The Cygnus C Support Library

Full Configuration

`libc` 1.4
May 1993

Steve Chamberlain
Roland Pesch
Cygnus Support

sac@cygnus.com, pesch@cygnus.com *The Cygnus C Support Library*

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

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

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

1 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’.

1.1 abort—abnormal termination of a program

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

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

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

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

`abort` does not return to its caller.

ANSI C requires `abort`.

Supporting OS subroutines required: `getpid`, `kill`.

1.2 abs—integer absolute value (magnitude)

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

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

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

The result is a nonnegative integer.

abs is ANSI.

No supporting OS subroutines are required.

1.3 `assert`—Macro for Debugging Diagnostics

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

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

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

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

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))
```

`assert` does not return a value.

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

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

1.4 atexit—request execution of functions at program exit

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

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.

`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).

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

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

1.5 atof, atoff—string to double or float

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

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 strtodf(*s*, NULL).

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.

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

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

1.6 atoi, atol—string to integer

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

`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).`

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

`atoi` is ANSI.

No supporting OS subroutines are required.

1.7 atol—convert string to long

```
long atol(const char *s);
```

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

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

atol is ANSI.

No supporting OS subroutines are required.

1.8 bsearch—binary search

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

`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 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.

`bsearch` is ANSI.

No supporting OS subroutines are required.

1.9 calloc—allocate space for arrays

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

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

If successful, a pointer to the newly allocated space.

If unsuccessful, `NULL`.

`calloc` is ANSI.

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

1.10 div—divide two integers

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

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

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`'.

When d is zero, the `quot` member of the result has the same sign as n and the largest representable magnitude.

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

`div` is ANSI, but the behavior for zero d is not specified by the standard.

No supporting OS subroutines are required.

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

```
#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);
```

`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.

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

None of these functions are ANSI C.

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

1.12 gvcvt, gcvtf—format double or float as string

```
#include <stdlib.h>

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

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*g’—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*).

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

Neither function is ANSI C.

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

1.13 ecvtbuf, fcvtbuf—double or float to string

```
#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);
```

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

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

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

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

Neither function is ANSI C.

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

1.14 exit—end program execution

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

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.

exit does not return to its caller.

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

Supporting OS subroutines required: _exit.

1.15 getenv—look up environment variable

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

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.

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

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.

1.16 labs—long integer absolute value

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

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.

The result is a nonnegative long integer.

labs is ANSI.

No supporting OS subroutine calls are required.

1.17 ldiv—divide two long integers

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

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

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`'.

When d is zero, the `quot` member of the result has the same sign as n and the largest representable magnitude.

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

`ldiv` is ANSI, but the behavior for zero d is not specified by the standard.

No supporting OS subroutines are required.

1.18 malloc, realloc, free—manage memory

```
#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);
```

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.

`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.

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

Supporting OS subroutines required: `sbrk`, `write` (if WARN_VLIMIT).

1.19 mbtowc—minimal multibyte to wide char converter

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

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.

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

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

`mbtowc` requires no supporting OS subroutines.

1.20 qsort—sort an array

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

`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.

`qsort` does not return a result.

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

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

1.21 rand, srand—pseudo-random numbers

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

int _rand_r(void *reent);
void _srand_r(void *reent, unsigned int seed);
```

`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.

`_rand_r` and `_srand_r` are reentrant versions of `rand` and `srand`. The extra argument *reent* is a pointer to a reentrancy structure.

`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.

`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.

1.22 strtod, strtodf—string to double or float

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

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

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*. strtodf 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 _strtod_r is a reentrant version. The extra argument *reent* is a pointer to a reentrancy structure.

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.

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

1.23 strtol—string to long

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

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

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

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

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

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

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

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

`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`.

`strtol` is ANSI.

No supporting OS subroutines are required.

1.24 strtoul—string to unsigned long

```
#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);
```

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

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

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

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

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

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

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

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

`strtoul` is ANSI.

`strtoul` requires no supporting OS subroutines.

1.25 system—execute command string

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

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

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.

`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`.

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

POSIX.2 requires `system`, and requires that it invoke `/bin/sh`.

Supporting OS subroutines required: `_exit`, `execve`, `fork`, `wait`.

1.26 wctomb—minimal wide char to multibyte converter

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

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.

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

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

`wctomb` requires no supporting OS subroutines.

2 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.

2.1 isalnum—alphanumeric character predicate

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

`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`'.

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

`isalnum` is ANSI C.

No OS subroutines are required.

2.2 isalpha—alphabetic character predicate

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

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’.

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

isalpha is ANSI C.

No supporting OS subroutines are required.

2.3 isascii—ASCII character predicate

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

`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`'.

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

`isascii` is ANSI C.

No supporting OS subroutines are required.

2.4 iscntrl—control character predicate

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

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’.

iscntrl returns non-zero if *c* is a delete character or ordinary control character (0x7F or 0x00–0x1F).

iscntrl is ANSI C.

No supporting OS subroutines are required.

2.5 isdigit—decimal digit predicate

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

`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`'.

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

`isdigit` is ANSI C.

No supporting OS subroutines are required.

2.6 islower—lower-case character predicate

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

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'.

islower returns non-zero if *c* is a lower case letter (a–z).

islower is ANSI C.

No supporting OS subroutines are required.

2.7 isprint, isgraph—printable character predicates

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

`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`'.

`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.

`isprint` and `isgraph` are ANSI C.

No supporting OS subroutines are required.

2.8 ispunct—punctuation character predicate

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

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'.

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

ispunct is ANSI C.

No supporting OS subroutines are required.

2.9 isspace—whitespace character predicate

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

`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`'.

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

`isspace` is ANSI C.

No supporting OS subroutines are required.

2.10 isupper—uppercase character predicate

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

`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`’.

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

`isupper` is ANSI C.

No supporting OS subroutines are required.

2.11 isxdigit—hexadecimal digit predicate

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

`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`'.

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

`isxdigit` is ANSI C.

No supporting OS subroutines are required.

2.12 toascii—force integers to ASCII range

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

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’.

toascii returns integers between 0 and 127.

toascii is not ANSI C.

No supporting OS subroutines are required.

2.13 tolower—translate characters to lower case

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

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).

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.

tolower is ANSI C. _tolower is not recommended for portable programs.

No supporting OS subroutines are required.

2.14 toupper—translate characters to upper case

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

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).

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.

toupper is ANSI C. _toupper is not recommended for portable programs.

No supporting OS subroutines are required.

3 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.

3.1 clearerr—clear file or stream error indicator

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

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

Use clearerr to reset both of these indicators.

See ferror and feof to query the two indicators.

clearerr does not return a result.

ANSI C requires clearerr.

No supporting OS subroutines are required.

3.2 fclose—close a file

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

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)`).

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

`fclose` is required by ANSI C.

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

3.3 feof—test for end of file

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

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

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

feof is required by ANSI C.

No supporting OS subroutines are required.

3.4 ferror—test whether read/write error has occurred

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

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

See clearerr to reset the error indicator.

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

ANSI C requires ferror.

No supporting OS subroutines are required.

3.5 fflush—flush buffered file output

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

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.

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

ANSI C requires `fflush`.

No supporting OS subroutines are required.

3.6 fgetc—get a character from a file or stream

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

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`.

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.

ANSI C requires `fgetc`.

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

3.7 fgetpos—record position in a stream or file

```
#include <stdio.h>
int fgetpos(FILE *fp, fpos_t *pos);
```

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.

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

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.

3.8 fgets—get character string from a file or stream

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

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

fgets 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.

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.)

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

3.9 fiprintf—format output to file (integer only)

```
#include <stdio.h>

int fiprintf(FILE *fd, const char *format, ...);
```

`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.

`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`.

`fiprintf` is not required by ANSI C.

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

3.10 fopen—open a file

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

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

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

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

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

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

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

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

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

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

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

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

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

`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`.

`fopen` is required by ANSI C.

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

3.11 fdopen—turn open file into a stream

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

`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`.

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

`fdopen` is ANSI.

3.12 fputc—write a character on a stream or file

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

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

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

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

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

`fputc` is required by ANSI C.

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

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

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

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

If successful, the result is `0`; otherwise, the result is `EOF`.

ANSI C requires `fputs`, but does not specify that the result on success must be `0`; any non-negative value is permitted.

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

3.14 `fread`—read array elements from a file

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

`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.

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

ANSI C requires `fread`.

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

3.15 freopen—open a file using an existing file descriptor

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

Use this variant of `fopen` if you wish to specify a particular file descriptor *fp* (notably `stdin`, `stdout`, or `stderr`) for the file.

If *fp* was associated with another file or stream, `freopen` closes that other file or stream (but ignores any errors while closing it).

file and *mode* are used just as in `fopen`.

If successful, the result is the same as the argument *fp*. If the file cannot be opened as specified, the result is `NULL`.

ANSI C requires `freopen`.

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

3.16 fseek—set file position

```
#include <stdio.h>
int fseek(FILE *fp, long offset, int whence)
```

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.

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).

ANSI C requires fseek.

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

3.17 fsetpos—restore position of a stream or file

```
#include <stdio.h>
int fsetpos(FILE *fp, const fpos_t *pos);
```

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.

fgetpos returns 0 when successful. If fgetpos fails, the result is 1. The reason for failure is indicated in errno: either ESPIPE (the stream identified by *fp* doesn’t support repositioning) or EINVAL (invalid file position).

ANSI C requires fsetpos, but does not specify the nature of **pos* beyond identifying it as written by fgetpos.

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

3.18 `ftell`—return position in a stream or file

```
#include <stdio.h>
long ftell(FILE *fp);
```

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`.

`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`.

`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.

3.19 fwrite—write array elements

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

`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.

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.

ANSI C requires `fwrite`.

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

3.20 getc—read a character (macro)

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

getc is a macro, defined in stdio.h. You can use getc to get the next single character from the file or stream identified by *fp*. As a side effect, getc advances the file's current position indicator.

For a subroutine version of this macro, see fgetc.

The 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.

ANSI C requires getc; it suggests, but does not require, that getc be implemented as a macro. The standard explicitly permits macro implementations of getc to use the argument more than once; therefore, in a portable program, you should not use an expression with side effects as the getc argument.

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

3.21 getchar—read a character (macro)

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

int _getchar_r(void *reent);
```

getchar is a macro, defined in stdio.h. You can use getchar to get the next single character from the standard input stream. As a side effect, getchar advances the standard input's current position indicator.

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

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)'.

ANSI C requires getchar; it suggests, but does not require, that getchar be implemented as a macro.

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

3.22 gets—get character string (obsolete, use fgets instead)

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

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.

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

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

3.23 iprintf—write formatted output (integer only)

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

`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.

`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`.

`iprintf` is not required by ANSI C.

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

3.24 mktemp, mkstemp—generate unused file name

```
#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);
```

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.

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 -1.

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

Supporting OS subroutines required: getpid, open, stat.

3.25 perror—print an error message on standard error

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

void _perror_r(void *reent, char *prefix);
```

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.

`perror` returns no result.

ANSI C requires `perror`, but the strings issued vary from one implementation to another.

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

3.26 putc—write a character (macro)

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

`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`.

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

ANSI C requires `putc`; it suggests, but does not require, that `putc` be implemented as a macro. The standard explicitly permits macro implementations of `putc` to use the *fp* argument more than once; therefore, in a portable program, you should not use an expression with side effects as this argument.

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

3.27 putchar—write a character (macro)

```
#include <stdio.h>
int putchar(int ch);

int _putchar_r(void *reent, int ch);
```

putchar is a macro, defined in stdio.h. putchar writes its argument to the standard output stream, after converting it from an int to an unsigned char.

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

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

ANSI C requires putchar; it suggests, but does not require, that putchar be implemented as a macro.

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

3.28 puts—write a character string

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

int _puts_r(void *reent, const char *s);
```

`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.

If successful, the result is `0`; otherwise, the result is `EOF`.

ANSI C requires `puts`, but does not specify that the result on success must be `0`; any non-negative value is permitted.

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

3.29 remove—delete a file's name

```
#include <stdio.h>
int remove(char *filename);

int _remove_r(void *reent, char *filename);
```

Use `remove` to dissolve the association between a particular filename (the string at *filename*) and the file it represents. After calling `remove` with a particular filename, you will no longer be able to open the file by that name.

In this implementation, you may use `remove` on an open file without error; existing file descriptors for the file will continue to access the file's data until the program using them closes the file.

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

`remove` returns `0` if it succeeds, `-1` if it fails.

ANSI C requires `remove`, but only specifies that the result on failure be nonzero. The behavior of `remove` when you call it on an open file may vary among implementations.

Supporting OS subroutine required: `unlink`.

3.30 rename—rename a file

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

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

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.

The result is either `0` (when successful) or `-1` (when the file could not be renamed).

ANSI C requires `rename`, but only specifies that the result on failure be nonzero. The effects of using the name of an existing file as **new* may vary from one implementation to another.

Supporting OS subroutines required: `link`, `unlink`.

3.31 rewind—reinitialize a file or stream

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

`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.

`rewind` does not return a result.

ANSI C requires `rewind`.

No supporting OS subroutines are required.

3.32 setbuf—specify full buffering for a file or stream

```
#include <stdio.h>
void setbuf(FILE *fp, char *buf);
```

`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.

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*.

`setbuf` does not return a result.

Both ANSI C and the System V Interface Definition (Issue 2) require `setbuf`. However, they differ on the meaning of a `NULL` buffer pointer: the SVID issue 2 specification says that a `NULL` buffer pointer requests unbuffered output. For maximum portability, avoid `NULL` buffer pointers.

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

3.33 setvbuf—specify file or stream buffering

```
#include <stdio.h>
int setvbuf(FILE *fp, char *buf,
    int mode, size_t size);
```

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.

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*.

A `0` result indicates success, `EOF` failure (invalid *mode* or *size* can cause failure).

Both ANSI C and the System V Interface Definition (Issue 2) require `setvbuf`. However, they differ on the meaning of a `NULL` buffer pointer: the SVID issue 2 specification says that a `NULL` buffer pointer requests unbuffered output. For maximum portability, avoid `NULL` buffer pointers.

Both specifications describe the result on failure only as a nonzero value.

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

3.34 `siprintf`—write formatted output (integer only)

```
#include <stdio.h>

int siprintf(char *str, const char *format [, arg, ...]);
```

`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.

`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.

`siprintf` is not required by ANSI C.

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

3.35 printf, fprintf, sprintf—format output

```
#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, ...]);
```

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 (other than %), which are copied unchanged to the output, and conversion specifications, each of which is introduced by %. (To include % in the output, use %% in the format string.) A conversion specification has the following form:

%[*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.

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. Similarily, 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`)

  `o` 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`).

`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`.

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

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

3.36 scanf, fscanf, sscanf—scan and format input

```
#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, ...]);
```

scanf scans a series of input fields from standard input, one character at a time. Each field is interpreted according to a format specifier passed to scanf in the format string at **format*. scanf stores the interpreted input from each field at the address passed to it as the corresponding argument following *format*. You must supply the same number of format specifiers and address arguments as there are input fields.

There must be sufficient address arguments for the given format specifiers; if not the results are unpredictable and likely disasterous. Excess address arguments are merely ignored.

scanf often produces unexpected results if the input diverges from an expected pattern. Since the combination of gets or fgets followed by sscanf is safe and easy, that is the preferred way to be certain that a program is synchronized with input at the end of a line.

fscanf and sscanf are identical to scanf, other than the source of input: fscanf reads from a file, and sscanf from a string.

The string at **format* is a character sequence composed of zero or more directives. Directives are composed of one or more whitespace characters, non-whitespace characters, and format specifications.

Whitespace characters are blank ( ), tab (\t), or newline (\n). When scanf encounters a whitespace character in the format string it will read (but not store) all consecutive whitespace characters up to the next non-whitespace character in the input.

Non-whitespace characters are all other ASCII characters except the percent sign (%). When scanf encounters a non-whitespace character in the format string it will read, but not store a matching non-whitespace character.

Format specifications tell `scanf` to read and convert characters from the input field into specific types of values, and store then in the locations specified by the address arguments.

Trailing whitespace is left unread unless explicitly matched in the format string.

The format specifiers must begin with a percent sign (`%`) and have the following form:

> `%[*][`*width*`][`*size*`]`*type*

Each format specification begins with the percent character (`%`). The other fields are:

`*`
: an optional marker; if present, it suppresses interpretation and assignment of this input field.

width
: an optional maximum field width: a decimal integer, which controls the maximum number of characters that will be read before converting the current input field. If the input field has fewer than *width* characters, `scanf` reads all the characters in the field, and then proceeds with the next field and its format specification.

  If a whitespace or a non-convertable character occurs before *width* character are read, the characters up to that character are read, converted, and stored. Then `scanf` proceeds to the next format specification.

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

```
Modifier   Type(s)
   h       d, i, o, u, x     convert input to short,
                             store in short object

   h       D, I, O, U, X     no effect
           e, f, c, s, n, p

   l       d, i, o, u, x     convert input to long,
                             store in long object

   l       e, f, g           convert input to double
                             store in a double object

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

   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)`.
`o`	Reads an octal integer into the corresponding *arg*: `(int *arg)`.
`O`	Reads an octal integer into the corresponding *arg*: `(long *arg)`.
`u`	Reads an unsigned decimal integer into the corresponding *arg*: `(unsigned int *arg)`.
`U`	Reads an unsigned decimal integer into the corresponding *arg*: `(unsigned long *arg)`.
`x,X`	Read a hexadecimal integer into the corresponding *arg*: `(int *arg)`.
`e, f, g`	Read a floating point number into the corresponding *arg*: `(float *arg)`.
`E, F, G`	Read a floating point number into the corresponding *arg*: `(double *arg)`.
`i`	Reads a decimal, octal or hexadecimal integer into the corresponding *arg*: `(int *arg)`.
`I`	Reads a decimal, octal or hexadecimal integer into the corresponding *arg*: `(long *arg)`.
`n`	Stores the number of characters read in the corresponding *arg*: `(int *arg)`.
`p`	Stores a scanned pointer. ANSI C leaves the details to each implementation; this implementation treats `%p` exactly the same as `%U`. Corresponding *arg*: `(void **arg)`.

A *pattern* of characters surrounded by square brackets can be used instead of the `s` type character. *pattern* is a set of characters which define a search set of possible characters making up the `scanf` input field. If the first character in the brackets is a caret (`^`), the search set is inverted to include all ASCII characters except those between the brackets.

There is also a range facility which you can use as a shortcut. `%[0-9]` matches all decimal digits. The hyphen must not be the first or last character in the set. The character prior to the hyphen must be lexically less than the character after it.

Here are some *pattern* examples:

`%[abcd]` matches strings containing only `a`, `b`, `c`, and `d`.

`%[^abcd]` matches strings containing any characters except `a`, `b`, `c`, or `d`

`%[A-DW-Z]`
matches strings containing `A`, `B`, `C`, `D`, `W`, `X`, `Y`, `Z`

`%[z-a]` matches the characters `z`, `-`, and `a`

Floating point numbers (for field types `e`, `f`, `g`, `E`, `F`, `G`) must correspond to the following general form:

```
[+/-] ddddd[.]ddd [E|e[+|-]ddd]
```

where objects inclosed in square brackets are optional, and `ddd` represents decimal, octal, or hexadecimal digits.

`scanf` returns the number of input fields successfully scanned, converted and stored; the return value does not include scanned fields which were not stored.

If `scanf` attempts to read at end-of-file, the return value is `EOF`.

If no fields were stored, the return value is `0`.

`scanf` might stop scanning a particular field before reaching the normal field end character, or may terminate entirely.

`scanf` stops scanning and storing the current field and moves to the next input field (if any) in any of the following situations:

- The assignment suppressing character (`*`) appears after the `%` in the format specification; the current input field is scanned but not stored.
- *width* characters have been read (*width* is a width specification, a positive decimal integer).
- The next character read cannot be converted under the the current format (for example, if a `Z` is read when the format is decimal).
- The next character in the input field does not appear in the search set (or does appear in the inverted search set).

When `scanf` stops scanning the current input field for one of these reasons, the next character is considered unread and used as the first character of the following input field, or the first character in a subsequent read operation on the input.

`scanf` will terminate under the following circumstances:

- The next character in the input field conflicts with a corresponding non-whitespace character in the format string.
- The next character in the input field is `EOF`.
- The format string has been exhausted.

When the format string contains a character sequence that is not part of a format specification, the same character sequence must appear in the input; `scanf` will scan but not store the matched characters. If a conflict occurs, the first conflicting character remains in the input as if it had never been read.

`scanf` is ANSI C.

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

3.37 tmpfile—create a temporary file

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

FILE *_tmpfile_r(void *reent);
```

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.

`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.

Both ANSI C and the System V Interface Definition (Issue 2) require `tmpfile`.

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

`tmpfile` also requires the global pointer `environ`.

3.38 tmpnam, tempnam—name for a temporary file

```
#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);
```

Use either of these functions to generate a name for a temporary file. The generated name is guaranteed to avoid collision with other files (for up to TMP_MAX calls of either function).

tmpnam generates file names with the value of P_tmpdir (defined in 'stdio.h') as the leading directory component of the path.

You can use the tmpnam argument *s* to specify a suitable area of memory for the generated filename; otherwise, you can call tmpnam(NULL) to use an internal static buffer.

tempnam allows you more control over the generated filename: you can use the argument *dir* to specify the path to a directory for temporary files, and you can use the argument *pfx* to specify a prefix for the base filename.

If *dir* is NULL, tempnam will attempt to use the value of environment variable TMPDIR instead; if there is no such value, tempnam uses the value of P_tmpdir (defined in 'stdio.h').

If you don't need any particular prefix to the basename of temporary files, you can pass NULL as the *pfx* argument to tempnam.

_tmpnam_r and _tempnam_r are reentrant versions of tmpnam and tempnam respectively. The extra argument *reent* is a pointer to a reentrancy structure.

The generated filenames are suitable for temporary files, but do not in themselves make files temporary. Files with these names must still be explicitly removed when you no longer want them.

If you supply your own data area *s* for tmpnam, you must ensure that it has room for at least L_tmpnam elements of type char.

Both `tmpnam` and `tempnam` return a pointer to the newly generated filename.

ANSI C requires `tmpnam`, but does not specify the use of `P_tmpdir`. The System V Interface Definition (Issue 2) requires both `tmpnam` and `tempnam`.

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

The global pointer `environ` is also required.

3.39 vprintf, vfprintf, vsprintf—format argument list

```
#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);
```

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 va_list object (initialized by va_start) rather than directly accepting a variable number of arguments.

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.

ANSI C requires all three functions.

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

4 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’.

4.1 bcmp—compare two memory areas

```
#include <string.h>
int bcmp(const char *s1, const char *s2, size_t n);
```

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`.

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*.

`bcmp` requires no supporting OS subroutines.

4.2 bcopy—copy memory regions

```
#include <string.h>
void bcopy(const char *in, char  *out, size_t n);
```

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`.

`bcopy` requires no supporting OS subroutines.

4.3 bzero—initialize memory to zero

```
#include <string.h>
void bzero(char *b, size_t length);
```

bzero initializes *length* bytes of memory, starting at address *b*, to zero.

bzero does not return a result.

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.

4.4 index—search for character in string

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

This function finds the first occurence of *c* (converted to a char) in the string pointed to by *string* (including the terminating null character).

This function is identical to `strchr`.

Returns a pointer to the located character, or a null pointer if *c* does not occur in *string*.

`index` requires no supporting OS subroutines.

4.5 memchr—find character in memory

```
#include <string.h>
void *memchr(const void *src, int c, size_t length);
```

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.

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.

memchr> is ANSI C.

memchr requires no supporting OS subroutines.

4.6 memcmp—compare two memory areas

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

This function compares not more than *n* characters of the object pointed to by *s1* with the object pointed to by *s2*.

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*.

`memcmp` is ANSI C.

`memcmp` requires no supporting OS subroutines.

4.7 memcpy—copy memory regions

```
#include <string.h>
void* memcpy(void *out, const void *in, size_t n);
```

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.

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

`memcpy` is ANSI C.

`memcpy` requires no supporting OS subroutines.

4.8 memmove—move possibly overlapping memory

```
#include <string.h>
void *memmove(void *dst, const void *src, size_t length);
```

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.

The function returns *dst* as passed.

`memmove` is ANSI C.

`memmove` requires no supporting OS subroutines.

4.9 memset—set an area of memory

```
#include <string.h>
void *memset(const void *dst, int c, size_t length);
```

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.

memset returns the value of *m*.

memset is ANSI C.

memset requires no supporting OS subroutines.

4.10 rindex—reverse search for character in string

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

This function finds the last occurence of *c* (converted to a char) in the string pointed to by *string* (including the terminating null character).

This function is identical to `strrchr`.

Returns a pointer to the located character, or a null pointer if *c* does not occur in *string*.

`rindex` requires no supporting OS subroutines.

4.11 strcat—concatenate strings

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

`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*.

This function returns the initial value of *dst*

`strcat` is ANSI C.

`strcat` requires no supporting OS subroutines.

4.12 strchr—search for character in string

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

This function finds the first occurence of *c* (converted to a char) in the string pointed to by *string* (including the terminating null character).

Returns a pointer to the located character, or a null pointer if *c* does not occur in *string*.

`strchr` is ANSI C.

`strchr` requires no supporting OS subroutines.

4.13 strcmp—character string compare

```
#include <string.h>
int strcmp(const char *a, const char *b);
```

`strcmp` compares the string at *a* to the string at *b*.

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.

`strcmp` is ANSI C.

`strcmp` requires no supporting OS subroutines.

4.14 strcoll—locale specific character string compare

```
#include <string.h>
int strcoll(const char *stra, const char * strb);
```

`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.

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.

`strcoll` is ANSI C.

`strcoll` requires no supporting OS subroutines.

4.15 strcpy—copy string

```
#include <string.h>
char *strcpy(char *dst, const char *src);
```

`strcpy` copies the string pointed to by *src* (including the terminating null character) to the array pointed to by *dst*.

This function returns the initial value of *dst*.

`strcpy` is ANSI C.

`strcpy` requires no supporting OS subroutines.

4.16 strcspn—count chars not in string

```
size_t strcspn(const char *s1, const char *s2);
```

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).

`strcspn` returns the length of the substring found.

`strcspn` is ANSI C.

`strcspn` requires no supporting OS subroutines.

4.17 strerror—convert error number to string

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

`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

`ENFILE` File table overflow

`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

`ENOTBLK` Block device required

`ENOTDIR` Not a directory

`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

This function returns a pointer to a string. Your application must not modify that string.

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.

`strerror` requires no supporting OS subroutines.

4.18 strlen—character string length

```
#include <string.h>
size_t strlen(const char *str);
```

The `strlen` function works out the length of the string starting at **str* by counting chararacters until it reaches a `NULL` character.

`strlen` returns the character count.

`strlen` is ANSI C.

`strlen` requires no supporting OS subroutines.

4.19 strncat—concatenate strings

```
#include <string.h>
char *strncat(char *dst, const char *src, size_t length);
```

`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

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`.

This function returns the initial value of *dst*

`strncat` is ANSI C.

`strncat` requires no supporting OS subroutines.

4.20 strncmp—character string compare

```
#include <string.h>
int strncmp(const char *a, const char * b, size_t length);
```

`strncmp` compares up to *length* characters from the string at *a* to the string at *b*.

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.

`strncmp` is ANSI C.

`strncmp` requires no supporting OS subroutines.

4.21 strncpy—counted copy string

```
#include <string.h>
char *strncpy(char *dst, const char *src, size_t length);
```

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.

This function returns the initial value of *dst*.

strncpy is ANSI C.

strncpy requires no supporting OS subroutines.

4.22 strpbrk—find chars in string

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

This function locates the first occurence in the string pointed to by *s1* of any character in string pointed to by *s2* (excluding the terminating null character).

strpbrk returns a pointer to the character found in *s1*, or a null pointer if no character from *s2* occurs in *s1*.

strpbrk requires no supporting OS subroutines.

4.23 strrchr—reverse search for character in string

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

This function finds the last occurence of *c* (converted to a char) in the string pointed to by *string* (including the terminating null character).

Returns a pointer to the located character, or a null pointer if *c* does not occur in *string*.

`strrchr` is ANSI C.

`strrchr` requires no supporting OS subroutines.

4.24 strspn—find initial match

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

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).

`strspn` returns the length of the segment found.

`strspn` is ANSI C.

`strspn` requires no supporting OS subroutines.

4.25 strstr—find string segment

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

Locates the first occurence 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 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.

`strstr` is ANSI C.

`strstr` requires no supporting OS subroutines.

4.26 strtok—get next token from a string

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

char *_strtok_r(void *reent,
    const char *source, const char *delimiters)
```

A series of calls to `strtok` break the string starting at `*source` into a sequence of tokens. The tokens are delimited from one another by characters from the string at `*delimiters`, at the outset. The first call to `strtok` normally has a string address as the first argument; subsequent calls can use `NULL` as the first argument, to continue searching the same string. You can continue searching a single string with different delimiters by using a different delimiter string on each call.

`strtok` begins by searching for any character not in the *delimiters* string: the first such character is the beginning of a token (and its address will be the result of the `strtok` call). `strtok` then continues searching until it finds another delimiter character; it replaces that character by `NULL` and returns. (If `strtok` comes to the end of the `*source` string without finding any more delimiters, the entire remainder of the string is treated as the next token). `strtok` starts its search at `*source`, unless you pass `NULL` as the first argument; if *source* is `NULL`, `strtok` continues searching from the end of the last search. Exploiting the `NULL` first argument leads to non-reentrant code. You can easily circumvent this problem by saving the last delimiter address in your application, and always using it to pass a non-null *source* argument.

`_strtok_r` performs the same function as `strtok`, but is reentrant. The extra argument *reent* is a pointer to a reentrancy structure.

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

`strtok` is ANSI C.

`strtok` requires no supporting OS subroutines.

4.27 strxfrm—transform string

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

This function transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is such that if the `strcmp` function is applied to the two transformed strings, it returns a value greater than, equal to, or less than zero, correspoinding to the result of a `strcoll` function applied to the same two original strings.

No more than *n* characters are placed into the resulting array pointed to by *s1*, including the terminating null character. If *n* is zero, *s1* may be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

With a C locale, this function just copies.

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.

`strxfrm` is ANSI C.

`strxfrm` requires no supporting OS subroutines.

5 Signal Handling ('signal.h')

A *signal* is an event that interrupts the normal flow of control in your program. Your operating environment normally defines the full set of signals available (see 'sys/signal.h'), as well as the default means of dealing with them—typically, either printing an error message and aborting your program, or ignoring the signal.

All systems support at least the following signals:

`SIGABRT` Abnormal termination of a program; raised by the <<abort>> function.

`SIGFPE` A domain error in arithmetic, such as overflow, or division by zero.

`SIGILL` Attempt to execute as a function data that is not executable.

`SIGINT` Interrupt; an interactive attention signal.

`SIGSEGV` An attempt to access a memory location that is not available.

`SIGTERM` A request that your program end execution.

Two functions are available for dealing with asynchronous signals—one to allow your program to send signals to itself (this is called *raising* a signal), and one to specify subroutines (called *handlers* to handle particular signals that you anticipate may occur—whether raised by your own program or the operating environment.

To support these functions, 'signal.h' defines three macros:

`SIG_DFL` Used with the `signal` function in place of a pointer to a handler subroutine, to select the operating environment's default handling of a signal.

`SIG_IGN` Used with the `signal` function in place of a pointer to a handler, to ignore a particular signal.

`SIG_ERR` Returned by the `signal` function in place of a pointer to a handler, to indicate that your request to set up a handler could not be honored for some reason.

'signal.h' also defines an integral type, `sig_atomic_t`. This type is not used in any function declarations; it exists only to allow your signal handlers to declare a static storage location where they may store a signal value. (Static storage is not otherwise reliable from signal handlers.)

5.1 raise—send a signal

```
#include <signal.h>
int raise(int sig);

int _raise_r(void *reent, int sig);
```

Send the signal *sig* (one of the macros from 'sys/signal.h'). This interrupts your program's normal flow of execution, and allows a signal handler (if you've defined one, using `signal`) to take control.

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

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.

ANSI C requires `raise`, but allows the full set of signal numbers to vary from one implementation to another.

Required OS subroutines: `getpid`, `kill`.

5.2 signal—specify handler subroutine for a signal

```
#include <signal.h>
void ( * signal(int sig, void(*func)(int)) )(int);

void ( * _signal_r(void *reent,
    int sig, void(*func)(int)) )(int);

int raise (int sig);

int _raise_r (void *reent, int sig);
```

`signal, raise` provide a simple signal/raise implementation for embedded targets.

`signal` allows you to request changed treatment for a particular signal *sig*. You can use one of the predefined macros `SIG_DFL` (select system default handling) or `SIG_IGN` (ignore this signal) as the value of *func*; otherwise, *func* is a function pointer that identifies a subroutine in your program as the handler for this signal.

Some of the execution environment for signal handlers is unpredictable; notably, the only library function required to work correctly from within a signal handler is `signal` itself, and only when used to redefine the handler for the current signal value.

Static storage is likewise unreliable for signal handlers, with one exception: if you declare a static storage location as ‘`volatile sig_atomic_t`’, then you may use that location in a signal handler to store signal values.

If your signal handler terminates using `return` (or implicit return), your program's execution continues at the point where it was when the signal was raised (whether by your program itself, or by an external event). Signal handlers can also use functions such as `exit` and `abort` to avoid returning.

`raise` sends the signal sig to the executing program. It returns zero if successful, non-zero if unsuccessful.

The alternate functions `_signal_r, _raise_r` are the reentrant versions. The extra argument *reent* is a pointer to a reentrancy structure.

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).

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.

6 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).

`tm_wday` Day of week: the number of days since Sunday.

`tm_yday` 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.

6.1 asctime—format time as string

```
#include <time.h>
char *asctime(const struct tm *timp);

#include <time.h>
char *_asctime_r(const struct tm *timp, void *reent);
```

Format the time value at *timp* 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.

`_asctime_r` provides the same function as `asctime`, but is reentrant. The extra argument *reent* is a pointer to a reentrancy structure.

A pointer to the string containing a formatted timestamp.

ANSI C requires `asctime`.

`asctime` requires no supporting OS subroutines.

6.2 clock—cumulative processor time

```
#include <time.h>
clock_t clock(void);
```

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.

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.

ANSI C requires clock and CLOCKS_PER_SEC.

Supporting OS subroutine required: times.

6.3 ctime—convert time to local and format as string

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

Convert the time value at *timp* 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`).

A pointer to the string containing a formatted timestamp.

ANSI C requires `ctime`.

`ctime` requires no supporting OS subroutines.

6.4 difftime—subtract two times

```
#include <time.h>
double difftime(time_t tim1, time_t tim2);
```

Subtracts the two times in the arguments: '*tim1* - *tim2*'.

The difference (in seconds) between *tim2* and *tim1*, as a `double`.

ANSI C requires `difftime`, and defines its result to be in seconds in all implementations.

`difftime` requires no supporting OS subroutines.

6.5 gmtime—convert time to UTC traditional form

```
#include <time.h>
struct tm *gmtime(const time_t *timep
```

`gmtime` assumes the time at *timep* 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.

A pointer to the traditional time representation (`struct tm`).

ANSI C requires `gmtime`.

`gmtime` requires no supporting OS subroutines.

6.6 localtime—convert time to local representation

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

`localtime` converts the time at *timep* 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`.

A pointer to the traditional time representation (`struct tm`).

ANSI C requires `localtime`.

`localtime` requires no supporting OS subroutines.

6.7 mktime—convert time to arithmetic representation

```
#include <time.h>
time_t mktime(struct tm *timp);
```

`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`.

If the contents of the structure at *timp* do not form a valid calendar time representation, the result is `-1`. Otherwise, the result is the time, converted to a `time_t` value.

ANSI C requires `mktime`.

`mktime` requires no supporting OS subroutines.

6.8 strftime—flexible calendar time formatter

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

`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, '%'.

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.

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.

6.9 time—get current calendar time (as single number)

```
#include <time.h>
time_t time(time_t *t);
```

`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`.

A `-1` result means the current time is not available; otherwise the result represents the current time.

ANSI C requires `time`.

Supporting OS subroutine required: Some implementations require `gettimeofday`.

7 Locale ('locale.h')

A *locale* is the name for a collection of parameters (affecting collating sequences and formatting conventions) that may be different depending on location or culture. The "C" locale is the only one defined in the ANSI C standard.

This is a minimal implementation, supporting only the required ``C'' value for locale; strings representing other locales are not honored. (``'' is also accepted; it represents the default locale for an implementation, here equivalent to ``C''.

'locale.h' defines the structure lconv to collect the information on a locale, with the following fields:

char *decimal_point
: The decimal point character used to format "ordinary" numbers (all numbers except those referring to amounts of money). ``.'' in the C locale.

char *thousands_sep
: The character (if any) used to separate groups of digits, when formatting ordinary numbers. ``'' in the C locale.

char *grouping
: Specifications for how many digits to group (if any grouping is done at all) when formatting ordinary numbers. The *numeric value* of each character in the string represents the number of digits for the next group, and a value of 0 (that is, the string's trailing NULL) means to continue grouping digits using the last value specified. Use CHAR_MAX to indicate that no further grouping is desired. ``'' in the C locale.

char *int_curr_symbol
: The international currency symbol (first three characters), if any, and the character used to separate it from numbers. ``'' in the C locale.

char *currency_symbol
: The local currency symbol, if any. ``'' in the C locale.

char *mon_decimal_point
: The symbol used to delimit fractions in amounts of money. ``'' in the C locale.

char *mon_thousands_sep
: Similar to thousands_sep, but used for amounts of money. ``'' in the C locale.

char *mon_grouping
: Similar to grouping, but used for amounts of money. ``'' in the C locale.

char *positive_sign
: A string to flag positive amounts of money when formatting. ``'' in the C locale.

`char *negative_sign`
: A string to flag negative amounts of money when formatting. `''` in the C locale.

`char int_frac_digits`
: The number of digits to display when formatting amounts of money to international conventions. `CHAR_MAX` (the largest number representable as a `char`) in the C locale.

`char frac_digits`
: The number of digits to display when formatting amounts of money to local conventions. `CHAR_MAX` in the C locale.

`char p_cs_precedes`
: `1` indicates the local currency symbol is used before a *positive or zero* formatted amount of money; `0` indicates the currency symbol is placed after the formatted number. `CHAR_MAX` in the C locale.

`char p_sep_by_space`
: `1` indicates the local currency symbol must be separated from *positive or zero* numbers by a space; `0` indicates that it is immediately adjacent to numbers. `CHAR_MAX` in the C locale.

`char n_cs_precedes`
: `1` indicates the local currency symbol is used before a *negative* formatted amount of money; `0` indicates the currency symbol is placed after the formatted number. `CHAR_MAX` in the C locale.

`char n_sep_by_space`
: `1` indicates the local currency symbol must be separated from *negative* numbers by a space; `0` indicates that it is immediately adjacent to numbers. `CHAR_MAX` in the C locale.

`char p_sign_posn`
: Controls the position of the *positive* sign for numbers representing money. `0` means parentheses surround the number; `1` means the sign is placed before both the number and the currency symbol; `2` means the sign is placed after both the number and the currency symbol; `3` means the sign is placed just before the currency symbol; and `4` means the sign is placed just after the currency symbol. `CHAR_MAX` in the C locale.

`char n_sign_posn`
: Controls the position of the *negative* sign for numbers representing money, using the same rules as `p_sign_posn`. `CHAR_MAX` in the C locale.

7.1 setlocale, localeconv—select or query locale

```
#include <locale.h>
char *setlocale(int category, const char *locale);
lconv *localeconv(void);

char *_setlocale_r(void *reent,
    int category, const char *locale);
lconv *_localeconv_r(void *reent);
```

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 for *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 "LC_", 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.

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).

ANSI C requires `setlocale`, but the only locale required across all implementations is the C locale.

No supporting OS subroutines are required.

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

Signal functions:

```
_raise_r        _signal_r
```

Stdlib functions:

```
_dtoa_r         _realloc_r      _strtoul_r
_free_r         _srand_r        _system_r
_malloc_r       _strtod_r
_rand_r         _strtol_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
```

9 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`.

9.1 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 8 [Reentrancy], page 155).

The bridge between these two interpretations of `errno` is straightforward: the C library routines with OS interface calls capture the `errno` values returned globally, and record them in the appropriate field of the reentrancy structure (so that you can query them using the `errno` macro from '`errno.h`').

This mechanism becomes visible when you write stub routines for OS interfaces. You must include '`errno.h`', then disable the macro, like this:

```
#include <errno.h>
#undef errno
extern int errno;
```

The examples in this chapter include this treatment of `errno`.

`_exit` Exit a program without cleaning up files. If your system doesn't provide this, it is best to avoid linking with subroutines that require it (`exit`, `system`).

`close` Close a file. Minimal implementation:

```
int close(int file){
    return -1;
}
```

`environ` A pointer to a list of environment variables and their values. For a minimal environment, this empty list is adequate:

```
char *__env[1] = { 0 };
char **environ = __env;
```

`execve` Transfer control to a new process. Minimal implementation (for a system without processes):

```
#include <errno.h>
#undef errno
extern int errno;
int execve(char *name, char **argv, char **env){
  errno=ENOMEM;
  return -1;
}
```

`fork` Create a new process. Minimal implementation (for a system without processes):

```
#include <errno.h>
#undef errno
extern int errno;
int fork() {
  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;
}
```

`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;
  heap_end += incr;
  return (caddr_t) prev_heap_end;
}
```

`stat` Status of a file (by name). Minimal implementation:

```
int stat(char *file, struct stat *st) {
  st->st_mode = S_IFCHR;
  return 0;
}
```

`times` Timing information for current process. Minimal implementation:

```
int times(struct tms *buf){
  return -1;
}
```

`unlink` Remove a file's directory entry. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int unlink(char *name){
  errno=ENOENT;
  return -1;
}
```

`wait` Wait for a child process. Minimal implementation:

```
#include <errno.h>
#undef errno
extern int errno;
int wait(int *status) {
  errno=ECHILD;
  return -1;
}
```

`write` Write 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;
}
```

9.2 Reentrant covers for OS subroutines

Since the system subroutines are used by other library routines that require reentrancy, 'libc.a' provides cover routines (for example, the reentrant version of fork is _fork_r). These cover routines are consistent with the other reentrant subroutines in this library, and achieve reentrancy by using a reserved global data block (see Chapter 8 [Reentrancy], page 155).

_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);
```

10 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).

10.1 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`.

10.1.1 Initialize variable argument list

```
#include <stdarg.h>
void va_start(va_list ap, rightmost);
```

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).

`va_start` does not return a result.

ANSI C requires `va_start`.

10.1.2 Extract a value from argument list

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

`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.

`va_arg` returns the next argument, an object of type *type*.

ANSI C requires `va_arg`.

10.1.3 Abandon a variable argument list

```
#include <stdarg.h>
void va_end(va_list ap);
```

Use `va_end` to declare that your program will not use the variable argument list *ap* any further.

`va_end` does not return a result.

ANSI C requires `va_end`.

10.2 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.

10.2.1 Declare variable arguments

```
#include <varargs.h>
function(va_alist)
va_dcl
```

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`.

These macros cannot be used in a context where a return is syntactically possible.

va_alist and *va_dcl* were the most widespread method of declaring variable argument lists prior to ANSI C.

10.2.2 Initialize variable argument list

```
#include <varargs.h>
va_list ap;
va_start(ap);
```

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*.

va_start does not return a result.

va_start is also defined as a macro in ANSI C, but the definitions are incompatible; the ANSI version has another parameter besides *ap*.

10.2.3 Extract a value from argument list

```
#include <varargs.h>
type va_arg(va_list ap, type);
```

`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*.

`va_arg` returns the next argument, an object of type *type*.

The `va_arg` defined in '`varargs.h`' has the same syntax and usage as the ANSI C version from '`stdarg.h`'.

10.2.4 Abandon a variable argument list

```
#include <varargs.h>
va_end(va_list ap);
```

Use `va_end` to declare that your program will not use the variable argument list *ap* any further.

`va_end` does not return a result.

The `va_end` defined in '`varargs.h`' has the same syntax and usage as the ANSI C version from '`stdarg.h`'.

Index

F

G

I

K

L

M

O

P

Q

R

S

T

U

V

W

The body of this manual is set in
cmr10 at 10.95pt,
with headings in
and examples in `cmtt10 at 10.95pt`.
cmti10 at 10.95pt and
cmsl10 at 10.95pt
are used for emphasis.

Table of Contents