s* is stored in *ptr* (if *ptr* is not `NULL`).

The alternate function `_wcstoull_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

`wcstoull_l` is like `wcstoull` but performs the conversion based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`wcstoull`, `wcstoull_l` return `0` and sets `errno` to `EINVAL` if the value of *base* is not supported.

`wcstoull`, `wcstoull_l` return the converted value, if any. If no conversion was made, `0` is returned.

`wcstoull`, `wcstoull_l` return `ULLONG_MAX` if the magnitude of the converted value is too large, and sets `errno` to `ERANGE`.

Portability

`wcstoull` is ANSI. `wcstoull_l` is a GNU extension.

`wcstoull` requires no supporting OS subroutines.

2.48 system—execute command string

Synopsis

```
#include <stdlib.h>
int system(char *s);

int _system_r(void *reent, char *s);
```

Description

Use `system` to pass a command string `*s` to `/bin/sh` on your system, and wait for it to finish executing.

Use “`system(NULL)`” to test whether your system has `/bin/sh` available.

The alternate function `_system_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`system(NULL)` returns a non-zero value if `/bin/sh` is available, and `0` if it is not.

With a command argument, the result of `system` is the exit status returned by `/bin/sh`.

Portability

ANSI C requires `system`, but leaves the nature and effects of a command processor undefined. ANSI C does, however, specify that `system(NULL)` return zero or nonzero to report on the existence of a command processor.

POSIX.2 requires `system`, and requires that it invoke a `sh`. Where `sh` is found is left unspecified.

Supporting OS subroutines required: `_exit`, `_execve`, `_fork_r`, `_wait_r`.

2.49 `utoa`—unsigned integer to string

Synopsis

```
#include <stdlib.h>
char *utoa(unsigned value, char *str, int base);
char *__utoa(unsigned value, char *str, int base);
```

Description

`utoa` converts the unsigned integer [<value>] to a null-terminated string using the specified base, which must be between 2 and 36, inclusive. *str* should be an array long enough to contain the converted value, which in the worst case is sizeof(int)*8+1 bytes.

Returns

A pointer to the string, *str*, or NULL if *base* is invalid.

Portability

`utoa` is non-ANSI.

No supporting OS subroutine calls are required.

2.50 wcstombs—minimal wide char string to multibyte string converter

Synopsis

```
#include <stdlib.h>
size_t wcstombs(char *restrict s, const wchar_t *restrict pwc, size_t n);
```

Description

When _MB_CAPABLE is not defined, this is a minimal ANSI-conforming implementation of `wcstombs`. In this case, all wide-characters are expected to represent single bytes and so are converted simply by casting to char.

When _MB_CAPABLE is defined, this routine calls `_wcstombs_r` to perform the conversion, passing a state variable to allow state dependent decoding. The result is based on the locale setting which may be restricted to a defined set of locales.

Returns

This implementation of `wcstombs` returns `0` if *s* is `NULL` or is the empty string; it returns `-1` if _MB_CAPABLE and one of the wide-char characters does not represent a valid multi-byte character; otherwise it returns the minimum of: `n` or the number of bytes that are transferred to `s`, not including the nul terminator.

If the return value is -1, the state of the `pwc` string is indeterminate. If the input has a length of 0, the output string will be modified to contain a wchar_t nul terminator if `n` > 0.

Portability

`wcstombs` is required in the ANSI C standard. However, the precise effects vary with the locale.

`wcstombs` requires no supporting OS subroutines.

2.51 wctomb—minimal wide char to multibyte converter

Synopsis

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

Description

When _MB_CAPABLE is not defined, this is a minimal ANSI-conforming implementation of `wctomb`. The only "wide characters" recognized are single bytes, and they are "converted" to themselves.

When _MB_CAPABLE is defined, this routine calls `_wctomb_r` to perform the conversion, passing a state variable to allow state dependent decoding. The result is based on the locale setting which may be restricted to a defined set of locales.

Each call to `wctomb` modifies `*s` unless *s* is a null pointer or _MB_CAPABLE is defined and *wchar* is invalid.

Returns

This implementation of `wctomb` returns `0` if *s* is `NULL`; it returns `-1` if _MB_CAPABLE is enabled and the wchar is not a valid multi-byte character, it returns `1` if _MB_CAPABLE is not defined or the wchar is in reality a single byte character, otherwise it returns the number of bytes in the multi-byte character.

Portability

`wctomb` is required in the ANSI C standard. However, the precise effects vary with the locale.

`wctomb` requires no supporting OS subroutines.

3 Character Type Macros and Functions (‘ctype.h’)

This chapter groups macros (which are also available as subroutines) to classify characters into several categories (alphabetic, numeric, control characters, whitespace, and so on), or to perform simple character mappings.

The header file ‘ctype.h’ defines the macros.

3.1 isalnum, isalnum_l—alphanumeric character predicate

Synopsis

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

#include <ctype.h>
int isalnum_l(int c, locale_t locale);
```

Description

`isalnum` is a macro which classifies singlebyte charset values by table lookup. It is a predicate returning non-zero for alphabetic or numeric ASCII characters, and `0` for other arguments. It is defined only if *c* is representable as an unsigned char or if *c* is EOF.

`isalnum_l` is like `isalnum` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '`#undef isalnum`' or '`#undef isalnum_l`'.

Returns

`isalnum`,`isalnum_l` return non-zero if *c* is a letter or a digit.

Portability

`isalnum` is ANSI C. `isalnum_l` is POSIX-1.2008.

No OS subroutines are required.

3.2 isalpha, isalpha_l—alphabetic character predicate

Synopsis

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

#include <ctype.h>
int isalpha_l(int c, locale_t locale);
```

Description

isalpha is a macro which classifies singlebyte charset values by table lookup. It is a predicate returning non-zero when *c* represents an alphabetic ASCII character, and 0 otherwise. It is defined only if *c* is representable as an unsigned char or if *c* is EOF.

isalpha_l is like isalpha but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining the macro using ‘#undef isalpha’ or ‘#undef isalpha_l’.

Returns

isalpha, isalpha_l return non-zero if *c* is a letter.

Portability

isalpha is ANSI C. isalpha_l is POSIX-1.2008.

No supporting OS subroutines are required.

3.3 isascii, isascii_l—ASCII character predicate

Synopsis

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

#include <ctype.h>
int isascii_l(int c, locale_t locale);
```

Description

`isascii` is a macro which returns non-zero when *c* is an ASCII character, and 0 otherwise. It is defined for all integer values.

`isascii_l` is like `isascii` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '`#undef isascii`' or '`#undef isascii_l`'.

Returns

`isascii`, `isascii_l` return non-zero if the low order byte of *c* is in the range 0 to 127 (`0x00`–`0x7F`).

Portability

`isascii` is ANSI C. `isascii_l` is a GNU extension.

No supporting OS subroutines are required.

3.4 isblank, isblank_l—blank character predicate

Synopsis

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

#include <ctype.h>
int isblank_l(int c, locale_t locale);
```

Description

isblank is a function which classifies singlebyte charset values by table lookup. It is a predicate returning non-zero for blank characters, and 0 for other characters. It is defined only if *c* is representable as an unsigned char or if *c* is EOF.

isblank_l is like isblank but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

isblank, isblank_l return non-zero if *c* is a blank character.

Portability

isblank is C99. isblank_l is POSIX-1.2008.

No supporting OS subroutines are required.

3.5 iscntrl, iscntrl_l—control character predicate

Synopsis

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

#include <ctype.h>
int iscntrl_l(int c, locale_t locale);
```

Description

`iscntrl` is a macro which classifies singlebyte charset values by table lookup. It is a predicate returning non-zero for control characters, and 0 for other characters. It is defined only if *c* is representable as an unsigned char or if *c* is EOF.

`iscntrl_l` is like `iscntrl` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '`#undef iscntrl`' or '`#undef iscntrl_l`'.

Returns

`iscntrl`, `iscntrl_l` return non-zero if *c* is a delete character or ordinary control character.

Portability

`iscntrl` is ANSI C. `iscntrl_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.6 isdigit, isdigit_l—decimal digit predicate

Synopsis

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

#include <ctype.h>
int isdigit_l(int c, locale_t locale);
```

Description

isdigit is a macro which classifies singlebyte charset values by table lookup. It is a predicate returning non-zero for decimal digits, and 0 for other characters. It is defined only if *c* is representable as an unsigned char or if *c* is EOF.

isdigit_l is like isdigit but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining the macro using ‘#undef isdigit’ or ‘#undef isdigit_l’.

Returns

isdigit, isdigit_l return non-zero if *c* is a decimal digit (0–9).

Portability

isdigit is ANSI C. isdigit_l is POSIX-1.2008.

No supporting OS subroutines are required.

3.7 islower, islower_l—lowercase character predicate

Synopsis

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

#include <ctype.h>
int islower_l(int c, locale_t locale);
```

Description

islower is a macro which classifies singlebyte charset values by table lookup. It is a predicate returning non-zero for minuscules (lowercase alphabetic characters), and 0 for other characters. It is defined only if *c* is representable as an unsigned char or if *c* is EOF.

islower_l is like islower but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '#undef islower' or '#undef islower_l'.

Returns

islower, islower_l return non-zero if *c* is a lowercase letter.

Portability

islower is ANSI C. islower_l is POSIX-1.2008.

No supporting OS subroutines are required.

3.8 isprint, isgraph, isprint_l, isgraph_l—printable character predicates

Synopsis

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

#include <ctype.h>
int isprint_l(int c, locale_t locale);
int isgraph_l(int c, locale_t locale);
```

Description

`isprint` is a macro which classifies singlebyte charset values by table lookup. It is a predicate returning non-zero for printable characters, and 0 for other character arguments. It is defined only if *c* is representable as an unsigned char or if *c* is EOF.

`isgraph` behaves identically to `isprint`, except that space characters are excluded.

`isprint_l`, `isgraph_l` are like `isprint`, `isgraph` but perform the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining either macro using ‘`#undef isprint`’ or ‘`#undef isgraph`’, or ‘`#undef isprint_l`’ or ‘`#undef isgraph_l`’.

Returns

`isprint`, `isprint_l` return non-zero if *c* is a printing character. `isgraph`, `isgraph_l` return non-zero if *c* is a printing character except spaces.

Portability

`isprint` and `isgraph` are ANSI C.

No supporting OS subroutines are required.

3.9 ispunct, ispunct_l—punctuation character predicate

Synopsis

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

#include <ctype.h>
int ispunct_l(int c, locale_t locale);
```

Description

`ispunct` is a macro which classifies singlebyte charset values by table lookup. It is a predicate returning non-zero for printable punctuation characters, and 0 for other characters. It is defined only if *c* is representable as an unsigned char or if *c* is EOF.

`ispunct_l` is like `ispunct` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '`#undef ispunct`' or '`#undef ispunct_l`'.

Returns

`ispunct`, `ispunct_l` return non-zero if *c* is a printable punctuation character.

Portability

`ispunct` is ANSI C. `ispunct_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.10 isspace, isspace_l—whitespace character predicate

Synopsis

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

#include <ctype.h>
int isspace_l(int c, locale_t locale);
```

Description

isspace is a macro which classifies singlebyte charset values by table lookup. It is a predicate returning non-zero for whitespace characters, and 0 for other characters. It is defined only when isascii(c) is true or c is EOF.

isspace_l is like isspace but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining the macro using ‘#undef isspace’ or ‘#undef isspace_l’.

Returns

isspace, isspace_l return non-zero if c is a space, tab, carriage return, new line, vertical tab, or formfeed (0x09–0x0D, 0x20), or one of the other space characters in non-ASCII charsets.

Portability

isspace is ANSI C. isspace_l is POSIX-1.2008.

No supporting OS subroutines are required.

3.11 `isupper`, `isupper_l`—uppercase character predicate

Synopsis

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

#include <ctype.h>
int isupper_l(int c, locale_t locale);
```

Description

`isupper` is a macro which classifies singlebyte charset values by table lookup. It is a predicate returning non-zero for uppercase letters (`A`–`Z`), and 0 for other characters.

`isupper_l` is like `isupper` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining the macro using '`#undef isupper`' or '`#undef isupper_l`'.

Returns

`isupper`, `isupper_l` return non-zero if *c* is an uppercase letter.

Portability

`isupper` is ANSI C. `isupper_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.12 isxdigit, isxdigit_l—hexadecimal digit predicate

Synopsis

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

#include <ctype.h>
int isxdigit_l(int c, locale_t locale);
```

Description

isxdigit is a macro which classifies singlebyte charset values by table lookup. It is a predicate returning non-zero for hexadecimal digits, and 0 for other characters. It is defined only if *c* is representable as an unsigned char or if *c* is EOF.

isxdigit_l is like isxdigit but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining the macro using ‘#undef isxdigit’ or ‘#undef isxdigit_l’.

Returns

isxdigit, isxdigit_l return non-zero if *c* is a hexadecimal digit (0–9, a–f, or A–F).

Portability

isxdigit is ANSI C. isxdigit_l is POSIX-1.2008.

No supporting OS subroutines are required.

3.13 `toascii`, `toascii_l`—force integers to ASCII range

Synopsis

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

#include <ctype.h>
int toascii_l(int c, locale_t locale);
```

Description

`toascii` is a macro which coerces integers to the ASCII range (0–127) by zeroing any higher-order bits.

`toascii_l` is like `toascii` but performs the function based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining this macro using '`#undef toascii`' or '`#undef toascii_l`'.

Returns

`toascii`, `toascii_l` return integers between 0 and 127.

Portability

`toascii` is X/Open, BSD and POSIX-1.2001, but marked obsolete in POSIX-1.2008. `toascii_l` is a GNU extension.

No supporting OS subroutines are required.

3.14 tolower, tolower_l—translate characters to lowercase

Synopsis

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

#include <ctype.h>
int tolower_l(int c, locale_t locale);
```

Description

`tolower` is a macro which converts uppercase characters to lowercase, leaving all other characters unchanged. It is only defined when *c* is an integer in the range `EOF` to `255`.

`tolower_l` is like `tolower` but performs the function based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining this macro using ‘`#undef tolower`’ or ‘`#undef tolower_l`’.

`_tolower` performs the same conversion as `tolower`, but should only be used when *c* is known to be an uppercase character (`A`–`Z`).

Returns

`tolower`, `tolower_l` return the lowercase equivalent of *c* when *c* is an uppercase character, and *c* otherwise.

`_tolower` returns the lowercase equivalent of *c* when it is a character between `A` and `Z`. If *c* is not one of these characters, the behaviour of `_tolower` is undefined.

Portability

`tolower` is ANSI C. `_tolower` is not recommended for portable programs. `tolower_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.15 toupper, toupper_l—translate characters to uppercase

Synopsis

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

#include <ctype.h>
int toupper_l(int c, locale_t locale);
```

Description

`toupper` is a macro which converts lowercase characters to uppercase, leaving all other characters unchanged. It is only defined when *c* is an integer in the range `EOF` to `255`.

`toupper_l` is like `toupper` but performs the function based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You can use a compiled subroutine instead of the macro definition by undefining this macro using '`#undef toupper`' or '`#undef toupper_l`'.

`_toupper` performs the same conversion as `toupper`, but should only be used when *c* is known to be a lowercase character (`a`–`z`).

Returns

`toupper`, `toupper_l` return the uppercase equivalent of *c* when *c* is a lowercase character, and *c* otherwise.

`_toupper` returns the uppercase equivalent of *c* when it is a character between `a` and `z`. If *c* is not one of these characters, the behaviour of `_toupper` is undefined.

Portability

`toupper` is ANSI C. `_toupper` is not recommended for portable programs. `toupper_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.16 iswalnum, iswalnum_l—alphanumeric wide character test

Synopsis

```
#include <wctype.h>
int iswalnum(wint_t c);

#include <wctype.h>
int iswalnum_l(wint_t c, locale_t locale);
```

Description

`iswalnum` is a function which classifies wide-character values that are alphanumeric.

`iswalnum_l` is like `iswalnum` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`iswalnum`, `iswalnum_l` return non-zero if *c* is a alphanumeric wide character.

Portability

`iswalnum` is C99. `iswalnum_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.17 iswalpha, iswalpha_l—alphabetic wide character test

Synopsis

```
#include <wctype.h>
int iswalpha(wint_t c);

#include <wctype.h>
int iswalpha_l(wint_t c, locale_t locale);
```

Description

`iswalpha` is a function which classifies wide-character values that are alphabetic.

`iswalpha_l` is like `iswalpha` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`iswalpha`, `iswalpha_l` return non-zero if *c* is an alphabetic wide character.

Portability

`iswalpha` is C99. `iswalpha_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.18 iswcntrl, iswcntrl_l—control wide character test

Synopsis

```
#include <wctype.h>
int iswcntrl(wint_t c);

#include <wctype.h>
int iswcntrl_l(wint_t c, locale_t locale);
```

Description

`iswcntrl` is a function which classifies wide-character values that are categorized as control characters.

`iswcntrl_l` is like `iswcntrl` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`iswcntrl`, `iswcntrl_l` return non-zero if *c* is a control wide character.

Portability

`iswcntrl` is C99. `iswcntrl_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.19 iswblank, iswblank_l—blank wide character test

Synopsis

```
#include <wctype.h>
int iswblank(wint_t c);

#include <wctype.h>
int iswblank_l(wint_t c, locale_t locale);
```

Description

`iswblank` is a function which classifies wide-character values that are categorized as blank.

`iswblank_l` is like `iswblank` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`iswblank`, `iswblank_l` return non-zero if *c* is a blank wide character.

Portability

`iswblank` is C99. `iswblank_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.20 iswdigit, iswdigit_l—decimal digit wide character test

Synopsis

```
#include <wctype.h>
int iswdigit(wint_t c);

#include <wctype.h>
int iswdigit_l(wint_t c, locale_t locale);
```

Description

`iswdigit` is a function which classifies wide-character values that are decimal digits.

`iswdigit_l` is like `iswdigit` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`iswdigit`, `iswdigit_l` return non-zero if *c* is a decimal digit wide character.

Portability

`iswdigit` is C99. `iswdigit_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.21 iswgraph, iswgraph_l—graphic wide character test

Synopsis

```
#include <wctype.h>
int iswgraph(wint_t c);

#include <wctype.h>
int iswgraph_l(wint_t c, locale_t locale);
```

Description

`iswgraph` is a function which classifies wide-character values that are graphic.

`iswgraph_l` is like `iswgraph` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`iswgraph`, `iswgraph_l` return non-zero if *c* is a graphic wide character.

Portability

`iswgraph` is C99. `iswgraph_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.22 iswlower, iswlower_l—lowercase wide character test

Synopsis

```
#include <wctype.h>
int iswlower(wint_t c);

#include <wctype.h>
int iswlower_l(wint_t c, locale_t locale);
```

Description

`iswlower` is a function which classifies wide-character values that are categorized as lowercase.

`iswlower_l` is like `iswlower` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`iswlower`, `iswlower_l` return non-zero if *c* is a lowercase wide character.

Portability

`iswlower` is C99. `iswlower_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.23 iswprint, iswprint_l—printable wide character test

Synopsis

```
#include <wctype.h>
int iswprint(wint_t c);

#include <wctype.h>
int iswprint_l(wint_t c, locale_t locale);
```

Description

`iswprint` is a function which classifies wide-character values that are printable.

`iswprint_l` is like `iswprint` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`iswprint`, `iswprint_l` return non-zero if *c* is a printable wide character.

Portability

`iswprint` is C99. `iswprint_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.24 iswpunct, iswpunct_l—punctuation wide character test

Synopsis

```
#include <wctype.h>
int iswpunct(wint_t c);

#include <wctype.h>
int iswpunct_l(wint_t c, locale_t locale);
```

Description

`iswpunct` is a function which classifies wide-character values that are punctuation.

`iswpunct_l` is like `iswpunct` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`iswpunct`, `iswpunct_l` return non-zero if *c* is a punctuation wide character.

Portability

`iswpunct` is C99. `iswpunct_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.25 iswspace, iswspace_l—whitespace wide character test

Synopsis

```
#include <wctype.h>
int iswspace(wint_t c);

#include <wctype.h>
int iswspace_l(wint_t c, locale_t locale);
```

Description

`iswspace` is a function which classifies wide-character values that are categorized as whitespace.

`iswspace_l` is like `iswspace` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`iswspace`, `iswspace_l` return non-zero if *c* is a whitespace wide character.

Portability

`iswspace` is C99. `iswspace_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.26 iswupper, iswupper_l—uppercase wide character test

Synopsis

```
#include <wctype.h>
int iswupper(wint_t c);

#include <wctype.h>
int iswupper_l(wint_t c, locale_t locale);
```

Description

`iswupper` is a function which classifies wide-character values that are categorized as uppercase.

`iswupper_l` is like `iswupper` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`iswupper`, `iswupper_l` return non-zero if *c* is an uppercase wide character.

Portability

`iswupper` is C99. `iswupper_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.27 iswxdigit, iswxdigit_l—hexadecimal digit wide character test

Synopsis

```
#include <wctype.h>
int iswxdigit(wint_t c);

#include <wctype.h>
int iswxdigit_l(wint_t c, locale_t locale);
```

Description

`iswxdigit` is a function which classifies wide character values that are hexadecimal digits.

`iswxdigit_l` is like `iswxdigit` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`iswxdigit`, `iswxdigit_l` return non-zero if *c* is a hexadecimal digit wide character.

Portability

`iswxdigit` is C99. `iswxdigit_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.28 iswctype, iswctype_l—extensible wide-character test

Synopsis

```
#include <wctype.h>
int iswctype(wint_t c, wctype_t desc);

#include <wctype.h>
int iswctype_l(wint_t c, wctype_t desc, locale_t locale);
```

Description

`iswctype` is a function which classifies wide-character values using the wide-character test specified by *desc*.

`iswctype_l` is like `iswctype` but performs the check based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`iswctype`, `iswctype_l` return non-zero if and only if *c* matches the test specified by *desc*. If *desc* is unknown, zero is returned.

Portability

`iswctype` is C99. `iswctype_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.29 wctype, wctype_l—get wide-character classification type

Synopsis

```
#include <wctype.h>
wctype_t wctype(const char *c);

#include <wctype.h>
wctype_t wctype_l(const char *c, locale_t locale);
```

Description

`wctype` is a function which takes a string *c* and gives back the appropriate wctype_t type value associated with the string, if one exists. The following values are guaranteed to be recognized: "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper", and "xdigit".

`wctype_l` is like `wctype` but performs the function based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`wctype`, `wctype_l` return 0 and sets `errno` to `EINVAL` if the given name is invalid. Otherwise, it returns a valid non-zero wctype_t value.

Portability

`wctype` is C99. `wctype_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.30 towlower, towlower_l—translate wide characters to lowercase

Synopsis

```
#include <wctype.h>
wint_t towlower(wint_t c);

#include <wctype.h>
wint_t towlower_l(wint_t c, locale_t locale);
```

Description

towlower is a function which converts uppercase wide characters to lowercase, leaving all other characters unchanged.

towlower_l is like towlower but performs the function based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

towlower, towlower_l return the lowercase equivalent of *c* when it is a uppercase wide character; otherwise, it returns the input character.

Portability

towlower is C99. towlower_l is POSIX-1.2008.

No supporting OS subroutines are required.

3.31 towupper, towupper_l—translate wide characters to uppercase

Synopsis

```
#include <wctype.h>
wint_t towupper(wint_t c);

#include <wctype.h>
wint_t towupper_l(wint_t c, locale_t locale);
```

Description

`towupper` is a function which converts lowercase wide characters to uppercase, leaving all other characters unchanged.

`towupper_l` is like `towupper` but performs the function based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`towupper`, `towupper_l` return the uppercase equivalent of *c* when it is a lowercase wide character, otherwise, it returns the input character.

Portability

`towupper` is C99. `towupper_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.32 towctrans, towctrans_l—extensible wide-character translation

Synopsis

```
#include <wctype.h>
wint_t towctrans(wint_t c, wctrans_t w);

#include <wctype.h>
wint_t towctrans_l(wint_t c, wctrans_t w, locale_t locale);
```

Description

`towctrans` is a function which converts wide characters based on a specified translation type *w*. If the translation type is invalid or cannot be applied to the current character, no change to the character is made.

`towctrans_l` is like `towctrans` but performs the function based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`towctrans`, `towctrans_l` return the translated equivalent of *c* when it is a valid for the given translation, otherwise, it returns the input character. When the translation type is invalid, `errno` is set to `EINVAL`.

Portability

`towctrans` is C99. `towctrans_l` is POSIX-1.2008.

No supporting OS subroutines are required.

3.33 wctrans, wctrans_l—get wide-character translation type

Synopsis

```
#include <wctype.h>
wctrans_t wctrans(const char *c);

#include <wctype.h>
wctrans_t wctrans_l(const char *c, locale_t locale);
```

Description

`wctrans` is a function which takes a string *c* and gives back the appropriate wctrans_t type value associated with the string, if one exists. The following values are guaranteed to be recognized: "tolower" and "toupper".

`wctrans_l` is like `wctrans` but performs the function based on the locale specified by the locale object locale. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

Returns

`wctrans`, `wctrans_l` return 0 and sets `errno` to `EINVAL` if the given name is invalid. Otherwise, it returns a valid non-zero wctrans_t value.

Portability

`wctrans` is C99. `wctrans_l` is POSIX-1.2008.

No supporting OS subroutines are required.

4 Input and Output (‘stdio.h’)

This chapter comprises functions to manage files or other input/output streams. Among these functions are subroutines to generate or scan strings according to specifications from a format string.

The underlying facilities for input and output depend on the host system, but these functions provide a uniform interface.

The corresponding declarations are in ‘stdio.h’.

The reentrant versions of these functions use macros

```
_stdin_r(reent)
_stdout_r(reent)
_stderr_r(reent)
```

instead of the globals `stdin`, `stdout`, and `stderr`. The argument *reent* is a pointer to a reentrancy structure.

4.1 clearerr, clearerr_unlocked—clear file or stream error indicator

Synopsis

```
#include <stdio.h>
void clearerr(FILE *fp);

#define _BSD_SOURCE
#include <stdio.h>
void clearerr_unlocked(FILE *fp);
```

Description

The `stdio` functions maintain an error indicator with each file pointer *fp*, to record whether any read or write errors have occurred on the associated file or stream. Similarly, it maintains an end-of-file indicator to record whether there is no more data in the file.

Use `clearerr` to reset both of these indicators.

See `ferror` and `feof` to query the two indicators.

`clearerr_unlocked` is a non-thread-safe version of `clearerr`. `clearerr_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `clearerr_unlocked` is equivalent to `clearerr`.

Returns

`clearerr` does not return a result.

Portability

ANSI C requires `clearerr`.

`clearerr_unlocked` is a BSD extension also provided by GNU libc.

No supporting OS subroutines are required.

4.2 diprintf, vdiprintf—print to a file descriptor (integer only)

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
int diprintf(int fd, const char *format, ...);
int vdiprintf(int fd, const char *format, va_list ap);
int _diprintf_r(struct _reent *ptr, int fd,
    const char *format, ...);
int _vdiprintf_r(struct _reent *ptr, int fd,
    const char *format, va_list ap);
```

Description

`diprintf` and `vdiprintf` are similar to `dprintf` and `vdprintf`, except that only integer format specifiers are processed.

The functions `_diprintf_r` and `_vdiprintf_r` are simply reentrant versions of the functions above.

Returns

Similar to `dprintf` and `vdprintf`.

Portability

This set of functions is an integer-only extension, and is not portable.

Supporting OS subroutines required: `sbrk`, `write`.

4.3 dprintf, vdprintf—print to a file descriptor

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
int dprintf(int fd, const char *restrict format, ...);
int vdprintf(int fd, const char *restrict format,
    va_list ap);
int _dprintf_r(struct _reent *ptr, int fd,
    const char *restrict format, ...);
int _vdprintf_r(struct _reent *ptr, int fd,
    const char *restrict format, va_list ap);
```

Description

`dprintf` and `vdprintf` allow printing a format, similarly to `printf`, but write to a file descriptor instead of to a `FILE` stream.

The functions `_dprintf_r` and `_vdprintf_r` are simply reentrant versions of the functions above.

Returns

The return value and errors are exactly as for `write`, except that `errno` may also be set to `ENOMEM` if the heap is exhausted.

Portability

This function is originally a GNU extension in glibc and is not portable.

Supporting OS subroutines required: `sbrk`, `write`.

4.4 fclose—close a file

Synopsis

```
#include <stdio.h>
int fclose(FILE *fp);
int _fclose_r(struct _reent *reent, FILE *fp);
```

Description

If the file or stream identified by *fp* is open, `fclose` closes it, after first ensuring that any pending data is written (by calling `fflush(fp)`).

The alternate function `_fclose_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`fclose` returns `0` if successful (including when *fp* is `NULL` or not an open file); otherwise, it returns `EOF`.

Portability

`fclose` is required by ANSI C.

Required OS subroutines: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.5 `fcloseall`—close all files

Synopsis

```
#include <stdio.h>
int fcloseall(void);
int _fcloseall_r (struct _reent *ptr);
```

Description

`fcloseall` closes all files in the current reentrancy struct's domain. The function `_fcloseall_r` is the same function, except the reentrancy struct is passed in as the *ptr* argument.

This function is not recommended as it closes all streams, including the std streams.

Returns

`fclose` returns `0` if all closes are successful. Otherwise, EOF is returned.

Portability

`fcloseall` is a glibc extension.

Required OS subroutines: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.6 fdopen—turn open file into a stream

Synopsis

```
#include <stdio.h>
FILE *fdopen(int fd, const char *mode);
FILE *_fdopen_r(struct _reent *reent,
    int fd, const char *mode);
```

Description

`fdopen` produces a file descriptor of type `FILE *`, from a descriptor for an already-open file (returned, for example, by the system subroutine `open` rather than by `fopen`). The *mode* argument has the same meanings as in `fopen`.

Returns

File pointer or `NULL`, as for `fopen`.

Portability

`fdopen` is ANSI.

4.7 `feof`, `feof_unlocked`—test for end of file

Synopsis

```
#include <stdio.h>
int feof(FILE *fp);

#define _BSD_SOURCE
#include <stdio.h>
int feof_unlocked(FILE *fp);
```

Description

`feof` tests whether or not the end of the file identified by *fp* has been reached.

`feof_unlocked` is a non-thread-safe version of `feof`. `feof_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `feof_unlocked` is equivalent to `feof`.

Returns

`feof` returns `0` if the end of file has not yet been reached; if at end of file, the result is nonzero.

Portability

`feof` is required by ANSI C.

`feof_unlocked` is a BSD extension also provided by GNU libc.

No supporting OS subroutines are required.

4.8 ferror, ferror_unlocked—test whether read/write error has occurred

Synopsis

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

#define _BSD_SOURCE
#include <stdio.h>
int ferror_unlocked(FILE *fp);
```

Description

The stdio functions maintain an error indicator with each file pointer *fp*, to record whether any read or write errors have occurred on the associated file or stream. Use ferror to query this indicator.

See clearerr to reset the error indicator.

ferror_unlocked is a non-thread-safe version of ferror. ferror_unlocked may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then ferror_unlocked is equivalent to ferror.

Returns

ferror returns 0 if no errors have occurred; it returns a nonzero value otherwise.

Portability

ANSI C requires ferror.

ferror_unlocked is a BSD extension also provided by GNU libc.

No supporting OS subroutines are required.

4.9 fflush, fflush_unlocked—flush buffered file output

Synopsis

```
#include <stdio.h>
int fflush(FILE *fp);

#define _BSD_SOURCE
#include <stdio.h>
int fflush_unlocked(FILE *fp);

#include <stdio.h>
int _fflush_r(struct _reent *reent, FILE *fp);

#define _BSD_SOURCE
#include <stdio.h>
int _fflush_unlocked_r(struct _reent *reent, FILE *fp);
```

Description

The `stdio` output functions can buffer output before delivering it to the host system, in order to minimize the overhead of system calls.

Use `fflush` to deliver any such pending output (for the file or stream identified by *fp*) to the host system.

If *fp* is `NULL`, `fflush` delivers pending output from all open files.

Additionally, if *fp* is a seekable input stream visiting a file descriptor, set the position of the file descriptor to match next unread byte, useful for obeying POSIX semantics when ending a process without consuming all input from the stream.

`fflush_unlocked` is a non-thread-safe version of `fflush`. `fflush_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `fflush_unlocked` is equivalent to `fflush`.

The alternate functions `_fflush_r` and `_fflush_unlocked_r` are reentrant versions, where the extra argument *reent* is a pointer to a reentrancy structure, and *fp* must not be NULL.

Returns

`fflush` returns `0` unless it encounters a write error; in that situation, it returns `EOF`.

Portability

ANSI C requires `fflush`. The behavior on input streams is only specified by POSIX, and not all implementations follow POSIX rules.

`fflush_unlocked` is a BSD extension also provided by GNU libc.

No supporting OS subroutines are required.

4.10 fgetc, fgetc_unlocked—get a character from a file or stream

Synopsis

```
#include <stdio.h>
int fgetc(FILE *fp);

#define _BSD_SOURCE
#include <stdio.h>
int fgetc_unlocked(FILE *fp);

#include <stdio.h>
int _fgetc_r(struct _reent *ptr, FILE *fp);

#define _BSD_SOURCE
#include <stdio.h>
int _fgetc_unlocked_r(struct _reent *ptr, FILE *fp);
```

Description

Use `fgetc` to get the next single character from the file or stream identified by *fp*. As a side effect, `fgetc` advances the file's current position indicator.

For a macro version of this function, see `getc`.

`fgetc_unlocked` is a non-thread-safe version of `fgetc`. `fgetc_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `fgetc_unlocked` is equivalent to `fgetc`.

The functions `_fgetc_r` and `_fgetc_unlocked_r` are simply reentrant versions that are passed the additional reentrant structure pointer argument: *ptr*.

Returns

The next character (read as an `unsigned char`, and cast to `int`), unless there is no more data, or the host system reports a read error; in either of these situations, `fgetc` returns `EOF`.

You can distinguish the two situations that cause an `EOF` result by using the `ferror` and `feof` functions.

Portability

ANSI C requires `fgetc`.

`fgetc_unlocked` is a BSD extension also provided by GNU libc.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.11 fgetpos—record position in a stream or file

Synopsis

```
#include <stdio.h>
int fgetpos(FILE *restrict fp, fpos_t *restrict pos);
int _fgetpos_r(struct _reent *ptr, FILE *restrict fp, fpos_t *restrict pos);
```

Description

Objects of type `FILE` can have a "position" that records how much of the file your program has already read. Many of the `stdio` functions depend on this position, and many change it as a side effect.

You can use `fgetpos` to report on the current position for a file identified by *fp*; `fgetpos` will write a value representing that position at `*pos`. Later, you can use this value with `fsetpos` to return the file to this position.

In the current implementation, `fgetpos` simply uses a character count to represent the file position; this is the same number that would be returned by `ftell`.

Returns

`fgetpos` returns `0` when successful. If `fgetpos` fails, the result is `1`. Failure occurs on streams that do not support positioning; the global `errno` indicates this condition with the value `ESPIPE`.

Portability

`fgetpos` is required by the ANSI C standard, but the meaning of the value it records is not specified beyond requiring that it be acceptable as an argument to `fsetpos`. In particular, other conforming C implementations may return a different result from `ftell` than what `fgetpos` writes at `*pos`.

No supporting OS subroutines are required.

4.12 fgets, fgets_unlocked—get character string from a file or stream

Synopsis

```
#include <stdio.h>
char *fgets(char *restrict buf, int n, FILE *restrict fp);

#define _GNU_SOURCE
#include <stdio.h>
char *fgets_unlocked(char *restrict buf, int n, FILE *restrict fp);

#include <stdio.h>
char *_fgets_r(struct _reent *ptr, char *restrict buf, int n, FILE *restrict fp);

#include <stdio.h>
char *_fgets_unlocked_r(struct _reent *ptr, char *restrict buf, int n, FILE *restrict
```

Description

Reads at most *n-1* characters from *fp* until a newline is found. The characters including to the newline are stored in *buf*. The buffer is terminated with a 0.

`fgets_unlocked` is a non-thread-safe version of `fgets`. `fgets_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `fgets_unlocked` is equivalent to `fgets`.

The functions `_fgets_r` and `_fgets_unlocked_r` are simply reentrant versions that are passed the additional reentrant structure pointer argument: *ptr*.

Returns

`fgets` returns the buffer passed to it, with the data filled in. If end of file occurs with some data already accumulated, the data is returned with no other indication. If no data are read, NULL is returned instead.

Portability

`fgets` should replace all uses of `gets`. Note however that `fgets` returns all of the data, while `gets` removes the trailing newline (with no indication that it has done so.)

`fgets_unlocked` is a GNU extension.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.13 fgetwc, getwc, fgetwc_unlocked, getwc_unlocked—get a wide character from a file or stream

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *fp);

#define _GNU_SOURCE
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc_unlocked(FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t _fgetwc_r(struct _reent *ptr, FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t _fgetwc_unlocked_r(struct _reent *ptr, FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *fp);

#define _GNU_SOURCE
#include <stdio.h>
#include <wchar.h>
wint_t getwc_unlocked(FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t _getwc_r(struct _reent *ptr, FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t _getwc_unlocked_r(struct _reent *ptr, FILE *fp);
```

Description

Use `fgetwc` to get the next wide character from the file or stream identified by *fp*. As a side effect, `fgetwc` advances the file's current position indicator.

`fgetwc_unlocked` is a non-thread-safe version of `fgetwc`. `fgetwc_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `fgetwc_unlocked` is equivalent to `fgetwc`.

The getwc and getwc_unlocked functions or macros functions identically to fgetwc and fgetwc_unlocked. It may be implemented as a macro, and may evaluate its argument more than once. There is no reason ever to use it.

_fgetwc_r, _getwc_r, _fgetwc_unlocked_r, and _getwc_unlocked_r are simply reentrant versions of the above functions that are passed the additional reentrant structure pointer argument: *ptr*.

Returns
The next wide character cast to wint_t, unless there is no more data, or the host system reports a read error; in either of these situations, fgetwc and getwc return WEOF.

You can distinguish the two situations that cause an EOF result by using the ferror and feof functions.

Portability
fgetwc and getwc are required by C99 and POSIX.1-2001.

fgetwc_unlocked and getwc_unlocked are GNU extensions.

4.14 fgetws, fgetws_unlocked—get wide character string from a file or stream

Synopsis

```
#include <wchar.h>
wchar_t *fgetws(wchar_t *__restrict ws, int n,
    FILE *__restrict fp);

#define _GNU_SOURCE
#include <wchar.h>
wchar_t *fgetws_unlocked(wchar_t *__restrict ws, int n,
    FILE *__restrict fp);

#include <wchar.h>
wchar_t *_fgetws_r(struct _reent *ptr, wchar_t *ws,
    int n, FILE *fp);

#include <wchar.h>
wchar_t *_fgetws_unlocked_r(struct _reent *ptr, wchar_t *ws,
    int n, FILE *fp);
```

Description

Reads at most *n-1* wide characters from *fp* until a newline is found. The wide characters including to the newline are stored in *ws*. The buffer is terminated with a 0.

`fgetws_unlocked` is a non-thread-safe version of `fgetws`. `fgetws_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `fgetws_unlocked` is equivalent to `fgetws`.

The `_fgetws_r` and `_fgetws_unlocked_r` functions are simply reentrant version of the above and are passed an additional reentrancy structure pointer: *ptr*.

Returns

`fgetws` returns the buffer passed to it, with the data filled in. If end of file occurs with some data already accumulated, the data is returned with no other indication. If no data are read, NULL is returned instead.

Portability

`fgetws` is required by C99 and POSIX.1-2001.

`fgetws_unlocked` is a GNU extension.

4.15 fileno, fileno_unlocked—return file descriptor associated with stream

Synopsis

```
#include <stdio.h>
int fileno(FILE *fp);

#define _BSD_SOURCE
#include <stdio.h>
int fileno_unlocked(FILE *fp);
```

Description

You can use `fileno` to return the file descriptor identified by *fp*.

`fileno_unlocked` is a non-thread-safe version of `fileno`. `fileno_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `fileno_unlocked` is equivalent to `fileno`.

Returns

`fileno` returns a non-negative integer when successful. If *fp* is not an open stream, `fileno` returns -1.

Portability

`fileno` is not part of ANSI C. POSIX requires `fileno`.

`fileno_unlocked` is a BSD extension also provided by GNU libc.

Supporting OS subroutines required: none.

4.16 fmemopen—open a stream around a fixed-length string

Synopsis

```
#include <stdio.h>
FILE *fmemopen(void *restrict buf, size_t size,
    const char *restrict mode);
```

Description

`fmemopen` creates a seekable `FILE` stream that wraps a fixed-length buffer of *size* bytes starting at *buf*. The stream is opened with *mode* treated as in `fopen`, where append mode starts writing at the first NUL byte. If *buf* is NULL, then *size* bytes are automatically provided as if by `malloc`, with the initial size of 0, and *mode* must contain `+` so that data can be read after it is written.

The stream maintains a current position, which moves according to bytes read or written, and which can be one past the end of the array. The stream also maintains a current file size, which is never greater than *size*. If *mode* starts with `r`, the position starts at `0`, and file size starts at *size* if *buf* was provided. If *mode* starts with `w`, the position and file size start at `0`, and if *buf* was provided, the first byte is set to NUL. If *mode* starts with `a`, the position and file size start at the location of the first NUL byte, or else *size* if *buf* was provided.

When reading, NUL bytes have no significance, and reads cannot exceed the current file size. When writing, the file size can increase up to *size* as needed, and NUL bytes may be embedded in the stream (see `open_memstream` for an alternative that automatically enlarges the buffer). When the stream is flushed or closed after a write that changed the file size, a NUL byte is written at the current position if there is still room; if the stream is not also open for reading, a NUL byte is additionally written at the last byte of *buf* when the stream has exceeded *size*, so that a write-only *buf* is always NUL-terminated when the stream is flushed or closed (and the initial *size* should take this into account). It is not possible to seek outside the bounds of *size*. A NUL byte written during a flush is restored to its previous value when seeking elsewhere in the string.

Returns

The return value is an open FILE pointer on success. On error, `NULL` is returned, and `errno` will be set to EINVAL if *size* is zero or *mode* is invalid, ENOMEM if *buf* was NULL and memory could not be allocated, or EMFILE if too many streams are already open.

Portability

This function is being added to POSIX 200x, but is not in POSIX 2001.

Supporting OS subroutines required: `sbrk`.

4.17 fopen—open a file

Synopsis

```
#include <stdio.h>
FILE *fopen(const char *file, const char *mode);

FILE *_fopen_r(struct _reent *reent,
    const char *file, const char *mode);
```

Description

`fopen` initializes the data structures needed to read or write a file. Specify the file's name as the string at *file*, and the kind of access you need to the file with the string at *mode*.

The alternate function `_fopen_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Three fundamental kinds of access are available: read, write, and append. `*mode` must begin with one of the three characters ‘`r`’, ‘`w`’, or ‘`a`’, to select one of these:

`r` Open the file for reading; the operation will fail if the file does not exist, or if the host system does not permit you to read it.

`w` Open the file for writing *from the beginning* of the file: effectively, this always creates a new file. If the file whose name you specified already existed, its old contents are discarded.

`a` Open the file for appending data, that is writing from the end of file. When you open a file this way, all data always goes to the current end of file; you cannot change this using `fseek`.

Some host systems distinguish between “binary” and “text” files. Such systems may perform data transformations on data written to, or read from, files opened as “text”. If your system is one of these, then you can append a ‘`b`’ to any of the three modes above, to specify that you are opening the file as a binary file (the default is to open the file as a text file).

‘`rb`’, then, means “read binary”; ‘`wb`’, “write binary”; and ‘`ab`’, “append binary”.

To make C programs more portable, the ‘`b`’ is accepted on all systems, whether or not it makes a difference.

Finally, you might need to both read and write from the same file. You can also append a ‘`+`’ to any of the three modes, to permit this. (If you want to append both ‘`b`’ and ‘`+`’, you can do it in either order: for example, `"rb+"` means the same thing as `"r+b"` when used as a mode string.)

Use `"r+"` (or `"rb+"`) to permit reading and writing anywhere in an existing file, without discarding any data; `"w+"` (or `"wb+"`) to create a new file (or begin by discarding all data from an old one) that permits reading and writing anywhere in it; and `"a+"` (or `"ab+"`) to permit reading anywhere in an existing file, but writing only at the end.

Returns

`fopen` returns a file pointer which you can use for other file operations, unless the file you requested could not be opened; in that situation, the result is `NULL`. If the reason for failure was an invalid string at *mode*, `errno` is set to `EINVAL`.

Portability
`fopen` is required by ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `open`, `read`, `sbrk`, `write`.

4.18 fopencookie—open a stream with custom callbacks

Synopsis

```
#include <stdio.h>
FILE *fopencookie(const void *cookie, const char *mode,
    cookie_io_functions_t functions);
```

Description

`fopencookie` creates a `FILE` stream where I/O is performed using custom callbacks. The callbacks are registered via the structure:

typedef ssize_t (*cookie_read_function_t)(void *_cookie, char *_buf, size_t _n); typedef ssize_t (*cookie_write_function_t)(void *_cookie, const char *_buf, size_t _n); typedef int (*cookie_seek_function_t)(void *_cookie, off_t *_off, int _whence); typedef int (*cookie_close_function_t)(void *_cookie);

```
typedef struct
{
cookie_read_function_t *read;
cookie_write_function_t *write;
cookie_seek_function_t *seek;
cookie_close_function_t *close;
} cookie_io_functions_t;
```

The stream is opened with *mode* treated as in `fopen`. The callbacks *functions.read* and *functions.write* may only be NULL when *mode* does not require them.

functions.read should return -1 on failure, or else the number of bytes read (0 on EOF). It is similar to `read`, except that *cookie* will be passed as the first argument.

functions.write should return -1 on failure, or else the number of bytes written. It is similar to `write`, except that *cookie* will be passed as the first argument.

functions.seek should return -1 on failure, and 0 on success, with *_off* set to the current file position. It is a cross between `lseek` and `fseek`, with the *_whence* argument interpreted in the same manner. A NULL *functions.seek* makes the stream behave similarly to a pipe in relation to stdio functions that require positioning.

functions.close should return -1 on failure, or 0 on success. It is similar to `close`, except that *cookie* will be passed as the first argument. A NULL *functions.close* merely flushes all data then lets `fclose` succeed. A failed close will still invalidate the stream.

Read and write I/O functions are allowed to change the underlying buffer on fully buffered or line buffered streams by calling `setvbuf`. They are also not required to completely fill or empty the buffer. They are not, however, allowed to change streams from unbuffered to buffered or to change the state of the line buffering flag. They must also be prepared to have read or write calls occur on buffers other than the one most recently specified.

Returns

The return value is an open FILE pointer on success. On error, `NULL` is returned, and `errno` will be set to EINVAL if a function pointer is missing or *mode* is invalid, ENOMEM if the stream cannot be created, or EMFILE if too many streams are already open.

Portability
This function is a newlib extension, copying the prototype from Linux. It is not portable. See also the `funopen` interface from BSD.

Supporting OS subroutines required: `sbrk`.

4.19 fpurge—discard pending file I/O

Synopsis

```
#include <stdio.h>
int fpurge(FILE *fp);

int _fpurge_r(struct _reent *reent, FILE *fp);

#include <stdio.h>
#include <stdio_ext.h>
void  __fpurge(FILE *fp);
```

Description

Use `fpurge` to clear all buffers of the given stream. For output streams, this discards data not yet written to disk. For input streams, this discards any data from `ungetc` and any data retrieved from disk but not yet read via `getc`. This is more severe than `fflush`, and generally is only needed when manually altering the underlying file descriptor of a stream.

`__fpurge` behaves exactly like `fpurge` but does not return a value.

The alternate function `_fpurge_r` is a reentrant version, where the extra argument *reent* is a pointer to a reentrancy structure, and *fp* must not be NULL.

Returns

`fpurge` returns `0` unless *fp* is not valid, in which case it returns `EOF` and sets `errno`.

Portability

These functions are not portable to any standard.

No supporting OS subroutines are required.

4.20 fputc, fputc_unlocked—write a character on a stream or file

Synopsis

```
#include <stdio.h>
int fputc(int ch, FILE *fp);

#define _BSD_SOURCE
#include <stdio.h>
int fputc_unlocked(int ch, FILE *fp);

#include <stdio.h>
int _fputc_r(struct _rent *ptr, int ch, FILE *fp);

#include <stdio.h>
int _fputc_unlocked_r(struct _rent *ptr, int ch, FILE *fp);
```

Description

`fputc` converts the argument *ch* from an `int` to an `unsigned char`, then writes it to the file or stream identified by *fp*.

If the file was opened with append mode (or if the stream cannot support positioning), then the new character goes at the end of the file or stream. Otherwise, the new character is written at the current value of the position indicator, and the position indicator oadvances by one.

For a macro version of this function, see `putc`.

`fputc_unlocked` is a non-thread-safe version of `fputc`. `fputc_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `fputc_unlocked` is equivalent to `fputc`.

The `_fputc_r` and `_fputc_unlocked_r` functions are simply reentrant versions of the above that take an additional reentrant structure argument: *ptr*.

Returns

If successful, `fputc` returns its argument *ch*. If an error intervenes, the result is `EOF`. You can use '`ferror(fp)`' to query for errors.

Portability

`fputc` is required by ANSI C.

`fputc_unlocked` is a BSD extension also provided by GNU libc.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.21 fputs, fputs_unlocked—write a character string in a file or stream

Synopsis

```
#include <stdio.h>
int fputs(const char *restrict s, FILE *restrict fp);

#define _GNU_SOURCE
#include <stdio.h>
int fputs_unlocked(const char *restrict s, FILE *restrict fp);

#include <stdio.h>
int _fputs_r(struct _reent *ptr, const char *restrict s, FILE *restrict fp);

#include <stdio.h>
int _fputs_unlocked_r(struct _reent *ptr, const char *restrict s, FILE *restrict fp);
```

Description

`fputs` writes the string at *s* (but without the trailing null) to the file or stream identified by *fp*.

`fputs_unlocked` is a non-thread-safe version of `fputs`. `fputs_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `fputs_unlocked` is equivalent to `fputs`.

`_fputs_r` and `_fputs_unlocked_r` are simply reentrant versions of the above that take an additional reentrant struct pointer argument: *ptr*.

Returns

If successful, the result is `0`; otherwise, the result is `EOF`.

Portability

ANSI C requires `fputs`, but does not specify that the result on success must be `0`; any non-negative value is permitted.

`fputs_unlocked` is a GNU extension.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.22 fputwc, putwc, fputwc_unlocked, putwc_unlocked—write a wide character on a stream or file

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t fputwc(wchar_t wc, FILE *fp);

#define _GNU_SOURCE
#include <stdio.h>
#include <wchar.h>
wint_t fputwc_unlocked(wchar_t wc, FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t _fputwc_r(struct _reent *ptr, wchar_t wc, FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t _fputwc_unlocked_r(struct _reent *ptr, wchar_t wc, FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t putwc(wchar_t wc, FILE *fp);

#define _GNU_SOURCE
#include <stdio.h>
#include <wchar.h>
wint_t putwc_unlocked(wchar_t wc, FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t _putwc_r(struct _reent *ptr, wchar_t wc, FILE *fp);

#include <stdio.h>
#include <wchar.h>
wint_t _putwc_unlocked_r(struct _reent *ptr, wchar_t wc, FILE *fp);
```

Description

`fputwc` writes the wide character argument *wc* to the file or stream identified by *fp*.

If the file was opened with append mode (or if the stream cannot support positioning), then the new wide character goes at the end of the file or stream. Otherwise, the new wide character is written at the current value of the position indicator, and the position indicator oadvances by one.

`fputwc_unlocked` is a non-thread-safe version of `fputwc`. `fputwc_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called

while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `fputwc_unlocked` is equivalent to `fputwc`.

The `putwc` and `putwc_unlocked` functions or macros function identically to `fputwc` and `fputwc_unlocked`. They may be implemented as a macro, and may evaluate its argument more than once. There is no reason ever to use them.

The `_fputwc_r`, `_putwc_r`, `_fputwc_unlocked_r`, and `_putwc_unlocked_r` functions are simply reentrant versions of the above that take an additional reentrant structure argument: *ptr*.

Returns
If successful, `fputwc` and `putwc` return their argument *wc*. If an error intervenes, the result is `EOF`. You can use ‘`ferror(fp)`’ to query for errors.

Portability
`fputwc` and `putwc` are required by C99 and POSIX.1-2001.

`fputwc_unlocked` and `putwc_unlocked` are GNU extensions.

4.23 fputws, fputws_unlocked—write a wide character string in a file or stream

Synopsis

```
#include <wchar.h>
int fputws(const wchar_t *__restrict ws, FILE *__restrict fp);

#define _GNU_SOURCE
#include <wchar.h>
int fputws_unlocked(const wchar_t *__restrict ws, FILE *__restrict fp);

#include <wchar.h>
int _fputws_r(struct _reent *ptr, const wchar_t *ws,
    FILE *fp);

#include <wchar.h>
int _fputws_unlocked_r(struct _reent *ptr, const wchar_t *ws,
    FILE *fp);
```

Description

`fputws` writes the wide character string at *ws* (but without the trailing null) to the file or stream identified by *fp*.

`fputws_unlocked` is a non-thread-safe version of `fputws`. `fputws_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `fputws_unlocked` is equivalent to `fputws`.

`_fputws_r` and `_fputws_unlocked_r` are simply reentrant versions of the above that take an additional reentrant struct pointer argument: *ptr*.

Returns

If successful, the result is a non-negative integer; otherwise, the result is `-1` to indicate an error.

Portability

`fputws` is required by C99 and POSIX.1-2001.

`fputws_unlocked` is a GNU extension.

4.24 fread, fread_unlocked—read array elements from a file

Synopsis

```
#include <stdio.h>
size_t fread(void *restrict buf, size_t size, size_t count,
    FILE *restrict fp);

#define _BSD_SOURCE
#include <stdio.h>
size_t fread_unlocked(void *restrict buf, size_t size, size_t count,
    FILE *restrict fp);

#include <stdio.h>
size_t _fread_r(struct _reent *ptr, void *restrict buf,
    size_t size, size_t count, FILE *restrict fp);

#include <stdio.h>
size_t _fread_unlocked_r(struct _reent *ptr, void *restrict buf,
    size_t size, size_t count, FILE *restrict fp);
```

Description

`fread` attempts to copy, from the file or stream identified by *fp*, *count* elements (each of size *size*) into memory, starting at *buf*. `fread` may copy fewer elements than *count* if an error, or end of file, intervenes.

`fread` also advances the file position indicator (if any) for *fp* by the number of *characters* actually read.

`fread_unlocked` is a non-thread-safe version of `fread`. `fread_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `fread_unlocked` is equivalent to `fread`.

`_fread_r` and `_fread_unlocked_r` are simply reentrant versions of the above that take an additional reentrant structure pointer argument: *ptr*.

Returns

The result of `fread` is the number of elements it succeeded in reading.

Portability

ANSI C requires `fread`.

`fread_unlocked` is a BSD extension also provided by GNU libc.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.25 `freopen`—open a file using an existing file descriptor

Synopsis

```
#include <stdio.h>
FILE *freopen(const char *restrict file, const char *restrict mode,
    FILE *restrict fp);
FILE *_freopen_r(struct _reent *ptr, const char *restrict file,
    const char *restrict mode, FILE *restrict fp);
```

Description

Use this variant of `fopen` if you wish to specify a particular file descriptor *fp* (notably `stdin`, `stdout`, or `stderr`) for the file.

If *fp* was associated with another file or stream, `freopen` closes that other file or stream (but ignores any errors while closing it).

file and *mode* are used just as in `fopen`.

If *file* is `NULL`, the underlying stream is modified rather than closed. The file cannot be given a more permissive access mode (for example, a *mode* of "w" will fail on a read-only file descriptor), but can change status such as append or binary mode. If modification is not possible, failure occurs.

Returns

If successful, the result is the same as the argument *fp*. If the file cannot be opened as specified, the result is `NULL`.

Portability

ANSI C requires `freopen`.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `open`, `read`, `sbrk`, `write`.

4.26 fseek, fseeko—set file position

Synopsis

```
#include <stdio.h>
int fseek(FILE *fp, long offset, int whence);
int fseeko(FILE *fp, off_t offset, int whence);
int _fseek_r(struct _reent *ptr, FILE *fp,
    long offset, int whence);
int _fseeko_r(struct _reent *ptr, FILE *fp,
    off_t offset, int whence);
```

Description

Objects of type FILE can have a “position” that records how much of the file your program has already read. Many of the stdio functions depend on this position, and many change it as a side effect.

You can use fseek/fseeko to set the position for the file identified by *fp*. The value of *offset* determines the new position, in one of three ways selected by the value of *whence* (defined as macros in ‘stdio.h’):

SEEK_SET—*offset* is the absolute file position (an offset from the beginning of the file) desired. *offset* must be positive.

SEEK_CUR—*offset* is relative to the current file position. *offset* can meaningfully be either positive or negative.

SEEK_END—*offset* is relative to the current end of file. *offset* can meaningfully be either positive (to increase the size of the file) or negative.

See ftell/ftello to determine the current file position.

Returns

fseek/fseeko return 0 when successful. On failure, the result is EOF. The reason for failure is indicated in errno: either ESPIPE (the stream identified by *fp* doesn’t support repositioning) or EINVAL (invalid file position).

Portability

ANSI C requires fseek.

fseeko is defined by the Single Unix specification.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.

4.27 __fsetlocking—set or query locking mode on FILE stream

Synopsis

```
#include <stdio.h>
#include <stdio_ext.h>
int __fsetlocking(FILE *fp, int type);
```

Description

This function sets how the stdio functions handle locking of FILE *fp*. The following values describe *type*:

`FSETLOCKING_INTERNAL` is the default state, where stdio functions automatically lock and unlock the stream.

`FSETLOCKING_BYCALLER` means that automatic locking in stdio functions is disabled. Applications which set this take all responsibility for file locking themselves.

`FSETLOCKING_QUERY` returns the current locking mode without changing it.

Returns

`__fsetlocking` returns the current locking mode of *fp*.

Portability

This function originates from Solaris and is also provided by GNU libc.

No supporting OS subroutines are required.

4.28 fsetpos—restore position of a stream or file

Synopsis

```
#include <stdio.h>
int fsetpos(FILE *fp, const fpos_t *pos);
int _fsetpos_r(struct _reent *ptr, FILE *fp,
    const fpos_t *pos);
```

Description

Objects of type FILE can have a “position” that records how much of the file your program has already read. Many of the stdio functions depend on this position, and many change it as a side effect.

You can use fsetpos to return the file identified by *fp* to a previous position **pos* (after first recording it with fgetpos).

See fseek for a similar facility.

Returns

fgetpos returns 0 when successful. If fgetpos fails, the result is 1. The reason for failure is indicated in errno: either ESPIPE (the stream identified by *fp* doesn't support repositioning) or EINVAL (invalid file position).

Portability

ANSI C requires fsetpos, but does not specify the nature of **pos* beyond identifying it as written by fgetpos.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.

4.29 `ftell`, `ftello`—return position in a stream or file

Synopsis

```
#include <stdio.h>
long ftell(FILE *fp);
off_t ftello(FILE *fp);
long _ftell_r(struct _reent *ptr, FILE *fp);
off_t _ftello_r(struct _reent *ptr, FILE *fp);
```

Description

Objects of type `FILE` can have a "position" that records how much of the file your program has already read. Many of the `stdio` functions depend on this position, and many change it as a side effect.

The result of `ftell`/`ftello` is the current position for a file identified by *fp*. If you record this result, you can later use it with `fseek`/`fseeko` to return the file to this position. The difference between `ftell` and `ftello` is that `ftell` returns `long` and `ftello` returns `off_t`.

In the current implementation, `ftell`/`ftello` simply uses a character count to represent the file position; this is the same number that would be recorded by `fgetpos`.

Returns

`ftell`/`ftello` return the file position, if possible. If they cannot do this, they return `-1L`. Failure occurs on streams that do not support positioning; the global `errno` indicates this condition with the value `ESPIPE`.

Portability

`ftell` is required by the ANSI C standard, but the meaning of its result (when successful) is not specified beyond requiring that it be acceptable as an argument to `fseek`. In particular, other conforming C implementations may return a different result from `ftell` than what `fgetpos` records.

`ftello` is defined by the Single Unix specification.

No supporting OS subroutines are required.

4.30 funopen, fropen, fwopen—open a stream with custom callbacks

Synopsis

```
#include <stdio.h>
FILE *funopen(const void *cookie,
    int (*readfn) (void *cookie, char *buf, int n),
    int (*writefn) (void *cookie, const char *buf, int n),
    fpos_t (*seekfn) (void *cookie, fpos_t off, int whence),
    int (*closefn) (void *cookie));
FILE *fropen(const void *cookie,
    int (*readfn) (void *cookie, char *buf, int n));
FILE *fwopen(const void *cookie,
    int (*writefn) (void *cookie, const char *buf, int n));
```

Description

`funopen` creates a `FILE` stream where I/O is performed using custom callbacks. At least one of *readfn* and *writefn* must be provided, which determines whether the stream behaves with mode `<"r">`, `<"w">`, or `<"r+">`.

readfn should return -1 on failure, or else the number of bytes read (0 on EOF). It is similar to `read`, except that `<int>` rather than `<size_t>` bounds a transaction size, and *cookie* will be passed as the first argument. A NULL *readfn* makes attempts to read the stream fail.

writefn should return -1 on failure, or else the number of bytes written. It is similar to `write`, except that `<int>` rather than `<size_t>` bounds a transaction size, and *cookie* will be passed as the first argument. A NULL *writefn* makes attempts to write the stream fail.

seekfn should return (fpos_t)-1 on failure, or else the current file position. It is similar to `lseek`, except that *cookie* will be passed as the first argument. A NULL *seekfn* makes the stream behave similarly to a pipe in relation to stdio functions that require positioning. This implementation assumes fpos_t and off_t are the same type.

closefn should return -1 on failure, or 0 on success. It is similar to `close`, except that *cookie* will be passed as the first argument. A NULL *closefn* merely flushes all data then lets `fclose` succeed. A failed close will still invalidate the stream.

Read and write I/O functions are allowed to change the underlying buffer on fully buffered or line buffered streams by calling `setvbuf`. They are also not required to completely fill or empty the buffer. They are not, however, allowed to change streams from unbuffered to buffered or to change the state of the line buffering flag. They must also be prepared to have read or write calls occur on buffers other than the one most recently specified.

The functions `fropen` and `fwopen` are convenience macros around `funopen` that only use the specified callback.

Returns

The return value is an open FILE pointer on success. On error, `NULL` is returned, and `errno` will be set to EINVAL if a function pointer is missing, ENOMEM if the stream cannot be created, or EMFILE if too many streams are already open.

Portability
This function is a newlib extension, copying the prototype from BSD. It is not portable. See also the `fopencookie` interface from Linux.

Supporting OS subroutines required: `sbrk`.

4.31 fwide—set and determine the orientation of a FILE stream

Synopsis

```
#include <wchar.h>
int fwide(FILE *fp, int mode);

int _fwide_r(struct _reent *ptr, FILE *fp, int mode);
```

Description

When *mode* is zero, the `fwide` function determines the current orientation of *fp*. It returns a value > 0 if *fp* is wide-character oriented, i.e. if wide character I/O is permitted but char I/O is disallowed. It returns a value < 0 if *fp* is byte oriented, i.e. if char I/O is permitted but wide character I/O is disallowed. It returns zero if *fp* has no orientation yet; in this case the next I/O operation might change the orientation (to byte oriented if it is a char I/O operation, or to wide-character oriented if it is a wide character I/O operation).

Once a stream has an orientation, it cannot be changed and persists until the stream is closed, unless the stream is re-opened with freopen, which removes the orientation of the stream.

When *mode* is non-zero, the `fwide` function first attempts to set *fp*'s orientation (to wide-character oriented if *mode* > 0, or to byte oriented if *mode* < 0). It then returns a value denoting the current orientation, as above.

Returns

The `fwide` function returns *fp*'s orientation, after possibly changing it. A return value > 0 means wide-character oriented. A return value < 0 means byte oriented. A return value of zero means undecided.

Portability

C99, POSIX.1-2001.

4.32 fwrite, fwrite_unlocked—write array elements

Synopsis

```
#include <stdio.h>
size_t fwrite(const void *restrict buf, size_t size,
    size_t count, FILE *restrict fp);

#define _BSD_SOURCE
#include <stdio.h>
size_t fwrite_unlocked(const void *restrict buf, size_t size,
    size_t count, FILE *restrict fp);

#include <stdio.h>
size_t _fwrite_r(struct _reent *ptr, const void *restrict buf, size_t size,
    size_t count, FILE *restrict fp);

#include <stdio.h>
size_t _fwrite_unlocked_r(struct _reent *ptr, const void *restrict buf, size_t size,
    size_t count, FILE *restrict fp);
```

Description

`fwrite` attempts to copy, starting from the memory location *buf*, *count* elements (each of size *size*) into the file or stream identified by *fp*. `fwrite` may copy fewer elements than *count* if an error intervenes.

`fwrite` also advances the file position indicator (if any) for *fp* by the number of *characters* actually written.

`fwrite_unlocked` is a non-thread-safe version of `fwrite`. `fwrite_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `fwrite_unlocked` is equivalent to `fwrite`.

`_fwrite_r` and `_fwrite_unlocked_r` are simply reentrant versions of the above that take an additional reentrant structure argument: *ptr*.

Returns

If `fwrite` succeeds in writing all the elements you specify, the result is the same as the argument *count*. In any event, the result is the number of complete elements that `fwrite` copied to the file.

Portability

ANSI C requires `fwrite`.

`fwrite_unlocked` is a BSD extension also provided by GNU libc.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.33 getc—read a character (macro)

Synopsis

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

#include <stdio.h>
int _getc_r(struct _reent *ptr, FILE *fp);
```

Description

getc is a macro, defined in stdio.h. You can use getc to get the next single character from the file or stream identified by *fp*. As a side effect, getc advances the file's current position indicator.

For a subroutine version of this macro, see fgetc.

The _getc_r function is simply the reentrant version of getc which passes an additional reentrancy structure pointer argument: *ptr*.

Returns

The next character (read as an unsigned char, and cast to int), unless there is no more data, or the host system reports a read error; in either of these situations, getc returns EOF.

You can distinguish the two situations that cause an EOF result by using the ferror and feof functions.

Portability

ANSI C requires getc; it suggests, but does not require, that getc be implemented as a macro. The standard explicitly permits macro implementations of getc to use the argument more than once; therefore, in a portable program, you should not use an expression with side effects as the getc argument.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.

4.34 getc_unlocked—non-thread-safe version of getc (macro)

Synopsis

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

#include <stdio.h>
int _getc_unlocked_r(FILE *fp);
```

Description

`getc_unlocked` is a non-thread-safe version of `getc` declared in `stdio.h`. `getc_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `getc_unlocked` is equivalent to `getc`.

The `_getc_unlocked_r` function is simply the reentrant version of `get_unlocked` which passes an additional reentrancy structure pointer argument: *ptr*.

Returns

See `getc`.

Portability

POSIX 1003.1 requires `getc_unlocked`. `getc_unlocked` may be implemented as a macro, so arguments should not have side-effects.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.35 getchar—read a character (macro)

Synopsis

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

int _getchar_r(struct _reent *reent);
```

Description

`getchar` is a macro, defined in `stdio.h`. You can use `getchar` to get the next single character from the standard input stream. As a side effect, `getchar` advances the standard input's current position indicator.

The alternate function `_getchar_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

The next character (read as an `unsigned char`, and cast to `int`), unless there is no more data, or the host system reports a read error; in either of these situations, `getchar` returns `EOF`.

You can distinguish the two situations that cause an `EOF` result by using ‘`ferror(stdin)`’ and ‘`feof(stdin)`’.

Portability

ANSI C requires `getchar`; it suggests, but does not require, that `getchar` be implemented as a macro.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.36 getchar_unlocked—non-thread-safe version of getchar (macro)

Synopsis

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

#include <stdio.h>
int _getchar_unlocked_r(struct _reent *ptr);
```

Description

`getchar_unlocked` is a non-thread-safe version of `getchar` declared in `stdio.h`. `getchar_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `getchar_unlocked` is equivalent to `getchar`.

The `_getchar_unlocked_r` function is simply the reentrant version of `getchar_unlocked` which passes an addtional reentrancy structure pointer argument: *ptr*.

Returns

See `getchar`.

Portability

POSIX 1003.1 requires `getchar_unlocked`. `getchar_unlocked` may be implemented as a macro.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.37 getdelim—read a line up to a specified line delimiter

Synopsis

```
#include <stdio.h>
int getdelim(char **bufptr, size_t *n,
    int delim, FILE *fp);
```

Description

`getdelim` reads a file *fp* up to and possibly including a specified delimiter *delim*. The line is read into a buffer pointed to by *bufptr* and designated with size **n*. If the buffer is not large enough, it will be dynamically grown by `getdelim`. As the buffer is grown, the pointer to the size *n* will be updated.

Returns

`getdelim` returns `-1` if no characters were successfully read; otherwise, it returns the number of bytes successfully read. At end of file, the result is nonzero.

Portability

`getdelim` is a glibc extension.

No supporting OS subroutines are directly required.

4.38 `getline`—read a line from a file

Synopsis

```
#include <stdio.h>
ssize_t getline(char **bufptr, size_t *n, FILE *fp);
```

Description

`getline` reads a file *fp* up to and possibly including the newline character. The line is read into a buffer pointed to by *bufptr* and designated with size **n*. If the buffer is not large enough, it will be dynamically grown by `getdelim`. As the buffer is grown, the pointer to the size *n* will be updated.

`getline` is equivalent to getdelim(bufptr, n, '\n', fp);

Returns

`getline` returns `-1` if no characters were successfully read, otherwise, it returns the number of bytes successfully read. at end of file, the result is nonzero.

Portability

`getline` is a glibc extension.

No supporting OS subroutines are directly required.

4.39 gets—get character string (obsolete, use fgets instead)

Synopsis

```
#include <stdio.h>

char *gets(char *buf);

char *_gets_r(struct _reent *reent, char *buf);
```

Description

Reads characters from standard input until a newline is found. The characters up to the newline are stored in *buf*. The newline is discarded, and the buffer is terminated with a 0.

This is a *dangerous* function, as it has no way of checking the amount of space available in *buf*. One of the attacks used by the Internet Worm of 1988 used this to overrun a buffer allocated on the stack of the finger daemon and overwrite the return address, causing the daemon to execute code downloaded into it over the connection.

The alternate function `_gets_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`gets` returns the buffer passed to it, with the data filled in. If end of file occurs with some data already accumulated, the data is returned with no other indication. If end of file occurs with no data in the buffer, NULL is returned.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.40 getw—read a word (int)

Synopsis

```
#include <stdio.h>
int getw(FILE *fp);
```

Description

`getw` is a function, defined in `stdio.h`. You can use `getw` to get the next word from the file or stream identified by *fp*. As a side effect, `getw` advances the file's current position indicator.

Returns

The next word (read as an `int`), unless there is no more data or the host system reports a read error; in either of these situations, `getw` returns `EOF`. Since `EOF` is a valid `int`, you must use `ferror` or `feof` to distinguish these situations.

Portability

`getw` is a remnant of K&R C; it is not part of any ISO C Standard. `fread` should be used instead. In fact, this implementation of `getw` is based upon `fread`.

Supporting OS subroutines required: `fread`.

4.41 getwchar, getwchar_unlocked—read a wide character from standard input

Synopsis

```
#include <wchar.h>
wint_t getwchar(void);

#define _GNU_SOURCE
#include <wchar.h>
wint_t getwchar_unlocked(void);

#include <wchar.h>
wint_t _getwchar_r(struct _reent *reent);

#include <wchar.h>
wint_t _getwchar_unlocked_r(struct _reent *reent);
```

Description

`getwchar` function or macro is the wide character equivalent of the `getchar` function. You can use `getwchar` to get the next wide character from the standard input stream. As a side effect, `getwchar` advances the standard input's current position indicator.

`getwchar_unlocked` is a non-thread-safe version of `getwchar`. `getwchar_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `getwchar_unlocked` is equivalent to `getwchar`.

The alternate functions `_getwchar_r` and `_getwchar_unlocked_r` are reentrant versions of the above. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

The next wide character cast to `wint_t`, unless there is no more data, or the host system reports a read error; in either of these situations, `getwchar` returns `WEOF`.

You can distinguish the two situations that cause an `WEOF` result by using ‘`ferror(stdin)`’ and ‘`feof(stdin)`’.

Portability

`getwchar` is required by C99.

`getwchar_unlocked` is a GNU extension.

4.42 mktemp, mkstemp, mkostemp, mkstemps,

Synopsis

```
#include <stdlib.h>
char *mktemp(char *path);
char *mkdtemp(char *path);
int mkstemp(char *path);
int mkstemps(char *path, int suffixlen);
int mkostemp(char *path, int flags);
int mkostemps(char *path, int suffixlen, int flags);

char *_mktemp_r(struct _reent *reent, char *path);
char *_mkdtemp_r(struct _reent *reent, char *path);
int *_mkstemp_r(struct _reent *reent, char *path);
int *_mkstemps_r(struct _reent *reent, char *path, int len);
int *_mkostemp_r(struct _reent *reent, char *path,
    int flags);
int *_mkostemps_r(struct _reent *reent, char *path, int len,
    int flags);
```

Description

`mktemp`, `mkstemp`, and `mkstemps` attempt to generate a file name that is not yet in use for any existing file. `mkstemp` and `mkstemps` create the file and open it for reading and writing; `mktemp` simply generates the file name (making `mktemp` a security risk). `mkostemp` and `mkostemps` allow the addition of other `open` flags, such as `O_CLOEXEC`, `O_APPEND`, or `O_SYNC`. On platforms with a separate text mode, `mkstemp` forces `O_BINARY`, while `mkostemp` allows the choice between `O_BINARY`, `O_TEXT`, or 0 for default. `mkdtemp` attempts to create a directory instead of a file, with a permissions mask of 0700.

You supply a simple pattern for the generated file name, as the string at *path*. The pattern should be a valid filename (including path information if you wish) ending with at least six '`X`' characters. The generated filename will match the leading part of the name you supply, with the trailing '`X`' characters replaced by some combination of digits and letters. With `mkstemps`, the '`X`' characters end *suffixlen* bytes before the end of the string.

The alternate functions `_mktemp_r`, `_mkdtemp_r`, `_mkstemp_r`, `_mkostemp_r`, `_mkostemps_r`, and `_mkstemps_r` are reentrant versions. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`mktemp` returns the pointer *path* to the modified string representing an unused filename, unless it could not generate one, or the pattern you provided is not suitable for a filename; in that case, it returns `NULL`. Be aware that there is an inherent race between generating the name and attempting to create a file by that name; you are advised to use `O_EXCL|O_CREAT`.

`mkdtemp` returns the pointer *path* to the modified string if the directory was created, otherwise it returns `NULL`.

`mkstemp`, `mkstemps`, `mkostemp`, and `mkostemps` return a file descriptor to the newly created file, unless it could not generate an unused filename, or the pattern you provided is not suitable for a filename; in that case, it returns `-1`.

Notes
Never use `mktemp`. The generated filenames are easy to guess and there's a race between the test if the file exists and the creation of the file. In combination this makes `mktemp` prone to attacks and using it is a security risk. Whenever possible use `mkstemp` instead. It doesn't suffer the race condition.

Portability
ANSI C does not require either `mktemp` or `mkstemp`; the System V Interface Definition requires `mktemp` as of Issue 2. POSIX 2001 requires `mkstemp`, and POSIX 2008 requires `mkdtemp` while deprecating `mktemp`. `mkstemps`, `mkostemp`, and `mkostemps` are not standardized.

Supporting OS subroutines required: `getpid`, `mkdir`, `open`, `stat`.

4.43 open_memstream, open_wmemstream—open a write stream around an arbitrary-length string

Synopsis

```
#include <stdio.h>
FILE *open_memstream(char **restrict buf,
    size_t *restrict size);

#include <wchar.h>
FILE *open_wmemstream(wchar_t **restrict buf,
    size_t *restrict size);
```

Description

`open_memstream` creates a seekable, byte-oriented `FILE` stream that wraps an arbitrary-length buffer, created as if by `malloc`. The current contents of **buf* are ignored; this implementation uses **size* as a hint of the maximum size expected, but does not fail if the hint was wrong. The parameters *buf* and *size* are later stored through following any call to `fflush` or `fclose`, set to the current address and usable size of the allocated string; although after fflush, the pointer is only valid until another stream operation that results in a write. Behavior is undefined if the user alters either **buf* or **size* prior to `fclose`.

`open_wmemstream` is like `open_memstream` just with the associated stream being wide-oriented. The size set in *size* in subsequent operations is the number of wide characters.

The stream is write-only, since the user can directly read **buf* after a flush; see `fmemopen` for a way to wrap a string with a readable stream. The user is responsible for calling `free` on the final **buf* after `fclose`.

Any time the stream is flushed, a NUL byte is written at the current position (but is not counted in the buffer length), so that the string is always NUL-terminated after at most **size* bytes (or wide characters in case of `open_wmemstream`). However, data previously written beyond the current stream offset is not lost, and the NUL value written during a flush is restored to its previous value when seeking elsewhere in the string.

Returns

The return value is an open FILE pointer on success. On error, `NULL` is returned, and `errno` will be set to EINVAL if *buf* or *size* is NULL, ENOMEM if memory could not be allocated, or EMFILE if too many streams are already open.

Portability

POSIX.1-2008

Supporting OS subroutines required: `sbrk`.

4.44 perror—print an error message on standard error

Synopsis

```
#include <stdio.h>
void perror(char *prefix);

void _perror_r(struct _reent *reent, char *prefix);
```

Description

Use `perror` to print (on standard error) an error message corresponding to the current value of the global variable `errno`. Unless you use `NULL` as the value of the argument *prefix*, the error message will begin with the string at *prefix*, followed by a colon and a space (`: `). The remainder of the error message is one of the strings described for `strerror`.

The alternate function `_perror_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`perror` returns no result.

Portability

ANSI C requires `perror`, but the strings issued vary from one implementation to another.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.45 putc—write a character (macro)

Synopsis

```
#include <stdio.h>
int putc(int ch, FILE *fp);

#include <stdio.h>
int _putc_r(struct _reent *ptr, int ch, FILE *fp);
```

Description

`putc` is a macro, defined in `stdio.h`. `putc` writes the argument *ch* to the file or stream identified by *fp*, after converting it from an `int` to an `unsigned char`.

If the file was opened with append mode (or if the stream cannot support positioning), then the new character goes at the end of the file or stream. Otherwise, the new character is written at the current value of the position indicator, and the position indicator advances by one.

For a subroutine version of this macro, see `fputc`.

The `_putc_r` function is simply the reentrant version of `putc` that takes an additional reentrant structure argument: *ptr*.

Returns

If successful, `putc` returns its argument *ch*. If an error intervenes, the result is `EOF`. You can use '`ferror(fp)`' to query for errors.

Portability

ANSI C requires `putc`; it suggests, but does not require, that `putc` be implemented as a macro. The standard explicitly permits macro implementations of `putc` to use the *fp* argument more than once; therefore, in a portable program, you should not use an expression with side effects as this argument.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.46 putc_unlocked—non-thread-safe version of putc (macro)

Synopsis

```
#include <stdio.h>
int putc_unlocked(int ch, FILE *fp);

#include <stdio.h>
int _putc_unlocked_r(struct _reent *ptr, int ch, FILE *fp);
```

Description

`putc_unlocked` is a non-thread-safe version of `putc` declared in `stdio.h`. `putc_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `putc_unlocked` is equivalent to `putc`.

The function `_putc_unlocked_r` is simply the reentrant version of `putc_unlocked` that takes an additional reentrant structure pointer argument: *ptr*.

Returns

See `putc`.

Portability

POSIX 1003.1 requires `putc_unlocked`. `putc_unlocked` may be implemented as a macro, so arguments should not have side-effects.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.47 putchar—write a character (macro)

Synopsis

```
#include <stdio.h>
int putchar(int ch);

int _putchar_r(struct _reent *reent, int ch);
```

Description

`putchar` is a macro, defined in `stdio.h`. `putchar` writes its argument to the standard output stream, after converting it from an `int` to an `unsigned char`.

The alternate function `_putchar_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

If successful, `putchar` returns its argument *ch*. If an error intervenes, the result is `EOF`. You can use '`ferror(stdin)`' to query for errors.

Portability

ANSI C requires `putchar`; it suggests, but does not require, that `putchar` be implemented as a macro.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.48 putchar_unlocked—non-thread-safe version of putchar (macro)

Synopsis

```
#include <stdio.h>
int putchar_unlocked(int ch);
```

Description

putchar_unlocked is a non-thread-safe version of putchar declared in stdio.h. putchar_unlocked may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then putchar_unlocked is equivalent to putchar.

Returns

See putchar.

Portability

POSIX 1003.1 requires putchar_unlocked. putchar_unlocked may be implemented as a macro.

Supporting OS subroutines required: close, fstat, isatty, lseek, read, sbrk, write.

4.49 puts—write a character string

Synopsis

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

int _puts_r(struct _reent *reent, const char *s);
```

Description

`puts` writes the string at *s* (followed by a newline, instead of the trailing null) to the standard output stream.

The alternate function `_puts_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

If successful, the result is a nonnegative integer; otherwise, the result is `EOF`.

Portability

ANSI C requires `puts`, but does not specify that the result on success must be `0`; any non-negative value is permitted.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.50 putw—write a word (int)

Synopsis

```
#include <stdio.h>
int putw(int w, FILE *fp);
```

Description

putw is a function, defined in stdio.h. You can use putw to write a word to the file or stream identified by *fp*. As a side effect, putw advances the file's current position indicator.

Returns

Zero on success, EOF on failure.

Portability

putw is a remnant of K&R C; it is not part of any ISO C Standard. fwrite should be used instead. In fact, this implementation of putw is based upon fwrite.

Supporting OS subroutines required: fwrite.

4.51 putwchar, putwchar_unlocked—write a wide character to standard output

Synopsis

```
#include <wchar.h>
wint_t putwchar(wchar_t wc);

#include <wchar.h>
wint_t putwchar_unlocked(wchar_t wc);

#include <wchar.h>
wint_t _putwchar_r(struct _reent *reent, wchar_t wc);

#include <wchar.h>
wint_t _putwchar_unlocked_r(struct _reent *reent, wchar_t wc);
```

Description

The `putwchar` function or macro is the wide-character equivalent of the `putchar` function. It writes the wide character wc to stdout.

`putwchar_unlocked` is a non-thread-safe version of `putwchar`. `putwchar_unlocked` may only safely be used within a scope protected by flockfile() (or ftrylockfile()) and funlockfile(). This function may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE *) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions. If threads are disabled, then `putwchar_unlocked` is equivalent to `putwchar`.

The alternate functions `_putwchar_r` and `_putwchar_unlocked_r` are reentrant versions of the above. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

If successful, `putwchar` returns its argument *wc*. If an error intervenes, the result is `EOF`. You can use '`ferror(stdin)`' to query for errors.

Portability

`putwchar` is required by C99.

`putwchar_unlocked` is a GNU extension.

4.52 remove—delete a file’s name

Synopsis

```
#include <stdio.h>
int remove(char *filename);

int _remove_r(struct _reent *reent, char *filename);
```

Description

Use `remove` to dissolve the association between a particular filename (the string at *filename*) and the file it represents. After calling `remove` with a particular filename, you will no longer be able to open the file by that name.

In this implementation, you may use `remove` on an open file without error; existing file descriptors for the file will continue to access the file’s data until the program using them closes the file.

The alternate function `_remove_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

`remove` returns `0` if it succeeds, `-1` if it fails.

Portability

ANSI C requires `remove`, but only specifies that the result on failure be nonzero. The behavior of `remove` when you call it on an open file may vary among implementations.

Supporting OS subroutine required: `unlink`.

4.53 rename—rename a file

Synopsis

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

Description

Use `rename` to establish a new name (the string at *new*) for a file now known by the string at *old*. After a successful `rename`, the file is no longer accessible by the string at *old*.

If `rename` fails, the file named `*old` is unaffected. The conditions for failure depend on the host operating system.

Returns

The result is either `0` (when successful) or `-1` (when the file could not be renamed).

Portability

ANSI C requires `rename`, but only specifies that the result on failure be nonzero. The effects of using the name of an existing file as `*new` may vary from one implementation to another.

Supporting OS subroutines required: `link`, `unlink`, or `rename`.

4.54 rewind—reinitialize a file or stream

Synopsis

```
#include <stdio.h>
void rewind(FILE *fp);
void _rewind_r(struct _reent *ptr, FILE *fp);
```

Description

`rewind` returns the file position indicator (if any) for the file or stream identified by *fp* to the beginning of the file. It also clears any error indicator and flushes any pending output.

Returns

`rewind` does not return a result.

Portability

ANSI C requires `rewind`.

No supporting OS subroutines are required.

4.55 `setbuf`—specify full buffering for a file or stream

Synopsis

```
#include <stdio.h>
void setbuf(FILE *fp, char *buf);
```

Description

`setbuf` specifies that output to the file or stream identified by *fp* should be fully buffered. All output for this file will go to a buffer (of size `BUFSIZ`, specified in '`stdio.h`'). Output will be passed on to the host system only when the buffer is full, or when an input operation intervenes.

You may, if you wish, supply your own buffer by passing a pointer to it as the argument *buf*. It must have size `BUFSIZ`. You can also use `NULL` as the value of *buf*, to signal that the `setbuf` function is to allocate the buffer.

Warnings

You may only use `setbuf` before performing any file operation other than opening the file.

If you supply a non-null *buf*, you must ensure that the associated storage continues to be available until you close the stream identified by *fp*.

Returns

`setbuf` does not return a result.

Portability

Both ANSI C and the System V Interface Definition (Issue 2) require `setbuf`. However, they differ on the meaning of a `NULL` buffer pointer: the SVID issue 2 specification says that a `NULL` buffer pointer requests unbuffered output. For maximum portability, avoid `NULL` buffer pointers.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.56 setbuffer—specify full buffering for a file or stream with size

Synopsis

```
#include <stdio.h>
void setbuffer(FILE *fp, char *buf, int size);
```

Description

`setbuffer` specifies that output to the file or stream identified by *fp* should be fully buffered. All output for this file will go to a buffer (of size *size*). Output will be passed on to the host system only when the buffer is full, or when an input operation intervenes.

You may, if you wish, supply your own buffer by passing a pointer to it as the argument *buf*. It must have size *size*. You can also use `NULL` as the value of *buf*, to signal that the `setbuffer` function is to allocate the buffer.

Warnings

You may only use `setbuffer` before performing any file operation other than opening the file.

If you supply a non-null *buf*, you must ensure that the associated storage continues to be available until you close the stream identified by *fp*.

Returns

`setbuffer` does not return a result.

Portability

This function comes from BSD not ANSI or POSIX.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.57 `setlinebuf`—specify line buffering for a file or stream

Synopsis

```
#include <stdio.h>
void setlinebuf(FILE *fp);
```

Description

`setlinebuf` specifies that output to the file or stream identified by *fp* should be line buffered. This causes the file or stream to pass on output to the host system at every newline, as well as when the buffer is full, or when an input operation intervenes.

Warnings

You may only use `setlinebuf` before performing any file operation other than opening the file.

Returns

`setlinebuf` returns as per setvbuf.

Portability

This function comes from BSD not ANSI or POSIX.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.58 setvbuf—specify file or stream buffering

Synopsis

```
#include <stdio.h>
int setvbuf(FILE *fp, char *buf,
    int mode, size_t size);
```

Description

Use `setvbuf` to specify what kind of buffering you want for the file or stream identified by *fp*, by using one of the following values (from `stdio.h`) as the *mode* argument:

`_IONBF` Do not use a buffer: send output directly to the host system for the file or stream identified by *fp*.

`_IOFBF` Use full output buffering: output will be passed on to the host system only when the buffer is full, or when an input operation intervenes.

`_IOLBF` Use line buffering: pass on output to the host system at every newline, as well as when the buffer is full, or when an input operation intervenes.

Use the *size* argument to specify how large a buffer you wish. You can supply the buffer itself, if you wish, by passing a pointer to a suitable area of memory as *buf*. Otherwise, you may pass `NULL` as the *buf* argument, and `setvbuf` will allocate the buffer.

Warnings

You may only use `setvbuf` before performing any file operation other than opening the file.

If you supply a non-null *buf*, you must ensure that the associated storage continues to be available until you close the stream identified by *fp*.

Returns

A `0` result indicates success, `EOF` failure (invalid *mode* or *size* can cause failure).

Portability

Both ANSI C and the System V Interface Definition (Issue 2) require `setvbuf`. However, they differ on the meaning of a `NULL` buffer pointer: the SVID issue 2 specification says that a `NULL` buffer pointer requests unbuffered output. For maximum portability, avoid `NULL` buffer pointers.

Both specifications describe the result on failure only as a nonzero value.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.59 siprintf, fiprintf, iprintf, sniprintf, asiprintf, asniprintf—format output (integer only)

Synopsis

```
#include <stdio.h>

int iprintf(const char *format, ...);
int fiprintf(FILE *fd, const char *format , ...);
int siprintf(char *str, const char *format, ...);
int sniprintf(char *str, size_t size, const char *format,
    ...);
int asiprintf(char **strp, const char *format, ...);
char *asniprintf(char *str, size_t *size,
    const char *format, ...);

int _iprintf_r(struct _reent *ptr, const char *format, ...);
int _fiprintf_r(struct _reent *ptr, FILE *fd,
    const char *format, ...);
int _siprintf_r(struct _reent *ptr, char *str,
    const char *format, ...);
int _sniprintf_r(struct _reent *ptr, char *str, size_t size,
    const char *format, ...);
int _asiprintf_r(struct _reent *ptr, char **strp,
    const char *format, ...);
char *_asniprintf_r(struct _reent *ptr, char *str,
    size_t *size, const char *format, ...);
```

Description

`iprintf`, `fiprintf`, `siprintf`, `sniprintf`, `asiprintf`, and `asniprintf` are the same as `printf`, `fprintf`, `sprintf`, `snprintf`, `asprintf`, and `asnprintf`, respectively, except that they restrict usage to non-floating-point format specifiers.

`_iprintf_r`, `_fiprintf_r`, `_asiprintf_r`, `_siprintf_r`, `_sniprintf_r`, `_asniprintf_r` are simply reentrant versions of the functions above.

Returns

Similar to `printf`, `fprintf`, `sprintf`, `snprintf`, `asprintf`, and `asnprintf`.

Portability

`iprintf`, `fiprintf`, `siprintf`, `sniprintf`, `asiprintf`, and `asniprintf` are newlib extensions.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.60 siscanf, fiscanf, iscanf—scan and format non-floating input

Synopsis

```
#include <stdio.h>

int iscanf(const char *format, ...);
int fiscanf(FILE *fd, const char *format, ...);
int siscanf(const char *str, const char *format, ...);

int _iscanf_r(struct _reent *ptr, const char *format, ...);
int _fiscanf_r(struct _reent *ptr, FILE *fd,
    const char *format, ...);
int _siscanf_r(struct _reent *ptr, const char *str,
    const char *format, ...);
```

Description

`iscanf`, `fiscanf`, and `siscanf` are the same as `scanf`, `fscanf`, and `sscanf` respectively, only that they restrict the available formats to non-floating-point format specifiers.

The routines `_iscanf_r`, `_fiscanf_r`, and `_siscanf_r` are reentrant versions of `iscanf`, `fiscanf`, and `siscanf` that take an additional first argument pointing to a reentrancy structure.

Returns

`iscanf` returns the number of input fields successfully scanned, converted and stored; the return value does not include scanned fields which were not stored.

If `iscanf` attempts to read at end-of-file, the return value is `EOF`.

If no fields were stored, the return value is `0`.

Portability

`iscanf`, `fiscanf`, and `siscanf` are newlib extensions.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.61 sprintf, fprintf, printf, snprintf, asprintf, asnprintf—format output

Synopsis

```
#include <stdio.h>

int printf(const char *restrict format, ...);
int fprintf(FILE *restrict fd, const char *restrict format, ...);
int sprintf(char *restrict str, const char *restrict format, ...);
int snprintf(char *restrict str, size_t size, const char *restrict format,
    ...);
int asprintf(char **restrict strp, const char *restrict format, ...);
char *asnprintf(char *restrict str, size_t *restrict size, const char *restrict format
    ...);

int _printf_r(struct _reent *ptr, const char *restrict format, ...);
int _fprintf_r(struct _reent *ptr, FILE *restrict fd,
    const char *restrict format, ...);
int _sprintf_r(struct _reent *ptr, char *restrict str,
    const char *restrict format, ...);
int _snprintf_r(struct _reent *ptr, char *restrict str, size_t size,
    const char *restrict format, ...);
int _asprintf_r(struct _reent *ptr, char **restrict strp,
    const char *restrict format, ...);
char *_asnprintf_r(struct _reent *ptr, char *restrict str,
    size_t *restrict size, const char *restrict format, ...);
```

Description

`printf` accepts a series of arguments, applies to each a format specifier from `*format`, and writes the formatted data to `stdout`, without a terminating NUL character. The behavior of `printf` is undefined if there are not enough arguments for the format. `printf` returns when it reaches the end of the format string. If there are more arguments than the format requires, excess arguments are ignored.

`fprintf` is like `printf`, except that output is directed to the stream *fd* rather than `stdout`.

`sprintf` is like `printf`, except that output is directed to the buffer *str*, and a terminating NUL is output. Behavior is undefined if more output is generated than the buffer can hold.

`snprintf` is like `sprintf`, except that output is limited to at most *size* bytes, including the terminating `NUL`. As a special case, if *size* is 0, *str* can be NULL, and `snprintf` merely calculates how many bytes would be printed.

`asprintf` is like `sprintf`, except that the output is stored in a dynamically allocated buffer, *pstr*, which should be freed later with `free`.

`asnprintf` is like `sprintf`, except that the return type is either the original *str* if it was large enough, or a dynamically allocated string if the output exceeds **size*; the length of the result is returned in **size*. When dynamic allocation occurs, the contents of the original *str* may have been modified.

For sprintf, snprintf, and asnprintf, the behavior is undefined if the output *str overlaps with one of the arguments. Behavior is also undefined if the argument for %n within *format overlaps another argument.

format is a pointer to a character string containing two types of objects: ordinary characters (other than %), which are copied unchanged to the output, and conversion specifications, each of which is introduced by %. (To include % in the output, use %% in the format string.) A conversion specification has the following form:

```
%[pos][flags][width][.prec][size]type
```

The fields of the conversion specification have the following meanings:

- *pos*

  Conversions normally consume arguments in the order that they are presented. However, it is possible to consume arguments out of order, and reuse an argument for more than one conversion specification (although the behavior is undefined if the same argument is requested with different types), by specifying *pos*, which is a decimal integer followed by '$'. The integer must be between 1 and <NL_ARGMAX> from limits.h, and if argument %n$ is requested, all earlier arguments must be requested somewhere within *format*. If positional parameters are used, then all conversion specifications except for %% must specify a position. This positional parameters method is a POSIX extension to the C standard definition for the functions.

- *flags*

  flags is an optional sequence of characters which control output justification, numeric signs, decimal points, trailing zeros, and octal and hex prefixes. The flag characters are minus (-), plus (+), space ( ), zero (0), sharp (#), and quote (’). They can appear in any combination, although not all flags can be used for all conversion specification types.

  ’ A POSIX extension to the C standard. However, this implementation presently treats it as a no-op, which is the default behavior for the C locale, anyway. (If it did what it is supposed to, when *type* were i, d, u, f, F, g, or G, the integer portion of the conversion would be formatted with thousands’ grouping wide characters.)

  - The result of the conversion is left justified, and the right is padded with blanks. If you do not use this flag, the result is right justified, and padded on the left.

  + The result of a signed conversion (as determined by *type* of d, i, a, A, e, E, f, F, g, or G) will always begin with a plus or minus sign. (If you do not use this flag, positive values do not begin with a plus sign.)

  " " (space) If the first character of a signed conversion specification is not a sign, or if a signed conversion results in no characters, the result will begin with a space. If the space ( ) flag and the plus (+) flag both appear, the space flag is ignored.

  0 If the *type* character is d, i, o, u, x, X, a, A, e, E, f, F, g, or G: leading zeros are used to pad the field width (following any indication of sign or

base); no spaces are used for padding. If the zero (`0`) and minus (`-`) flags both appear, the zero (`0`) flag will be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision *prec* is specified, the zero (`0`) flag is ignored.

Note that `0` is interpreted as a flag, not as the beginning of a field width.

`#` The result is to be converted to an alternative form, according to the *type* character.

The alternative form output with the # flag depends on the *type* character:

`o` Increases precision to force the first digit of the result to be a zero.

`x` A non-zero result will have a `0x` prefix.

`X` A non-zero result will have a `0X` prefix.

`a, A, e, E, f, or F`
The result will always contain a decimal point even if no digits follow the point. (Normally, a decimal point appears only if a digit follows it.) Trailing zeros are removed.

`g or G` The result will always contain a decimal point even if no digits follow the point. Trailing zeros are not removed.

`all others`
Undefined.

- *width*

  width is an optional minimum field width. You can either specify it directly as a decimal integer, or indirectly by using instead an asterisk (`*`), in which case an `int` argument is used as the field width. If positional arguments are used, then the width must also be specified positionally as `*m$`, with m as a decimal integer. Negative field widths are treated as specifying the minus (`-`) flag for left justfication, along with a positive field width. The resulting format may be wider than the specified width.

- *prec*

  prec is an optional field; if present, it is introduced with '`.`' (a period). You can specify the precision either directly as a decimal integer or indirectly by using an asterisk (`*`), in which case an `int` argument is used as the precision. If positional arguments are used, then the precision must also be specified positionally as `*m$`, with m as a decimal integer. Supplying a negative precision is equivalent to omitting the precision. If only a period is specified the precision is zero. The effect depends on the conversion *type*.

  `d, i, o, u, x, or X`
  Minimum number of digits to appear. If no precision is given, defaults to 1.

  `a or A` Number of digits to appear after the decimal point. If no precision is given, the precision defaults to the minimum needed for an exact representation.

  `e, E, f or F`
  Number of digits to appear after the decimal point. If no precision is given, the precision defaults to 6.

`g or G` Maximum number of significant digits. A precision of 0 is treated the same as a precision of 1. If no precision is given, the precision defaults to 6.

`s or S` Maximum number of characters to print from the string. If no precision is given, the entire string is printed.

`all others`
undefined.

- *size*

size is an optional modifier that changes the data type that the corresponding argument has. Behavior is unspecified if a size is given that does not match the *type*.

`hh` With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument should be converted to a `signed char` or `unsigned char` before printing.

With `n`, specifies that the argument is a pointer to a `signed char`.

`h` With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument should be converted to a `short` or `unsigned short` before printing.

With `n`, specifies that the argument is a pointer to a `short`.

`l` With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument is a `long` or `unsigned long`.

With `c`, specifies that the argument has type `wint_t`.

With `s`, specifies that the argument is a pointer to `wchar_t`.

With `n`, specifies that the argument is a pointer to a `long`.

With `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G`, has no effect (because of vararg promotion rules, there is no need to distinguish between `float` and `double`).

`ll` With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument is a `long long` or `unsigned long long`.

With `n`, specifies that the argument is a pointer to a `long long`.

`j` With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument is an `intmax_t` or `uintmax_t`.

With `n`, specifies that the argument is a pointer to an `intmax_t`.

`z` With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument is a `size_t`.

With `n`, specifies that the argument is a pointer to a `size_t`.

`t` With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument is a `ptrdiff_t`.

With `n`, specifies that the argument is a pointer to a `ptrdiff_t`.

`L` With `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G`, specifies that the argument is a `long double`.

- *type*

type specifies what kind of conversion `printf` performs. Here is a table of these:

`%` Prints the percent character (`%`).

`c` Prints *arg* as single character. If the `l` size specifier is in effect, a multibyte character is printed.

`C` Short for `%lc`. A POSIX extension to the C standard.

`s` Prints the elements of a pointer to `char` until the precision or a null character is reached. If the `l` size specifier is in effect, the pointer is to an array of `wchar_t`, and the string is converted to multibyte characters before printing.

`S` Short for `%ls`. A POSIX extension to the C standard.

`d or i` Prints a signed decimal integer; takes an `int`. Leading zeros are inserted as necessary to reach the precision. A value of 0 with a precision of 0 produces an empty string.

`D` Newlib extension, short for `%ld`.

`o` Prints an unsigned octal integer; takes an `unsigned`. Leading zeros are inserted as necessary to reach the precision. A value of 0 with a precision of 0 produces an empty string.

`O` Newlib extension, short for `%lo`.

`u` Prints an unsigned decimal integer; takes an `unsigned`. Leading zeros are inserted as necessary to reach the precision. A value of 0 with a precision of 0 produces an empty string.

`U` Newlib extension, short for `%lu`.

`x` Prints an unsigned hexadecimal integer (using `abcdef` as digits beyond `9`); takes an `unsigned`. Leading zeros are inserted as necessary to reach the precision. A value of 0 with a precision of 0 produces an empty string.

`X` Like `x`, but uses `ABCDEF` as digits beyond `9`.

`f` Prints a signed value of the form `[-]9999.9999`, with the precision determining how many digits follow the decimal point; takes a `double` (remember that `float` promotes to `double` as a vararg). The low order digit is rounded to even. If the precision results in at most DECIMAL_DIG digits, the result is rounded correctly; if more than DECIMAL_DIG digits are printed, the result is only guaranteed to round back to the original value.

If the value is infinite, the result is `inf`, and no zero padding is performed. If the value is not a number, the result is `nan`, and no zero padding is performed.

`F` Like `f`, but uses `INF` and `NAN` for non-finite numbers.

`e` Prints a signed value of the form `[-]9.9999e[+|-]999`; takes a `double`. The digit before the decimal point is non-zero if the value is non-zero. The precision determines how many digits appear between `.` and `e`, and the exponent always contains at least two digits. The value zero has an exponent of zero. If the value is not finite, it is printed like `f`.

`E` Like `e`, but using `E` to introduce the exponent, and like `F` for non-finite values.

`g` Prints a signed value in either `f` or `e` form, based on the given value and precision—an exponent less than -4 or greater than the precision selects the `e` form. Trailing zeros and the decimal point are printed only if necessary; takes a `double`.

`G` Like `g`, except use `F` or `E` form.

`a` Prints a signed value of the form `[-]0x1.ffffp[+|-]9`; takes a `double`. The letters `abcdef` are used for digits beyond `9`. The precision determines how many digits appear after the decimal point. The exponent contains at least one digit, and is a decimal value representing the power of 2; a value of 0 has an exponent of 0. Non-finite values are printed like `f`.

`A` Like `a`, except uses `X`, `P`, and `ABCDEF` instead of lower case.

`n` Takes a pointer to `int`, and stores a count of the number of bytes written so far. No output is created.

`p` Takes a pointer to `void`, and prints it in an implementation-defined format. This implementation is similar to `%#tx`), except that `0x` appears even for the NULL pointer.

`m` Prints the output of `strerror(errno)`; no argument is required. A GNU extension.

`_printf_r`, `_fprintf_r`, `_asprintf_r`, `_sprintf_r`, `_snprintf_r`, `_asnprintf_r` are simply reentrant versions of the functions above.

Returns
On success, `sprintf` and `asprintf` return the number of bytes in the output string, except the concluding `NUL` is not counted. `snprintf` returns the number of bytes that would be in the output string, except the concluding `NUL` is not counted. `printf` and `fprintf` return the number of characters transmitted. `asnprintf` returns the original *str* if there was enough room, otherwise it returns an allocated string.

If an error occurs, the result of `printf`, `fprintf`, `snprintf`, and `asprintf` is a negative value, and the result of `asnprintf` is NULL. No error returns occur for `sprintf`. For `printf` and `fprintf`, `errno` may be set according to `fputc`. For `asprintf` and `asnprintf`, `errno` may be set to ENOMEM if allocation fails, and for `snprintf`, `errno` may be set to EOVERFLOW if *size* or the output length exceeds INT_MAX.

Bugs
The “”’ (quote) flag does not work when locale’s thousands_sep is not empty.

Portability
ANSI C requires `printf`, `fprintf`, `sprintf`, and `snprintf`. `asprintf` and `asnprintf` are newlib extensions.

The ANSI C standard specifies that implementations must support at least formatted output of up to 509 characters. This implementation has no inherent limit.

Depending on how newlib was configured, not all format specifiers are supported.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.62 sscanf, fscanf, scanf—scan and format input

Synopsis

```
#include <stdio.h>

int scanf(const char *restrict format, ...);
int fscanf(FILE *restrict fd, const char *restrict format, ...);
int sscanf(const char *restrict str, const char *restrict format, ...);

int _scanf_r(struct _reent *ptr, const char *restrict format, ...);
int _fscanf_r(struct _reent *ptr, FILE *restrict fd,
    const char *restrict format, ...);
int _sscanf_r(struct _reent *ptr, const char *restrict str,
    const char *restrict format, ...);
```

Description

`scanf` scans a series of input fields from standard input, one character at a time. Each field is interpreted according to a format specifier passed to `scanf` in the format string at `*format`. `scanf` stores the interpreted input from each field at the address passed to it as the corresponding argument following *format*. You must supply the same number of format specifiers and address arguments as there are input fields.

There must be sufficient address arguments for the given format specifiers; if not the results are unpredictable and likely disasterous. Excess address arguments are merely ignored.

`scanf` often produces unexpected results if the input diverges from an expected pattern. Since the combination of `gets` or `fgets` followed by `sscanf` is safe and easy, that is the preferred way to be certain that a program is synchronized with input at the end of a line.

`fscanf` and `sscanf` are identical to `scanf`, other than the source of input: `fscanf` reads from a file, and `sscanf` from a string.

The routines `_scanf_r`, `_fscanf_r`, and `_sscanf_r` are reentrant versions of `scanf`, `fscanf`, and `sscanf` that take an additional first argument pointing to a reentrancy structure.

The string at `*format` is a character sequence composed of zero or more directives. Directives are composed of one or more whitespace characters, non-whitespace characters, and format specifications.

Whitespace characters are blank ( ), tab (`\t`), or newline (`\n`). When `scanf` encounters a whitespace character in the format string it will read (but not store) all consecutive whitespace characters up to the next non-whitespace character in the input.

Non-whitespace characters are all other ASCII characters except the percent sign (`%`). When `scanf` encounters a non-whitespace character in the format string it will read, but not store a matching non-whitespace character.

Format specifications tell `scanf` to read and convert characters from the input field into specific types of values, and store then in the locations specified by the address arguments.

Trailing whitespace is left unread unless explicitly matched in the format string.

The format specifiers must begin with a percent sign (`%`) and have the following form:

```
%[*][width][size]type
```

Each format specification begins with the percent character (`%`). The other fields are:

- *

  an optional marker; if present, it suppresses interpretation and assignment of this input field.

- *width*

  an optional maximum field width: a decimal integer, which controls the maximum number of characters that will be read before converting the current input field. If the input field has fewer than *width* characters, `scanf` reads all the characters in the field, and then proceeds with the next field and its format specification.

  If a whitespace or a non-convertable character occurs before *width* character are read, the characters up to that character are read, converted, and stored. Then `scanf` proceeds to the next format specification.

- *size*

  `h`, `j`, `l`, `L`, `t`, and `z` are optional size characters which override the default way that `scanf` interprets the data type of the corresponding argument.

Modifier	**Type(s)**	
hh	d, i, o, u, x, n	convert input to char, store in char object
h	d, i, o, u, x, n	convert input to short, store in short object
h	D, I, O, U, X, e, f, c, s, p	no effect
j	d, i, o, u, x, n	convert input to intmax_t, store in intmax_t object
j	all others	no effect
l	d, i, o, u, x, n	convert input to long, store in long object
l	e, f, g	convert input to double, store in a double object
l	D, I, O, U, X, c, s, p	no effect
ll	d, i, o, u, x, n	convert to long long, store in long long object
L	d, i, o, u, x, n	convert to long long, store in long long object
L	e, f, g, E, G	convert to long double, store in long double object
L	all others	no effect
t	d, i, o, u, x, n	convert input to ptrdiff_t, store in ptrdiff_t object
t	all others	no effect
z	d, i, o, u, x, n	convert input to size_t, store in size_t object
z	all others	no effect

- *type*

  A character to specify what kind of conversion `scanf` performs. Here is a table of the conversion characters:

  `%` No conversion is done; the percent character (`%`) is stored.

  `c` Scans one character. Corresponding *arg*: `(char *arg)`.

  `s` Reads a character string into the array supplied. Corresponding *arg*: `(char arg[])`.

`[`*pattern*`]`
: Reads a non-empty character string into memory starting at *arg*. This area must be large enough to accept the sequence and a terminating null character which will be added automatically. (*pattern* is discussed in the paragraph following this table). Corresponding *arg*: `(char *arg)`.

`d`
: Reads a decimal integer into the corresponding *arg*: `(int *arg)`.

`D`
: Reads a decimal integer into the corresponding *arg*: `(long *arg)`.

`o`
: Reads an octal integer into the corresponding *arg*: `(int *arg)`.

`O`
: Reads an octal integer into the corresponding *arg*: `(long *arg)`.

`u`
: Reads an unsigned decimal integer into the corresponding *arg*: `(unsigned int *arg)`.

`U`
: Reads an unsigned decimal integer into the corresponding *arg*: `(unsigned long *arg)`.

`x,X`
: Read a hexadecimal integer into the corresponding *arg*: `(int *arg)`.

`e, f, g`
: Read a floating-point number into the corresponding *arg*: `(float *arg)`.

`E, F, G`
: Read a floating-point number into the corresponding *arg*: `(double *arg)`.

`i`
: Reads a decimal, octal or hexadecimal integer into the corresponding *arg*: `(int *arg)`.

`I`
: Reads a decimal, octal or hexadecimal integer into the corresponding *arg*: `(long *arg)`.

`n`
: Stores the number of characters read in the corresponding *arg*: `(int *arg)`.

`p`
: Stores a scanned pointer. ANSI C leaves the details to each implementation; this implementation treats `%p` exactly the same as `%U`. Corresponding *arg*: `(void **arg)`.

A *pattern* of characters surrounded by square brackets can be used instead of the `s` type character. *pattern* is a set of characters which define a search set of possible characters making up the `scanf` input field. If the first character in the brackets is a caret (`^`), the search set is inverted to include all ASCII characters except those between the brackets. There is also a range facility which you can use as a shortcut. `%[0-9]` matches all decimal digits. The hyphen must not be the first or last character in the set. The character prior to the hyphen must be lexically less than the character after it.

Here are some *pattern* examples:

`%[abcd]`
: matches strings containing only `a`, `b`, `c`, and `d`.

`%[^abcd]`
: matches strings containing any characters except `a`, `b`, `c`, or `d`

`%[A-DW-Z]`
: matches strings containing `A`, `B`, `C`, `D`, `W`, `X`, `Y`, `Z`

`%[z-a]`
: matches the characters `z`, `-`, and `a`

Floating point numbers (for field types `e`, `f`, `g`, `E`, `F`, `G`) must correspond to the following general form:

```
[+/-] ddddd[.]ddd [E|e[+|-]ddd]
```

where objects inclosed in square brackets are optional, and `ddd` represents decimal, octal, or hexadecimal digits.

Returns
`scanf` returns the number of input fields successfully scanned, converted and stored; the return value does not include scanned fields which were not stored.

If `scanf` attempts to read at end-of-file, the return value is `EOF`.

If no fields were stored, the return value is `0`.

`scanf` might stop scanning a particular field before reaching the normal field end character, or may terminate entirely.

`scanf` stops scanning and storing the current field and moves to the next input field (if any) in any of the following situations:

- The assignment suppressing character (`*`) appears after the `%` in the format specification; the current input field is scanned but not stored.
- *width* characters have been read (*width* is a width specification, a positive decimal integer).
- The next character read cannot be converted under the the current format (for example, if a `Z` is read when the format is decimal).
- The next character in the input field does not appear in the search set (or does appear in the inverted search set).

When `scanf` stops scanning the current input field for one of these reasons, the next character is considered unread and used as the first character of the following input field, or the first character in a subsequent read operation on the input.

`scanf` will terminate under the following circumstances:

- The next character in the input field conflicts with a corresponding non-whitespace character in the format string.
- The next character in the input field is `EOF`.
- The format string has been exhausted.

When the format string contains a character sequence that is not part of a format specification, the same character sequence must appear in the input; `scanf` will scan but not store the matched characters. If a conflict occurs, the first conflicting character remains in the input as if it had never been read.

Portability
`scanf` is ANSI C.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.63 stdio_ext,__fbufsize,__fpending,__flbf,__freadable,__fwritable,__freading,__fwriting—access internals of FILE structure

Synopsis

```
#include <stdio.h>
#include <stdio_ext.h>
size_t __fbufsize(FILE *fp);
size_t __fpending(FILE *fp);
int __flbf(FILE *fp);
int __freadable(FILE *fp);
int __fwritable(FILE *fp);
int __freading(FILE *fp);
int __fwriting(FILE *fp);
```

Description

These functions provides access to the internals of the FILE structure *fp*.

Returns

`__fbufsize` returns the number of bytes in the buffer of stream *fp*.

`__fpending` returns the number of bytes in the output buffer of stream *fp*.

`__flbf` returns nonzero if stream *fp* is line-buffered, and `0` if not.

`__freadable` returns nonzero if stream *fp* may be read, and `0` if not.

`__fwritable` returns nonzero if stream *fp* may be written, and `0` if not.

`__freading` returns nonzero if stream *fp* if the last operation on it was a read, or if it read-only, and `0` if not.

`__fwriting` returns nonzero if stream *fp* if the last operation on it was a write, or if it write-only, and `0` if not.

Portability

These functions originate from Solaris and are also provided by GNU libc.

No supporting OS subroutines are required.

4.64 swprintf, fwprintf, wprintf—wide character format output

Synopsis

```
#include <wchar.h>

int wprintf(const wchar_t *format, ...);
int fwprintf(FILE *__restrict fd,
    const wchar_t *__restrict format, ...);
int swprintf(wchar_t *__restrict str, size_t size,
    const wchar_t *__restrict format, ...);

int _wprintf_r(struct _reent *ptr, const wchar_t *format, ...);
int _fwprintf_r(struct _reent *ptr, FILE *fd,
    const wchar_t *format, ...);
int _swprintf_r(struct _reent *ptr, wchar_t *str,
    size_t size, const wchar_t *format, ...);
```

Description

`wprintf` accepts a series of arguments, applies to each a format specifier from `*format`, and writes the formatted data to `stdout`, without a terminating NUL wide character. The behavior of `wprintf` is undefined if there are not enough arguments for the format or if any argument is not the right type for the corresponding conversion specifier. `wprintf` returns when it reaches the end of the format string. If there are more arguments than the format requires, excess arguments are ignored.

`fwprintf` is like `wprintf`, except that output is directed to the stream *fd* rather than `stdout`.

`swprintf` is like `wprintf`, except that output is directed to the buffer *str* with a terminating wide `NUL`, and the resulting string length is limited to at most *size* wide characters, including the terminating `NUL`. It is considered an error if the output (including the terminating wide-`NULL`) does not fit into *size* wide characters. (This error behavior is not the same as for `snprintf`, which `swprintf` is otherwise completely analogous to. While `snprintf` allows the needed size to be known simply by giving *size*=0, `swprintf` does not, giving an error instead.)

For `swprintf` the behavior is undefined if the output `*str` overlaps with one of the arguments. Behavior is also undefined if the argument for `%n` within `*format` overlaps another argument.

format is a pointer to a wide character string containing two types of objects: ordinary characters (other than `%`), which are copied unchanged to the output, and conversion specifications, each of which is introduced by `%`. (To include `%` in the output, use `%%` in the format string.) A conversion specification has the following form:

```
%[pos][flags][width][.prec][size]type
```

The fields of the conversion specification have the following meanings:

- *pos*

  Conversions normally consume arguments in the order that they are presented. However, it is possible to consume arguments out of order, and reuse an argument for more

than one conversion specification (although the behavior is undefined if the same argument is requested with different types), by specifying *pos*, which is a decimal integer followed by ’$’. The integer must be between 1 and <NL_ARGMAX> from limits.h, and if argument `%n$` is requested, all earlier arguments must be requested somewhere within *format*. If positional parameters are used, then all conversion specifications except for `%%` must specify a position. This positional parameters method is a POSIX extension to the C standard definition for the functions.

- *flags*

  flags is an optional sequence of characters which control output justification, numeric signs, decimal points, trailing zeros, and octal and hex prefixes. The flag characters are minus (`-`), plus (`+`), space ( ), zero (`0`), sharp (`#`), and quote (`’`). They can appear in any combination, although not all flags can be used for all conversion specification types.

  `’` A POSIX extension to the C standard. However, this implementation presently treats it as a no-op, which is the default behavior for the C locale, anyway. (If it did what it is supposed to, when *type* were `i`, `d`, `u`, `f`, `F`, `g`, or `G`, the integer portion of the conversion would be formatted with thousands’ grouping wide characters.)

  `-` The result of the conversion is left justified, and the right is padded with blanks. If you do not use this flag, the result is right justified, and padded on the left.

  `+` The result of a signed conversion (as determined by *type* of `d`, `i`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G`) will always begin with a plus or minus sign. (If you do not use this flag, positive values do not begin with a plus sign.)

  `" " (space)` If the first character of a signed conversion specification is not a sign, or if a signed conversion results in no characters, the result will begin with a space. If the space ( ) flag and the plus (`+`) flag both appear, the space flag is ignored.

  `0` If the *type* character is `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G`: leading zeros are used to pad the field width (following any indication of sign or base); no spaces are used for padding. If the zero (`0`) and minus (`-`) flags both appear, the zero (`0`) flag will be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision *prec* is specified, the zero (`0`) flag is ignored.

  Note that `0` is interpreted as a flag, not as the beginning of a field width.

  `#` The result is to be converted to an alternative form, according to the *type* character.

  The alternative form output with the # flag depends on the *type* character:

  `o` Increases precision to force the first digit of the result to be a zero.

  `x` A non-zero result will have a `0x` prefix.

  `X` A non-zero result will have a `0X` prefix.

`a, A, e, E, f, or F`
: The result will always contain a decimal point even if no digits follow the point. (Normally, a decimal point appears only if a digit follows it.) Trailing zeros are removed.

`g or G`
: The result will always contain a decimal point even if no digits follow the point. Trailing zeros are not removed.

`all others`
: Undefined.

- *width*

  width is an optional minimum field width. You can either specify it directly as a decimal integer, or indirectly by using instead an asterisk (`*`), in which case an `int` argument is used as the field width. If positional arguments are used, then the width must also be specified positionally as `*m$`, with m as a decimal integer. Negative field widths are treated as specifying the minus (`-`) flag for left justfication, along with a positive field width. The resulting format may be wider than the specified width.

- *prec*

  prec is an optional field; if present, it is introduced with '`.`' (a period). You can specify the precision either directly as a decimal integer or indirectly by using an asterisk (`*`), in which case an `int` argument is used as the precision. If positional arguments are used, then the precision must also be specified positionally as `*m$`, with m as a decimal integer. Supplying a negative precision is equivalent to omitting the precision. If only a period is specified the precision is zero. The effect depends on the conversion *type*.

  `d, i, o, u, x, or X`
  : Minimum number of digits to appear. If no precision is given, defaults to 1.

  `a or A`
  : Number of digits to appear after the decimal point. If no precision is given, the precision defaults to the minimum needed for an exact representation.

  `e, E, f or F`
  : Number of digits to appear after the decimal point. If no precision is given, the precision defaults to 6.

  `g or G`
  : Maximum number of significant digits. A precision of 0 is treated the same as a precision of 1. If no precision is given, the precision defaults to 6.

  `s or S`
  : Maximum number of characters to print from the string. If no precision is given, the entire string is printed.

  `all others`
  : undefined.

- *size*

  size is an optional modifier that changes the data type that the corresponding argument has. Behavior is unspecified if a size is given that does not match the *type*.

  `hh`
  : With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument should be converted to a `signed char` or `unsigned char` before printing.

    With `n`, specifies that the argument is a pointer to a `signed char`.

`h` With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument should be converted to a `short` or `unsigned short` before printing.

With `n`, specifies that the argument is a pointer to a `short`.

`l` With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument is a `long` or `unsigned long`.

With `c`, specifies that the argument has type `wint_t`.

With `s`, specifies that the argument is a pointer to `wchar_t`.

With `n`, specifies that the argument is a pointer to a `long`.

With `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G`, has no effect (because of vararg promotion rules, there is no need to distinguish between `float` and `double`).

`ll` With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument is a `long long` or `unsigned long long`.

With `n`, specifies that the argument is a pointer to a `long long`.

`j` With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument is an `intmax_t` or `uintmax_t`.

With `n`, specifies that the argument is a pointer to an `intmax_t`.

`z` With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument is a `size_t`.

With `n`, specifies that the argument is a pointer to a `size_t`.

`t` With `d`, `i`, `o`, `u`, `x`, or `X`, specifies that the argument is a `ptrdiff_t`.

With `n`, specifies that the argument is a pointer to a `ptrdiff_t`.

`L` With `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G`, specifies that the argument is a `long double`.

- *type*

  type specifies what kind of conversion `wprintf` performs. Here is a table of these:

  `%` Prints the percent character (`%`).

  `c` If no `l` qualifier is present, the int argument shall be converted to a wide character as if by calling the btowc() function and the resulting wide character shall be written. Otherwise, the wint_t argument shall be converted to wchar_t, and written.

  `C` Short for `%lc`. A POSIX extension to the C standard.

  `s` If no `l` qualifier is present, the application shall ensure that the argument is a pointer to a character array containing a character sequence beginning in the initial shift state. Characters from the array shall be converted as if by repeated calls to the mbrtowc() function, with the conversion state described by an mbstate_t object initialized to zero before the first character is converted, and written up to (but not including) the terminating null wide character. If the precision is specified, no more than that many wide characters shall be written. If the precision is not specified, or is greater than the size of the array, the application shall ensure that the array contains a null wide character.

If an `l` qualifier is present, the application shall ensure that the argument is a pointer to an array of type wchar_t. Wide characters from the array shall be written up to (but not including) a terminating null wide character. If no precision is specified, or is greater than the size of the array, the application shall ensure that the array contains a null wide character. If a precision is specified, no more than that many wide characters shall be written.

`S` Short for `%ls`. A POSIX extension to the C standard.

`d or i` Prints a signed decimal integer; takes an `int`. Leading zeros are inserted as necessary to reach the precision. A value of 0 with a precision of 0 produces an empty string.

`o` Prints an unsigned octal integer; takes an `unsigned`. Leading zeros are inserted as necessary to reach the precision. A value of 0 with a precision of 0 produces an empty string.

`u` Prints an unsigned decimal integer; takes an `unsigned`. Leading zeros are inserted as necessary to reach the precision. A value of 0 with a precision of 0 produces an empty string.

`x` Prints an unsigned hexadecimal integer (using `abcdef` as digits beyond `9`); takes an `unsigned`. Leading zeros are inserted as necessary to reach the precision. A value of 0 with a precision of 0 produces an empty string.

`X` Like `x`, but uses `ABCDEF` as digits beyond `9`.

`f` Prints a signed value of the form `[-]9999.9999`, with the precision determining how many digits follow the decimal point; takes a `double` (remember that `float` promotes to `double` as a vararg). The low order digit is rounded to even. If the precision results in at most DECIMAL_DIG digits, the result is rounded correctly; if more than DECIMAL_DIG digits are printed, the result is only guaranteed to round back to the original value.

If the value is infinite, the result is `inf`, and no zero padding is performed. If the value is not a number, the result is `nan`, and no zero padding is performed.

`F` Like `f`, but uses `INF` and `NAN` for non-finite numbers.

`e` Prints a signed value of the form `[-]9.9999e[+|-]999`; takes a `double`. The digit before the decimal point is non-zero if the value is non-zero. The precision determines how many digits appear between `.` and `e`, and the exponent always contains at least two digits. The value zero has an exponent of zero. If the value is not finite, it is printed like `f`.

`E` Like `e`, but using `E` to introduce the exponent, and like `F` for non-finite values.

`g` Prints a signed value in either `f` or `e` form, based on the given value and precision—an exponent less than -4 or greater than the precision selects the `e` form. Trailing zeros and the decimal point are printed only if necessary; takes a `double`.

`G` Like `g`, except use `F` or `E` form.

`a` Prints a signed value of the form `[-]0x1.ffffp[+|-]9`; takes a `double`. The letters `abcdef` are used for digits beyond `9`. The precision determines how many digits appear after the decimal point. The exponent contains at least one digit, and is a decimal value representing the power of 2; a value of 0 has an exponent of 0. Non-finite values are printed like `f`.

`A` Like `a`, except uses `X`, `P`, and `ABCDEF` instead of lower case.

`n` Takes a pointer to `int`, and stores a count of the number of bytes written so far. No output is created.

`p` Takes a pointer to `void`, and prints it in an implementation-defined format. This implementation is similar to `%#tx`), except that `0x` appears even for the NULL pointer.

`m` Prints the output of `strerror(errno)`; no argument is required. A GNU extension.

`_wprintf_r`, `_fwprintf_r`, `_swprintf_r`, are simply reentrant versions of the functions above.

Returns

On success, `swprintf` return the number of wide characters in the output string, except the concluding `NUL` is not counted. `wprintf` and `fwprintf` return the number of characters transmitted.

If an error occurs, the result of `wprintf`, `fwprintf`, and `swprintf` is a negative value. For `wprintf` and `fwprintf`, `errno` may be set according to `fputwc`. For `swprintf`, `errno` may be set to EOVERFLOW if *size* is greater than INT_MAX / sizeof (wchar_t), or when the output does not fit into *size* wide characters (including the terminating wide `NULL`).

Bugs

The “”’ (quote) flag does not work when locale’s thousands_sep is not empty.

Portability

POSIX-1.2008 with extensions; C99 (compliant except for POSIX extensions).

Depending on how newlib was configured, not all format specifiers are supported.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.65 swscanf, fwscanf, wscanf—scan and format wide character input

Synopsis

```
#include <stdio.h>

int wscanf(const wchar_t *__restrict format, ...);
int fwscanf(FILE *__restrict fd,
    const wchar_t *__restrict format, ...);
int swscanf(const wchar_t *__restrict str,
    const wchar_t *__restrict format, ...);

int _wscanf_r(struct _reent *ptr, const wchar_t *format, ...);
int _fwscanf_r(struct _reent *ptr, FILE *fd,
    const wchar_t *format, ...);
int _swscanf_r(struct _reent *ptr, const wchar_t *str,
    const wchar_t *format, ...);
```

Description

`wscanf` scans a series of input fields from standard input, one wide character at a time. Each field is interpreted according to a format specifier passed to `wscanf` in the format string at `*format`. `wscanf` stores the interpreted input from each field at the address passed to it as the corresponding argument following *format*. You must supply the same number of format specifiers and address arguments as there are input fields.

There must be sufficient address arguments for the given format specifiers; if not the results are unpredictable and likely disasterous. Excess address arguments are merely ignored.

`wscanf` often produces unexpected results if the input diverges from an expected pattern. Since the combination of `gets` or `fgets` followed by `swscanf` is safe and easy, that is the preferred way to be certain that a program is synchronized with input at the end of a line.

`fwscanf` and `swscanf` are identical to `wscanf`, other than the source of input: `fwscanf` reads from a file, and `swscanf` from a string.

The routines `_wscanf_r`, `_fwscanf_r`, and `_swscanf_r` are reentrant versions of `wscanf`, `fwscanf`, and `swscanf` that take an additional first argument pointing to a reentrancy structure.

The string at `*format` is a wide character sequence composed of zero or more directives. Directives are composed of one or more whitespace characters, non-whitespace characters, and format specifications.

Whitespace characters are blank ( ), tab (`\t`), or newline (`\n`). When `wscanf` encounters a whitespace character in the format string it will read (but not store) all consecutive whitespace characters up to the next non-whitespace character in the input.

Non-whitespace characters are all other ASCII characters except the percent sign (`%`). When `wscanf` encounters a non-whitespace character in the format string it will read, but not store a matching non-whitespace character.

Format specifications tell `wscanf` to read and convert characters from the input field into specific types of values, and store then in the locations specified by the address arguments.

Trailing whitespace is left unread unless explicitly matched in the format string.

The format specifiers must begin with a percent sign (%) and have the following form:

```
%[*][width][size]type
```

Each format specification begins with the percent character (%). The other fields are:

- *

  an optional marker; if present, it suppresses interpretation and assignment of this input field.

- *width*

  an optional maximum field width: a decimal integer, which controls the maximum number of characters that will be read before converting the current input field. If the input field has fewer than *width* characters, `wscanf` reads all the characters in the field, and then proceeds with the next field and its format specification.

  If a whitespace or a non-convertable wide character occurs before *width* character are read, the characters up to that character are read, converted, and stored. Then `wscanf` proceeds to the next format specification.

- *size*

  `h`, `j`, `l`, `L`, `t`, and `z` are optional size characters which override the default way that `wscanf` interprets the data type of the corresponding argument.

Modifier	**Type(s)**	
hh	d, i, o, u, x, n	convert input to char, store in char object
h	d, i, o, u, x, n	convert input to short, store in short object
h	e, f, c, s, p	no effect
j	d, i, o, u, x, n	convert input to intmax_t, store in intmax_t object
j	all others	no effect
l	d, i, o, u, x, n	convert input to long, store in long object
l	e, f, g	convert input to double, store in a double object
l	c, s, [	the input is stored in a wchar_t object
l	p	no effect
ll	d, i, o, u, x, n	convert to long long, store in long long object
L	d, i, o, u, x, n	convert to long long, store in long long object
L	e, f, g, E, G	convert to long double, store in long double object
L	all others	no effect
t	d, i, o, u, x, n	convert input to ptrdiff_t, store in ptrdiff_t object
t	all others	no effect
z	d, i, o, u, x, n	convert input to size_t, store in size_t object
z	all others	no effect

- *type*

  A character to specify what kind of conversion `wscanf` performs. Here is a table of the conversion characters:

  `%` No conversion is done; the percent character (%) is stored.

`c` Scans one wide character. Corresponding *arg*: `(char *arg)`. Otherwise, if an `l` specifier is present, the corresponding *arg* is a `(wchar_t *arg)`.

`s` Reads a character string into the array supplied. Corresponding *arg*: `(char arg[])`. If an `l` specifier is present, the corresponding *arg* is a `(wchar_t *arg)`.

`[`*pattern*`]`
Reads a non-empty character string into memory starting at *arg*. This area must be large enough to accept the sequence and a terminating null character which will be added automatically. (*pattern* is discussed in the paragraph following this table). Corresponding *arg*: `(char *arg)`. If an `l` specifier is present, the corresponding *arg* is a `(wchar_t *arg)`.

`d` Reads a decimal integer into the corresponding *arg*: `(int *arg)`.

`o` Reads an octal integer into the corresponding *arg*: `(int *arg)`.

`u` Reads an unsigned decimal integer into the corresponding *arg*: `(unsigned int *arg)`.

`x,X` Read a hexadecimal integer into the corresponding *arg*: `(int *arg)`.

`e, f, g` Read a floating-point number into the corresponding *arg*: `(float *arg)`.

`E, F, G` Read a floating-point number into the corresponding *arg*: `(double *arg)`.

`i` Reads a decimal, octal or hexadecimal integer into the corresponding *arg*: `(int *arg)`.

`n` Stores the number of characters read in the corresponding *arg*: `(int *arg)`.

`p` Stores a scanned pointer. ANSI C leaves the details to each implementation; this implementation treats `%p` exactly the same as `%U`. Corresponding *arg*: `(void **arg)`.

A *pattern* of characters surrounded by square brackets can be used instead of the `s` type character. *pattern* is a set of characters which define a search set of possible characters making up the `wscanf` input field. If the first character in the brackets is a caret (`^`), the search set is inverted to include all ASCII characters except those between the brackets. There is no range facility as is defined in the corresponding non-wide character scanf functions. Ranges are not part of the POSIX standard.

Here are some *pattern* examples:

`%[abcd]` matches wide character strings containing only `a`, `b`, `c`, and `d`.

`%[^abcd]` matches wide character strings containing any characters except `a`, `b`, `c`, or `d`.

`%[A-DW-Z]`
Note: No wide character ranges, so this expression matches wide character strings containing `A`, `-`, `D`, `W`, `Z`.

Floating point numbers (for field types `e`, `f`, `g`, `E`, `F`, `G`) must correspond to the following general form:

```
[+/-] ddddd[.]ddd [E|e[+|-]ddd]
```

where objects inclosed in square brackets are optional, and `ddd` represents decimal, octal, or hexadecimal digits.

Returns

`wscanf` returns the number of input fields successfully scanned, converted and stored; the return value does not include scanned fields which were not stored.

If `wscanf` attempts to read at end-of-file, the return value is `EOF`.

If no fields were stored, the return value is `0`.

`wscanf` might stop scanning a particular field before reaching the normal field end character, or may terminate entirely.

`wscanf` stops scanning and storing the current field and moves to the next input field (if any) in any of the following situations:

- The assignment suppressing character (`*`) appears after the `%` in the format specification; the current input field is scanned but not stored.
- *width* characters have been read (*width* is a width specification, a positive decimal integer).
- The next wide character read cannot be converted under the the current format (for example, if a `Z` is read when the format is decimal).
- The next wide character in the input field does not appear in the search set (or does appear in the inverted search set).

When `wscanf` stops scanning the current input field for one of these reasons, the next character is considered unread and used as the first character of the following input field, or the first character in a subsequent read operation on the input.

`wscanf` will terminate under the following circumstances:

- The next wide character in the input field conflicts with a corresponding non-whitespace character in the format string.
- The next wide character in the input field is `WEOF`.
- The format string has been exhausted.

When the format string contains a wide character sequence that is not part of a format specification, the same wide character sequence must appear in the input; `wscanf` will scan but not store the matched characters. If a conflict occurs, the first conflicting wide character remains in the input as if it had never been read.

Portability

`wscanf` is C99, POSIX-1.2008.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.66 `tmpfile`—create a temporary file

Synopsis

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

FILE *_tmpfile_r(struct _reent *reent);
```

Description

Create a temporary file (a file which will be deleted automatically), using a name generated by `tmpnam`. The temporary file is opened with the mode `"wb+"`, permitting you to read and write anywhere in it as a binary file (without any data transformations the host system may perform for text files).

The alternate function `_tmpfile_r` is a reentrant version. The argument *reent* is a pointer to a reentrancy structure.

Returns

`tmpfile` normally returns a pointer to the temporary file. If no temporary file could be created, the result is NULL, and `errno` records the reason for failure.

Portability

Both ANSI C and the System V Interface Definition (Issue 2) require `tmpfile`.

Supporting OS subroutines required: `close`, `fstat`, `getpid`, `isatty`, `lseek`, `open`, `read`, `sbrk`, `write`.

`tmpfile` also requires the global pointer `environ`.

4.67 tmpnam, tempnam—name for a temporary file

Synopsis

```
#include <stdio.h>
char *tmpnam(char *s);
char *tempnam(char *dir, char *pfx);
char *_tmpnam_r(struct _reent *reent, char *s);
char *_tempnam_r(struct _reent *reent, char *dir, char *pfx);
```

Description

Use either of these functions to generate a name for a temporary file. The generated name is guaranteed to avoid collision with other files (for up to `TMP_MAX` calls of either function).

`tmpnam` generates file names with the value of `P_tmpdir` (defined in ‘`stdio.h`’) as the leading directory component of the path.

You can use the `tmpnam` argument *s* to specify a suitable area of memory for the generated filename; otherwise, you can call `tmpnam(NULL)` to use an internal static buffer.

`tempnam` allows you more control over the generated filename: you can use the argument *dir* to specify the path to a directory for temporary files, and you can use the argument *pfx* to specify a prefix for the base filename.

If *dir* is `NULL`, `tempnam` will attempt to use the value of environment variable `TMPDIR` instead; if there is no such value, `tempnam` uses the value of `P_tmpdir` (defined in ‘`stdio.h`’).

If you don’t need any particular prefix to the basename of temporary files, you can pass `NULL` as the *pfx* argument to `tempnam`.

`_tmpnam_r` and `_tempnam_r` are reentrant versions of `tmpnam` and `tempnam` respectively. The extra argument *reent* is a pointer to a reentrancy structure.

Warnings

The generated filenames are suitable for temporary files, but do not in themselves make files temporary. Files with these names must still be explicitly removed when you no longer want them.

If you supply your own data area *s* for `tmpnam`, you must ensure that it has room for at least `L_tmpnam` elements of type `char`.

Returns

Both `tmpnam` and `tempnam` return a pointer to the newly generated filename.

Portability

ANSI C requires `tmpnam`, but does not specify the use of `P_tmpdir`. The System V Interface Definition (Issue 2) requires both `tmpnam` and `tempnam`.

Supporting OS subroutines required: `close`, `fstat`, `getpid`, `isatty`, `lseek`, `open`, `read`, `sbrk`, `write`.

The global pointer `environ` is also required.

4.68 ungetc—push data back into a stream

Synopsis

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

int _ungetc_r(struct _reent *reent, int c, FILE *stream);
```

Description

`ungetc` is used to return bytes back to *stream* to be read again. If *c* is EOF, the stream is unchanged. Otherwise, the unsigned char *c* is put back on the stream, and subsequent reads will see the bytes pushed back in reverse order. Pushed byes are lost if the stream is repositioned, such as by `fseek`, `fsetpos`, or `rewind`.

The underlying file is not changed, but it is possible to push back something different than what was originally read. Ungetting a character will clear the end-of-stream marker, and decrement the file position indicator. Pushing back beyond the beginning of a file gives unspecified behavior.

The alternate function `_ungetc_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

The character pushed back, or `EOF` on error.

Portability

ANSI C requires `ungetc`, but only requires a pushback buffer of one byte; although this implementation can handle multiple bytes, not all can. Pushing back a signed char is a common application bug.

Supporting OS subroutines required: `sbrk`.

4.69 ungetwc—push wide character data back into a stream

Synopsis

```
#include <stdio.h>
#include <wchar.h>
wint_t ungetwc(wint_t wc, FILE *stream);

wint_t _ungetwc_r(struct _reent *reent, wint_t wc, FILE *stream);
```

Description

`ungetwc` is used to return wide characters back to *stream* to be read again. If *wc* is WEOF, the stream is unchanged. Otherwise, the wide character *wc* is put back on the stream, and subsequent reads will see the wide chars pushed back in reverse order. Pushed wide chars are lost if the stream is repositioned, such as by `fseek`, `fsetpos`, or `rewind`.

The underlying file is not changed, but it is possible to push back something different than what was originally read. Ungetting a character will clear the end-of-stream marker, and decrement the file position indicator. Pushing back beyond the beginning of a file gives unspecified behavior.

The alternate function `_ungetwc_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

The wide character pushed back, or `WEOF` on error.

Portability

C99

4.70 vfprintf, vprintf, vsprintf, vsnprintf, vasprintf, vasnprintf—format argument list

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
int vprintf(const char *fmt, va_list list);
int vfprintf(FILE *fp, const char *fmt, va_list list);
int vsprintf(char *str, const char *fmt, va_list list);
int vsnprintf(char *str, size_t size, const char *fmt,
    va_list list);
int vasprintf(char **strp, const char *fmt, va_list list);
char *vasnprintf(char *str, size_t *size, const char *fmt,
    va_list list);

int _vprintf_r(struct _reent *reent, const char *fmt,
    va_list list);
int _vfprintf_r(struct _reent *reent, FILE *fp,
    const char *fmt, va_list list);
int _vsprintf_r(struct _reent *reent, char *str,
    const char *fmt, va_list list);
int _vasprintf_r(struct _reent *reent, char **str,
    const char *fmt, va_list list);
int _vsnprintf_r(struct _reent *reent, char *str,
    size_t size, const char *fmt, va_list list);
char *_vasnprintf_r(struct _reent *reent, char *str,
    size_t *size, const char *fmt, va_list list);
```

Description

`vprintf`, `vfprintf`, `vasprintf`, `vsprintf`, `vsnprintf`, and `vasnprintf` are (respectively) variants of `printf`, `fprintf`, `asprintf`, `sprintf`, `snprintf`, and `asnprintf`. They differ only in allowing their caller to pass the variable argument list as a `va_list` object (initialized by `va_start`) rather than directly accepting a variable number of arguments. The caller is responsible for calling `va_end`.

`_vprintf_r`, `_vfprintf_r`, `_vasprintf_r`, `_vsprintf_r`, `_vsnprintf_r`, and `_vasnprintf_r` are reentrant versions of the above.

Returns

The return values are consistent with the corresponding functions.

Portability

ANSI C requires `vprintf`, `vfprintf`, `vsprintf`, and `vsnprintf`. The remaining functions are newlib extensions.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.71 vfscanf, vscanf, vsscanf—format argument list

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
int vscanf(const char *fmt, va_list list);
int vfscanf(FILE *fp, const char *fmt, va_list list);
int vsscanf(const char *str, const char *fmt, va_list list);

int _vscanf_r(struct _reent *reent, const char *fmt,
    va_list list);
int _vfscanf_r(struct _reent *reent, FILE *fp, const char *fmt,
    va_list list);
int _vsscanf_r(struct _reent *reent, const char *str,
    const char *fmt, va_list list);
```

Description

vscanf, vfscanf, and vsscanf are (respectively) variants of scanf, fscanf, and sscanf. They differ only in allowing their caller to pass the variable argument list as a va_list object (initialized by va_start) rather than directly accepting a variable number of arguments.

Returns

The return values are consistent with the corresponding functions: vscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields which were not stored.

If vscanf attempts to read at end-of-file, the return value is EOF.

If no fields were stored, the return value is 0.

The routines _vscanf_r, _vfscanf_f, and _vsscanf_r are reentrant versions which take an additional first parameter which points to the reentrancy structure.

Portability

These are GNU extensions.

Supporting OS subroutines required:

4.72 vfwprintf, vwprintf, vswprintf—wide character format argument list

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>
int vwprintf(const wchar_t *__restrict fmt, va_list list);
int vfwprintf(FILE *__restrict fp,
    const wchar_t *__restrict fmt, va_list list);
int vswprintf(wchar_t * __restrict str, size_t size,
    const wchar_t *__ restrict fmt, va_list list);

int _vwprintf_r(struct _reent *reent, const wchar_t *fmt,
    va_list list);
int _vfwprintf_r(struct _reent *reent, FILE *fp,
    const wchar_t *fmt, va_list list);
int _vswprintf_r(struct _reent *reent, wchar_t *str,
    size_t size, const wchar_t *fmt, va_list list);
```

Description

`vwprintf`, `vfwprintf` and `vswprintf` are (respectively) variants of `wprintf`, `fwprintf` and `swprintf`. They differ only in allowing their caller to pass the variable argument list as a `va_list` object (initialized by `va_start`) rather than directly accepting a variable number of arguments. The caller is responsible for calling `va_end`.

`_vwprintf_r`, `_vfwprintf_r` and `_vswprintf_r` are reentrant versions of the above.

Returns

The return values are consistent with the corresponding functions.

Portability

POSIX-1.2008 with extensions; C99 (compliant except for POSIX extensions).

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

See Also

`wprintf`, `fwprintf` and `swprintf`.

4.73 vfwscanf, vwscanf, vswscanf—scan and format argument list from wide character input

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
int vwscanf(const wchar_t *__restrict fmt, va_list list);
int vfwscanf(FILE *__restrict fp,
    const wchar_t *__restrict fmt, va_list list);
int vswscanf(const wchar_t *__restrict str,
    const wchar_t *__restrict fmt, va_list list);

int _vwscanf(struct _reent *reent, const wchar_t *fmt,
    va_list list);
int _vfwscanf(struct _reent *reent, FILE *fp,
    const wchar_t *fmt, va_list list);
int _vswscanf(struct _reent *reent, const wchar_t *str,
    const wchar_t *fmt, va_list list);
```

Description

vwscanf, vfwscanf, and vswscanf are (respectively) variants of wscanf, fwscanf, and swscanf. They differ only in allowing their caller to pass the variable argument list as a va_list object (initialized by va_start) rather than directly accepting a variable number of arguments.

Returns

The return values are consistent with the corresponding functions: vwscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields which were not stored.

If vwscanf attempts to read at end-of-file, the return value is EOF.

If no fields were stored, the return value is 0.

The routines _vwscanf, _vfwscanf, and _vswscanf are reentrant versions which take an additional first parameter which points to the reentrancy structure.

Portability

C99, POSIX-1.2008

4.74 viprintf, vfiprintf, vsiprintf, vsniprintf, vasiprintf, vasniprintf—format argument list (integer only)

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
int viprintf(const char *fmt, va_list list);
int vfiprintf(FILE *fp, const char *fmt, va_list list);
int vsiprintf(char *str, const char *fmt, va_list list);
int vsniprintf(char *str, size_t size, const char *fmt,
    va_list list);
int vasiprintf(char **strp, const char *fmt, va_list list);
char *vasniprintf(char *str, size_t *size, const char *fmt,
    va_list list);

int _viprintf_r(struct _reent *reent, const char *fmt,
    va_list list);
int _vfiprintf_r(struct _reent *reent, FILE *fp,
    const char *fmt, va_list list);
int _vsiprintf_r(struct _reent *reent, char *str,
    const char *fmt, va_list list);
int _vsniprintf_r(struct _reent *reent, char *str,
    size_t size, const char *fmt, va_list list);
int _vasiprintf_r(struct _reent *reent, char **str,
    const char *fmt, va_list list);
char *_vasniprintf_r(struct _reent *reent, char *str,
    size_t *size, const char *fmt, va_list list);
```

Description

`viprintf`, `vfiprintf`, `vasiprintf`, `vsiprintf`, `vsniprintf`, and `vasniprintf` are (respectively) variants of `iprintf`, `fiprintf`, `asiprintf`, `siprintf`, `sniprintf`, and `asniprintf`. They differ only in allowing their caller to pass the variable argument list as a `va_list` object (initialized by `va_start`) rather than directly accepting a variable number of arguments. The caller is responsible for calling `va_end`.

`_viprintf_r`, `_vfiprintf_r`, `_vasiprintf_r`, `_vsiprintf_r`, `_vsniprintf_r`, and `_vasniprintf_r` are reentrant versions of the above.

Returns

The return values are consistent with the corresponding functions:

Portability

All of these functions are newlib extensions.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

4.75 viscanf, vfiscanf, vsiscanf—format argument list

Synopsis

```
#include <stdio.h>
#include <stdarg.h>
int viscanf(const char *fmt, va_list list);
int vfiscanf(FILE *fp, const char *fmt, va_list list);
int vsiscanf(const char *str, const char *fmt, va_list list);

int _viscanf_r(struct _reent *reent, const char *fmt,
    va_list list);
int _vfiscanf_r(struct _reent *reent, FILE *fp, const char *fmt,
    va_list list);
int _vsiscanf_r(struct _reent *reent, const char *str,
    const char *fmt, va_list list);
```

Description

viscanf, vfiscanf, and vsiscanf are (respectively) variants of iscanf, fiscanf, and siscanf. They differ only in allowing their caller to pass the variable argument list as a va_list object (initialized by va_start) rather than directly accepting a variable number of arguments.

Returns

The return values are consistent with the corresponding functions: viscanf returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields which were not stored.

If viscanf attempts to read at end-of-file, the return value is EOF.

If no fields were stored, the return value is 0.

The routines _viscanf_r, _vfiscanf_f, and _vsiscanf_r are reentrant versions which take an additional first parameter which points to the reentrancy structure.

Portability

These are newlib extensions.

Supporting OS subroutines required:

5 Strings and Memory (‘string.h’)

This chapter describes string-handling functions and functions for managing areas of memory. The corresponding declarations are in ‘string.h’.

5.1 bcmp—compare two memory areas

Synopsis

```
#include <strings.h>
int bcmp(const void *s1, const void *s2, size_t n);
```

Description

This function compares not more than *n* bytes of the object pointed to by *s1* with the object pointed to by *s2*.

This function is identical to `memcmp`.

Returns

The function returns an integer greater than, equal to or less than zero according to whether the object pointed to by *s1* is greater than, equal to or less than the object pointed to by *s2*.

Portability

`bcmp` requires no supporting OS subroutines.

5.2 bcopy—copy memory regions

Synopsis

```
#include <strings.h>
void bcopy(const void *in, void *out, size_t n);
```

Description

This function copies *n* bytes from the memory region pointed to by *in* to the memory region pointed to by *out*.

This function is implemented in term of `memmove`.

Portability

`bcopy` requires no supporting OS subroutines.

5.3 bzero—initialize memory to zero

Synopsis

```
#include <strings.h>
void bzero(void *b, size_t length);
```

Description
`bzero` initializes *length* bytes of memory, starting at address *b*, to zero.

Returns
`bzero` does not return a result.

Portability
`bzero` is in the Berkeley Software Distribution. Neither ANSI C nor the System V Interface Definition (Issue 2) require `bzero`.

`bzero` requires no supporting OS subroutines.

5.4 index—search for character in string

Synopsis

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

Description

This function finds the first occurence of *c* (converted to a char) in the string pointed to by *string* (including the terminating null character).

This function is identical to `strchr`.

Returns

Returns a pointer to the located character, or a null pointer if *c* does not occur in *string*.

Portability

`index` requires no supporting OS subroutines.

5.5 memccpy—copy memory regions with end-token check

Synopsis

```
#include <string.h>
void* memccpy(void *restrict out, const void *restrict in,
    int endchar, size_t n);
```

Description

This function copies up to *n* bytes from the memory region pointed to by *in* to the memory region pointed to by *out*. If a byte matching the *endchar* is encountered, the byte is copied and copying stops.

If the regions overlap, the behavior is undefined.

Returns

`memccpy` returns a pointer to the first byte following the *endchar* in the *out* region. If no byte matching *endchar* was copied, then `NULL` is returned.

Portability

`memccpy` is a GNU extension.

`memccpy` requires no supporting OS subroutines.

5.6 memchr—find character in memory

Synopsis

```
#include <string.h>
void *memchr(const void *src, int c, size_t length);
```

Description

This function searches memory starting at `*src` for the character *c*. The search only ends with the first occurrence of *c*, or after *length* characters; in particular, `NUL` does not terminate the search.

Returns

If the character *c* is found within *length* characters of `*src`, a pointer to the character is returned. If *c* is not found, then `NULL` is returned.

Portability

`memchr` is ANSI C.

`memchr` requires no supporting OS subroutines.

5.7 memcmp—compare two memory areas

Synopsis

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

Description

This function compares not more than *n* characters of the object pointed to by *s1* with the object pointed to by *s2*.

Returns

The function returns an integer greater than, equal to or less than zero according to whether the object pointed to by *s1* is greater than, equal to or less than the object pointed to by *s2*.

Portability

`memcmp` is ANSI C.

`memcmp` requires no supporting OS subroutines.

5.8 memcpy—copy memory regions

Synopsis

```
#include <string.h>
void* memcpy(void *restrict out, const void *restrict in,
    size_t n);
```

Description

This function copies *n* bytes from the memory region pointed to by *in* to the memory region pointed to by *out*.

If the regions overlap, the behavior is undefined.

Returns

`memcpy` returns a pointer to the first byte of the *out* region.

Portability

`memcpy` is ANSI C.

`memcpy` requires no supporting OS subroutines.

5.9 memmem—find memory segment

Synopsis

```
#include <string.h>
void *memmem(const void *s1, size_t l1, const void *s2,
    size_t l2);
```

Description

Locates the first occurrence in the memory region pointed to by *s1* with length *l1* of the sequence of bytes pointed to by *s2* of length *l2*. If you already know the lengths of your haystack and needle, `memmem` is much faster than `strstr`.

Returns

Returns a pointer to the located segment, or a null pointer if *s2* is not found. If *l2* is 0, *s1* is returned.

Portability

`memmem` is a newlib extension.

`memmem` requires no supporting OS subroutines.

5.10 memmove—move possibly overlapping memory

Synopsis

```
#include <string.h>
void *memmove(void *dst, const void *src, size_t length);
```

Description

This function moves *length* characters from the block of memory starting at `*src` to the memory starting at `*dst`. `memmove` reproduces the characters correctly at `*dst` even if the two areas overlap.

Returns

The function returns *dst* as passed.

Portability

`memmove` is ANSI C.

`memmove` requires no supporting OS subroutines.

5.11 mempcpy—copy memory regions and return end pointer

Synopsis

```
#include <string.h>
void* mempcpy(void *out, const void *in, size_t n);
```

Description

This function copies *n* bytes from the memory region pointed to by *in* to the memory region pointed to by *out*.

If the regions overlap, the behavior is undefined.

Returns

`mempcpy` returns a pointer to the byte following the last byte copied to the *out* region.

Portability

`mempcpy` is a GNU extension.

`mempcpy` requires no supporting OS subroutines.

5.12 memrchr—reverse search for character in memory

Synopsis

```
#include <string.h>
void *memrchr(const void *src, int c, size_t length);
```

Description

This function searches memory starting at *length* bytes beyond `*src` backwards for the character *c*. The search only ends with the first occurrence of *c*; in particular, `NUL` does not terminate the search.

Returns

If the character *c* is found within *length* characters of `*src`, a pointer to the character is returned. If *c* is not found, then `NULL` is returned.

Portability

`memrchr` is a GNU extension.

`memrchr` requires no supporting OS subroutines.

5.13 `memset`—set an area of memory

Synopsis

```
#include <string.h>
void *memset(void *dst, int c, size_t length);
```

Description

This function converts the argument *c* into an unsigned char and fills the first *length* characters of the array pointed to by *dst* to the value.

Returns

`memset` returns the value of *dst*.

Portability

`memset` is ANSI C.

`memset` requires no supporting OS subroutines.

5.14 rawmemchr—find character in memory

Synopsis

```
#include <string.h>
void *rawmemchr(const void *src, int c);
```

Description

This function searches memory starting at **src* for the character *c*. The search only ends with the first occurrence of *c*; in particular, NUL does not terminate the search. No bounds checking is performed, so this function should only be used when it is certain that the character *c* will be found.

Returns

A pointer to the first occurance of character *c*.

Portability

rawmemchr is a GNU extension.

rawmemchr requires no supporting OS subroutines.

5.15 `rindex`—reverse search for character in string

Synopsis

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

Description

This function finds the last occurence of *c* (converted to a char) in the string pointed to by *string* (including the terminating null character).

This function is identical to `strrchr`.

Returns

Returns a pointer to the located character, or a null pointer if *c* does not occur in *string*.

Portability

`rindex` requires no supporting OS subroutines.

5.16 stpcpy—copy string returning a pointer to its end

Synopsis

```
#include <string.h>
char *stpcpy(char *restrict dst, const char *restrict src);
```

Description

stpcpy copies the string pointed to by *src* (including the terminating null character) to the array pointed to by *dst*.

Returns

This function returns a pointer to the end of the destination string, thus pointing to the trailing ’\0’.

Portability

stpcpy is a GNU extension, candidate for inclusion into POSIX/SUSv4.

stpcpy requires no supporting OS subroutines.

5.17 stpncpy—counted copy string returning a pointer to its end

Synopsis

```
#include <string.h>
char *stpncpy(char *restrict dst, const char *restrict src,
    size_t length);
```

Description

`stpncpy` copies not more than *length* characters from the the string pointed to by *src* (including the terminating null character) to the array pointed to by *dst*. If the string pointed to by *src* is shorter than *length* characters, null characters are appended to the destination array until a total of *length* characters have been written.

Returns

This function returns a pointer to the end of the destination string, thus pointing to the trailing '\0', or, if the destination string is not null-terminated, pointing to dst + n.

Portability

`stpncpy` is a GNU extension, candidate for inclusion into POSIX/SUSv4.

`stpncpy` requires no supporting OS subroutines.

5.18 strcasecmp—case-insensitive character string compare

Synopsis

```
#include <strings.h>
int strcasecmp(const char *a, const char *b);
```

Description

`strcasecmp` compares the string at *a* to the string at *b* in a case-insensitive manner.

Returns

If `*a` sorts lexicographically after `*b` (after both are converted to lowercase), `strcasecmp` returns a number greater than zero. If the two strings match, `strcasecmp` returns zero. If `*a` sorts lexicographically before `*b`, `strcasecmp` returns a number less than zero.

Portability

`strcasecmp` is in the Berkeley Software Distribution.

`strcasecmp` requires no supporting OS subroutines. It uses tolower() from elsewhere in this library.

5.19 `strcasestr`—case-insensitive character string search

Synopsis

```
#include <string.h>
char *strcasestr(const char *s, const char *find);
```

Description

`strcasestr` searchs the string *s* for the first occurrence of the sequence *find*. `strcasestr` is identical to `strstr` except the search is case-insensitive.

Returns

A pointer to the first case-insensitive occurrence of the sequence *find* or `NULL` if no match was found.

Portability

`strcasestr` is in the Berkeley Software Distribution.

`strcasestr` requires no supporting OS subroutines. It uses tolower() from elsewhere in this library.

5.20 strcat—concatenate strings

Synopsis

```
#include <string.h>
char *strcat(char *restrict dst, const char *restrict src);
```

Description

strcat appends a copy of the string pointed to by *src* (including the terminating null character) to the end of the string pointed to by *dst*. The initial character of *src* overwrites the null character at the end of *dst*.

Returns

This function returns the initial value of *dst*

Portability

strcat is ANSI C.

strcat requires no supporting OS subroutines.

5.21 `strchr`—search for character in string

Synopsis

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

Description

This function finds the first occurence of *c* (converted to a char) in the string pointed to by *string* (including the terminating null character).

Returns

Returns a pointer to the located character, or a null pointer if *c* does not occur in *string*.

Portability

`strchr` is ANSI C.

`strchr` requires no supporting OS subroutines.

5.22 strchrnul—search for character in string

Synopsis

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

Description

This function finds the first occurence of *c* (converted to a char) in the string pointed to by *string* (including the terminating null character).

Returns

Returns a pointer to the located character, or a pointer to the concluding null byte if *c* does not occur in *string*.

Portability

`strchrnul` is a GNU extension.

`strchrnul` requires no supporting OS subroutines. It uses strchr() and strlen() from elsewhere in this library.

5.23 `strcmp`—character string compare

Synopsis

```
#include <string.h>
int strcmp(const char *a, const char *b);
```

Description

`strcmp` compares the string at *a* to the string at *b*.

Returns

If `*a` sorts lexicographically after `*b`, `strcmp` returns a number greater than zero. If the two strings match, `strcmp` returns zero. If `*a` sorts lexicographically before `*b`, `strcmp` returns a number less than zero.

Portability

`strcmp` is ANSI C.

`strcmp` requires no supporting OS subroutines.

5.24 strcoll—locale-specific character string compare

Synopsis

```
#include <string.h>
int strcoll(const char *stra, const char * strb);
```

Description

`strcoll` compares the string pointed to by *stra* to the string pointed to by *strb*, using an interpretation appropriate to the current `LC_COLLATE` state.

(NOT Cygwin:) The current implementation of `strcoll` simply uses `strcmp` and does not support any language-specific sorting.

Returns

If the first string is greater than the second string, `strcoll` returns a number greater than zero. If the two strings are equivalent, `strcoll` returns zero. If the first string is less than the second string, `strcoll` returns a number less than zero.

Portability

`strcoll` is ANSI C.

`strcoll` requires no supporting OS subroutines.

5.25 `strcpy`—copy string

Synopsis

```
#include <string.h>
char *strcpy(char *dst, const char *src);
```

Description

`strcpy` copies the string pointed to by *src* (including the terminating null character) to the array pointed to by *dst*.

Returns

This function returns the initial value of *dst*.

Portability

`strcpy` is ANSI C.

`strcpy` requires no supporting OS subroutines.

5.26 strcspn—count characters not in string

Synopsis

```
size_t strcspn(const char *s1, const char *s2);
```

Description

This function computes the length of the initial part of the string pointed to by *s1* which consists entirely of characters *NOT* from the string pointed to by *s2* (excluding the terminating null character).

Returns

`strcspn` returns the length of the substring found.

Portability

`strcspn` is ANSI C.

`strcspn` requires no supporting OS subroutines.

5.27 strerror, strerror_l—convert error number to string

Synopsis

```
#include <string.h>
char *strerror(int errnum);
char *strerror_l(int errnum, locale_t locale);
char *_strerror_r(struct _reent ptr, int errnum,
    int internal, int *error);
```

Description

`strerror` converts the error number *errnum* into a string. The value of *errnum* is usually a copy of `errno`. If `errnum` is not a known error number, the result points to an empty string.

`strerror_l` is like `strerror` but creates a string in a format as expected in locale *locale*. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

This implementation of `strerror` prints out the following strings for each of the values defined in '`errno.h`':

`0` Success

`E2BIG` Arg list too long

`EACCES` Permission denied

`EADDRINUSE`
Address already in use

`EADDRNOTAVAIL`
Address not available

`EADV` Advertise error

`EAFNOSUPPORT`
Address family not supported by protocol family

`EAGAIN` No more processes

`EALREADY` Socket already connected

`EBADF` Bad file number

`EBADMSG` Bad message

`EBUSY` Device or resource busy

`ECANCELED`
Operation canceled

`ECHILD` No children

`ECOMM` Communication error

`ECONNABORTED`
Software caused connection abort

`ECONNREFUSED`
Connection refused

`ECONNRESET`
: Connection reset by peer

`EDEADLK`
: Deadlock

`EDESTADDRREQ`
: Destination address required

`EEXIST`
: File exists

`EDOM`
: Mathematics argument out of domain of function

`EFAULT`
: Bad address

`EFBIG`
: File too large

`EHOSTDOWN`
: Host is down

`EHOSTUNREACH`
: Host is unreachable

`EIDRM`
: Identifier removed

`EILSEQ`
: Illegal byte sequence

`EINPROGRESS`
: Connection already in progress

`EINTR`
: Interrupted system call

`EINVAL`
: Invalid argument

`EIO`
: I/O error

`EISCONN`
: Socket is already connected

`EISDIR`
: Is a directory

`ELIBACC`
: Cannot access a needed shared library

`ELIBBAD`
: Accessing a corrupted shared library

`ELIBEXEC`
: Cannot exec a shared library directly

`ELIBMAX`
: Attempting to link in more shared libraries than system limit

`ELIBSCN`
: `.lib` section in a.out corrupted

`EMFILE`
: File descriptor value too large

`EMLINK`
: Too many links

`EMSGSIZE`
: Message too long

`EMULTIHOP`
: Multihop attempted

`ENAMETOOLONG`
: File or path name too long

`ENETDOWN`
: Network interface is not configured

`ENETRESET`
: Connection aborted by network

`ENETUNREACH`
: Network is unreachable

`ENFILE` Too many open files in system

`ENOBUFS` No buffer space available

`ENODATA` No data

`ENODEV` No such device

`ENOENT` No such file or directory

`ENOEXEC` Exec format error

`ENOLCK` No lock

`ENOLINK` Virtual circuit is gone

`ENOMEM` Not enough space

`ENOMSG` No message of desired type

`ENONET` Machine is not on the network

`ENOPKG` No package

`ENOPROTOOPT`
: Protocol not available

`ENOSPC` No space left on device

`ENOSR` No stream resources

`ENOSTR` Not a stream

`ENOSYS` Function not implemented

`ENOTBLK` Block device required

`ENOTCONN` Socket is not connected

`ENOTDIR` Not a directory

`ENOTEMPTY`
: Directory not empty

`ENOTRECOVERABLE`
: State not recoverable

`ENOTSOCK` Socket operation on non-socket

`ENOTSUP` Not supported

`ENOTTY` Not a character device

`ENXIO` No such device or address

`EOPNOTSUPP`
: Operation not supported on socket

EOVERFLOW
: Value too large for defined data type

EOWNERDEAD
: Previous owner died

EPERM
: Not owner

EPIPE
: Broken pipe

EPROTO
: Protocol error

EPROTOTYPE
: Protocol wrong type for socket

EPROTONOSUPPORT
: Unknown protocol

ERANGE
: Result too large

EREMOTE
: Resource is remote

EROFS
: Read-only file system

ESHUTDOWN
: Can’t send after socket shutdown

ESOCKTNOSUPPORT
: Socket type not supported

ESPIPE
: Illegal seek

ESRCH
: No such process

ESRMNT
: Srmount error

ESTRPIPE
: Strings pipe error

ETIME
: Stream ioctl timeout

ETIMEDOUT
: Connection timed out

ETXTBSY
: Text file busy

EWOULDBLOCK
: Operation would block (usually same as EAGAIN)

EXDEV
: Cross-device link

`_strerror_r` is a reentrant version of the above.

Returns
This function returns a pointer to a string. Your application must not modify that string.

Portability
ANSI C requires `strerror`, but does not specify the strings used for each error number.
`strerror_l` is POSIX-1.2008.

Although this implementation of `strerror` is reentrant (depending on `_user_strerror`), ANSI C declares that subsequent calls to `strerror` may overwrite the result string; therefore portable code cannot depend on the reentrancy of this subroutine.

Although this implementation of `strerror` guarantees a non-null result with a NUL-terminator, some implementations return `NULL` on failure. Although POSIX allows `strerror` to set `errno` to EINVAL on failure, this implementation does not do so (unless you provide `_user_strerror`).

POSIX recommends that unknown *errnum* result in a message including that value, however it is not a requirement and this implementation does not provide that information (unless you provide `_user_strerror`).

This implementation of `strerror` provides for user-defined extensibility. `errno.h` defines *__ELASTERROR*, which can be used as a base for user-defined error values. If the user supplies a routine named `_user_strerror`, and *errnum* passed to `strerror` does not match any of the supported values, `_user_strerror` is called with three arguments. The first is of type *int*, and is the *errnum* value unknown to `strerror`. The second is of type *int*, and matches the *internal* argument of `_strerror_r`; this should be zero if called from `strerror` and non-zero if called from any other function; `_user_strerror` can use this information to satisfy the POSIX rule that no other standardized function can overwrite a static buffer reused by `strerror`. The third is of type *int **, and matches the *error* argument of `_strerror_r`; if a non-zero value is stored into that location (usually *EINVAL*), then `strerror` will set `errno` to that value, and the XPG variant of `strerror_r` will return that value instead of zero or *ERANGE*. `_user_strerror` returns a *char ** value; returning *NULL* implies that the user function did not choose to handle *errnum*. The default `_user_strerror` returns *NULL* for all input values. Note that `_user_sterror` must be thread-safe, and only denote errors via the third argument rather than modifying `errno`, if `strerror` and `strerror_r` are are to comply with POSIX.

`strerror` requires no supporting OS subroutines.

5.28 strerror_r—convert error number to string and copy to buffer

Synopsis

```
#include <string.h>
#ifdef _GNU_SOURCE
char *strerror_r(int errnum, char *buffer, size_t n);
#else
int strerror_r(int errnum, char *buffer, size_t n);
#endif
```

Description

`strerror_r` converts the error number *errnum* into a string and copies the result into the supplied *buffer* for a length up to *n*, including the NUL terminator. The value of *errnum* is usually a copy of `errno`. If `errnum` is not a known error number, the result is the empty string.

See `strerror` for how strings are mapped to `errnum`.

Returns

There are two variants: the GNU version always returns a NUL-terminated string, which is *buffer* if all went well, but which is another pointer if *n* was too small (leaving *buffer* untouched). If the return is not *buffer*, your application must not modify that string. The POSIX version returns 0 on success, *EINVAL* if `errnum` was not recognized, and *ERANGE* if *n* was too small. The variant chosen depends on macros that you define before inclusion of `string.h`.

Portability

`strerror_r` with a *char ** result is a GNU extension. `strerror_r` with an *int* result is required by POSIX 2001. This function is compliant only if `_user_strerror` is not provided, or if it is thread-safe and uses separate storage according to whether the second argument of that function is non-zero. For more details on `_user_strerror`, see the `strerror` documentation.

POSIX states that the contents of *buf* are unspecified on error, although this implementation guarantees a NUL-terminated string for all except *n* of 0.

POSIX recommends that unknown *errnum* result in a message including that value, however it is not a requirement and this implementation provides only an empty string (unless you provide `_user_strerror`). POSIX also recommends that unknown *errnum* fail with EINVAL even when providing such a message, however it is not a requirement and this implementation will return success if `_user_strerror` provided a non-empty alternate string without assigning into its third argument.

`strerror_r` requires no supporting OS subroutines.

5.29 `strlen`—character string length

Synopsis

```
#include <string.h>
size_t strlen(const char *str);
```

Description

The `strlen` function works out the length of the string starting at `*str` by counting chararacters until it reaches a `NULL` character.

Returns

`strlen` returns the character count.

Portability

`strlen` is ANSI C.

`strlen` requires no supporting OS subroutines.

5.30 strlwr—force string to lowercase

Synopsis

```
#include <string.h>
char *strlwr(char *a);
```

Description

`strlwr` converts each character in the string at *a* to lowercase.

Returns

`strlwr` returns its argument, *a*.

Portability

`strlwr` is not widely portable.

`strlwr` requires no supporting OS subroutines.

5.31 strncasecmp—case-insensitive character string compare

Synopsis

```
#include <strings.h>
int strncasecmp(const char *a, const char * b, size_t length);
```

Description

`strncasecmp` compares up to *length* characters from the string at *a* to the string at *b* in a case-insensitive manner.

Returns

If `*a` sorts lexicographically after `*b` (after both are converted to lowercase), `strncasecmp` returns a number greater than zero. If the two strings are equivalent, `strncasecmp` returns zero. If `*a` sorts lexicographically before `*b`, `strncasecmp` returns a number less than zero.

Portability

`strncasecmp` is in the Berkeley Software Distribution.

`strncasecmp` requires no supporting OS subroutines. It uses tolower() from elsewhere in this library.

5.32 strncat—concatenate strings

Synopsis

```
#include <string.h>
char *strncat(char *restrict dst, const char *restrict src,
    size_t length);
```

Description

`strncat` appends not more than *length* characters from the string pointed to by *src* (including the terminating null character) to the end of the string pointed to by *dst*. The initial character of *src* overwrites the null character at the end of *dst*. A terminating null character is always appended to the result

Warnings

Note that a null is always appended, so that if the copy is limited by the *length* argument, the number of characters appended to *dst* is `n + 1`.

Returns

This function returns the initial value of *dst*

Portability

`strncat` is ANSI C.

`strncat` requires no supporting OS subroutines.

5.33 strncmp—character string compare

Synopsis

```
#include <string.h>
int strncmp(const char *a, const char * b, size_t length);
```

Description

`strncmp` compares up to *length* characters from the string at *a* to the string at *b*.

Returns

If `*a` sorts lexicographically after `*b`, `strncmp` returns a number greater than zero. If the two strings are equivalent, `strncmp` returns zero. If `*a` sorts lexicographically before `*b`, `strncmp` returns a number less than zero.

Portability

`strncmp` is ANSI C.

`strncmp` requires no supporting OS subroutines.

5.34 strncpy—counted copy string

Synopsis

```
#include <string.h>
char *strncpy(char *restrict dst, const char *restrict src,
    size_t length);
```

Description

`strncpy` copies not more than *length* characters from the the string pointed to by *src* (including the terminating null character) to the array pointed to by *dst*. If the string pointed to by *src* is shorter than *length* characters, null characters are appended to the destination array until a total of *length* characters have been written.

Returns

This function returns the initial value of *dst*.

Portability

`strncpy` is ANSI C.

`strncpy` requires no supporting OS subroutines.

5.35 `strnstr`—find string segment

Synopsis

```
#include <string.h>
size_t strnstr(const char *s1, const char *s2, size_t n);
```

Description

Locates the first occurrence in the string pointed to by *s1* of the sequence of limited to the *n* characters in the string pointed to by *s2*

Returns

Returns a pointer to the located string segment, or a null pointer if the string *s2* is not found. If *s2* points to a string with zero length, *s1* is returned.

Portability

`strnstr` is a BSD extension.

`strnstr` requires no supporting OS subroutines.

5.36 strnlen—character string length

Synopsis

```
#include <string.h>
size_t strnlen(const char *str, size_t n);
```

Description

The strnlen function works out the length of the string starting at *str by counting chararacters until it reaches a NUL character or the maximum: n number of characters have been inspected.

Returns

strnlen returns the character count or n.

Portability

strnlen is a GNU extension.

strnlen requires no supporting OS subroutines.

5.37 `strpbrk`—find characters in string

Synopsis

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

Description

This function locates the first occurence in the string pointed to by *s1* of any character in string pointed to by *s2* (excluding the terminating null character).

Returns

`strpbrk` returns a pointer to the character found in *s1*, or a null pointer if no character from *s2* occurs in *s1*.

Portability

`strpbrk` requires no supporting OS subroutines.

5.38 strrchr—reverse search for character in string

Synopsis

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

Description

This function finds the last occurence of *c* (converted to a char) in the string pointed to by *string* (including the terminating null character).

Returns

Returns a pointer to the located character, or a null pointer if *c* does not occur in *string*.

Portability

`strrchr` is ANSI C.

`strrchr` requires no supporting OS subroutines.

5.39 `strsignal`—convert signal number to string

Synopsis

```
#include <string.h>
char *strsignal(int signal);
```

Description

`strsignal` converts the signal number *signal* into a string. If *signal* is not a known signal number, the result will be of the form `"Unknown signal NN"` where NN is the *signal* is a decimal number.

Returns

This function returns a pointer to a string. Your application must not modify that string.

Portability

POSIX.1-2008 C requires `strsignal`, but does not specify the strings used for each signal number.

`strsignal` requires no supporting OS subroutines.

5.40 strspn—find initial match

Synopsis

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Description

This function computes the length of the initial segment of the string pointed to by *s1* which consists entirely of characters from the string pointed to by *s2* (excluding the terminating null character).

Returns

`strspn` returns the length of the segment found.

Portability

`strspn` is ANSI C.

`strspn` requires no supporting OS subroutines.

5.41 `strstr`—find string segment

Synopsis

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Description

Locates the first occurrence in the string pointed to by *s1* of the sequence of characters in the string pointed to by *s2* (excluding the terminating null character).

Returns

Returns a pointer to the located string segment, or a null pointer if the string *s2* is not found. If *s2* points to a string with zero length, *s1* is returned.

Portability

`strstr` is ANSI C.

`strstr` requires no supporting OS subroutines.

5.42 strtok, strtok_r, strsep—get next token from a string

Synopsis

```
#include <string.h>
char *strtok(char *restrict source,
    const char *restrict delimiters);
char *strtok_r(char *restrict source,
    const char *restrict delimiters,
    char **lasts);
char *strsep(char **source_ptr, const char *delimiters);
```

Description

The `strtok` function is used to isolate sequential tokens in a null-terminated string, `*source`. These tokens are delimited in the string by at least one of the characters in `*delimiters`. The first time that `strtok` is called, `*source` should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer instead. The separator string, `*delimiters`, 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 separator character itself with a null character. When no more tokens remain, a null pointer is returned.

The `strtok_r` function has the same behavior as `strtok`, except a pointer to placeholder `*lasts` must be supplied by the caller.

The `strsep` function is similar in behavior to `strtok`, except a pointer to the string pointer must be supplied `source_ptr` and the function does not skip leading delimiters. When the string starts with a delimiter, the delimiter is changed to the null character and the empty string is returned. Like `strtok_r` and `strtok`, the `*source_ptr` is updated to the next character following the last delimiter found or NULL if the end of string is reached with no more delimiters.

Returns

`strtok`, `strtok_r`, and `strsep` all return a pointer to the next token, or `NULL` if no more tokens can be found. For `strsep`, a token may be the empty string.

Notes

`strtok` is unsafe for multi-threaded applications. `strtok_r` and `strsep` are thread-safe and should be used instead.

Portability

`strtok` is ANSI C. `strtok_r` is POSIX. `strsep` is a BSD extension.

`strtok`, `strtok_r`, and `strsep` require no supporting OS subroutines.

5.43 strupr—force string to uppercase

Synopsis

```
#include <string.h>
char *strupr(char *a);
```

Description

`strupr` converts each character in the string at *a* to uppercase.

Returns

`strupr` returns its argument, *a*.

Portability

`strupr` is not widely portable.

`strupr` requires no supporting OS subroutines.

5.44 strverscmp—version string compare

Synopsis

```
#define _GNU_SOURCE
#include <string.h>
int strverscmp(const char *a, const char *b);
```

Description

strverscmp compares the string at *a* to the string at *b* in a version-logical order.

Returns

If *a version-sorts after *b, strverscmp returns a number greater than zero. If the two strings match, strverscmp returns zero. If *a version-sorts before *b, strverscmp returns a number less than zero.

Portability

strverscmp is a GNU extension.

strverscmp requires no supporting OS subroutines. It uses isdigit() from elsewhere in this library.

5.45 `strxfrm`—transform string

Synopsis

```
#include <string.h>
size_t strxfrm(char *restrict s1, const char *restrict s2,
    size_t n);
```

Description

This function transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is such that if the `strcmp` function is applied to the two transformed strings, it returns a value greater than, equal to, or less than zero, correspoinding to the result of a `strcoll` function applied to the same two original strings.

No more than *n* characters are placed into the resulting array pointed to by *s1*, including the terminating null character. If *n* is zero, *s1* may be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

(NOT Cygwin:) The current implementation of `strxfrm` simply copies the input and does not support any language-specific transformations.

Returns

The `strxfrm` function returns the length of the transformed string (not including the terminating null character). If the value returned is *n* or more, the contents of the array pointed to by *s1* are indeterminate.

Portability

`strxfrm` is ANSI C.

`strxfrm` requires no supporting OS subroutines.

5.46 swab—swap adjacent bytes

Synopsis

```
#include <unistd.h>
void swab(const void *in, void *out, ssize_t n);
```

Description

This function copies *n* bytes from the memory region pointed to by *in* to the memory region pointed to by *out*, exchanging adjacent even and odd bytes.

Portability

`swab` requires no supporting OS subroutines.

5.47 wcscasecmp—case-insensitive wide character string compare

Synopsis

```
#include <wchar.h>
int wcscasecmp(const wchar_t *a, const wchar_t *b);
```

Description

`wcscasecmp` compares the wide character string at *a* to the wide character string at *b* in a case-insensitive manner.

Returns

If `*a` sorts lexicographically after `*b` (after both are converted to uppercase), `wcscasecmp` returns a number greater than zero. If the two strings match, `wcscasecmp` returns zero. If `*a` sorts lexicographically before `*b`, `wcscasecmp` returns a number less than zero.

Portability

POSIX-1.2008

`wcscasecmp` requires no supporting OS subroutines. It uses tolower() from elsewhere in this library.

5.48 wcsdup—wide character string duplicate

Synopsis

```
#include <wchar.h>
wchar_t *wcsdup(const wchar_t *str);

#include <wchar.h>
wchar_t *_wcsdup_r(struct _reent *ptr, const wchar_t *str);
```

Description

`wcsdup` allocates a new wide character string using `malloc`, and copies the content of the argument *str* into the newly allocated string, thus making a copy of *str*.

Returns

`wcsdup` returns a pointer to the copy of *str* if enough memory for the copy was available. Otherwise it returns NULL and errno is set to ENOMEM.

Portability

POSIX-1.2008

5.49 wcsncasecmp—case-insensitive wide character string compare

Synopsis

```
#include <wchar.h>
int wcsncasecmp(const wchar_t *a, const wchar_t * b, size_t length);
```

Description

`wcsncasecmp` compares up to *length* wide characters from the string at *a* to the string at *b* in a case-insensitive manner.

Returns

If `*a` sorts lexicographically after `*b` (after both are converted to uppercase), `wcsncasecmp` returns a number greater than zero. If the two strings are equivalent, `wcsncasecmp` returns zero. If `*a` sorts lexicographically before `*b`, `wcsncasecmp` returns a number less than zero.

Portability

POSIX-1.2008

`wcsncasecmp` requires no supporting OS subroutines. It uses tolower() from elsewhere in this library.

6 Wide Character Strings (‘wchar.h’)

This chapter describes wide-character string-handling functions and managing areas of memory containing wide characters. The corresponding declarations are in ‘wchar.h’.

6.1 wmemchr—find a wide character in memory

Synopsis

```
#include <wchar.h>
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

Description

The `wmemchr` function locates the first occurrence of *c* in the initial *n* wide characters of the object pointed to be *s*. This function is not affected by locale and all wchar_t values are treated identically. The null wide character and wchar_t values not corresponding to valid characters are not treated specially.

If *n* is zero, *s* must be a valid pointer and the function behaves as if no valid occurrence of *c* is found.

Returns

The `wmemchr` function returns a pointer to the located wide character, or a null pointer if the wide character does not occur in the object.

Portability

`wmemchr` is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.2 wmemcmp—compare wide characters in memory

Synopsis

```
#include <wchar.h>
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

The wmemcmp function compares the first *n* wide characters of the object pointed to by *s1* to the first *n* wide characters of the object pointed to by *s2*. This function is not affected by locale and all wchar_t values are treated identically. The null wide character and wchar_t values not corresponding to valid characters are not treated specially.

If *n* is zero, *s1* and *s2* must be a valid pointers and the function behaves as if the two objects compare equal.

Returns

The wmemcmp function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*.

Portability

wmemcmp is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.3 wmemcpy—copy wide characters in memory

Synopsis

```
#include <wchar.h>
wchar_t *wmemcpy(wchar_t *__restrict d,
    const wchar_t *__restrict s, size_t n);
```

Description

The wmemcpy function copies *n* wide characters from the object pointed to by *s* to the object pointed to be *d*. This function is not affected by locale and all wchar_t values are treated identically. The null wide character and wchar_t values not corresponding to valid characters are not treated specially.

If *n* is zero, *d* and *s* must be a valid pointers, and the function copies zero wide characters.

Returns

The wmemcpy function returns the value of *d*.

Portability

wmemcpy is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.4 wmemmove—copy wide characters in memory with overlapping areas

Synopsis

```
#include <wchar.h>
wchar_t *wmemmove(wchar_t *d, const wchar_t *s, size_t n);
```

Description

The wmemmove function copies *n* wide characters from the object pointed to by *s* to the object pointed to by *d*. Copying takes place as if the *n* wide characters from the object pointed to by *s* are first copied into a temporary array of *n* wide characters that does not overlap the objects pointed to by *d* or *s*, and then the *n* wide characters from the temporary array are copied into the object pointed to by *d*.

This function is not affected by locale and all wchar_t values are treated identically. The null wide character and wchar_t values not corresponding to valid characters are not treated specially.

If *n* is zero, *d* and *s* must be a valid pointers, and the function copies zero wide characters.

Returns

The wmemmove function returns the value of *d*.

Portability

wmemmove is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.5 wmempcpy—copy wide characters in memory and return end pointer

Synopsis

```
#define _GNU_SOURCE
#include <wchar.h>
wchar_t *wmempcpy(wchar_t *d,
    const wchar_t *s, size_t n);
```

Description

The `wmemcpy` function copies *n* wide characters from the object pointed to by *s* to the object pointed to be *d*. This function is not affected by locale and all wchar_t values are treated identically. The null wide character and wchar_t values not corresponding to valid characters are not treated specially.

If *n* is zero, *d* and *s* must be a valid pointers, and the function copies zero wide characters.

Returns

`wmempcpy` returns a pointer to the wide character following the last wide character copied to the *out* region.

Portability

`wmempcpy` is a GNU extension.

No supporting OS subroutines are required.

6.6 wmemset—set wide characters in memory

Synopsis

```
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

Description

The wmemset function copies the value of *c* into each of the first *n* wide characters of the object pointed to by *s*. This function is not affected by locale and all wchar_t values are treated identically. The null wide character and wchar_t values not corresponding to valid characters are not treated specially.

If *n* is zero, *s* must be a valid pointer and the function copies zero wide characters.

Returns

The wmemset function returns the value of *s*.

Portability

wmemset is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.7 wcscat—concatenate two wide-character strings

Synopsis

```
#include <wchar.h>
wchar_t *wcscat(wchar_t *__restrict s1,
    const wchar_t *__restrict s2);
```

Description

The `wcscat` function appends a copy of the wide-character string pointed to by *s2* (including the terminating null wide-character code) to the end of the wide-character string pointed to by *s1*. The initial wide-character code of *s2* overwrites the null wide-character code at the end of *s1*. If copying takes place between objects that overlap, the behaviour is undefined.

Returns

The `wcscat` function returns *s1*; no return value is reserved to indicate an error.

Portability

`wcscat` is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.8 wcschr—wide-character string scanning operation

Synopsis

```
#include <wchar.h>
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

Description

The `wcschr` function locates the first occurrence of *c* in the wide-character string pointed to by *s*. The value of *c* must be a character representable as a type wchar_t and must be a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character string.

Returns

Upon completion, `wcschr` returns a pointer to the wide-character code, or a null pointer if the wide-character code is not found.

Portability

`wcschr` is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.9 wcscmp—compare two wide-character strings

Synopsis

```
#include <wchar.h>
int wcscmp(const wchar_t *s1, *s2);
```

Description

The `wcscmp` function compares the wide-character string pointed to by *s1* to the wide-character string pointed to by *s2*.

The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared.

Returns

Upon completion, `wcscmp` returns an integer greater than, equal to or less than 0, if the wide-character string pointed to by *s1* is greater than, equal to or less than the wide-character string pointed to by *s2* respectively.

Portability

`wcscmp` is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.10 wcscoll—locale-specific wide-character string compare

Synopsis

```
#include <wchar.h>
int wcscoll(const wchar_t *stra, const wchar_t * strb);
```

Description

`wcscoll` compares the wide-character string pointed to by *stra* to the wide-character string pointed to by *strb*, using an interpretation appropriate to the current `LC_COLLATE` state.

(NOT Cygwin:) The current implementation of `wcscoll` simply uses `wcscmp` and does not support any language-specific sorting.

Returns

If the first string is greater than the second string, `wcscoll` returns a number greater than zero. If the two strings are equivalent, `wcscoll` returns zero. If the first string is less than the second string, `wcscoll` returns a number less than zero.

Portability

`wcscoll` is ISO/IEC 9899/AMD1:1995 (ISO C).

6.11 wcscpy—copy a wide-character string

Synopsis

```
#include <wchar.h>
wchar_t *wcscpy(wchar_t *__restrict s1,
    const wchar_t *__restrict s2);
```

Description

The `wcscpy` function copies the wide-character string pointed to by *s2* (including the terminating null wide-character code) into the array pointed to by *s1*. If copying takes place between objects that overlap, the behaviour is undefined.

Returns

The `wcscpy` function returns *s1*; no return value is reserved to indicate an error.

Portability

`wcscpy` is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.12 wcpcpy—copy a wide-character string returning a pointer to its end

Synopsis

```
#include <wchar.h>
wchar_t *wcpcpy(wchar_t *s1, const wchar_t *s2);
```

Description

The wcpcpy function copies the wide-character string pointed to by *s2* (including the terminating null wide-character code) into the array pointed to by *s1*. If copying takes place between objects that overlap, the behaviour is undefined.

Returns

This function returns a pointer to the end of the destination string, thus pointing to the trailing ’\0’.

Portability

wcpcpy is a GNU extension.

No supporting OS subroutines are required.

6.13 wcscspn—get length of a complementary wide substring

Synopsis

```
#include <wchar.h>
size_t wcscspn(const wchar_t *s, wchar_t *set);
```

Description

The `wcscspn` function computes the length of the maximum initial segment of the wide-character string pointed to by *s* which consists entirely of wide-character codes not from the wide-character string pointed to by *set*.

Returns

The `wcscspn` function returns the length of the initial substring of *s1*; no return value is reserved to indicate an error.

Portability

`wcscspn` is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.14 wcsftime—convert date and time to a formatted wide-character string

Synopsis

```
#include <time.h>
#include <wchar.h>
size_t wcsftime(wchar_t *s, size_t maxsize,
    const wchar_t *format, const struct tm *timp);
```

Description

`wcsftime` is equivalent to `strftime`, except that:

- The argument s points to the initial element of an array of wide characters into which the generated output is to be placed.
- The argument maxsize indicates the limiting number of wide characters.
- The argument format is a wide-character string and the conversion specifiers are replaced by corresponding sequences of wide characters.
- The return value indicates the number of wide characters.

(The difference in all of the above being wide characters versus regular characters.) See `strftime` for the details of the format specifiers.

Returns

When the formatted time takes up no more than *maxsize* wide characters, the result is the length of the formatted wide string. Otherwise, if the formatting operation was abandoned due to lack of room, the result is `0`, and the wide-character string starting at *s* corresponds to just those parts of `*format` that could be completely filled in within the *maxsize* limit.

Portability

C99 and POSIX require `wcsftime`, but do not specify the contents of `*s` when the formatted string would require more than *maxsize* characters. Unrecognized specifiers and fields of `timp` that are out of range cause undefined results. Since some formats expand to 0 bytes, it is wise to set `*s` to a nonzero value beforehand to distinguish between failure and an empty string. This implementation does not support `s` being NULL, nor overlapping `s` and `format`.

`wcsftime` requires no supporting OS subroutines.

See Also

`strftime`

6.15 wcslcat—concatenate wide-character strings to specified length

Synopsis

```
#include <wchar.h>
size_t wcslcat(wchar_t *dst, const wchar_t *src, size_t siz);
```

Description

The `wcslcat` function appends wide characters from *src* to end of the *dst* wide-character string so that the resultant wide-character string is not more than *siz* wide characters including the terminating null wide-character code. A terminating null wide character is always added unless *siz* is 0. Thus, the maximum number of wide characters that can be appended from *src* is *siz* - 1. If copying takes place between objects that overlap, the behaviour is undefined.

Returns

Wide-character string length of initial *dst* plus the wide-character string length of *src* (does not include terminating null wide-characters). If the return value is greater than or equal to *siz*, then truncation occurred and not all wide characters from *src* were appended.

Portability

No supporting OS subroutines are required.

6.16 wcslcpy—copy a wide-character string to specified length

Synopsis

```
#include <wchar.h>
size_t wcslcpy(wchar_t *dst, const wchar_t *src, size_t siz);
```

Description

`wcslcpy` copies wide characters from *src* to *dst* such that up to *siz* - 1 characters are copied. A terminating null is appended to the result, unless *siz* is zero.

Returns

`wcslcpy` returns the number of wide characters in *src*, not including the terminating null wide character. If the return value is greater than or equal to *siz*, then not all wide characters were copied from *src* and truncation occurred.

Portability

No supporting OS subroutines are required.

6.17 `wcslen`—get wide-character string length

Synopsis

```
#include <wchar.h>
size_t wcslen(const wchar_t *s);
```

Description

The `wcslen` function computes the number of wide-character codes in the wide-character string to which *s* points, not including the terminating null wide-character code.

Returns

The `wcslen` function returns the length of *s*; no return value is reserved to indicate an error.

Portability

`wcslen` is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.18 wcsncat—concatenate part of two wide-character strings

Synopsis

```
#include <wchar.h>
wchar_t *wcsncat(wchar_t *__restrict s1,
    const wchar_t *__restrict s2, size_t n);
```

Description

The `wcsncat` function appends not more than *n* wide-character codes (a null wide-character code and wide-character codes that follow it are not appended) from the array pointed to by *s2* to the end of the wide-character string pointed to by *s1*. The initial wide-character code of *s2* overwrites the null wide-character code at the end of *s1*. A terminating null wide-character code is always appended to the result. If copying takes place between objects that overlap, the behaviour is undefined.

Returns

The `wcsncat` function returns *s1*; no return value is reserved to indicate an error.

Portability

`wcsncat` is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.19 wcsncmp—compare part of two wide-character strings

Synopsis

```
#include <wchar.h>
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

The `wcsncmp` function compares not more than *n* wide-character codes (wide-character codes that follow a null wide-character code are not compared) from the array pointed to by *s1* to the array pointed to by *s2*.

The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of wide-character codes that differ in the objects being compared.

Returns

Upon successful completion, `wcsncmp` returns an integer greater than, equal to or less than 0, if the possibly null-terminated array pointed to by *s1* is greater than, equal to or less than the possibly null-terminated array pointed to by *s2* respectively.

Portability

`wcsncmp` is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.20 wcsncpy—copy part of a wide-character string

Synopsis

```
#include <wchar.h>
wchar_t *wcsncpy(wchar_t *__restrict s1,
    const wchar_t *__restrict s2, size_t n);
```

Description

The `wcsncpy` function copies not more than *n* wide-character codes (wide-character codes that follow a null wide-character code are not copied) from the array pointed to by *s2* to the array pointed to by *s1*. If copying takes place between objects that overlap, the behaviour is undefined. Note that if *s1* contains more than *n* wide characters before its terminating null, the result is not null-terminated.

If the array pointed to by *s2* is a wide-character string that is shorter than *n* wide-character codes, null wide-character codes are appended to the copy in the array pointed to by *s1*, until *n* wide-character codes in all are written.

Returns

The `wcsncpy` function returns *s1*; no return value is reserved to indicate an error.

Portability

ISO/IEC 9899; POSIX.1.

No supporting OS subroutines are required.

6.21 wcpncpy—copy part of a wide-character string returning a pointer to its end

Synopsis

```
#include <wchar.h>
wchar_t *wcpncpy(wchar_t *__restrict s1,
    const wchar_t *__restrict s2, size_t n);
```

Description

The `wcpncpy` function copies not more than n wide-character codes (wide-character codes that follow a null wide-character code are not copied) from the array pointed to by *s2* to the array pointed to by *s1*. If copying takes place between objects that overlap, the behaviour is undefined.

If the array pointed to by *s2* is a wide-character string that is shorter than *n* wide-character codes, null wide-character codes are appended to the copy in the array pointed to by *s1*, until *n* wide-character codes in all are written.

Returns

The `wcpncpy` function returns *s1*; no return value is reserved to indicate an error.

Portability

`wcpncpy` is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.22 wcsnlen—get fixed-size wide-character string length

Synopsis

```
#include <wchar.h>
size_t wcsnlen(const wchar_t *s, size_t maxlen);
```

Description

The `wcsnlen` function computes the number of wide-character codes in the wide-character string pointed to by *s* not including the terminating L’\0’ wide character but at most *maxlen* wide characters.

Returns

`wcsnlen` returns the length of *s* if it is less then *maxlen*, or *maxlen* if there is no L’\0’ wide character in first *maxlen* characters.

Portability

`wcsnlen` is a GNU extension.

`wcsnlen` requires no supporting OS subroutines.

6.23 wcspbrk—-scan wide-character string for a wide-character code

Synopsis

```
#include <wchar.h>
wchar_t *wcspbrk(const wchar_t *s, const wchar_t *set);
```

Description

The `wcspbrk` function locates the first occurrence in the wide-character string pointed to by *s* of any wide-character code from the wide-character string pointed to by *set*.

Returns

Upon successful completion, `wcspbrk` returns a pointer to the wide-character code or a null pointer if no wide-character code from *set* occurs in *s*.

Portability

`wcspbrk` is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.24 wcsrchr—wide-character string scanning operation

Synopsis

```
#include <wchar.h>
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
```

Description

The `wcsrchr` function locates the last occurrence of *c* in the wide-character string pointed to by *s*. The value of *c* must be a character representable as a type wchar_t and must be a wide-character code corresponding to a valid character in the current locale. The terminating null wide-character code is considered to be part of the wide-character string.

Returns

Upon successful completion, `wcsrchr` returns a pointer to the wide-character code or a null pointer if *c* does not occur in the wide-character string.

Portability

`wcsrchr` is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.25 wcsspn—get length of a wide substring

Synopsis

```
#include <wchar.h>
size_t wcsspn(const wchar_t *s, const wchar_t *set);
```

Description

The `wcsspn` function computes the length of the maximum initial segment of the wide-character string pointed to by *s* which consists entirely of wide-character codes from the wide-character string pointed to by *set*.

Returns

The wcsspn() function returns the length *s1*; no return value is reserved to indicate an error.

Portability

`wcsspn` is ISO/IEC 9899/AMD1:1995 (ISO C).

No supporting OS subroutines are required.

6.26 wcsstr—find a wide-character substring

Synopsis

```
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *__restrict big,
    const wchar_t *__restrict little);
```

Description

The `wcsstr` function locates the first occurrence in the wide-character string pointed to by *big* of the sequence of wide characters (excluding the terminating null wide character) in the wide-character string pointed to by *little*.

Returns

On successful completion, `wcsstr` returns a pointer to the located wide-character string, or a null pointer if the wide-character string is not found.

If *little* points to a wide-character string with zero length, the function returns *big*.

Portability

`wcsstr` is ISO/IEC 9899/AMD1:1995 (ISO C).

6.27 wcstok—get next token from a string

Synopsis

```
#include <wchar.h>
wchar_t *wcstok(wchar_t *__restrict source,
    const wchar_t *__restrict delimiters,
    wchar_t **__restrict lasts);
```

Description

The `wcstok` function is the wide-character equivalent of the `strtok_r` function (which in turn is the same as the `strtok` function with an added argument to make it thread-safe).

The `wcstok` function is used to isolate (one at a time) sequential tokens in a null-terminated wide-character string, `*source`. A token is defined as a substring not containing any wide-characters from `*delimiters`.

The first time that `wcstok` is called, `*source` should be specified with the wide-character string to be searched, and `*lasts`–but not `lasts`, which must be non-NULL–may be random; subsequent calls, wishing to obtain further tokens from the same string, should pass a null pointer for `*source` instead but must supply `*lasts` unchanged from the last call. The separator wide-character string, `*delimiters`, must be supplied each time and may change between calls. A pointer to placeholder `*lasts` must be supplied by the caller, and is set each time as needed to save the state by `wcstok`. Every call to `wcstok` with `*source == NULL` must pass the value of `*lasts` as last set by `wcstok`.

The `wcstok` function returns a pointer to the beginning of each subsequent token in the string, after replacing the separator wide-character itself with a null wide-character. When no more tokens remain, a null pointer is returned.

Returns

`wcstok` returns a pointer to the first wide character of a token, or `NULL` if there is no token.

Notes

`wcstok` is thread-safe (unlike `strtok`, but like `strtok_r`). `wcstok` writes into the string being searched.

Portability

`wcstok` is C99 and POSIX.1-2001.

`wcstok` requires no supporting OS subroutines.

6.28 wcswidth—number of column positions of a wide-character string

Synopsis

```
#include <wchar.h>
int wcswidth(const wchar_t *pwcs, size_t n);
```

Description

The `wcswidth` function shall determine the number of column positions required for *n* wide-character codes (or fewer than *n* wide-character codes if a null wide-character code is encountered before *n* wide-character codes are exhausted) in the string pointed to by *pwcs*.

Returns

The `wcswidth` function either shall return 0 (if *pwcs* points to a null wide-character code), or return the number of column positions to be occupied by the wide-character string pointed to by *pwcs*, or return -1 (if any of the first *n* wide-character codes in the wide-character string pointed to by *pwcs* is not a printable wide-character code).

Portability

`wcswidth` has been introduced in the Single UNIX Specification Volume 2. `wcswidth` has been marked as an extension in the Single UNIX Specification Volume 3.

6.29 wcsxfrm—locale-specific wide-character string transformation

Synopsis

```
#include <wchar.h>
int wcsxfrm(wchar_t *__restrict stra,
    const wchar_t *__restrict strb, size_t n);
```

Description

`wcsxfrm` transforms the wide-character string pointed to by *strb* to the wide-character string pointed to by *stra*, Comparing two transformed wide strings with `wcscmp` should return the same result as comparing the original strings with `wcscoll`. No more than *n* wide characters are transformed, including the trailing null character.

If *n* is 0, *stra* may be a NULL pointer.

(NOT Cygwin:) The current implementation of `wcsxfrm` simply uses `wcslcpy` and does not support any language-specific transformations.

Returns

`wcsxfrm` returns the length of the transformed wide character string. if the return value is greater or equal to *n*, the content of *stra* is undefined.

Portability

`wcsxfrm` is ISO/IEC 9899/AMD1:1995 (ISO C).

6.30 wcwidth—number of column positions of a wide-character code

Synopsis

```
#include <wchar.h>
int wcwidth(const wint_t wc);
```

Description

The wcwidth function shall determine the number of column positions required for the wide character *wc*. The application shall ensure that the value of *wc* is a character representable as a wint_t (combining Unicode surrogate pairs into single 21-bit Unicode code points), and is a wide-character code corresponding to a valid character in the current locale.

Returns

The wcwidth function shall either return 0 (if *wc* is a null wide-character code), or return the number of column positions to be occupied by the wide-character code *wc*, or return -1 (if *wc* does not correspond to a printable wide-character code).

Portability

wcwidth has been introduced in the Single UNIX Specification Volume 2. wcwidth has been marked as an extension in the Single UNIX Specification Volume 3.

7 Signal Handling (‘signal.h’)

A *signal* is an event that interrupts the normal flow of control in your program. Your operating environment normally defines the full set of signals available (see ‘sys/signal.h’), as well as the default means of dealing with them—typically, either printing an error message and aborting your program, or ignoring the signal.

All systems support at least the following signals:

`SIGABRT` Abnormal termination of a program; raised by the `abort` function.

`SIGFPE` A domain error in arithmetic, such as overflow, or division by zero.

`SIGILL` Attempt to execute as a function data that is not executable.

`SIGINT` Interrupt; an interactive attention signal.

`SIGSEGV` An attempt to access a memory location that is not available.

`SIGTERM` A request that your program end execution.

Two functions are available for dealing with asynchronous signals—one to allow your program to send signals to itself (this is called *raising* a signal), and one to specify subroutines (called *handlers* to handle particular signals that you anticipate may occur—whether raised by your own program or the operating environment.

To support these functions, ‘signal.h’ defines three macros:

`SIG_DFL` Used with the `signal` function in place of a pointer to a handler subroutine, to select the operating environment's default handling of a signal.

`SIG_IGN` Used with the `signal` function in place of a pointer to a handler, to ignore a particular signal.

`SIG_ERR` Returned by the `signal` function in place of a pointer to a handler, to indicate that your request to set up a handler could not be honored for some reason.

‘signal.h’ also defines an integral type, `sig_atomic_t`. This type is not used in any function declarations; it exists only to allow your signal handlers to declare a static storage location where they may store a signal value. (Static storage is not otherwise reliable from signal handlers.)

7.1 psignal—print a signal message on standard error

Synopsis

```
#include <stdio.h>
void psignal(int signal, const char *prefix);
```

Description

Use `psignal` to print (on standard error) a signal message corresponding to the value of the signal number *signal*. Unless you use `NULL` as the value of the argument *prefix*, the signal message will begin with the string at *prefix*, followed by a colon and a space (`: `). The remainder of the signal message is one of the strings described for `strsignal`.

Returns

`psignal` returns no result.

Portability

POSIX.1-2008 requires `psignal`, but the strings issued vary from one implementation to another.

Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

7.2 raise—send a signal

Synopsis

```
#include <signal.h>
int raise(int sig);

int _raise_r(void *reent, int sig);
```

Description

Send the signal *sig* (one of the macros from ‘sys/signal.h’). This interrupts your program's normal flow of execution, and allows a signal handler (if you've defined one, using `signal`) to take control.

The alternate function `_raise_r` is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

The result is `0` if *sig* was successfully raised, `1` otherwise. However, the return value (since it depends on the normal flow of execution) may not be visible, unless the signal handler for *sig* terminates with a `return` or unless `SIG_IGN` is in effect for this signal.

Portability

ANSI C requires `raise`, but allows the full set of signal numbers to vary from one implementation to another.

Required OS subroutines: `getpid`, `kill`.

7.3 signal—specify handler subroutine for a signal

Synopsis

```
#include <signal.h>
void (*signal(int sig, void(*func)(int))) (int);

void (*_signal_r(void *reent, int sig, void(*func)(int))) (int);
```

Description

`signal` provides a simple signal-handling implementation for embedded targets.

`signal` allows you to request changed treatment for a particular signal *sig*. You can use one of the predefined macros `SIG_DFL` (select system default handling) or `SIG_IGN` (ignore this signal) as the value of *func*; otherwise, *func* is a function pointer that identifies a subroutine in your program as the handler for this signal.

Some of the execution environment for signal handlers is unpredictable; notably, the only library function required to work correctly from within a signal handler is `signal` itself, and only when used to redefine the handler for the current signal value.

Static storage is likewise unreliable for signal handlers, with one exception: if you declare a static storage location as '`volatile sig_atomic_t`', then you may use that location in a signal handler to store signal values.

If your signal handler terminates using `return` (or implicit return), your program's execution continues at the point where it was when the signal was raised (whether by your program itself, or by an external event). Signal handlers can also use functions such as `exit` and `abort` to avoid returning.

The alternate function `_signal_r` is the reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

Returns

If your request for a signal handler cannot be honored, the result is `SIG_ERR`; a specific error number is also recorded in `errno`.

Otherwise, the result is the previous handler (a function pointer or one of the predefined macros).

Portability

ANSI C requires `signal`.

No supporting OS subroutines are required to link with `signal`, but it will not have any useful effects, except for software generated signals, without an operating system that can actually raise exceptions.

8 Time Functions (‘time.h’)

This chapter groups functions used either for reporting on time (elapsed, current, or compute time) or to perform calculations based on time.

The header file ‘time.h’ defines three types. clock_t and time_t are both used for representations of time particularly suitable for arithmetic. (In this implementation, quantities of type clock_t have the highest resolution possible on your machine, and quantities of type time_t resolve to seconds.) size_t is also defined if necessary for quantities representing sizes.

‘time.h’ also defines the structure tm for the traditional representation of Gregorian calendar time as a series of numbers, with the following fields:

tm_sec Seconds, between 0 and 60 inclusive (60 allows for leap seconds).

tm_min Minutes, between 0 and 59 inclusive.

tm_hour Hours, between 0 and 23 inclusive.

tm_mday Day of the month, between 1 and 31 inclusive.

tm_mon Month, between 0 (January) and 11 (December).

tm_year Year (since 1900), can be negative for earlier years.

tm_wday Day of week, between 0 (Sunday) and 6 (Saturday).

tm_yday Number of days elapsed since last January 1, between 0 and 365 inclusive.

tm_isdst Daylight Savings Time flag: positive means DST in effect, zero means DST not in effect, negative means no information about DST is available. Although for mktime(), negative means that it should decide if DST is in effect or not.

8.1 asctime—format time as string

Synopsis

```
#include <time.h>
char *asctime(const struct tm *clock);
char *_asctime_r(const struct tm *clock, char *buf);
```

Description

Format the time value at *clock* into a string of the form

```
Wed Jun 15 11:38:07 1988\n\0
```

The string is generated in a static buffer; each call to `asctime` overwrites the string generated by previous calls.

Returns

A pointer to the string containing a formatted timestamp.

Portability

ANSI C requires `asctime`.

`asctime` requires no supporting OS subroutines.

8.2 clock—cumulative processor time

Synopsis

```
#include <time.h>
clock_t clock(void);
```

Description

Calculates the best available approximation of the cumulative amount of time used by your program since it started. To convert the result into seconds, divide by the macro CLOCKS_PER_SEC.

Returns

The amount of processor time used so far by your program, in units defined by the machine-dependent macro CLOCKS_PER_SEC. If no measurement is available, the result is (clock_t)-1.

Portability

ANSI C requires clock and CLOCKS_PER_SEC.

Supporting OS subroutine required: times.

8.3 `ctime`—convert time to local and format as string

Synopsis

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

Description

Convert the time value at *clock* to local time (like `localtime`) and format it into a string of the form

```
Wed Jun 15 11:38:07 1988\n\0
```

(like `asctime`).

Returns

A pointer to the string containing a formatted timestamp.

Portability

ANSI C requires `ctime`.

`ctime` requires no supporting OS subroutines.

8.4 difftime—subtract two times

Synopsis

```
#include <time.h>
double difftime(time_t tim1, time_t tim2);
```

Description
Subtracts the two times in the arguments: ‘*tim1* - *tim2*’.

Returns
The difference (in seconds) between *tim2* and *tim1*, as a `double`.

Portability
ANSI C requires `difftime`, and defines its result to be in seconds in all implementations.
`difftime` requires no supporting OS subroutines.

8.5 gmtime—convert time to UTC traditional form

Synopsis

```
#include <time.h>
struct tm *gmtime(const time_t *clock);
struct tm *gmtime_r(const time_t *clock, struct tm *res);
```

Description

`gmtime` takes the time at *clock* representing the number of elapsed seconds since 00:00:00 on January 1, 1970, Universal Coordinated Time (UTC, also known in some countries as GMT, Greenwich Mean time) and converts it to a `struct tm` representation.

`gmtime` constructs the traditional time representation in static storage; each call to `gmtime` or `localtime` will overwrite the information generated by previous calls to either function.

Returns

A pointer to the traditional time representation (`struct tm`).

Portability

ANSI C requires `gmtime`.

`gmtime` requires no supporting OS subroutines.

8.6 localtime—convert time to local representation

Synopsis

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

Description

`localtime` converts the time at *clock* into local time, then converts its representation from the arithmetic representation to the traditional representation defined by `struct tm`.

`localtime` constructs the traditional time representation in static storage; each call to `gmtime` or `localtime` will overwrite the information generated by previous calls to either function.

`mktime` is the inverse of `localtime`.

Returns

A pointer to the traditional time representation (`struct tm`).

Portability

ANSI C requires `localtime`.

`localtime` requires no supporting OS subroutines.

8.7 mktime—convert time to arithmetic representation

Synopsis

```
#include <time.h>
time_t mktime(struct tm *timp);
```

Description

`mktime` assumes the time at *timp* is a local time, and converts its representation from the traditional representation defined by `struct tm` into a representation suitable for arithmetic.

`localtime` is the inverse of `mktime`.

Returns

If the contents of the structure at *timp* do not form a valid calendar time representation, the result is `-1`. Otherwise, the result is the time, converted to a `time_t` value.

Portability

ANSI C requires `mktime`.

`mktime` requires no supporting OS subroutines.

8.8 strftime, strftime_l—convert date and time to a formatted string

Synopsis

```
#include <time.h>
size_t strftime(char *restrict s, size_t maxsize,
    const char *restrict format,
    const struct tm *restrict timp);
size_t strftime_l(char *restrict s, size_t maxsize,
    const char *restrict format,
    const struct tm *restrict timp,
    locale_t locale);
```

Description

`strftime` converts a `struct tm` representation of the time (at *timp*) into a null-terminated string, starting at *s* and occupying no more than *maxsize* characters.

`strftime_l` is like `strftime` but creates a string in a format as expected in locale *locale*. If *locale* is LC_GLOBAL_LOCALE or not a valid locale object, the behaviour is undefined.

You control the format of the output using the string at *format*. `*format` can contain two kinds of specifications: text to be copied literally into the formatted string, and time conversion specifications. Time conversion specifications are two- and three-character sequences beginning with ‘`%`’ (use ‘`%%`’ to include a percent sign in the output). Each defined conversion specification selects only the specified field(s) of calendar time data from `*timp`, and converts it to a string in one of the following ways:

`%a` The abbreviated weekday name according to the current locale. [tm_wday]

`%A` The full weekday name according to the current locale. In the default "C" locale, one of ‘`Sunday`’, ‘`Monday`’, ‘`Tuesday`’, ‘`Wednesday`’, ‘`Thursday`’, ‘`Friday`’, ‘`Saturday`’. [tm_wday]

`%b` The abbreviated month name according to the current locale. [tm_mon]

`%B` The full month name according to the current locale. In the default "C" locale, one of ‘`January`’, ‘`February`’, ‘`March`’, ‘`April`’, ‘`May`’, ‘`June`’, ‘`July`’, ‘`August`’, ‘`September`’, ‘`October`’, ‘`November`’, ‘`December`’. [tm_mon]

`%c` The preferred date and time representation for the current locale. [tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday]

`%C` The century, that is, the year divided by 100 then truncated. For 4-digit years, the result is zero-padded and exactly two characters; but for other years, there may a negative sign or more digits. In this way, ‘`%C%y`’ is equivalent to ‘`%Y`’. [tm_year]

`%d` The day of the month, formatted with two digits (from ‘`01`’ to ‘`31`’). [tm_mday]

`%D` A string representing the date, in the form ‘`"%m/%d/%y"`’. [tm_mday, tm_mon, tm_year]

`%e` The day of the month, formatted with leading space if single digit (from ‘`1`’ to ‘`31`’). [tm_mday]

`%Ex` In some locales, the E modifier selects alternative representations of certain modifiers `x`. In newlib, it is ignored, and treated as %`x`.

`%F` A string representing the ISO 8601:2000 date format, in the form '`"%Y-%m-%d"`'. [tm_mday, tm_mon, tm_year]

`%g` The last two digits of the week-based year, see specifier %G (from '`00`' to '`99`'). [tm_year, tm_wday, tm_yday]

`%G` The week-based year. In the ISO 8601:2000 calendar, week 1 of the year includes January 4th, and begin on Mondays. Therefore, if January 1st, 2nd, or 3rd falls on a Sunday, that day and earlier belong to the last week of the previous year; and if December 29th, 30th, or 31st falls on Monday, that day and later belong to week 1 of the next year. For consistency with %Y, it always has at least four characters. Example: "%G" for Saturday 2nd January 1999 gives "1998", and for Tuesday 30th December 1997 gives "1998". [tm_year, tm_wday, tm_yday]

`%h` Synonym for "%b". [tm_mon]

`%H` The hour (on a 24-hour clock), formatted with two digits (from '`00`' to '`23`'). [tm_hour]

`%I` The hour (on a 12-hour clock), formatted with two digits (from '`01`' to '`12`'). [tm_hour]

`%j` The count of days in the year, formatted with three digits (from '`001`' to '`366`'). [tm_yday]

`%k` The hour (on a 24-hour clock), formatted with leading space if single digit (from '`0`' to '`23`'). Non-POSIX extension (c.p. %I). [tm_hour]

`%l` The hour (on a 12-hour clock), formatted with leading space if single digit (from '`1`' to '`12`'). Non-POSIX extension (c.p. %H). [tm_hour]

`%m` The month number, formatted with two digits (from '`01`' to '`12`'). [tm_mon]

`%M` The minute, formatted with two digits (from '`00`' to '`59`'). [tm_min]

`%n` A newline character ('`\n`').

`%Ox` In some locales, the O modifier selects alternative digit characters for certain modifiers `x`. In newlib, it is ignored, and treated as %`x`.

`%p` Either '`AM`' or '`PM`' as appropriate, or the corresponding strings for the current locale. [tm_hour]

`%P` Same as '`%p`', but in lowercase. This is a GNU extension. [tm_hour]

`%r` Replaced by the time in a.m. and p.m. notation. In the "C" locale this is equivalent to "%I:%M:%S %p". In locales which don't define a.m./p.m. notations, the result is an empty string. [tm_sec, tm_min, tm_hour]

`%R` The 24-hour time, to the minute. Equivalent to "%H:%M". [tm_min, tm_hour]

`%s` The time elapsed, in seconds, since the start of the Unix epoch at 1970-01-01 00:00:00 UTC.

`%S` The second, formatted with two digits (from ‘`00`’ to ‘`60`’). The value 60 accounts for the occasional leap second. [tm_sec]

`%t` A tab character (‘`\t`’).

`%T` The 24-hour time, to the second. Equivalent to `"%H:%M:%S"`. [tm_sec, tm_min, tm_hour]

`%u` The weekday as a number, 1-based from Monday (from ‘`1`’ to ‘`7`’). [tm_wday]

`%U` The week number, where weeks start on Sunday, week 1 contains the first Sunday in a year, and earlier days are in week 0. Formatted with two digits (from ‘`00`’ to ‘`53`’). See also `%W`. [tm_wday, tm_yday]

`%V` The week number, where weeks start on Monday, week 1 contains January 4th, and earlier days are in the previous year. Formatted with two digits (from ‘`01`’ to ‘`53`’). See also `%G`. [tm_year, tm_wday, tm_yday]

`%w` The weekday as a number, 0-based from Sunday (from ‘`0`’ to ‘`6`’). [tm_wday]

`%W` The week number, where weeks start on Monday, week 1 contains the first Monday in a year, and earlier days are in week 0. Formatted with two digits (from ‘`00`’ to ‘`53`’). [tm_wday, tm_yday]

`%x` Replaced by the preferred date representation in the current locale. In the `"C"` locale this is equivalent to `"%m/%d/%y"`. [tm_mon, tm_mday, tm_year]

`%X` Replaced by the preferred time representation in the current locale. In the `"C"` locale this is equivalent to `"%H:%M:%S"`. [tm_sec, tm_min, tm_hour]

`%y` The last two digits of the year (from ‘`00`’ to ‘`99`’). [tm_year] (Implementation interpretation: always positive, even for negative years.)

`%Y` The full year, equivalent to `%C%y`. It will always have at least four characters, but may have more. The year is accurate even when tm_year added to the offset of 1900 overflows an int. [tm_year]

`%z` The offset from UTC. The format consists of a sign (negative is west of Greewich), two characters for hour, then two characters for minutes (-hhmm or +hhmm). If tm_isdst is negative, the offset is unknown and no output is generated; if it is zero, the offset is the standard offset for the current time zone; and if it is positive, the offset is the daylight savings offset for the current timezone. The offset is determined from the TZ environment variable, as if by calling tzset(). [tm_isdst]

`%Z` The time zone name. If tm_isdst is negative, no output is generated. Otherwise, the time zone name is based on the TZ environment variable, as if by calling tzset(). [tm_isdst]

`%%` A single character, ‘`%`’.

Returns

When the formatted time takes up no more than *maxsize* characters, the result is the length of the formatted string. Otherwise, if the formatting operation was abandoned due to lack

of room, the result is `0`, and the string starting at *s* corresponds to just those parts of `*format` that could be completely filled in within the *maxsize* limit.

Portability
ANSI C requires `strftime`, but does not specify the contents of `*s` when the formatted string would require more than *maxsize* characters. Unrecognized specifiers and fields of `timp` that are out of range cause undefined results. Since some formats expand to 0 bytes, it is wise to set `*s` to a nonzero value beforehand to distinguish between failure and an empty string. This implementation does not support `s` being NULL, nor overlapping `s` and `format`.

`strftime_l` is POSIX-1.2008.

`strftime` and `strftime_l` require no supporting OS subroutines.

Bugs
(NOT Cygwin:) `strftime` ignores the LC_TIME category of the current locale, hard-coding the "C" locale settings.

8.9 time—get current calendar time (as single number)

Synopsis

```
#include <time.h>
time_t time(time_t *t);
```

Description

time looks up the best available representation of the current time and returns it, encoded as a time_t. It stores the same value at *t* unless the argument is NULL.

Returns

A -1 result means the current time is not available; otherwise the result represents the current time.

Portability

ANSI C requires time.

Supporting OS subroutine required: Some implementations require gettimeofday.

8.10 `__tz_lock`, `__tz_unlock`—lock time zone global variables

Synopsis

```
#include "local.h"
void __tz_lock (void);
void __tz_unlock (void);
```

Description

The `tzset` facility functions call these functions when they need to ensure the values of global variables. The version of these routines supplied in the library use the lock API defined in sys/lock.h. If multiple threads of execution can call the time functions and give up scheduling in the middle, then you you need to define your own versions of these functions in order to safely lock the time zone variables during a call. If you do not, the results of `localtime`, `mktime`, `ctime`, and `strftime` are undefined.

The lock `__tz_lock` may not be called recursively; that is, a call `__tz_lock` will always lock all subsequent `__tz_lock` calls until the corresponding `__tz_unlock` call on the same thread is made.

8.11 tzset—set timezone characteristics from TZ environment variable

Synopsis

```
#include <time.h>
void tzset(void);
void _tzset_r (struct _reent *reent_ptr);
```

Description

`tzset` examines the TZ environment variable and sets up the three external variables: `_timezone`, `_daylight`, and `tzname`. The value of `_timezone` shall be the offset from the current time zone to GMT. The value of `_daylight` shall be 0 if there is no daylight savings time for the current time zone, otherwise it will be non-zero. The `tzname` array has two entries: the first is the name of the standard time zone, the second is the name of the daylight-savings time zone.

The TZ environment variable is expected to be in the following POSIX format:

stdoffset1[dst[offset2][,start[/time1],end[/time2]]]

where: std is the name of the standard time-zone (minimum 3 chars) offset1 is the value to add to local time to arrive at Universal time it has the form: hh[:mm[:ss]] dst is the name of the alternate (daylight-savings) time-zone (min 3 chars) offset2 is the value to add to local time to arrive at Universal time it has the same format as the std offset start is the day that the alternate time-zone starts time1 is the optional time that the alternate time-zone starts (this is in local time and defaults to 02:00:00 if not specified) end is the day that the alternate time-zone ends time2 is the time that the alternate time-zone ends (it is in local time and defaults to 02:00:00 if not specified)

Note that there is no white-space padding between fields. Also note that if TZ is null, the default is Universal GMT which has no daylight-savings time. If TZ is empty, the default EST5EDT is used.

The function `_tzset_r` is identical to `tzset` only it is reentrant and is used for applications that use multiple threads.

Returns

There is no return value.

Portability

`tzset` is part of the POSIX standard.

Supporting OS subroutine required: None

9 Locale (‘locale.h’)

A *locale* is the name for a collection of parameters (affecting collating sequences and formatting conventions) that may be different depending on location or culture. The "C" locale is the only one defined in the ANSI C standard.

This is a minimal implementation, supporting only the required "C" value for locale; strings representing other locales are not honored. ("" is also accepted; it represents the default locale for an implementation, here equivalent to "C").

‘locale.h’ defines the structure lconv to collect the information on a locale, with the following fields:

char *decimal_point
: The decimal point character used to format “ordinary” numbers (all numbers except those referring to amounts of money). "." in the C locale.

char *thousands_sep
: The character (if any) used to separate groups of digits, when formatting ordinary numbers. "" in the C locale.

char *grouping
: Specifications for how many digits to group (if any grouping is done at all) when formatting ordinary numbers. The *numeric value* of each character in the string represents the number of digits for the next group, and a value of 0 (that is, the string's trailing NULL) means to continue grouping digits using the last value specified. Use CHAR_MAX to indicate that no further grouping is desired. "" in the C locale.

char *int_curr_symbol
: The international currency symbol (first three characters), if any, and the character used to separate it from numbers. "" in the C locale.

char *currency_symbol
: The local currency symbol, if any. "" in the C locale.

char *mon_decimal_point
: The symbol used to delimit fractions in amounts of money. "" in the C locale.

char *mon_thousands_sep
: Similar to thousands_sep, but used for amounts of money. "" in the C locale.

char *mon_grouping
: Similar to grouping, but used for amounts of money. "" in the C locale.

char *positive_sign
: A string to flag positive amounts of money when formatting. "" in the C locale.

char *negative_sign
: A string to flag negative amounts of money when formatting. "" in the C locale.

char int_frac_digits
: The number of digits to display when formatting amounts of money to international conventions. CHAR_MAX (the largest number representable as a char) in the C locale.

`char frac_digits`
: The number of digits to display when formatting amounts of money to local conventions. `CHAR_MAX` in the C locale.

`char p_cs_precedes`
: `1` indicates the local currency symbol is used before a *positive or zero* formatted amount of money; `0` indicates the currency symbol is placed after the formatted number. `CHAR_MAX` in the C locale.

`char p_sep_by_space`
: `1` indicates the local currency symbol must be separated from *positive or zero* numbers by a space; `0` indicates that it is immediately adjacent to numbers. `CHAR_MAX` in the C locale.

`char n_cs_precedes`
: `1` indicates the local currency symbol is used before a *negative* formatted amount of money; `0` indicates the currency symbol is placed after the formatted number. `CHAR_MAX` in the C locale.

`char n_sep_by_space`
: `1` indicates the local currency symbol must be separated from *negative* numbers by a space; `0` indicates that it is immediately adjacent to numbers. `CHAR_MAX` in the C locale.

`char p_sign_posn`
: Controls the position of the *positive* sign for numbers representing money. `0` means parentheses surround the number; `1` means the sign is placed before both the number and the currency symbol; `2` means the sign is placed after both the number and the currency symbol; `3` means the sign is placed just before the currency symbol; and `4` means the sign is placed just after the currency symbol. `CHAR_MAX` in the C locale.

`char n_sign_posn`
: Controls the position of the *negative* sign for numbers representing money, using the same rules as `p_sign_posn`. `CHAR_MAX` in the C locale.

9.1 setlocale, localeconv—select or query locale

Synopsis

```
#include <locale.h>
char *setlocale(int category, const char *locale);
lconv *localeconv(void);

char *_setlocale_r(void *reent,
    int category, const char *locale);
lconv *_localeconv_r(void *reent);
```

Description

`setlocale` is the facility defined by ANSI C to condition the execution environment for international collating and formatting information; `localeconv` reports on the settings of the current locale.

This is a minimal implementation, supporting only the required `"POSIX"` and `"C"` values for *locale*; strings representing other locales are not honored unless _MB_CAPABLE is defined.

If _MB_CAPABLE is defined, POSIX locale strings are allowed, following the form

language[_TERRITORY][.charset][@modifier]

`"language"` is a two character string per ISO 639, or, if not available for a given language, a three character string per ISO 639-3. `"TERRITORY"` is a country code per ISO 3166. For `"charset"` and `"modifier"` see below.

Additionally to the POSIX specifier, the following extension is supported for backward compatibility with older implementations using newlib: `"C-charset"`. Instead of `"C-"`, you can also specify `"C."`. Both variations allow to specify language neutral locales while using other charsets than ASCII, for instance `"C.UTF-8"`, which keeps all settings as in the C locale, but uses the UTF-8 charset.

The following charsets are recognized: `"UTF-8"`, `"JIS"`, `"EUCJP"`, `"SJIS"`, `"KOI8-R"`, `"KOI8-U"`, `"GEORGIAN-PS"`, `"PT154"`, `"TIS-620"`, `"ISO-8859-x"` with 1 <= x <= 16, or `"CPxxx"` with xxx in [437, 720, 737, 775, 850, 852, 855, 857, 858, 862, 866, 874, 932, 1125, 1250, 1251, 1252, 1253, 1254, 1255, 1256, 1257, 1258].

Charsets are case insensitive. For instance, `"EUCJP"` and `"eucJP"` are equivalent. Charset names with dashes can also be written without dashes, as in `"UTF8"`, `"iso88591"` or `"koi8r"`. `"EUCJP"` and `"EUCKR"` are also recognized with dash, `"EUC-JP"` and `"EUC-KR"`.

Full support for all of the above charsets requires that newlib has been build with multibyte support and support for all ISO and Windows Codepage. Otherwise all singlebyte charsets are simply mapped to ASCII. Right now, only newlib for Cygwin is built with full charset support by default. Under Cygwin, this implementation additionally supports the charsets `"GBK"`, `"GB2312"`, `"eucCN"`, `"eucKR"`, and `"Big5"`. Cygwin does not support `"JIS"`.

Cygwin additionally supports locales from the file /usr/share/locale/locale.alias.

("" is also accepted; if given, the settings are read from the corresponding LC_* environment variables and $LANG according to POSIX rules.)

This implementation also supports the modifiers `"cjknarrow"` and `"cjkwide"`, which affect how the functions `wcwidth` and `wcswidth` handle characters from the "CJK Ambiguous Width" category of characters described at

http://www.unicode.org/reports/tr11/#Ambiguous. These characters have a width of 1 for singlebyte charsets and a width of 2 for multibyte charsets other than UTF-8. For UTF-8, their width depends on the language specifier: it is 2 for `"zh"` (Chinese), `"ja"` (Japanese), and `"ko"` (Korean), and 1 for everything else. Specifying `"cjknarrow"` or `"cjkwide"` forces a width of 1 or 2, respectively, independent of charset and language.

If you use `NULL` as the *locale* argument, `setlocale` returns a pointer to the string representing the current locale. The acceptable values for *category* are defined in '`locale.h`' as macros beginning with `"LC_"`.

`localeconv` returns a pointer to a structure (also defined in '`locale.h`') describing the locale-specific conventions currently in effect.

`_localeconv_r` and `_setlocale_r` are reentrant versions of `localeconv` and `setlocale` respectively. The extra argument *reent* is a pointer to a reentrancy structure.

Returns
A successful call to `setlocale` returns a pointer to a string associated with the specified category for the new locale. The string returned by `setlocale` is such that a subsequent call using that string will restore that category (or all categories in case of LC_ALL), to that state. The application shall not modify the string returned which may be overwritten by a subsequent call to `setlocale`. On error, `setlocale` returns `NULL`.

`localeconv` returns a pointer to a structure of type `lconv`, which describes the formatting and collating conventions in effect (in this implementation, always those of the C locale).

Portability
ANSI C requires `setlocale`, but the only locale required across all implementations is the C locale.

Notes
There is no ISO-8859-12 codepage. It's also refused by this implementation.

No supporting OS subroutines are required.

10 Reentrancy

Reentrancy is a characteristic of library functions which allows multiple processes to use the same address space with assurance that the values stored in those spaces will remain constant between calls. The Red Hat newlib implementation of the library functions ensures that whenever possible, these library functions are reentrant. However, there are some functions that can not be trivially made reentrant. Hooks have been provided to allow you to use these functions in a fully reentrant fashion.

These hooks use the structure `_reent` defined in '`reent.h`'. A variable defined as '`struct _reent`' is called a *reentrancy structure*. All functions which must manipulate global information are available in two versions. The first version has the usual name, and uses a single global instance of the reentrancy structure. The second has a different name, normally formed by prepending '`_`' and appending '`_r`', and takes a pointer to the particular reentrancy structure to use.

For example, the function `fopen` takes two arguments, *file* and *mode*, and uses the global reentrancy structure. The function `_fopen_r` takes the arguments, *struct_reent*, which is a pointer to an instance of the reentrancy structure, *file* and *mode*.

There are two versions of '`struct _reent`', a normal one and one for small memory systems, controlled by the `_REENT_SMALL` definition from the (automatically included) '`<sys/config.h>`'.

Each function which uses the global reentrancy structure uses the global variable `_impure_ptr`, which points to a reentrancy structure.

This means that you have two ways to achieve reentrancy. Both require that each thread of execution control initialize a unique global variable of type '`struct _reent`':

1. Use the reentrant versions of the library functions, after initializing a global reentrancy structure for each process. Use the pointer to this structure as the extra argument for all library functions.
2. Ensure that each thread of execution control has a pointer to its own unique reentrancy structure in the global variable `_impure_ptr`, and call the standard library subroutines.

The following functions are provided in both reentrant and non-reentrant versions.

Equivalent for errno variable:

```
_errno_r
```

Locale functions:

```
_localeconv_r  _setlocale_r
```

Equivalents for stdio variables:

```
_stdin_r        _stdout_r       _stderr_r
```

Stdio functions:

```
_fdopen_r       _perror_r       _tempnam_r
_fopen_r        _putchar_r      _tmpnam_r
_getchar_r      _puts_r         _tmpfile_r
_gets_r         _remove_r       _vfprintf_r
_iprintf_r      _rename_r       _vsnprintf_r
_mkstemp_r      _snprintf_r     _vsprintf_r
_mktemp_t       _sprintf_r
```

Signal functions:

```
_init_signal_r  _signal_r
_kill_r         __sigtramp_r
_raise_r
```

Stdlib functions:

```
_calloc_r       _mblen_r        _setenv_r
_dtoa_r         _mbstowcs_r     _srand_r
_free_r         _mbtowc_r       _strtod_r
_getenv_r       _memalign_r     _strtol_r
_mallinfo_r     _mstats_r       _strtoul_r
_malloc_r       _putenv_r       _system_r
_malloc_r       _rand_r         _wcstombs_r
_malloc_stats_r _realloc_r      _wctomb_r
```

String functions:

```
_strdup_r       _strtok_r
```

System functions:

```
_close_r        _link_r         _unlink_r
_execve_r       _lseek_r        _wait_r
_fcntl_r        _open_r         _write_r
_fork_r         _read_r
_fstat_r        _sbrk_r
_gettimeofday_r _stat_r
_getpid_r       _times_r
```

Time function:

```
_asctime_r
```

11 Miscellaneous Macros and Functions

This chapter describes miscellaneous routines not covered elsewhere.

11.1 `ffs`—find first bit set in a word

Synopsis

```
#include <strings.h>
int ffs(int word);
```

Description

`ffs` returns the first bit set in a word.

Returns
`ffs` returns 0 if *c* is 0, 1 if *c* is odd, 2 if *c* is a multiple of 2, etc.

Portability
`ffs` is not ANSI C.

No supporting OS subroutines are required.

11.2 __retarget_lock_init, __retarget_lock_init_recursive, __retarget_lock_close, __retarget_lock_close_recursive, __retarget_lock_acquire, __retarget_lock_acquire_recursive, __retarget_lock_try_acquire, __retarget_lock_try_acquire_recursive, __retarget_lock_release, __retarget_lock_release_recursive—locking routines

Synopsis

```
#include <lock.h>
struct __lock __lock___sinit_recursive_mutex;
struct __lock __lock___sfp_recursive_mutex;
struct __lock __lock___atexit_recursive_mutex;
struct __lock __lock___at_quick_exit_mutex;
struct __lock __lock___malloc_recursive_mutex;
struct __lock __lock___env_recursive_mutex;
struct __lock __lock___tz_mutex;
struct __lock __lock___dd_hash_mutex;
struct __lock __lock___arc4random_mutex;

void __retarget_lock_init (_LOCK_T * lock_ptr);
void __retarget_lock_init_recursive (_LOCK_T * lock_ptr);
void __retarget_lock_close (_LOCK_T lock);
void __retarget_lock_close_recursive (_LOCK_T lock);
void __retarget_lock_acquire (_LOCK_T lock);
void __retarget_lock_acquire_recursive (_LOCK_T lock);
int __retarget_lock_try_acquire (_LOCK_T lock);
int __retarget_lock_try_acquire_recursive (_LOCK_T lock);
void __retarget_lock_release (_LOCK_T lock);
void __retarget_lock_release_recursive (_LOCK_T lock);
```

Description

Newlib was configured to allow the target platform to provide the locking routines and static locks at link time. As such, a dummy default implementation of these routines and static locks is provided for single-threaded application to link successfully out of the box on bare-metal systems.

For multi-threaded applications the target platform is required to provide an implementation for **all** these routines and static locks. If some routines or static locks are missing, the link will fail with doubly defined symbols.

Portability

These locking routines and static lock are newlib-specific. Supporting OS subroutines are required for linking multi-threaded applications.

11.3 unctrl—get printable representation of a character

Synopsis

```
#include <unctrl.h>
char *unctrl(int c);
int unctrllen(int c);
```

Description

`unctrl` is a macro which returns the printable representation of *c* as a string. `unctrllen` is a macro which returns the length of the printable representation of *c*.

Returns

`unctrl` returns a string of the printable representation of *c*.

`unctrllen` returns the length of the string which is the printable representation of *c*.

Portability

`unctrl` and `unctrllen` are not ANSI C.

No supporting OS subroutines are required.

12 Overflow Protection

12.1 Stack Smashing Protection

Stack Smashing Protection is a compiler feature which emits extra code to check for stack smashing attacks. It depends on a canary, which is initialized with the process, and functions for process termination when an overflow is detected. These are private entry points intended solely for use by the compiler, and are used when any of the `-fstack-protector`, `-fstack-protector-all`, `-fstack-protector-explicit`, or `-fstack-protector-strong` compiler flags are enabled.

12.2 Object Size Checking

Object Size Checking is a feature which wraps certain functions with checks to prevent buffer overflows. These are enabled when compiling with optimization (`-O1` and higher) and `_FORTIFY_SOURCE` defined to 1, or for stricter checks, to 2.

The following functions use object size checking to detect buffer overflows when enabled:

String functions:

`bcopy`	`memmove`	`strcpy`
`bzero`	`mempcpy`	`strcat`
`explicit_bzero`	`memset`	`strncat`
`memcpy`	`stpcpy`	`strncpy`

Wide Character String functions:

`fgetws`	`wcrtomb`	`wcsrtombs`
`fgetws_unlocked`	`wcscat`	`wmemcpy`
`mbsnrtowcs`	`wcscpy`	`wmemmove`
`mbsrtowcs`	`wcsncat`	`wmempcpy`
`wcpcpy`	`wcsncpy`	`wmemset`
`wcpncpy`	`wcsnrtombs`	

Stdio functions:

`fgets`	`fread_unlocked`	`sprintf`
`fgets_unlocked`	`gets`	`vsnprintf`
`fread`	`snprintf`	`vsprintf`

Stdlib functions:

`mbstowcs`	`wcstombs`	`wctomb`

System functions:

`getcwd`	`read`	`ttyname_r`
`pread`	`readlink`	

13 System Calls

The C subroutine library depends on a handful of subroutine calls for operating system services. If you use the C library on a system that complies with the POSIX.1 standard (also known as IEEE 1003.1), most of these subroutines are supplied with your operating system.

If some of these subroutines are not provided with your system—in the extreme case, if you are developing software for a "bare board" system, without an OS—you will at least need to provide do-nothing stubs (or subroutines with minimal functionality) to allow your programs to link with the subroutines in `libc.a`.

13.1 Definitions for OS interface

This is the complete set of system definitions (primarily subroutines) required; the examples shown implement the minimal functionality required to allow `libc` to link, and fail gracefully where OS services are not available.

Graceful failure is permitted by returning an error code. A minor complication arises here: the C library must be compatible with development environments that supply fully functional versions of these subroutines. Such environments usually return error codes in a global `errno`. However, the Red Hat newlib C library provides a *macro* definition for `errno` in the header file '`errno.h`', as part of its support for reentrant routines (see Chapter 10 [Reentrancy], page 311).

The bridge between these two interpretations of `errno` is straightforward: the C library routines with OS interface calls capture the `errno` values returned globally, and record them in the appropriate field of the reentrancy structure (so that you can query them using the `errno` macro from '`errno.h`').

This mechanism becomes visible when you write stub routines for OS interfaces. You must include '`errno.h`', then disable the macro, like this:

```
#include <errno.h>
#undef errno
extern int errno;
```

The examples in this chapter include this treatment of `errno`.

`_exit` Exit a program without cleaning up files. If your system doesn't provide this, it is best to avoid linking with subroutines that require it (`exit`, `system`).

`close` Close a file. Minimal implementation:

```
int close(int file) {
  return -1;
}
```

`environ` A pointer to a list of environment variables and their values. For a minimal environment, this empty list is adequate:

```
char *__env[1] = { 0 };
char **environ = __env;
```

`execve` Transfer control to a new process. Minimal implementation (for a system without processes):

```
#include <errno.h>
#undef errno
extern int errno;
int execve(char *name, char **argv, char **env) {
  errno = ENOMEM;
  return -1;
}
```

`fork` Create a new process. Minimal implementation (for a system without processes):

```
#include <errno.h>
#undef errno
extern int errno;
int fork(void) {
  errno = EAGAIN;
  return -1;
}
```

`fstat` Status of an open file. For consistency with other minimal implementations in these examples, all files are regarded as character special devices. The '`sys/stat.h`' header file required is distributed in the '`include`' subdirectory for this C library.

```
#include <sys/stat.h>
int fstat(int file, struct stat *st) {
  st->st_mode = S_IFCHR;
  return 0;
}
```

`getpid` Process-ID; this is sometimes used to generate strings unlikely to conflict with other processes. Minimal implementation, for a system without processes:

```
int getpid(void) {
  return 1;
}
```

`isatty` Query whether output stream is a terminal. For consistency with the other minimal implementations, which only support output to `stdout`, this minimal implementation is suggested:

```
int isatty(int file) {
  return 1;
}
```

`kill` Send a signal. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int kill(int pid, int sig) {
  errno = EINVAL;
  return -1;
}
```

`link` Establish a new name for an existing file. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int link(char *old, char *new) {
  errno = EMLINK;
  return -1;
}
```

`lseek` Set position in a file. Minimal implementation:

```
int lseek(int file, int ptr, int dir) {
  return 0;
}
```

`open` Open a file. Minimal implementation:

```
int open(const char *name, int flags, int mode) {
  return -1;
}
```

`read` Read from a file. Minimal implementation:

```
int read(int file, char *ptr, int len) {
  return 0;
}
```

`sbrk` Increase program data space. As `malloc` and related functions depend on this, it is useful to have a working implementation. The following suffices for a standalone system; it exploits the symbol `_end` automatically defined by the GNU linker.

```
caddr_t sbrk(int incr) {
  extern char _end; /* Defined by the linker */
  static char *heap_end;
  char *prev_heap_end;

  if (heap_end == 0) {
    heap_end = &_end;
  }
  prev_heap_end = heap_end;
  if (heap_end + incr > stack_ptr) {
    write (1, "Heap and stack collision\n", 25);
    abort ();
  }

  heap_end += incr;
  return (caddr_t) prev_heap_end;
}
```

`stat` Status of a file (by name). Minimal implementation:

```
int stat(char *file, struct stat *st) {
```

```
  st->st_mode = S_IFCHR;
  return 0;
}
```

`times` Timing information for current process. Minimal implementation:

```
int times(struct tms *buf) {
  return -1;
}
```

`unlink` Remove a file's directory entry. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int unlink(char *name) {
  errno = ENOENT;
  return -1;
}
```

`wait` Wait for a child process. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int wait(int *status) {
  errno = ECHILD;
  return -1;
}
```

`write` Write to a file. '`libc`' subroutines will use this system routine for output to all files, *including* `stdout`—so if you need to generate any output, for example to a serial port for debugging, you should make your minimal `write` capable of doing this. The following minimal implementation is an incomplete example; it relies on a `outbyte` subroutine (not shown; typically, you must write this in assembler from examples provided by your hardware manufacturer) to actually perform the output.

```
int write(int file, char *ptr, int len) {
  int todo;

  for (todo = 0; todo < len; todo++) {
    outbyte (*ptr++);
  }
  return len;
}
```

13.2 Reentrant covers for OS subroutines

Since the system subroutines are used by other library routines that require reentrancy, '`libc.a`' provides cover routines (for example, the reentrant version of `fork` is `_fork_r`). These cover routines are consistent with the other reentrant subroutines in this library, and achieve reentrancy by using a reserved global data block (see Chapter 10 [Reentrancy], page 311).

13.2.1 _close_r—Reentrant version of close

Synopsis

```
#include <reent.h>
int _close_r(struct _reent *ptr, int fd);
```

Description

This is a reentrant version of `close`. It takes a pointer to the global data block, which holds `errno`.

13.2.2 _execve_r—Reentrant version of execve

Synopsis

```
#include <reent.h>
int _execve_r(struct _reent *ptr, const char *name,
    char *const argv[], char *const env[]);
```

Description

This is a reentrant version of `execve`. It takes a pointer to the global data block, which holds `errno`.

13.2.3 _fork_r—Reentrant version of fork

Synopsis

```
#include <reent.h>
int _fork_r(struct _reent *ptr);
```

Description

This is a reentrant version of `fork`. It takes a pointer to the global data block, which holds `errno`.

13.2.4 _wait_r—Reentrant version of wait

Synopsis

```
#include <reent.h>
int _wait_r(struct _reent *ptr, int *status);
```

Description

This is a reentrant version of `wait`. It takes a pointer to the global data block, which holds `errno`.

13.2.5 _fstat_r—Reentrant version of fstat

Synopsis

```
#include <reent.h>
int _fstat_r(struct _reent *ptr,
    int fd, struct stat *pstat);
```

Description

This is a reentrant version of `fstat`. It takes a pointer to the global data block, which holds `errno`.

13.2.6 `_link_r`—Reentrant version of link

Synopsis

```
#include <reent.h>
int _link_r(struct _reent *ptr,
    const char *old, const char *new);
```

Description

This is a reentrant version of `link`. It takes a pointer to the global data block, which holds `errno`.

13.2.7 `_lseek_r`—Reentrant version of lseek

Synopsis

```
#include <reent.h>
off_t _lseek_r(struct _reent *ptr,
    int fd, off_t pos, int whence);
```

Description

This is a reentrant version of `lseek`. It takes a pointer to the global data block, which holds `errno`.

13.2.8 _open_r—Reentrant version of open

Synopsis

```
#include <reent.h>
int _open_r(struct _reent *ptr,
    const char *file, int flags, int mode);
```

Description

This is a reentrant version of `open`. It takes a pointer to the global data block, which holds `errno`.

13.2.9 _read_r—Reentrant version of read

Synopsis

```
#include <reent.h>
_ssize_t _read_r(struct _reent *ptr,
    int fd, void *buf, size_t cnt);
```

Description

This is a reentrant version of `read`. It takes a pointer to the global data block, which holds `errno`.

13.2.10 _sbrk_r—Reentrant version of sbrk

Synopsis

```
#include <reent.h>
void *_sbrk_r(struct _reent *ptr, ptrdiff_t incr);
```

Description

This is a reentrant version of `sbrk`. It takes a pointer to the global data block, which holds `errno`.

13.2.11 _kill_r—Reentrant version of kill

Synopsis

```
#include <reent.h>
int _kill_r(struct _reent *ptr, int pid, int sig);
```

Description

This is a reentrant version of `kill`. It takes a pointer to the global data block, which holds `errno`.

13.2.12 `_getpid_r`—Reentrant version of getpid

Synopsis

```
#include <reent.h>
int _getpid_r(struct _reent *ptr);
```

Description

This is a reentrant version of `getpid`. It takes a pointer to the global data block, which holds `errno`.

We never need `errno`, of course, but for consistency we still must have the reentrant pointer argument.

13.2.13 _stat_r—Reentrant version of stat

Synopsis

```
#include <reent.h>
int _stat_r(struct _reent *ptr,
    const char *file, struct stat *pstat);
```

Description

This is a reentrant version of `stat`. It takes a pointer to the global data block, which holds `errno`.

13.2.14 _times_r—Reentrant version of times

Synopsis

```
#include <reent.h>
#include <sys/times.h>
clock_t _times_r(struct _reent *ptr, struct tms *ptms);
```

Description

This is a reentrant version of `times`. It takes a pointer to the global data block, which holds `errno`.

13.2.15 _unlink_r—Reentrant version of unlink

Synopsis

```
#include <reent.h>
int _unlink_r(struct _reent *ptr, const char *file);
```

Description

This is a reentrant version of `unlink`. It takes a pointer to the global data block, which holds `errno`.

13.2.16 `_write_r`—Reentrant version of write

Synopsis

```
#include <reent.h>
_ssize_t _write_r(struct _reent *ptr,
    int fd, const void *buf, size_t cnt);
```

Description

This is a reentrant version of `write`. It takes a pointer to the global data block, which holds `errno`.

14 Variable Argument Lists

The `printf` family of functions is defined to accept a variable number of arguments, rather than a fixed argument list. You can define your own functions with a variable argument list, by using macro definitions from either '`stdarg.h`' (for compatibility with ANSI C) or from '`varargs.h`' (for compatibility with a popular convention prior to ANSI C).

14.1 ANSI-standard macros, '`stdarg.h`'

In ANSI C, a function has a variable number of arguments when its parameter list ends in an ellipsis (`...`). The parameter list must also include at least one explicitly named argument; that argument is used to initialize the variable list data structure.

ANSI C defines three macros (`va_start`, `va_arg`, and `va_end`) to operate on variable argument lists. '`stdarg.h`' also defines a special type to represent variable argument lists: this type is called `va_list`.

14.1.1 Initialize variable argument list

Synopsis

```
#include <stdarg.h>
void va_start(va_list ap, rightmost);
```

Description

Use `va_start` to initialize the variable argument list *ap*, so that `va_arg` can extract values from it. *rightmost* is the name of the last explicit argument in the parameter list (the argument immediately preceding the ellipsis '`...`' that flags variable arguments in an ANSI C function header). You can only use `va_start` in a function declared using this ellipsis notation (not, for example, in one of its subfunctions).

Returns

`va_start` does not return a result.

Portability

ANSI C requires `va_start`.

14.1.2 Extract a value from argument list

Synopsis

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Description

`va_arg` returns the next unprocessed value from a variable argument list *ap* (which you must previously create with *va_start*). Specify the type for the value as the second parameter to the macro, *type*.

You may pass a `va_list` object *ap* to a subfunction, and use `va_arg` from the subfunction rather than from the function actually declared with an ellipsis in the header; however, in that case you may *only* use `va_arg` from the subfunction. ANSI C does not permit extracting successive values from a single variable-argument list from different levels of the calling stack.

There is no mechanism for testing whether there is actually a next argument available; you might instead pass an argument count (or some other data that implies an argument count) as one of the fixed arguments in your function call.

Returns

`va_arg` returns the next argument, an object of type *type*.

Portability

ANSI C requires `va_arg`.

14.1.3 Abandon a variable argument list

Synopsis

```
#include <stdarg.h>
void va_end(va_list ap);
```

Description

Use `va_end` to declare that your program will not use the variable argument list *ap* any further.

Returns

`va_end` does not return a result.

Portability

ANSI C requires `va_end`.

14.2 Traditional macros, '`varargs.h`'

If your C compiler predates ANSI C, you may still be able to use variable argument lists using the macros from the '`varargs.h`' header file. These macros resemble their ANSI counterparts, but have important differences in usage. In particular, since traditional C has no declaration mechanism for variable argument lists, two additional macros are provided simply for the purpose of defining functions with variable argument lists.

As with '`stdarg.h`', the type `va_list` is used to hold a data structure representing a variable argument list.

14.2.1 Declare variable arguments

Synopsis

```
#include <varargs.h>
function(va_alist)
va_dcl
```

Description

To use the '`varargs.h`' version of variable argument lists, you must declare your function with a call to the macro `va_alist` as its argument list, and use `va_dcl` as the declaration. *Do not use a semicolon after* `va_dcl`.

Returns

These macros cannot be used in a context where a return is syntactically possible.

Portability

va_alist and *va_dcl* were the most widespread method of declaring variable argument lists prior to ANSI C.

14.2.2 Initialize variable argument list

Synopsis

```
#include <varargs.h>
va_list ap;
va_start(ap);
```

Description

With the '`varargs.h`' macros, use `va_start` to initialize a data structure *ap* to permit manipulating a variable argument list. *ap* must have the type *va_alist*.

Returns

`va_start` does not return a result.

Portability

`va_start` is also defined as a macro in ANSI C, but the definitions are incompatible; the ANSI version has another parameter besides *ap*.

14.2.3 Extract a value from argument list

Synopsis

```
#include <varargs.h>
type va_arg(va_list ap, type);
```

Description

`va_arg` returns the next unprocessed value from a variable argument list *ap* (which you must previously create with *va_start*). Specify the type for the value as the second parameter to the macro, *type*.

Returns

`va_arg` returns the next argument, an object of type *type*.

Portability

The `va_arg` defined in '`varargs.h`' has the same syntax and usage as the ANSI C version from '`stdarg.h`'.

14.2.4 Abandon a variable argument list

Synopsis

```
#include <varargs.h>
va_end(va_list ap);
```

Description

Use `va_end` to declare that your program will not use the variable argument list *ap* any further.

Returns

`va_end` does not return a result.

Portability

The `va_end` defined in '`varargs.h`' has the same syntax and usage as the ANSI C version from '`stdarg.h`'.

Document Index

E

`errno` global vs macro 319
extra argument, reentrant fns 311

G

global reentrancy structure 311

L

linking the C library 319
list of overflow protected functions 317
list of reentrant functions 311

O

OS interface subroutines 319

R

reentrancy 311
reentrancy structure 311
reentrant function list 311

S

stubs 319
subroutines for OS interface 319

The body of this manual is set in
cmr10 at 10.95pt,
with headings in **cmb10 at 10.95pt**
and examples in `cmtt10 at 10.95pt`.
cmti10 at 10.95pt and
cmsl10 at 10.95pt
are used for emphasis.

Table of Contents

1 Introduction 1

2 Standard Utility Functions (‘stdlib.h’) 3

2.1 _Exit—end program execution with no cleanup processing 4
2.2 a64l, l64a—convert between radix-64 ASCII string and long ... 5
2.3 abort—abnormal termination of a program 6
2.4 abs—integer absolute value (magnitude) 7
2.5 assert—macro for debugging diagnostics 8
2.6 atexit—request execution of functions at program exit 9
2.7 atof, atoff—string to double or float 10
2.8 atoi, atol—string to integer 11
2.9 atoll—convert a string to a long long integer 12
2.10 bsearch—binary search 13
2.11 calloc—allocate space for arrays 14
2.12 div—divide two integers 15
2.13 ecvt, ecvtf, fcvt, fcvtf—double or float to string 16
2.14 gcvt, gcvtf—format double or float as string 17
2.15 ecvtbuf, fcvtbuf—double or float to string 18
2.16 __env_lock, __env_unlock—lock environ variable 19
2.17 exit—end program execution 20
2.18 getenv—look up environment variable 21
2.19 itoa—integer to string 22
2.20 labs—long integer absolute value 23
2.21 ldiv—divide two long integers 24
2.22 llabs—compute the absolute value of an long long integer.... 25
2.23 lldiv—divide two long long integers 26
2.24 malloc, realloc, free—manage memory 27
2.25 mallinfo, malloc_stats, mallopt—malloc support 29
2.26 __malloc_lock, __malloc_unlock—lock malloc pool 30
2.27 mblen—minimal multibyte length function 31
2.28 mbsrtowcs, mbsnrtowcs—convert a character string to a wide-character string 32
2.29 mbstowcs—minimal multibyte string to wide char converter .. 33
2.30 mbtowc—minimal multibyte to wide char converter 34
2.31 on_exit—request execution of function with argument at program exit 35
2.32 qsort—sort an array 36
2.33 rand, srand—pseudo-random numbers 37
2.34 random, srandom—pseudo-random numbers 38
2.35 rand48, drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48—pseudo-random number generators and initialization routines 39

2.36 `rpmatch`—determine whether response to question is affirmative or negative ... 41
2.37 `strtod`, `strtof`, `strtold`, `strtod_l`, `strtof_l`, `strtold_l`—string to double or float ... 42
2.38 `strtol`, `strtol_l`—string to long ... 44
2.39 `strtoll`, `strtoll_l`—string to long long ... 46
2.40 `strtoul`, `strtoul_l`—string to unsigned long ... 48
2.41 `strtoull`, `strtoull_l`—string to unsigned long long ... 50
2.42 `wcsrtombs`, `wcsnrtombs`—convert a wide-character string to a character string ... 52
2.43 `wcstod`, `wcstof`, `wcstold`, `wcstod_l`, `wcstof_l`, `wcstold_l`—wide char string to double or float ... 53
2.44 `wcstol`, `wcstol_l`—wide string to long ... 55
2.45 `wcstoll`, `wcstoll_l`—wide string to long long ... 57
2.46 `wcstoul`, `wcstoul_l`—wide string to unsigned long ... 59
2.47 `wcstoull`, `wcstoull_l`—wide string to unsigned long long ... 61
2.48 `system`—execute command string ... 63
2.49 `utoa`—unsigned integer to string ... 64
2.50 `wcstombs`—minimal wide char string to multibyte string converter ... 65
2.51 `wctomb`—minimal wide char to multibyte converter ... 66

3 Character Type Macros and Functions (‘`ctype.h`’) ... 67

3.1 `isalnum`, `isalnum_l`—alphanumeric character predicate ... 68
3.2 `isalpha`, `isalpha_l`—alphabetic character predicate ... 69
3.3 `isascii`, `isascii_l`—ASCII character predicate ... 70
3.4 `isblank`, `isblank_l`—blank character predicate ... 71
3.5 `iscntrl`, `iscntrl_l`—control character predicate ... 72
3.6 `isdigit`, `isdigit_l`—decimal digit predicate ... 73
3.7 `islower`, `islower_l`—lowercase character predicate ... 74
3.8 `isprint`, `isgraph`, `isprint_l`, `isgraph_l`—printable character predicates ... 75
3.9 `ispunct`, `ispunct_l`—punctuation character predicate ... 76
3.10 `isspace`, `isspace_l`—whitespace character predicate ... 77
3.11 `isupper`, `isupper_l`—uppercase character predicate ... 78
3.12 `isxdigit`, `isxdigit_l`—hexadecimal digit predicate ... 79
3.13 `toascii`, `toascii_l`—force integers to ASCII range ... 80
3.14 `tolower`, `tolower_l`—translate characters to lowercase ... 81
3.15 `toupper`, `toupper_l`—translate characters to uppercase ... 82
3.16 `iswalnum`, `iswalnum_l`—alphanumeric wide character test ... 83
3.17 `iswalpha`, `iswalpha_l`—alphabetic wide character test ... 84
3.18 `iswcntrl`, `iswcntrl_l`—control wide character test ... 85
3.19 `iswblank`, `iswblank_l`—blank wide character test ... 86
3.20 `iswdigit`, `iswdigit_l`—decimal digit wide character test ... 87
3.21 `iswgraph`, `iswgraph_l`—graphic wide character test ... 88
3.22 `iswlower`, `iswlower_l`—lowercase wide character test ... 89
3.23 `iswprint`, `iswprint_l`—printable wide character test ... 90

3.24 `iswpunct`, `iswpunct_l`—punctuation wide character test 91
3.25 `iswspace`, `iswspace_l`—whitespace wide character test 92
3.26 `iswupper`, `iswupper_l`—uppercase wide character test 93
3.27 `iswxdigit`, `iswxdigit_l`—hexadecimal digit wide character test 94
3.28 `iswctype`, `iswctype_l`—extensible wide-character test 95
3.29 `wctype`, `wctype_l`—get wide-character classification type 96
3.30 `towlower`, `towlower_l`—translate wide characters to lowercase 97
3.31 `towupper`, `towupper_l`—translate wide characters to uppercase 98
3.32 `towctrans`, `towctrans_l`—extensible wide-character translation 99
3.33 `wctrans`, `wctrans_l`—get wide-character translation type ... 100

4 Input and Output (‘`stdio.h`’) 101

4.1 `clearerr`, `clearerr_unlocked`—clear file or stream error indicator 102
4.2 `diprintf`, `vdiprintf`—print to a file descriptor (integer only) 103
4.3 `dprintf`, `vdprintf`—print to a file descriptor 104
4.4 `fclose`—close a file 105
4.5 `fcloseall`—close all files 106
4.6 `fdopen`—turn open file into a stream 107
4.7 `feof`, `feof_unlocked`—test for end of file 108
4.8 `ferror`, `ferror_unlocked`—test whether read/write error has occurred 109
4.9 `fflush`, `fflush_unlocked`—flush buffered file output 110
4.10 `fgetc`, `fgetc_unlocked`—get a character from a file or stream 111
4.11 `fgetpos`—record position in a stream or file 112
4.12 `fgets`, `fgets_unlocked`—get character string from a file or stream 113
4.13 `fgetwc`, `getwc`, `fgetwc_unlocked`, `getwc_unlocked`—get a wide character from a file or stream 114
4.14 `fgetws`, `fgetws_unlocked`—get wide character string from a file or stream 116
4.15 `fileno`, `fileno_unlocked`—return file descriptor associated with stream 117
4.16 `fmemopen`—open a stream around a fixed-length string 118
4.17 `fopen`—open a file 119
4.18 `fopencookie`—open a stream with custom callbacks 121
4.19 `fpurge`—discard pending file I/O 123
4.20 `fputc`, `fputc_unlocked`—write a character on a stream or file 124
4.21 `fputs`, `fputs_unlocked`—write a character string in a file or stream 125

4.22 fputwc, putwc, fputwc_unlocked, putwc_unlocked—write a wide character on a stream or file 126
4.23 fputws, fputws_unlocked—write a wide character string in a file or stream 128
4.24 fread, fread_unlocked—read array elements from a file 129
4.25 freopen—open a file using an existing file descriptor 130
4.26 fseek, fseeko—set file position 131
4.27 __fsetlocking—set or query locking mode on FILE stream 132
4.28 fsetpos—restore position of a stream or file 133
4.29 ftell, ftello—return position in a stream or file 134
4.30 funopen, fropen, fwopen—open a stream with custom callbacks 135
4.31 fwide—set and determine the orientation of a FILE stream 137
4.32 fwrite, fwrite_unlocked—write array elements 138
4.33 getc—read a character (macro) 139
4.34 getc_unlocked—non-thread-safe version of getc (macro) 140
4.35 getchar—read a character (macro) 141
4.36 getchar_unlocked—non-thread-safe version of getchar (macro) 142
4.37 getdelim—read a line up to a specified line delimiter 143
4.38 getline—read a line from a file 144
4.39 gets—get character string (obsolete, use fgets instead) 145
4.40 getw—read a word (int) 146
4.41 getwchar, getwchar_unlocked—read a wide character from standard input 147
4.42 mktemp, mkstemp, mkostemp, mkstemps, ... 148
4.43 open_memstream, open_wmemstream—open a write stream around an arbitrary-length string 150
4.44 perror—print an error message on standard error 151
4.45 putc—write a character (macro) 152
4.46 putc_unlocked—non-thread-safe version of putc (macro) 153
4.47 putchar—write a character (macro) 154
4.48 putchar_unlocked—non-thread-safe version of putchar (macro) 155
4.49 puts—write a character string 156
4.50 putw—write a word (int) 157
4.51 putwchar, putwchar_unlocked—write a wide character to standard output 158
4.52 remove—delete a file's name 159
4.53 rename—rename a file 160
4.54 rewind—reinitialize a file or stream 161
4.55 setbuf—specify full buffering for a file or stream 162
4.56 setbuffer—specify full buffering for a file or stream with size 163
4.57 setlinebuf—specify line buffering for a file or stream 164
4.58 setvbuf—specify file or stream buffering 165

4.59 siprintf, fiprintf, iprintf, sniprintf, asiprintf, asniprintf—format output (integer only) ... 166
4.60 siscanf, fiscanf, iscanf—scan and format non-floating input ... 167
4.61 sprintf, fprintf, printf, snprintf, asprintf, asnprintf—format output ... 168
4.62 sscanf, fscanf, scanf—scan and format input ... 175
4.63 stdio_ext,__fbufsize,__fpending,__flbf,__freadable,__fwritable,__freading,__fwriting—access internals of FILE structure ... 179
4.64 swprintf, fwprintf, wprintf—wide character format output ... 180
4.65 swscanf, fwscanf, wscanf—scan and format wide character input ... 186
4.66 tmpfile—create a temporary file ... 190
4.67 tmpnam, tempnam—name for a temporary file ... 191
4.68 ungetc—push data back into a stream ... 192
4.69 ungetwc—push wide character data back into a stream ... 193
4.70 vfprintf, vprintf, vsprintf, vsnprintf, vasprintf, vasnprintf—format argument list ... 194
4.71 vfscanf, vscanf, vsscanf—format argument list ... 195
4.72 vfwprintf, vwprintf, vswprintf—wide character format argument list ... 196
4.73 vfwscanf, vwscanf, vswscanf—scan and format argument list from wide character input ... 197
4.74 viprintf, vfiprintf, vsiprintf, vsniprintf, vasiprintf, vasniprintf—format argument list (integer only) ... 198
4.75 viscanf, vfiscanf, vsiscanf—format argument list ... 199

5 Strings and Memory (‘string.h’) ... 201

5.1 bcmp—compare two memory areas ... 202
5.2 bcopy—copy memory regions ... 203
5.3 bzero—initialize memory to zero ... 204
5.4 index—search for character in string ... 205
5.5 memccpy—copy memory regions with end-token check ... 206
5.6 memchr—find character in memory ... 207
5.7 memcmp—compare two memory areas ... 208
5.8 memcpy—copy memory regions ... 209
5.9 memmem—find memory segment ... 210
5.10 memmove—move possibly overlapping memory ... 211
5.11 mempcpy—copy memory regions and return end pointer ... 212
5.12 memrchr—reverse search for character in memory ... 213
5.13 memset—set an area of memory ... 214
5.14 rawmemchr—find character in memory ... 215
5.15 rindex—reverse search for character in string ... 216
5.16 stpcpy—copy string returning a pointer to its end ... 217
5.17 stpncpy—counted copy string returning a pointer to its end ... 218

5.18 strcasecmp—case-insensitive character string compare 219
5.19 strcasestr—case-insensitive character string search 220
5.20 strcat—concatenate strings 221
5.21 strchr—search for character in string 222
5.22 strchrnul—search for character in string 223
5.23 strcmp—character string compare 224
5.24 strcoll—locale-specific character string compare 225
5.25 strcpy—copy string 226
5.26 strcspn—count characters not in string 227
5.27 strerror, strerror_l—convert error number to string 228
5.28 strerror_r—convert error number to string and copy to buffer 233
5.29 strlen—character string length 234
5.30 strlwr—force string to lowercase 235
5.31 strncasecmp—case-insensitive character string compare 236
5.32 strncat—concatenate strings 237
5.33 strncmp—character string compare 238
5.34 strncpy—counted copy string 239
5.35 strnstr—find string segment 240
5.36 strnlen—character string length 241
5.37 strpbrk—find characters in string 242
5.38 strrchr—reverse search for character in string 243
5.39 strsignal—convert signal number to string 244
5.40 strspn—find initial match 245
5.41 strstr—find string segment 246
5.42 strtok, strtok_r, strsep—get next token from a string 247
5.43 strupr—force string to uppercase 248
5.44 strverscmp—version string compare 249
5.45 strxfrm—transform string 250
5.46 swab—swap adjacent bytes 251
5.47 wcscasecmp—case-insensitive wide character string compare 252
5.48 wcsdup—wide character string duplicate 253
5.49 wcsncasecmp—case-insensitive wide character string compare 254

6 Wide Character Strings ('wchar.h') 255

6.1 wmemchr—find a wide character in memory 256
6.2 wmemcmp—compare wide characters in memory 257
6.3 wmemcpy—copy wide characters in memory 258
6.4 wmemmove—copy wide characters in memory with overlapping areas 259
6.5 wmempcpy—copy wide characters in memory and return end pointer 260
6.6 wmemset—set wide characters in memory 261
6.7 wcscat—concatenate two wide-character strings 262
6.8 wcschr—wide-character string scanning operation 263
6.9 wcscmp—compare two wide-character strings 264

6.10 `wcscoll`—locale-specific wide-character string compare 265
6.11 `wcscpy`—copy a wide-character string 266
6.12 `wcpcpy`—copy a wide-character string returning a pointer to its end 267
6.13 `wcscspn`—get length of a complementary wide substring 268
6.14 `wcsftime`—convert date and time to a formatted wide-character string 269
6.15 `wcslcat`—concatenate wide-character strings to specified length 270
6.16 `wcslcpy`—copy a wide-character string to specified length ... 271
6.17 `wcslen`—get wide-character string length 272
6.18 `wcsncat`—concatenate part of two wide-character strings 273
6.19 `wcsncmp`—compare part of two wide-character strings 274
6.20 `wcsncpy`—copy part of a wide-character string 275
6.21 `wcpncpy`—copy part of a wide-character string returning a pointer to its end 276
6.22 `wcsnlen`—get fixed-size wide-character string length 277
6.23 `wcspbrk`—-scan wide-character string for a wide-character code 278
6.24 `wcsrchr`—wide-character string scanning operation 279
6.25 `wcsspn`—get length of a wide substring 280
6.26 `wcsstr`—find a wide-character substring 281
6.27 `wcstok`—get next token from a string 282
6.28 `wcswidth`—number of column positions of a wide-character string 283
6.29 `wcsxfrm`—locale-specific wide-character string transformation 284
6.30 `wcwidth`—number of column positions of a wide-character code 285

7 Signal Handling (‘`signal.h`’) 287

7.1 `psignal`—print a signal message on standard error 288
7.2 `raise`—send a signal 289
7.3 `signal`—specify handler subroutine for a signal 290

8 Time Functions (‘`time.h`’) 291

8.1 `asctime`—format time as string 292
8.2 `clock`—cumulative processor time 293
8.3 `ctime`—convert time to local and format as string 294
8.4 `difftime`—subtract two times 295
8.5 `gmtime`—convert time to UTC traditional form 296
8.6 `localtime`—convert time to local representation 297
8.7 `mktime`—convert time to arithmetic representation 298
8.8 `strftime`, `strftime_l`—convert date and time to a formatted string 299
8.9 `time`—get current calendar time (as single number) 303
8.10 `__tz_lock`, `__tz_unlock`—lock time zone global variables ... 304

8.11 tzset—set timezone characteristics from TZ environment variable 305

9 Locale ('locale.h') 307

9.1 setlocale, localeconv—select or query locale 309

10 Reentrancy 311

11 Miscellaneous Macros and Functions 313

11.1 ffs—find first bit set in a word 314
11.2 __retarget_lock_init, __retarget_lock_init_recursive, __retarget_lock_close, __retarget_lock_close_recursive, __retarget_lock_acquire, __retarget_lock_acquire_recursive, __retarget_lock_try_acquire, __retarget_lock_try_acquire_recursive, __retarget_lock_release, __retarget_lock_release_recursive—locking routines 315
11.3 unctrl—get printable representation of a character 316

12 Overflow Protection 317

12.1 Stack Smashing Protection 317
12.2 Object Size Checking 317

13 System Calls 319

13.1 Definitions for OS interface 319
13.2 Reentrant covers for OS subroutines 323
13.2.1 _close_r—Reentrant version of close 324
13.2.2 _execve_r—Reentrant version of execve 325
13.2.3 _fork_r—Reentrant version of fork 326
13.2.4 _wait_r—Reentrant version of wait 327
13.2.5 _fstat_r—Reentrant version of fstat 328
13.2.6 _link_r—Reentrant version of link 329
13.2.7 _lseek_r—Reentrant version of lseek 330
13.2.8 _open_r—Reentrant version of open 331
13.2.9 _read_r—Reentrant version of read 332
13.2.10 _sbrk_r—Reentrant version of sbrk 333
13.2.11 _kill_r—Reentrant version of kill 334
13.2.12 _getpid_r—Reentrant version of getpid 335
13.2.13 _stat_r—Reentrant version of stat 336
13.2.14 _times_r—Reentrant version of times 337
13.2.15 _unlink_r—Reentrant version of unlink 338
13.2.16 _write_r—Reentrant version of write 339

14 Variable Argument Lists 341
14.1 ANSI-standard macros, '`stdarg.h`' 341
14.1.1 Initialize variable argument list 342
14.1.2 Extract a value from argument list 343
14.1.3 Abandon a variable argument list 344
14.2 Traditional macros, '`varargs.h`' 344
14.2.1 Declare variable arguments 345
14.2.2 Initialize variable argument list 346
14.2.3 Extract a value from argument list 347
14.2.4 Abandon a variable argument list 348

Document Index 349