Stretch®

The C Support Library

Reference Manual

2007.11

© 2004 Stretch, Inc. All rights reserved. The Stretch logo, Stretch, and Extending the Possibilities are trademarks of Stretch, Inc. All other trademarks and brand names are the properties of their respective owners.

This publication is provided "AS IS." Stretch, Inc. (hereafter "Stretch") DOES NOT MAKE ANY WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF TITLE, NONINFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Information in this document is provided solely to enable system and software developers to use Stretch processors. Unless specifically set forth herein, there are no express or implied patent, copyright or any other intellectual property rights or licenses granted hereunder. Stretch does not warrant that the contents of this publication, whether individually or as one or more groups, meets your requirements or that the publication is error-free. This publication could include technical inaccuracies or typographical errors. Changes may be made to the information herein, and these changes may be incorporated in new editions of this publication.

Part #: RU-0000-0409-000

Version 1.8.0
Steve Chamberlain, Roland Pesch, and Cygnus Support
With modifications from Tensilica, Inc.
© 1992, 1993 Cygnus Support
© 1999, 2000, 2001, 2002 Tensilica, Inc.

"libm" includes software developed at SunPro, a Sun Microsystems, Inc. business. Permission to use, copy, modify, and distribute this software is freely granted, provided that this notice is preserved.

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.

Contents

Chapter 1 Introduction

Chapter 2 Standard Utility Functions (stdlib.h)

Chapter 3 Character Type Macros and Functions (ctype.h)

Chapter 4 Input and Output (stdio.h)

Stretch, Inc.

Chapter 5 Strings and Memory (string.h)

Stretch, Inc.

Stretch, Inc.

Chapter 1 Introduction

This reference manual describes the functions provided by the Cygnus `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 12, “System Calls”, for a description of the relevant operating system calls.

Chapter 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 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 programs 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.

Required OS subroutines

`getpid, kill`

2.2 abs—Integer absolute value (magnitude)

Synopsis

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

Description

`abs` returns |i| |*i*| , 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 non-negative integer.

Portability

`abs` is ANSI.

No supporting OS subroutines are required.

2.3 assert—Macro for Debugging Diagnostics

Synopsis

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

Description

Use this macro to embed debugging diagnostic statements in your programs. The argument *expression* should be an expression which evaluates to true (non-zero) 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
```

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.

Required OS subroutines

(only if enabled): close, fstat, getpid, isatty, kill, lseek, read, sbrk, write

2.4 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.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

2.5 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.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

2.6 atoi, atol—string to integer

Synopsis

```
#include <stdlib.h>
int atoi(const char *s);
long atol(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)
```

Returns

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

Portability

atoi is ANSI.

No supporting OS subroutines are required.

2.7 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.

2.8 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 `callocr` 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.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

2.9 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.

2.10 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.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

2.11 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`*. `gdvtf` produces corresponding character representations of `float` numbers.

`gcvt` uses the same rules as the `printf` format `%.`*`precision`*—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.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

Stretch, Inc.

2.12 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 representing 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.

Required OS subroutines

close, fstat, isatty, lseek, read, sbrk, write

2.13 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.

Required OS subroutines

`exit`

2.14 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`.

2.15 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.

2.16 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.

2.17 malloc, realloc, free—manage memory

Synopsis

```
#include <stdlib.h>
void *malloc(size_t nbytes);
void *realloc(void *aptr, size_t nbytes);
void free(void *aptr);
void *_malloc_r(void *reent, size_t nbytes);
void *_realloc_r(void *reent, void *aptr, size_t nbytes);
void _free_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 alternate functions `_malloc_r`, `_realloc_r`, and `_free_r` are reentrant versions. The extra argument *`reent`* is a pointer to a reentrancy structure.

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.

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

Required OS subroutines sbrk, write (if WARNLVLIMIT)

2.18 mbtowc—minimal multi-byte to wide char converter

Synopsis

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

Description

This is a minimal ANSI-conforming implementation of `mbtowc`. The 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. In this implementation, the argument *n* is ignored.

Returns

This implementation of `mbtowc` returns 0 if *s* is null; it returns 1 otherwise (reporting the length of the character “sequence” used).

Portability

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

`mbtowc` requires no supporting OS subroutines.

2.19 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).

Stretch, Inc.

2.20 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.

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.

2.21 strtod, strtof—string to double or float

Synopsis

```
#include <stdlib.h>
double strtod(const char *str, char **tail)
float strtof(const char *str, char **tail)
double _strtod_r(void *reent,
const char *str, char **tail);
```

Description

The function strtod parses the character string *str*, producing a substring which can be converted to a double value. The substring converted is the longest initial subsequence of *str*, beginning with the first non-whitespace character, that has the format:

```
[+|-] digits[.] [digits] [(e|E) [+|-] 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. 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*, strtof is identical to strtod except for its return type.

This implementation returns the nearest machine number to the input decimal string. Ties are broken by using the IEEE round-even rule.

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

Returns

strtod returns 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.

Required OS subroutines

close, fstat, isatty, lseek, read, sbrk, write

2.22 strtol—string to long

Synopsis

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

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

Returns

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

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

Portability

`strtol` is ANSI.

No supporting OS subroutines are required.

Stretch, Inc.

2.23 strtoul—string to unsigned long

Synopsis

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

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

Returns

`strtoul` returns the converted value, if any; 0 if no conversion was made.

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

Stretch, Inc.

Portability

`strtoul` is ANSI.

`strtoul` requires no supporting OS subroutines.

Stretch, Inc.

2.24 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 non-zero 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.

Required OS subroutines

`_exit, _execve, _fork_r, _wait_r`

Stretch, Inc.

2.25 wctomb—minimal wide char to multi-byte converter

Synopsis

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

Description

This is a minimal ANSI-conforming implementation of `wctomb`. The only "wide characters" recognized are single bytes, and they are "converted" to themselves.

Each call to `wctomb` copies the character *`wchar`* to `*`*`s`*, unless *`s`* is a null pointer.

Returns

This implementation of `wctomb` returns 0 if *`s`* is null; it returns 1 otherwise (reporting the length of the character "sequence" generated).

Portability

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

`wctomb` requires no supporting OS subroutines.

Chapter 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—alphanumeric character predicate

Synopsis

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

Description

`isalnum` is a macro which classifies ASCII integer 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 for all integer values.

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

Returns

`isalnum` returns non-zero if *c* is a letter (a-z or A-Z) or a digit (0-9).

Portability

`isalnum` is ANSI C.

No OS subroutines are required.

3.2 isalpha—alphabetic character predicate

Synopsis

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

Description

`isalpha` is a macro which classifies ASCII integer 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 when `isascii(c)` is true or *c* is `EOF`.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef isalpha`.

Returns

`isalpha` returns non-zero if *c* is a letter (a-z or A-Z).

Portability

`isalpha` is ANSI C.

No supporting OS subroutines are required.

3.3 isascii—ASCII character predicate

Synopsis

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

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.

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

Returns

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

Portability

`isascii` is ANSI C.

No supporting OS subroutines are required.

3.4 iscntrl—control character predicate

Synopsis

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

Description

`iscntrl` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for control characters, and 0 for other characters. It is defined only when `isascii(`*c*`)` is true or *c* is `EOF`.

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

Returns

`iscntrl` returns non-zero if *c* is a delete character or ordinary control character (`0x7F` or `0x00-0xlF`).

Portability

`iscntrl` is ANSI C.

No supporting OS subroutines are required.

3.5 isdigit—decimal digit predicate

Synopsis

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

Description

`isdigit` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for decimal digits, and 0 for other characters. It is defined only when `isascii(`*`c`*`)` is true or *c* is `EOF`.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef isdigit`.

Returns

`isdigit` returns non-zero if *c* is a decimal digit (0-9).

Portability

`isdigit` is ANSI C.

No supporting OS subroutines are required.

3.6 islower—lower-case character predicate

Synopsis

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

Description

`islower` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for minuscules (lower-case alphabetic characters), and 0 for other characters. It is defined only when `isascii(`*c*`)` is true or *c* is `EOF`.

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

Returns

`islower` returns non-zero if *c* is a lowercase letter (a-z).

Portability

`islower` is ANSI C.

No supporting OS subroutines are required.

3.7 isprint, isgraph—printable character predicates

Synopsis

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

Description

`isprint` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for printable characters, and 0 for other character arguments. It is defined only when `isascii(`*`c`*`)` is true or *c* is `EOF`.

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

Returns

`isprint` returns non-zero if *c* is a printing character, (`0x20-0x7E`). isgraph behaves identically to isprint, except that the space character (`0x20`) is excluded.

Portability

`isprint` and `isgraph` are ANSI C.

No supporting OS subroutines are required.

3.8 ispunct—punctuation character predicate

Synopsis

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

Description

`ispunct` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for printable punctuation characters, and 0 for other characters. It is defined only when `isascii(`*`c`*`)` is true or *`c`* is `EOF`.

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

Returns

`ispunct` returns non-zero if *`c`* is a printable punctuation character (`isgraph(`*`c`*`) && !isalnum(`*`c`*`)`).

Portability

`ispunct` is ANSI C.

No supporting OS subroutines are required.

3.9 isspace—whitespace character predicate

Synopsis

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

Description

`isspace` is a macro which classifies ASCII integer 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`.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef isspace`.

Returns

`isspace` returns non-zero if *c* is a space, tab, carriage return, new line, vertical tab, or formfeed (`0x09`–`0x0D`, `0x20`).

Portability

`isspace` is ANSI C.

No supporting OS subroutines are required.

3.10 isupper—uppercase character predicate

Synopsis

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

Description

`isupper` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for upper-case letters (A-Z), and 0 for other characters. It is defined only when `isascii(`*c*`)` is true or *c* is `EOF`.

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

Returns

`isupper` returns non-zero if *c* is a upper case letter (A-Z).

Portability

`isupper` is ANSI C.

No supporting OS subroutines are required.

3.11 isxdigit—hexadecimal digit predicate

Synopsis

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

Description

`isxdigit` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for hexadecimal digits, and 0 for other characters. It is defined only when `isascii(c)` is true or *c* is `EOF`.

You can use a compiled subroutine instead of the macro definition by undefining the macro using `#undef isxdigit`.

Returns

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

Portability

`isxdigit` is ANSI C.

No supporting OS subroutines are required.

3.12 toascii—force integers to ASCII range

Synopsis

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

Description

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

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

Returns

`toascii` returns integers between 0 and 127.

Portability

`toascii` is not ANSI C.

No supporting OS subroutines are required.

3.13 tolower—translate characters to lower case

Synopsis

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

Description

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

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

`_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` returns the lower-case equivalent of *c* when it is a character between A and Z, and *c* otherwise.

`_tolower` returns the lower-case 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.

No supporting OS subroutines are required.

3.14 toupper—translate characters to upper case

Synopsis

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

Description

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

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

`_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` returns the upper-case equivalent of *c* when it is a character between a and z, and *c* otherwise.

`_toupper` returns the upper-case 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.

No supporting OS subroutines are required.

Chapter 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—clear file or stream error indicator

Synopsis

```
#include <stdio.h>
void clearerr(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.

Returns

`clearerr` does not return a result.

Portability

ANSI C requires `clearerr`.

No supporting OS subroutines are required.

4.2 fclose—close a file

Synopsis

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

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`

Stretch, Inc.

4.3 feof—test for end of file

Synopsis

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

Description

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

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.

No supporting OS subroutines are required.

4.4 ferror—test whether read/write error has occurred

Synopsis

```
#include <stdio.h>
int ferror(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.

Returns

`ferror` returns 0 if no errors have occurred; it returns a non-zero value otherwise.

Portability

ANSI C requires `ferror`.

No supporting OS subroutines are required.

Stretch, Inc.

4.5 fflush—flush buffered file output

Synopsis

```
#include <stdio.h>
int fflush(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.

Returns

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

Portability

ANSI C requires `fflush`.

No supporting OS subroutines are required.

4.6 fgetc—get a character from a file or stream

Synopsis

```
#include <stdio.h>
int fgetc(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`.

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`.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.7 fgetpos—record position in a stream or file

Synopsis

```
#include <stdio.h>
int fgetpos(FILE *fp, 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 `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 l. 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.

4.8 fgets—get character string from a file or stream

Synopsis

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

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.

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.)

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.9 fiprintf—format output to file (integer only)

Synopsis

```
#include <stdio.h>
int fiprintf(FILE *fd, const char *format, ...) ;
```

Description

`fiprintf` is a restricted version of `fprintf`: it has the same arguments and behavior, save that it cannot perform any floating-point formatting—the f, g, G, e, and f type specifiers are not recognized.

Returns

`fiprintf` returns the number of bytes in the output string, save that the concluding null is not counted, `fiprintf` returns when the end of the format string is encountered. If an error occurs, `fiprintf` returns `EOF`.

Portability

`fiprintf` is not required by ANSI C.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.10 fopen—open a file

Synopsis

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

Description

fopen initializes the data structures needed to read or write a file. Specify the files 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.

Stretch, Inc.

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.

Required OS subroutines

close, fstat, isatty, lseek, open, read, sbrk, write

Stretch, Inc.

4.11 fdopen—turn open file into a stream

Synopsis

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

Stretch, Inc.

4.12 fputc—write a character on a stream or file

Synopsis

```
#include <stdio.h>
int fputc(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 advances by one.

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

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.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.13 fputs—write a character string in a file or stream

Synopsis

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

Description

`fputs` writes the string at *s* (but without the trailing null) to the file or stream identified by *fp*.

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.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.14 fread—read array elements from a file

Synopsis

```
#include <stdio.h>
size_t fread(void *buf, size_t size, size_t count,
             FILE *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.

Returns

The result of `fread` is the number of elements it succeeded in reading.

Portability

ANSI C requires `fread`.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.15 freopen—open a file using an existing file descriptor

Synopsis

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

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`.

Required OS subroutines

`close, fstat, isatty, lseek, open, read, sbrk, write`

4.16 fseek—set file position

Synopsis

```
#include <stdio.h>
int fseek(FILE *fp, long 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` 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` to determine the current file position.

Returns

`fseek` returns 0 when successful. If `fseek` fails, 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`.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.17 fsetpos—restore position of a stream or file

Synopsis

```
#include <stdio.h>
int fsetpos(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 l. 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`.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.18 ftell—return position in a stream or file

Synopsis

```
#include <stdio.h>
long ftell(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` is the current position for a file identified by *`fp`*. If you record this result, you can later use it with `fseek` to return the file to this position.

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

Returns

`ftell` returns the file position, if possible. If it cannot do this, it returns `-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.

No supporting OS subroutines are required.

4.19 fwrite—write array elements

Synopsis

```
#include <stdio.h>
size_t fwrite(const void *buf, size_t size,
              size_t count, FILE *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.

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`.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.20 getc—read a character (macro)

Synopsis

```
#include <stdio.h>
int getc(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 files current position indicator.

For a subroutine version of this macro, see `fgetc`.

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.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.21 getchar—read a character (macro)

Synopsis

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

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.22 gets—get character string

IMPORTANT! Obsolete, use `fgets` instead.

Synopsis

```
#include <stdio.h>
char *gets(char *buf);
char *_gets_r(void *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.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.23 iprintf—write formatted output (integer only)

Synopsis

```
#include <stdio.h>
int iprintf(const char *format, ...) ;
```

Description

`iprintf` is a restricted version of `printf`: it has the same arguments and behavior, save that it cannot perform any floating-point formatting: the `f`, `g`, `G`, `e`, and `f` type specifiers are not recognized.

Returns

`iprintf` returns the number of bytes in the output string, save that the concluding null is not counted, `iprintf` returns when the end of the format string is encountered. If an error occurs, `iprintf` returns `EOF`.

Portability

`iprintf` is not required by ANSI C.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.24 mktemp, mkstemp—generate unused file name

Synopsis

```
#include <stdio.h>
char *mktemp(char *path);
int mkstemp(char *path);
char *_mktemp_r(void *reent, char *path);
int *_mkstemp_r(void *reent, char *path);
```

Description

`mktemp` and `mkstemp` attempt to generate a file name that is not yet in use for any existing file, `mkstemp` creates the file and opens it for reading and writing; `mktemp` simply generates the file name.

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 some number of "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.

The alternate functions `_mktemp_r` and `_mkstemp_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`.

`mkstemp` returns 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 -l.

Portability

ANSI C does not require either `mktemp` or `mkstemp`; the System V Interface Definition requires `mktemp` as of Issue 2.

Required OS subroutines

`getpid, open, stat`

4.25 perror—print an error message on standard error

Synopsis

```
#include <stdio.h>
void perror(char *prefix);
void _perror_r(void *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.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.26 putc—write a character (macro)

Synopsis

```
#include <stdio.h>
int putc(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.

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.

Required OS subroutines

close, fstat, isatty, lseek, read, sbrk, write

4.27 putchar—write a character (macro)

Synopsis

```
#include <stdio.h>
int putchar(int ch);
int _putchar_r(void *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 `putcharr` 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.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.28 puts—write a character string

Synopsis

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

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.29 remove—delete a files name

Synopsis

```
#include <stdio.h>
int remove(char *filename);
int _remove_r(void *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 files 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, -l 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.

Required OS subroutines

`unlink`

4.30 rename—rename a file

Synopsis

```
#include <stdio.h>
int rename(const char *old, const char *new);
int _rename_r(void *reent, 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.

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

Returns

The result is either 0 (when successful) or -l (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.

Required OS subroutines

`link, unlink, or rename`

Stretch, Inc.

4.31 rewind—reinitialize a file or stream

Synopsis

```
#include <stdio.h>
void rewind(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.

4.32 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.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.33 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 non-zero value.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.34 siprintf—write formatted output (integer only)

Synopsis

```
#include <stdio.h>
int siprintf(char *str, const char *format [, arg, ...]);
```

Description

`siprintf` is a restricted version of `sprintf`: it has the same arguments and behavior, save that it cannot perform any floating-point formatting: the `f`, `g`, `G`, `e`, and `f` type specifiers are not recognized.

Returns

`siprintf` returns the number of bytes in the output string, save that the concluding null is not counted, `siprintf` returns when the end of the format string is encountered.

Portability

`siprintf` is not required by ANSI C.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.35 printf, fprintf, sprintf—format output

Synopsis

```
#include <stdio.h>
int printf(const char *format [, arg, …]);
int fprintf(FILE *fd, const char *format [, arg, …]);
int sprintf(char *str, const char *format [, arg, …]);
```

Description

`printf` accepts a series of arguments, applies to each a format specifier from *`format`, and writes the formatted data to stdout, terminated with a null 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` and `sprintf` are identical to `printf`, other than the destination of the formatted output: `fprintf` sends the output to a specified file *`fd`*, while `sprintf` stores the output in the specified `char` array *`str`*. For `sprintf`, the behavior is also undefined if the output *`*str`* overlaps with one of the arguments, *`format`* is a pointer to a charater string containing two types of objects: ordinary characters (otherthan %), 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:

```
% [flags] [width] [.prec] [size] [type]
```

The fields of the conversion specification have the following meanings:

`flags` an optional sequence of characters which control output justification, numeric signs, decimal points, trailing zeroes, and octal and hex prefixes. The flag characters are minus (-), plus (+), space ( ), zero (0), and sharp (#). They can appear in any combination.

`-` 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`*) 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.

Stretch, Inc.

- `0` If the type character is `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, or `G`: leading zeroes, 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 next character:
  - `0` 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.
  - `e`, `E` 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 zeroes are removed.
  - `g` or `G` same as `e` or `e`, but trailing zeroes 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. Negative field widths are not supported; if you attempt to specify a negative field width, it is interpreted as a minus (-) flag followed by a positive field width.

`prec` an optional field; if present, it is introduced with `.` (a period). This field gives the maximum number of characters to print in a conversion; the minimum number of digits of an integer to print, for conversions with type `d`, `i`, `o`, `u`, `x`, and `X`; the maximum number of significant digits, for the `g` and `G` conversions; or the number of digits to print after the decimal point, for `e`, `E`, and `f` conversions. 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. Supplying a negative precision is equivalent to omitting the precision. If only a period is specified the precision is zero. If a precision appears with any other conversion type than those listed here, the behavior is undefined.

`size` `h`, `l`, and `L` are optional size characters which override the default way that `printf` interprets the data type of the corresponding argument, h forces the following `d`, `i`, `o`, `u`, `x` or `X` conversion type to apply to a `short` or unsigned `short`, `h` also forces a following n type to apply to a pointer to a `short`. Similarly, an `l` forces the following `d`, `i`, `o`, `u`, `x` or `X` conversion type to apply to a `long` or unsigned `long`. `l` also forces a following `n` type to apply to a pointer to a `long`. If an `h` or an `l` appears with another conversion specifier, the behavior is undefined, `l` forces a following `e`, `E`, `f`, `g` or G conversion type to apply to a `long double` argument. If `l` appears with any other conversion type, the behavior is undefined.

`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

`s` prints characters until precision is reached or a null terminator is encountered; takes a string pointer

`d` prints a signed decimal integer; takes an `int` (same as `i`)

`i` prints a signed decimal integer; takes an `int` (same as `d`)

`0` prints a signed octal integer; takes an `int`

`u` prints an unsigned decimal integer; takes an `int`

`x` prints an unsigned hexadecimal integer (using `abcdef` as digits beyond 9); takes an `int`

`X` prints an unsigned hexadecimal integer (using `ABCDEF` as digits beyond 9); takes an `int`

`f` prints a signed value of the form `[-] 9999.9999`; takes a floating point number

`e` prints a signed value of the form

```
[-]9.9999e[+|-]999
```

takes a floating point number

`E` prints the same way as `e`, but using `E` to introduce the exponent; takes a floating point number

`g` prints a signed value in either `f` or `e` form, based on given value and precision—trailing zeros and the decimal point are printed only if necessary; takes a floating point number

`G` prints the same way as `g`, but using `E` for the exponent if an exponent is needed; takes a floating point number

`n` stores (in the same object) a count of the characters written; takes a pointer to `int`

`p` prints a pointer in an implementation-defined format. This implementation treats the pointer as an `unsigned long` (same as `Lu`).

Returns

`sprintf` returns the number of bytes in the output string, save that the concluding null is not counted, `printf` and `fprintf` return the number of characters transmitted. If an error occurs, `printf` and `fprintf` return `EOF`. No error returns occur for `sprintf`.

Portability

The ANSI C standard specifies that implementations must support at least formatted output of up to 509 characters.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.36 scanf, fscanf, sscanf—scan and format input

Synopsis

```
#include <stdio.h>
int scanf(const char *format [, arg, …]);
int fscanf(FILE *fd, const char *format [, arg, …]);
int sscanf(const char *str, const char *format [, arg, …]);
```

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 disastrous. 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 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-convertible 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`, `l`, and `L` are optional size characters which override the default way that `scanf` interprets the data type of the corresponding argument.

Modifiers	**Type(s)**	
`h`	`d, i. o, u, X`	convert input to short, store in short object

`h`	`D, I, 0, U,`
	`X, e, f, c,`
	`s, n, P`
		no effect

`1`	`d, i, o, u, x`	convert input to long, store in long object
`1`	`e, f, g`	convert input to double store in a double object

`1`	`D, I, O, U,`
	`X, c, s, n, p`
		no effect

`L`	`d, i, o, u, x`	convert to long double, store in long double
`L`	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`).

`0` Reads an octal integer into the corresponding *arg*: (`int *arg`).

`0` 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`).

`1` 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:

%[abed]	matches strings containing only a, b, c, and d
%Tabcd]	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 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.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

4.37 tmpfile—create a temporary file

Synopsis

```
#include <stdio.h>
FILE *tmpfile(void);
FILE *_tmpfile_r(void *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`.

Required OS subroutines

`close, fstat, getpid, isatty, lseek, open, read, sbrk, write`
`tmpfile` also requires the global pointer `environ`.

4.38 tmpnam, tempnam—name for a temporary file

Synopsis

```
#include <stdio.h>
char *tmpnam(char *s) ;
char *tempnam(char *dir, char *pfx);
char *_tmpnam_r(void *reent, char *s);
char *_tempnam_r(void *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 tmpmax 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 arguments to specify a suitable area of memory for the generated file name; otherwise, you can call tmpnam (null) to use an internal static buffer.

tempnam allows you more control over the generated file name: 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 file name.

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 areas 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 file name.

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.

Required OS subroutines

close, fstat, getpid, isatty, lseek, open, read, sbrk, write
The global pointer environ is also required.

4.39 vprintf, vfprintf, vsprintf—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 _vprintf_r(void *reent, const char *fmt,
               va_list list);
int _vfprintf_r(void *reent, FILE *fp, const char *fmt,
                va_list list);
int _vsprintf_r(void *reent, char *str, const char *fmt,
                va list list);
```

Description

`vprintf`, `vfprintf`, and `vsprintf` are (respectively) variants of `printf`, `fprintf`, and `sprintf`. They differ only in allowing their caller to pass the variable argument list as a `valist` object (initialized by `vastart`) rather than directly accepting a variable number of arguments.

Returns

The return values are consistent with the corresponding functions: `vsprintf` returns the number of bytes in the output string, save that the concluding null is not counted, `vprintf` and `vfprintf` return the number of characters transmitted. If an error occurs, `vprintf` and `vfprintf` return `EOF`. No error returns occur for `vsprintf`.

Portability

ANSI C requires all three functions.

Required OS subroutines

`close, fstat, isatty, lseek, read, sbrk, write`

Chapter 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 <string.h>
int bcmp(const char *s1, const char *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`.

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 <string.h>
void bcopy(const char *in, char  *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 <string.h>
void bzero (char *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 <string.h>
char *index(const char *string, int c)
```

Description

This function finds the first occurrence 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 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, null 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.

5.6 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.

5.7 memcpy—copy memory regions

Synopsis

```
#include <string.h>
void *memcpy(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

memcpy returns a pointer to the first byte of the out region.

Portability

memcpy is ANSI C.

memcpy requires no supporting OS subroutines.

5.8 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.

5.9 memset—set an area of memory

Synopsis

```
#include <string.h>
void *memset(const 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 *m*.

Portability

`memset` is ANSI C.

`memset` requires no supporting OS subroutines.

5.10 rindex—reverse search for character in string

Synopsis

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

Description

This function finds the last occurrence 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.

5.11 strcat—concatenate strings

Synopsis

```
#include <string.h>
char *strcat(char *dst, const char *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.

5.12 strchr—search for character in string

Synopsis

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

Description

This function finds the first occurrence 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.

5.13 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.

5.14 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.

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.

5.15 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.

5.16 strcspn—count chars 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.

Stretch, Inc.

5.17 strerror—convert error number to string

Synopsis

```
#include <string.h>
char *strerror(int errnum);
```

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.

This implementation of `strerror` prints out the following strings for each of the values defined in `errno.h`:

`E2BIG`	Arg list too long
`EACCES`	Permission denied
`EADV`	Advertise error
`EAGAIN`	No more processes
`EBADF`	Bad file number
`EBADMSG`	Bad message
`EBUSY`	Device or resource busy
`ECHILD`	No children
`ECOMM`	Communication error
`EDEADLK`	Deadlock
`EEXIST`	File exists
`EDOM`	Math argument
`EFAULT`	Bad address
`EFBIG`	File too large
`EIDRM`	Identifier removed
`EINTR`	Interrupted system call
`EINVAL`	Invalid argument
`EIO`	I/O error
`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`	Too many open files
`EMLINK`	Too many links
`EMULTIHOP`	Multihop attempted
`ENAMETOOLONG`	File or path name too long
`ENFILE`	Too many open files in system

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
ENOSPC	No space left on device
ENOSR	No stream resources
ENOSTR	Not a stream
ENOSYS	Function not implemented
ENOTBLK	Block device required
ENOTDIR	Not a directory
ENOTEMPTY	Directory not empty
ENOTTY	Not a character device
ENXIO	No such device or address
EPERM	Not owner
EPIPE	Broken pipe
EPROTO	Protocol error
ERANGE	Result too large
EREMOTE	Resource is remote
EROFS	Read-only file system
ESPIPE	Illegal seek
ESRCH	No such process
ESRMNT	Srmount error
ETIME	Stream ioctl timeout
ETXTBSY	Text file busy
EXDEV	Cross-device link

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.

Although this implementation of strerror is reentrant, ANSI C declares that subsequent calls to strerror may overwrite the result string; therefore portable code cannot depend on the reentrancy of this subroutine.

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 *errnum* as its argument.

userstrerror takes one argument of type int, and returns a character pointer. If *errnum* is unknown to _user_strerror, _user_strerror returns NULL. The default _user_strerror returns NULL for all input values.

strerror requires no supporting OS subroutines.

5.18 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 characters until it reaches a null character.

Returns

`strlen` returns the character count.

Portability

`strlen` is ANSI C.

`strlen` requires no supporting OS subroutines.

5.19 strlwr—force string to lower case

Synopsis

```
#include <string.h>
char *strlwr(char *a) ;
```

Description

`strlwr` converts each characters in the string at *a* to lower case.

Returns

`strlwr` returns its argument, *a*.

Portability

`strlwr` is not widely portable.

`strlwr` requires no supporting OS subroutines.

5.20 strncat—concatenate strings

Synopsis

```
#include <string.h>
char *strncat(char *dst, const char *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.

5.21 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.

5.22 strncpy—counted copy string

Synopsis

```
#include <string.h>
char *strncpy(char *dst, const char *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.

5.23 strpbrk—find chars in string

Synopsis

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2) ;
```

Description

This function locates the first occurrence 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.

Stretch, Inc.

5.24 strrchr—reverse search for character in string

Synopsis

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

Description

This function finds the last occurrence 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.

Stretch, Inc.

5.25 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.

5.26 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, the *s1* is returned.

Portability

`strstr` is ANSI C.

`strstr` requires no supporting OS subroutines.

5.27 strtok—get next token from a string

Synopsis

```
#include <string.h>
char *strtok(char *source, const char *delimiters)
char *strtok_r(char *source, const char *delimiters,
               char **lasts)
```

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.

Returns

`strtok` returns a pointer to the next token, or `NULL` if no more tokens can be found.

Portability

`strtok` is ANSI C.

`strtok` requires no supporting OS subroutines.

5.28 strupr—force string to uppercase

Synopsis

```
#include <string.h>
char *strupr(char *a);
```

Description

`strupr` converts each characters in the string at *a* to upper case.

Returns

`strupr` returns its argument, *a*.

Portability

`strupr` is not widely portable.

`strupr` requires no supporting OS subroutines.

Stretch, Inc.

5.29 strxfrm—transform string

Synopsis

```
#include <string.h>
size_t strxfrm(char *s1, const char *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, corresponding 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.

With a C locale, this function just copies.

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.

Chapter 6 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 environments 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.)

6.1 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 programs 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`

6.2 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` providesa simple signal 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 programs 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 `raise`, `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.

Chapter 7 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.
`tm_min`	Minutes.
`tm_hour`	Hours.
`tm_mday`	Day.
`tm_mon`	Month.
`tm_year`	Year (since 1900).
`tmwday`	Day of week: the number of days since Sunday.
`tmyday`	Number of days elapsed since last January 1.
`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.

7.1 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.

7.2 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 -1.

Portability

ANSI C requires `clock` and `CLOCKS_PER_SEC`.

Required OS subroutines

`times`

7.3 ctime—convert time to local and format as string

Synopsis

```
#include <time.h>
char *ctime(time_t *clock);
char *ctime_r(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.

7.4 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.

7.5 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` assumes the time at *`clock`* represents a local time, `gmtime` converts it to UTC (Universal Coordinated Time, also known in some countries as GMT, Greenwich Mean Time), then converts the representation from the arithmetic representation to the traditional representation defined by `struct tm`.

`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.

7.6 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.

7.7 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 -l. Otherwise, the result is the time, converted to a `time_t` value.

Portability

ANSI C requires `mktime`.

`mktime` requires no supporting OS subroutines.

7.8 strftime—flexible calendar time formatter

Synopsis

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
                const char *format,
                const struct tm *timp);
```

Description

`strftime` converts a `struct tm` representation of the time (at *`timp`*) into a string, starting at *`s`* and occupying no more than *`maxsize`* characters.

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-character sequences beginning with `%` (use `%%` to include a percent sign in the output). Each defined conversion specification selects a field of calendar time data from `*`*`timp`*, and converts it to a string in one of the following ways:

`%a` An abbreviation for the day of the week.

`%A` The full name for the day of the week.

`%b` An abbreviation for the month name.

`%B` The full name of the month.

`%c` A string representing the complete date and time, in the form

```
Mon Apr 01 13:13:13 1992
```

`%d` The day of the month, formatted with two digits.

`%H` The hour (on a 24-hour clock), formatted with two digits.

`%I` The hour (on a 12-hour clock), formatted with two digits.

`%j` The count of days in the year, formatted with three digits (from 001 to 366).

`%m` The month number, formatted with two digits.

`%M` The minute, formatted with two digits.

`%p` Either am or pm as appropriate.

`%S` The second, formatted with two digits.

`%U` The week number, formatted with two digits (from 00 to 53; week number 1 is taken as beginning with the first Sunday in a year). See also `%w`.

`%w` A single digit representing the day of the week: Sunday is day 0.

`%W` Another version of the week number: like `%u`, but counting week 1 as beginning with the first Monday in a year.

`o %x` A string representing the complete date, in a format like

```
Mon Apr 01 1992
```

`%X` A string representing the full time of day (hours, minutes, and seconds), in a format like

```
13:13:13
```

`%y` The last two digits of the year.

`%Y` The full year, formatted with four digits to include the century.

`%Z` Defined by ANSI C as eliciting the time zone if available; it is not available in this implementation (which accepts `%Z` but generates no output for it).

`%%` 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.

`strftime` requires no supporting OS subroutines.

7.9 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 -l result means the current time is not available; otherwise the result represents the current time.

Portability

ANSI C requires time.

Required OS subroutines

Some implementations require `gettimeofday`.

Chapter 8 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 strings 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; l 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.

8.1 setlocale, localeconv—select or query locale

Synopsis

```
#include <locale.h>
char *setlocale (int category, const char *locale);
struct lconv *localeconv(void);
char *_setlocale_r(void *reent, int category,
                   const char *locale);
struct 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 C value tor locale; strings representing other locales are not honored. (“” is also accepted; it represents the default locale for an implementation, here equivalent to “C”.)

If you use null as the locale argument, `setlocale` returns a pointer to the string representing the current locale (always “C” in this implementation). The acceptable values for category are defined in `locale.h` as macros beginning with `mlc_`, but this implementation does not check the values you pass in the *`category`* argument.

`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

`setlocale` returns either a pointer to a string naming the locale currently in effect (always “C” for this implementation), or, if the locale request cannot be honored, `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.

No supporting OS subroutines are required.

Chapter 9 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. Cygnus's 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`*.

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
_fopen_r
_getchar_r
_gets_r
_iprintf_r
_mkstemp_r
_mktemp_r
_perror_r
_putchar_r
_puts_r
_remove_r
_rename_r
_tempnam_r
_tmpfile_r
_tmpnam_r

signal **functions:**

_raise_r
_signal_r

stdlib **functions:**

_dtoa_r
_free_r
_malloc_r
_rand_r
_realloc_r
_srand_r
_strtod_r
_strtol_r
_strtoul_r
_system_r

string **functions:**

strtok r

System functions:

_close_r
_lseek_r
_stat_r
_fork_r
_open_r
_unlink_r
_fstat_r
_read_r
_wait_r
_link_r
_sbrk_r
_write_r

time **function:**

_asctime_r

Chapter 10 Miscellaneous Macros and Functions

This chapter describes miscellaneous routines not covered elsewhere.

10.1 unctrl—translate characters to upper case

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.

Chapter 11 Functions for Xtensa Processors

This chapter describes machine-dependent functions that are included in the C library when it is built for Xtensa processors.

Stretch, Inc.

11.1 setjmp—save stack environment

Synopsis

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Description

`setjmp` and `longjmp` are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program, `setjmp` saves the stack context/environment in *`env`* for later use by `longjmp`. The stack context will be invalidated if the function which called `setjmp` returns.

Returns

`setjmp` returns 0 if returning directly, and non-zero when returning from `longjmp` using the saved context.

Portability

`setjmp` is ANSI C and POSIX.1.

`setjmp` requires no supporting OS subroutines.

11.2 longjmp—non-local goto

Synopsis

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

Description

`longjmp` and `setjmp` are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program, `longjmp` restores the environment saved by the last call of `setjmp` with the corresponding *`env`* argument. After `longjmp` is completed, program execution continues as if the corresponding call of `setjmp` had just returned the value *`val`*. `longjmp` cannot cause 0 to be returned. If `longjmp` is invoked with a second argument of 0, 1 will be returned instead.

Returns

This function never returns.

Portability

`longjmp` is ANSI C and POSIX.1.

`longjmp` requires no supporting OS subroutines.

Stretch, Inc.

Chapter 12 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`.

12.1 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 Cygnus C library provides a macro definition for errno in the header file `errno.h`, as part of its support for reentrant routines (see Chapter 9, “Reentrancy”).

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[l] = { 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() {
   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() {
   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){
   errno = EIO;
   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;
}
```

Stretch, Inc.

`unlink` Remove a files 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 a character 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 `writechar` 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++) {
   writechar(*ptr++);
}
   return len;
}
```

12.2 Xtensa System Calls

The default Xtensa runtime environment does not include an operating system. Instead, the GNU low-level operating system support library (`libgloss`) implements the operating system routines. There are two versions of `libgloss` for Xtensa. The first provides minimal stub routines, similar to those described above (see Section 12.1, “Definitions for OS interface”, on page 12-1). The second version of `libgloss` uses the “semihosting” feature

Stretch, Inc.

of the Xtensa instruction set simulator (ISS) to implement the operating system routines on top of the host system. For example, using this library, an open system call can open a file on the host running the ISS. This semihosting version of `libgloss` is the default when running on the Xtensa ISS.

12.2.1 Base Xtensa System Calls

The minimal Xtensa `libgloss` implementation provides the following system routines:

`stat` Sets `errno` to `EIO` and returns -1.

`fstat` Assumes terminal I/O and sets `st_mode` to `S_IFCHR`. Returns 0.

`getpid` Returns 1.

`kill` Calls `_exit` if the specified process ID is 1 (as returned by `getpid`); otherwise, does nothing. Returns 0.

`isatty` Returns 1.

`sbrk` Grows the heap. If the end of the heap extends beyond the `_heap_sentry` symbol (set by the linker), sets `errno` to `ENOMEM` and returns -1. Note that unlike the semihosting version of `libgloss`, this version does not currently check if the heap collides with the stack.

`open` Sets `errno` to `EIO`. Returns -1.

`close` Does nothing. Returns 0.

`lseek` Sets `errno` to `ESPIPE`. Returns-1.

`unlink` Sets `errno` to `EIO`. Returns -1.

`inbyte` Reads a byte from the serial port.

`read` Reads the specified number of bytes from the serial port using `inbyte`. Ignores the file descriptor argument.

`outbyte` Writes a byte to the serial port.

`write` Writes the specified number of bytes to the serial port using `outbyte`. Ignores the file descriptor argument.

`print` Prints a string to the serial port.

`putnum` Prints a 32-bit number in hexadecimal to the serial port.

12.2.2 System Calls with the Xtensa ISS

The semihosting version of `libgloss` for Xtensa provides the following system routines (all the routines using ISS semihosting set errno and the return value based on the status of the host system call):

`stat` Sets `errno` to `EIO` and returns -1.

`fstat` Assumes terminal I/O and sets `st_mode` to `S_IFCHR`. Returns 0.

`getpid` Returns 1.

`kill` Calls `_exit` if the specified process ID is 1 (as returned by `getpid`); otherwise, does nothing. Returns 0.

`isatty` Returns 1.

`sbrk` Grows the heap. If the end of the heap extends beyond the `_heap_sentry` symbol (set by the linker) or if it collides with the stack, sets `errno` to `ENOMEM` and returns -1.

`open` Opens the specified file on the host using the ISS semihosting interface.

`close` Closes the specified host file using the ISS semihosting interface.

`lseek` Seeks to the specified location in a host file using the ISS semihosting interface.

`unlink` Unlinks (i.e., removes) a host file using the ISS semihosting interface.

`read` Reads from a host file using the ISS semihosting interface.

`write` Writes to a host file using the ISS semihosting interface.

`rename` Renames a host file using the ISS semihosting interface.

`gettimeofday` Get the host systems current time expressed in elapsed seconds and microseconds since January 1, 1970 using the ISS semihosting interface. (Only supported on Unix platforms.)

`time` Get the host systems current time in seconds since January 1,1970 using the ISS semihosting interface.

`times` Returns Xtensa simulation cycle count if the ISS was invoked with the `--pipe` option; otherwise, returns the count of Xtensa instructions executed.

`_exit` Terminates the Xtensa ISS simulation.

`creat` Creates a new host file using the ISS semihosting interface.

`link` Adds a hard link to a host file using the ISS semihosting interface. (Only supported on Unix platforms.)

12.3 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 9, “Reentrancy”).

`_open_r` A reentrant version of `open`. It takes a pointer to the global data block, which holds `errno`.

```
int _open_r(void *reent, const char *file,
            int flags, int mode) ;
```

`_close_r` A reentrant version of `close`. It takes a pointer to the global data block, which holds `errno`.

```
int _close_r(void *reent, int fd);
```

`_lseek_r` A reentrant version of `lseek`. It takes a pointer to the global data block, which holds `errno`.

```
off_t _lseek_r(void *reent, int fd,
               off_t pos, int whence);
```

`_read_r` A reentrant version of `read`. It takes a pointer to the global data block, which holds `errno`.

```
long _read_r(void *reent, int fd,
             void *buf, size_t cnt);
```

`_write_r` A reentrant version of `write`. It takes a pointer to the global data block, which holds `errno`.

```
long _write_r(void *reent, int fd,
              const void *buf, size_t cnt);
```

_fork_r

A reentrant version of fork. It takes a pointer to the global data block, which holds errno.

```
int _fork_r(void *reent);
```

_wait_r

A reentrant version of wait. It takes a pointer to the global data block, which holds errno.

```
int _wait_r(void *reent, int *status) ;
```

_stat_r

A reentrant version of stat. It takes a pointer to the global data block, which holds errno.

```
int _stat_r(void *reent, const char *file,
            struct stat *pstat);
```

_fstat_r

A reentrant version of fstat. It takes a pointer to the global data block, which holds errno.

```
int _fstat_r(void *reent, int fd,
             struct stat *pstat);
```

_link_r

A reentrant version of link. It takes a pointer to the global data block, which holds errno.

```
int _link_r(void *reent, const char *old,
            const char *new);
```

_unlink_r

A reentrant version of unlink. It takes a pointer to the global data block, which holds errno.

```
int _unlink_r(void *reent,
              const char *file);
```

_sbrk_r

A reentrant version of sbrk. It takes a pointer to the global data block, which holds errno.

```
char *sbrk r(void *reent, size t incr);
```

Chapter 13 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).

13.1 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.

13.1.1 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`.

13.1.2 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.

13.1.3 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`.

13.2 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.

13.2.1 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.

13.2.2 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*.

13.2.3 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`.

13.2.4 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`.

Index

Stretch, Inc.