Sega Dreamcast

Dreamcast GNUPro™ Toolkit Libraries

GNUPro C Library

GNUPro Math Library

GNU C++ Iostream library

Important Information

This documentation has been provided courtesy of CYGNUS. The contents are applicable to GNUPro™ Toolkit development, however, all references to development support offered by CYGNUS should be ignored.

Technical support for this product as it applies to the Sega Dreamcast™ development environment should be directed to Sega Third Party Developer Technical Support at 415/701-4060. Future updates and/or additional information may also be found at Sega's DTS Website at, **`http//:www.dts.sega.com/NextGen`**

This documentation has been prepared by Cygnus Technical Publications; contact the Cygnus Technical Publications staff: `doc@cygnus.com`.

GNUPro Warranty

The GNUPro Toolkit is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. This version of GNUPro Toolkit is supported for customers of Cygnus.

For non-customers, GNUPro Toolkit software has NO WARRANTY.

Because this software is licensed free of charge, there are no warranties for it, to the extent permitted by applicable law. Except when otherwise stated in writing, the copyright holders and/or other parties provide the software “as is” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the software is with you. Should the software prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event, unless required by applicable law or agreed to in writing, will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

How to contact Cygnus

Use the following information for contacting Cygnus.

Cygnus Headquarters

1325 Chesapeake Terrace
Sunnyvale, CA 94089 USA
Telephone (toll free): +1 800 CYGNUS-1
Telephone (main line): +1 408 542 9600
Telephone (hotline): +1 408 542 9601
FAX: +1-408 542 9699
(Faxes are answered 8 a.m.–5 p.m., Monday through Friday.)
email: `info@cygnus.com`
Website: `www.cygnus.com`.

Cygnus United Kingdom

36 Cambridge Place
Cambridge CB2 1NS
United Kingdom
Telephone: +44 1223 728728
FAX: +44 1223 728728
email: `info@cygnus.co.uk/`

Cygnus Japan

Nihon Cygnus
Madre Matsuda Building
4-13 Kioi-cho Chiyoda-ku
Tokyo 102
Telephone: +81 3 3234 3896
FAX: +81 3 3239 3300
email: `info@cygnus.co.jp`
Website: `http://www.cygnus.co.jp/`

Use the hotline (**+1 408 542 9601**) to get help, although the most reliable way to resolve problems with GNUPro Toolkit is by using email:

`bugs@cygnus.com`.

Contents

GNUPro Math Library

GNUPRO™ TOOLKIT

GNUPro C Library

June, 1998
98r1

CYGNUS

This documentation has been prepared by Cygnus Technical Publications; contact the Cygnus Technical Publications staff: **doc@cygnus.com**.

1

Standard Utility Functions (`stdlib.h`)

The following documentation groups utility functions, useful in a variety of programs, corresponding to declarations in the header file, `stdlib.h`.

abort

[abnormal termination of a program]

SYNOPSIS

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

DESCRIPTION

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

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

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

RETURNS

`abort` does not return to its caller.

COMPLIANCE

ANSI C requires `abort`.

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

abs

[integer absolute value (magnitude)]

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

The result is a nonnegative integer.

COMPLIANCE

`abs` is ANSI.

No supporting OS subroutines are required.

assert

[macro for debugging diagnostics]

SYNOPSIS

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

DESCRIPTION

Use the macro, `assert`, to embed debugging 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, as in the following example.

```
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, as in the following example.

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

RETURNS

`assert` does not return a value.

COMPLIANCE

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`.

atexit

[request execution of functions at program exit]

SYNOPSIS

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

DESCRIPTION

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

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

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

RETURNS

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

COMPLIANCE

`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`.

atof, atoff

[string to double or float]

SYNOPSIS

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

DESCRIPTION

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

The functions parse the character string, *s*, locating a substring which can be converted to a floating point value. The substring must match the following 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, if it 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)`.

RETURNS

`atof` returns the converted substring value, if any, as a `double`; or `0.0`, if no conversion could be performed. If the correct value is out of the range of representative 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`.

COMPLIANCE

`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`.

atoi, atol

[string to integer]

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

COMPLIANCE

`atoi` is ANSI.

No supporting OS subroutines are required.

bsearch

[binary search]

SYNOPSIS

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

DESCRIPTION

`bsearch` searches an array beginning at *`base`* for any element that matches *`key`*, using binary search. *`nmemb`* is the element count of the array; *`size`* is the size of each element. The array must be sorted in ascending order with respect to the comparison function, *`compar`* (*`compar`* being a variable, replaced with the appropriate comparison function as the last argument of `bsearch`).

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

RETURNS

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

COMPLIANCE

`bsearch` is ANSI.

No supporting OS subroutines are required.

calloc

[allocate space for arrays]

SYNOPSIS

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

void *calloc_r(void *reent, size_t <n>, <size_t> s);
```

DESCRIPTION

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

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

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

RETURNS

If successful, a pointer to the newly allocated space. If unsuccessful, `NULL`.

COMPLIANCE

`calloc` is ANSI.

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

div

[divide two integers]

SYNOPSIS

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

DESCRIPTION

`div` divides *n* by *d*, returning quotient and remainder as two integers in a structure, `div_t`.

RETURNS

The result is represented with the following example.

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

The previous example has the `quot` field representing the quotient, and the `rem` field representing the remainder.

For nonzero *d*, if *r*`=div(`*n*`, `*d*`);`, then *n* equals *r*`.rem + `*d*`*`*r*`.quot`.

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

COMPLIANCE

`div` is ANSI.

No supporting OS subroutines are required.

ecvt, ecvtf, fcvt, fcvtf

[double or float to string]

SYNOPSIS

```
#include <stdlib.h>
char *ecvt(double val, int chars, int *decpt, int *sgn);
char *ecvtf(float val, int chars, int *decpt, int *sgn);

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

DESCRIPTION

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

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

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

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

RETURNS

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

COMPLIANCE

None of these functions are ANSI C.

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

ecvtbuf, fcvtbuf

[double or float to string]

SYNOPSIS

```
#include <stdio.h>
char *ecvtbuf(double val, int chars, int *decpt,
                                  int *sgn, char *buf);

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

DESCRIPTION

`ecvtbuf` and `fcvtbuf` produce (`NULL`-terminated) strings of digits representing the `double` number, *`val`*.

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

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

RETURNS

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

COMPLIANCE

Neither function is ANSI C.

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

exit

[end program execution]

SYNOPSIS

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

DESCRIPTION

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

`exit` does two kinds of cleanup before ending execution of your program.

- It calls all application-defined cleanup functions you have enrolled with `atexit`.
- Files and streams are cleaned up: any pending output is delivered to the host system, each open file or stream is closed, and files created by `tmpfile` are deleted.

RETURNS

`exit` does not return to its caller.

COMPLIANCE

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

Supporting OS subroutines required: `_exit`.

getenv

[look up environment variable]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

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

COMPLIANCE

`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`.

gvcvt, gcvtf

[format double or float as string]

SYNOPSIS

```
#include <stdlib.h>
char *gcvt(double val, int precision, char *buf);
char *gcvtf(float val, int precision, char *buf);
```

DESCRIPTION

`gcvt` writes a fully formatted number as a `NULL`-terminated string in the buffer, `*buf`.

`gcvtf` produces corresponding character representations of `float` numbers.

`gcvt` uses the same rules as the `printf`-format, `%.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`*).

RETURNS

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

COMPLIANCE

Neither function is ANSI C.

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

labs

[long integer absolute value]

SYNOPSIS

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

DESCRIPTION

`labs` returns | *x* |, the absolute value of *`I`* (also called the magnitude of *`I`*). That is, if *`I`* is negative, the result is the opposite of *`I`*; but, if *`I`* is nonnegative, the result is *`I`*. The similar function, `abs`, uses and returns `int` rather than `long` values.

RETURNS

The result is a nonnegative long integer.

COMPLIANCE

`labs` is ANSI.

No supporting OS subroutine calls are required.

ldiv

[divide two long integers]

SYNOPSIS

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

DESCRIPTION

`ldiv` divides *n* by *d*, returning quotient and remainder as two long integers in a structure, `ldiv_t`.

RETURNS

The result is represented with the following example.

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

The previous example has the `quot` field representing the quotient, and `rem` representing the remainder.

For nonzero *d*, if *r*`=ldiv(`*n*`,`*d*`);`, then *n* equals *r*`.rem + `*d*`*`*r*`.quot.`

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

COMPLIANCE

`ldiv` is ANSI.

No supporting OS subroutines are required.

malloc, realloc, free

[manage memory]

SYNOPSIS

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

void *memalign(size_t align, size_t nbytes);

size_t malloc_usable_size(void * aptr);

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

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

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

DESCRIPTION

These functions manage a pool of system memory.

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

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

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

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

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

The `malloc_usable_size` function takes a pointer to a block allocated by `malloc`. It returns the amount of space that is available in the block.

This may or may not be more than the size requested from malloc, due to alignment or minimum size constraints.

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

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

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

See "__malloc_lock, __malloc_unlock" on page 25 for those functions.

These functions operate by calling the functions, _sbrk_r or sbrk, which allocates space. You may need to provide one of these functions for your system. _sbrk_r is called with a positive value to allocate more space, and with a negative value to release previously allocated space if it is no longer required. See "System Calls" on page 165, specifically, "Reentrant covers for OS subroutines" on page 171.

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

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

free does not return a result.

memalign returns a pointer to the newly allocated space.

malloc_usable_size returns the usable size.

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

memalign is part of SVR4.

malloc_usable_size is not portable.

Supporting OS subroutines required: sbrk.

mallinfo, malloc_stats, mallopt

[malloc support]

SYNOPSIS

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

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

DESCRIPTION

`mallinfo` returns a structure describing the current state of memory allocation. The structure is defined in `malloc.h`. The following fields are defined:

- `arena` is the total amount of space in the heap.
- `ordblks` is the number of chunks which are not in use.
- `uordblks` is the total amount of space allocated by `malloc`.
- `fordblks` is the total amount of space not in use.
- `keepcost` is the size of the top most memory block.

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

`mallopt` takes a parameter and a value. The parameters are defined in `malloc.h`, and may be one of the following:

- `M_TRIM_THRESHOLD` sets the maximum amount of unused space in the top most block before releasing it back to the system in free (the space is released by calling _sbrk_r with a negative argument).
- `M_TOP_PAD` is the amount of padding to allocate whenever `_sbrk_r` is called to allocate more space.

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

RETURNS

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

`malloc_stats` does not return a result.

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

COMPLIANCE `mallinfo` and `mallopt` are provided by SVR4, but `mallopt` takes different parameters on different systems.

`malloc_stats` is not portable.

__malloc_lock, __malloc_unlock

[lock malloc pool]

SYNOPSIS

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

DESCRIPTION

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

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

mmbtowc

[minimal multibyte to wide char converter]

SYNOPSIS

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

DESCRIPTION

This is a minimal ANSI-conforming implementation of `mbtowc`. The only "multibyte character sequences" recognized are single bytes, and they are "converted" to themselves.

Each call to `mbtowc` copies one character from `*`*`s`* to `*`*`pwc`*, unless *`s`* is a `NULL` pointer.

In this implementation, the argument, *`n`*, is ignored.

RETURNS

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

COMPLIANCE

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

`mbtowc` requires no supporting OS subroutines.

qsort

[sort an array]

SYNOPSIS

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

DESCRIPTION

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

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

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

RETURNS

`qsort` does not return a result.

COMPLIANCE

`qsort` meets ANSI standards (without specifying the sorting algorithm).

rand, srand

[pseudo-random numbers]

SYNOPSIS

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

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

DESCRIPTION

`rand` returns a different integer each time it is called; each integer is chosen by an algorithm designed to be unpredictable, so that you can use `rand` when you require a random number. The algorithm depends on a static variable called the *random seed*; starting with a given value of the random seed, and always producing 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.

RETURNS

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

`srand` does not return a result.

COMPLIANCE

`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.

strtod, strtodf

[string to double or float]

SYNOPSIS

```
#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);
```

DESCRIPTION

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

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

The substring contains no characters if *`str`* is empty, if it 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.

RETURNS

`strtod` returns the converted substring value, if any. If no conversion could be performed, 0 is returned. If the correct value is out of the range of representative 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`.

COMPLIANCE

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

strtol

[string to long]

SYNOPSIS

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

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

DESCRIPTION

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

If the value of *`base`* is `0`, the subject string is expected to look like a normal C integer constant: an optional sign, a possible `0x` indicating a hexadecimal base, and a number. If *`base`* is between 2 and 36, the expected form of the subject is a sequence of letters and digits representing an integer in the radix specified by *`base`*, with an optional plus or minus sign. The letters, `a`–`z` (or, 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 in the previous paragraphs. 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.

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

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

COMPLIANCE `strtol` is ANSI.

No supporting OS subroutines are required.

strtoul

[string to unsigned long]

SYNOPSIS

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

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

DESCRIPTION

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

If the value of `base` is zero, the subject string is expected to look like a normal C integer constant (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 `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 in the previous paragraphs. Finally, a pointer to the first character past the converted subject string is stored in `ptr`, if `ptr` is not `NULL`.

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

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

RETURNS `strtoul` returns the converted value, if any. 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`.

COMPLIANCE `strtoul` is ANSI.

`strtoul` requires no supporting OS subroutines.

system

[execute command string]

SYNOPSIS

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

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

DESCRIPTION

Use `system` to pass a command string, `*s`, to `/bin/sh` on your system, and wait for it to finish executing. Use `system(NULL)` to test whether your system has `/bin/sh` available.

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

RETURNS

`system(NULL)` returns a non-zero value if `/bin/sh` is available, and `0` if it is not. With a command argument, the result of `system` is the exit status returned by `/bin/sh`.

COMPLIANCE

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

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

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

wctomb

[minimal wide char to multibyte converter]

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

COMPLIANCE

`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`)

The following documentation groups macros (which are also available as subroutines) that classify characters into several categories (alphabetic, numeric, control characters, whitespace, and so on), or perform simple character mappings. The header file, `ctype.h`, defines the macros.

isalnum

[alphanumeric character predicate]

SYNOPSIS

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

DESCRIPTION

`isalnum` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero for alphabetic or numeric ASCII characters, and 0 for other arguments. It is defined for all integer values.

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

RETURNS

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

COMPLIANCE

`isalnum` is ANSI C.

No OS subroutines are required.

isalpha

[alphabetic character predicate]

SYNOPSIS

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

DESCRIPTION

`isalpha` is a macro which classifies ASCII integer values by table lookup. It is a predicate returning non-zero when *c* represents an alphabetic ASCII character, and 0 otherwise. It is defined only when `isascii(`*c*`)` is true or *c* is `EOF`.

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

RETURNS

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

COMPLIANCE

`isalpha` is ANSI C.

No supporting OS subroutines are required.

isascii

[ASCII character predicate]

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

COMPLIANCE

`isascii` is ANSI C.

No supporting OS subroutines are required.

iscntrl

[control character predicate]

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

COMPLIANCE

`iscntrl` is ANSI C.

No supporting OS subroutines are required.

isdigit

[decimal digit predicate]

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

COMPLIANCE

`isdigit` is ANSI C.

No supporting OS subroutines are required.

islower

[lower-case character predicate]

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

COMPLIANCE

`islower` is ANSI C.

No supporting OS subroutines are required.

isprint, isgraph

[printable character predicates]

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

COMPLIANCE

`isprint` and `isgraph` are ANSI C.

No supporting OS subroutines are required.

ispunct

[punctuation character predicate]

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

COMPLIANCE

`ispunct` is ANSI C.

No supporting OS subroutines are required.

isspace

[whitespace character predicate]

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

COMPLIANCE

`isspace` is ANSI C.

No supporting OS subroutines are required.

isupper

[uppercase character predicate]

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

COMPLIANCE

`isupper` is ANSI C.

No supporting OS subroutines are required.

isxdigit

[hexadecimal digit predicate]

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

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

COMPLIANCE

`isxdigit` is ANSI C.

No supporting OS subroutines are required.

toascii

[force integers to ASCII range]

SYNOPSIS

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

DESCRIPTION

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

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

RETURNS

`toascii` returns integers between 0 and 127.

COMPLIANCE

`toascii` is not ANSI C.

No supporting OS subroutines are required.

tolower

[translate characters to lower case]

SYNOPSIS

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

DESCRIPTION

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

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

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

RETURNS

`tolower` returns the lowercase equivalent of *c* when it is a character between `A` and `Z`, and *c*, otherwise.

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

COMPLIANCE

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

No supporting OS subroutines are required.

toupper

[translate characters to upper case]

SYNOPSIS

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

DESCRIPTION

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

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

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

RETURNS

`toupper` returns the uppercase equivalent of *c* when it is a character between `a` and `z`, and *c*, otherwise.

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

COMPLIANCE

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

No supporting OS subroutines are required.

3

Input and Output (`stdio.h`)

The following documentation comprises those functions that manage files or other input/output streams. Among these functions are subroutines to generate or scan strings according to specifications from a format string.

- “fseek” on page 71
- “ftell” on page 73
- “fwrite” on page 74
- “getc” on page 75
- “getchar” on page 76
- “gets” on page 77
- “iprintf” on page 78
- “mktemp, mkstemp” on page 79
- “perror” on page 80
- “printf, fprintf, sprintf” on page 81
- “putc” on page 86
- “putchar” on page 87
- “puts” on page 88
- “remove” on page 89
- “rename” on page 90
- “rewind” on page 91
- “scanf, fscanf, sscanf” on page 92
- “setbuf” on page 97
- “setvbuf” on page 98
- “siprintf” on page 99
- “tmpfile” on page 100
- “tmpnam, tempnam” on page 101
- “vprintf, vfprintf, vsprintf” on page 103

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 the following macros.

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

These reentrant versions are used instead of the globals, `stdin`, `stdout`, and `stderr`.

The argument, *`reent`*, is a pointer to a reentrancy structure.

clearerr

[clear file or stream error indicator]

SYNOPSIS

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

DESCRIPTION

The `stdio` functions maintain an error indicator with each file pointer, *`fp`*, to record whether any read or write errors have occurred on the associated file or stream. Similarly, it maintains an end-of-file (`EOF`) indicator to record whether there is no more data in the file. Use `clearerr` to reset both of these indicators. See `ferror` and `feof` to query the two indicators.

RETURNS

`clearerr` does not return a result.

COMPLIANCE

ANSI C requires `clearerr`.

No supporting OS subroutines are required.

fclose

[close a file]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

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

COMPLIANCE

`fclose` is required by ANSI C.

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

fdopen

[turn open file into a stream]

SYNOPSIS

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

FILE *_fdopen_r(void *reent, int fd, const char *mode);
```

DESCRIPTION

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

RETURNS

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

COMPLIANCE

`fdopen` is ANSI.

feof

[test for end of file]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

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

COMPLIANCE

`feof` is required by ANSI C.

No supporting OS subroutines are required.

ferror

[test whether read/write error has occurred]

SYNOPSIS

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

DESCRIPTION

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

See `clearerr` to reset the error indicator.

RETURNS

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

COMPLIANCE

ANSI C requires `ferror`.

No supporting OS subroutines are required.

fflush

[flush buffered file output]

SYNOPSIS

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

DESCRIPTION

The `stdio` output functions can buffer output before delivering it to the host system, in order to minimize the overhead of system calls. Use `fflush` to deliver any such pending output (for the file or stream identified by *`fp`*) to the host system. If *`fp`* is `NULL`, `fflush` delivers pending output from all open files.

RETURNS

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

COMPLIANCE

ANSI C requires `fflush`.

No supporting OS subroutines are required.

fgetc

[get a character from a file or stream]

SYNOPSIS

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

DESCRIPTION

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

For a macro version of this function, see "getc" on page 75.

RETURNS

The next character (read as `unsigned char`, and cast to `int`) is returned, 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.

COMPLIANCE

ANSI C requires `fgetc`.

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

fgetpos

[record position in a stream or file]

SYNOPSIS

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

DESCRIPTION

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

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

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

RETURNS

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

COMPLIANCE

`fgetpos` is required by ANSI C, 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.

fgets

[get character string from a file or stream]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

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

COMPLIANCE

`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`.

fiprintf

[format output to file (integer only)]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

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

COMPLIANCE

`fiprintf` is not required by ANSI C.

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

fopen

[open a file]

SYNOPSIS

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

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

DESCRIPTION

`fopen` initializes the data structures needed to read or write a file. Specify the 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`, in order to select any of the modes. The following documentation describes the access.

- `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, such as 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, 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*; `ab`, *append binary*.

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

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

Use `r+` (or `rb+`) to permit reading and writing anywhere in an existing file,

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

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

COMPLIANCE `fopen` is required by ANSI C.

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

fputc

[write a character on a stream or file]

SYNOPSIS

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

DESCRIPTION

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

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

For a macro version of this function, see “putc” on page 86.

RETURNS

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

COMPLIANCE

`fputc` is required by ANSI C.

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

fputs

[write a character string in a file or stream]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

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

COMPLIANCE

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`.

fread

[read array elements from a file]

SYNOPSIS

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

DESCRIPTION

`fread` attempts to copy, from the file or stream identified by *`fp`*, *`count`* elements (each of size, *`size`*) into memory, starting at *`buf`*. `fread` may copy fewer elements than `count` if an error, or end of file (`EOF`), intervenes.

`fread` also advances the file position indicator (if any) for *`fp`* by the number of *characters* actually read.

RETURNS

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

COMPLIANCE

ANSI C requires `fread`.

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

freopen

[open a file using an existing file descriptor]

SYNOPSIS

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

DESCRIPTION

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

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

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

RETURNS

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

COMPLIANCE

ANSI C requires `freopen`.

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

fseek

[set file position]

SYNOPSIS

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

DESCRIPTION

Objects of type, `FILE`, can have a *position* that records how much of the file your program has already read. Many of the `stdio` functions depend on this position, and many change it as a side effect. You can use `fseek` to set the position for the file identified by *`fp`*.

The value of *`offset`* determines the new position, in one of three ways, selected by the value of *`whence`* (defined as macros in `stdio.h`).

- `SEEK_SET`—*`offset`* is the absolute file position (an offset from the beginning of the file) desired. *`offset`* must be positive.
- `SEEK_CUR`—*`offset`* is relative to the current file position. *`offset`* can meaningfully be either positive or negative.
- `SEEK_END`—*`offset`* is relative to the current end of file. *`offset`* can meaningfully be either positive (to increase the size of the file) or negative.

See“ftell” on page 73 to determine the current file position.

RETURNS

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

COMPLIANCE

ANSI C requires `fseek`.

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

fsetpos

[restore position of a stream or file]

SYNOPSIS

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

DESCRIPTION

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

You can use `fsetpos` to return the file identified by *`fp`* to a previous position `*`*`pos`* (after first recording it with `fgetpos`).

See “fseek” on page 71 for a similar facility.

RETURNS

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

COMPLIANCE

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`.

ftell

[return position in a stream or file]

SYNOPSIS

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

DESCRIPTION

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

The result of `ftell` is the current position for a file identified by *`fp`*. If you record this result, you can later use it with `fseek` to return the file to this position.

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

RETURNS

`ftell` returns the file position, if possible. If it cannot do this, it returns `-1L`. Failure occurs on streams that do not support positioning; the global, `errno`, indicates this condition with the value, `ESPIPE`.

COMPLIANCE

`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.

fwrite

[write array elements]

SYNOPSIS

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

DESCRIPTION

`fwrite` attempts to copy, starting from the memory location, *`buf`*, *`count`* elements (each of size, *`size`*) into the file or stream identified by *`fp`*. `fwrite` may copy fewer elements than *`count`* if an error intervenes.

`fwrite` also advances the file position indicator (if any) for *`fp`* by the number of *characters* actually written.

RETURNS

If `fwrite` succeeds in writing all the elements you specify, the result is the same as the argument, *`count`*. In any event, the result is the number of complete elements that `fwrite` copied to the file.

COMPLIANCE

ANSI C requires `fwrite`.

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

getc

[read a character (macro)]

SYNOPSIS

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

DESCRIPTION

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

For a subroutine version of this macro, see "fgetc" on page 61.

RETURNS

The next character (read as `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.

COMPLIANCE

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`.

getchar

[read a character (macro)]

SYNOPSIS

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

int _getchar_r(void *reent);
```

DESCRIPTION

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

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

RETURNS

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

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

COMPLIANCE

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`.

gets

[get character string] (obsolete, use **fgets** instead)]

SYNOPSIS

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

char *_gets_r(void *reent, char *buf);
```

DESCRIPTION

`gets` 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`.

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

WARNING!

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 function 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.

RETURNS

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

COMPLIANCE

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

iprintf

[write formatted output (integer only)]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

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

COMPLIANCE

`iprintf` is not required by ANSI C.

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

mktemp, mkstemp

[generate unused file name]

SYNOPSIS

```
#include <stdio.h>
char *mktemp(char *path);
int mkstemp(char *path);

char *_mktemp_r(void *reent, char *path);
int *_mkstemp_r(void *reent, char *path);
```

DESCRIPTION

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

You supply a simple pattern for the generated file name, as the string at *`path`*. The pattern should be a valid filename (including path information if you wish) ending with some number of ***X*** characters. The generated filename will match the leading part of the name you supply, with the trailing ***X*** characters replaced by some combination of digits and letters.

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

RETURNS

`mktemp` returns the pointer, *`path`*, to the modified string representing an unused filename, unless it could not generate one, or the pattern you provided is not suitable for a filename; in that case, it returns `NULL`.

`mkstemp` returns a file descriptor to the newly created file, unless it could not generate an unused filename, or the pattern you provided is not suitable for a filename; in that case, it returns `-1`.

COMPLIANCE

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`.

perror

[print an error message on standard error]

SYNOPSIS

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

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

DESCRIPTION

Use `perror` to print (on standard error) an error message corresponding to the current value of the global variable, `errno`.

Unless you use `NULL` as the value of the argument, *`prefix`*, the error message will begin with the string at *`prefix`*, followed by a colon and a space (`: `). The remainder of the error message is one of the strings described for `strerror`.

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

RETURNS

`perror` returns no result.

COMPLIANCE

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`.

printf, fprintf, sprintf

[format output]

SYNOPSIS

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

int fprintf(FILE *fd, const char *format [, arg, ...]);

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

DESCRIPTION

`printf` accepts a series of arguments, applies to each a format specifier from `*format`, and writes the formatted data to `stdout`, terminated with a null character.

The behavior of `printf` is undefined if there are not enough arguments for the format. `printf` returns when it reaches the end of the `format` string. If there are more arguments than the format requires, excess arguments are ignored.

`fprintf` and `sprintf` are identical to `printf`, other than the destination of the formatted output: `fprintf` sends the output to a specified file, `fd`, while `sprintf` stores the output in the specified `char` array, `str`. For `sprintf`, the behavior is also undefined if the output string, `*str`, overlaps with one of the arguments. `format` is a pointer to a character string containing two types of objects: ordinary characters (other than `%`), which are copied unchanged to the output, and conversion specifications, each of which is introduced by `%`. (To include `%` in the output, use `%%` in the format string.)

A conversion specification uses fields in the following form.

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

The fields of the conversion specification (represented in the previous example of a conversion specification by `flags`, `width`, `.prec`, `size`, and `type`) have the following meanings.

- `[flags]`

  `flags`, an optional sequence of characters, controls 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.

  -

    With `-`, the *minus sign* flag, the result of the conversion is left justified, and the right is padded with blanks. If you do not use the minus sign flag, the result is right justified, and padded on the left.

`+`

With `+`, the *plus sign* flag, the result of a signed conversion (as determined by the specification for *`type`*) will always begin with a plus or minus sign.

IMPORTANT: If you don't use this flag, positive values won't 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). If the zero (`0`) and minus flags both appear, the zero flag will be ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if *`prec`* is specified, the zero flag is ignored.

IMPORTANT: Do not use spaces padding. Also, `0` is interpreted as a flag, not as the beginning of a field width.

`#`

With `#`, the result is to be converted to an alternative form, according to one of the following subsequent characters.

`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` stands for an optional minimum field width. Either specify it directly as a decimal integer, or indirectly by, instead, using an asterisk (`*`), in which case an `int` argument is used as the field width. Negative

field widths are not supported; if you try to specify a negative field width, it is interpreted as a minus flag (`-`), followed by a positive field width.

- *`[.prec]`*
  `prec` is 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 types, `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 the ones specified in this description, the behavior is undefined.
- *`[size]`*
  `h`, `l`, and `L` are optional *`size`* characters which override the default way that `printf` interprets the data type of the corresponding argument. `h` forces the following `d`, `i`, `o`, `u`, `x` or `X` conversion type to apply to a `short` or `unsigned short`. `h` also forces a following `n` *`type`* to apply to a pointer to a `short`. Similarly, an `l` forces the following `d`, `i`, `o`, `u`, `x` or `X` conversion type to apply to a `long` or `unsigned long`. `l` also forces a following `n` *`type`* to apply to a pointer to a `long`. If an `h` or an `l` appears with another conversion specifier, the behavior is undefined. `L` forces a following `e`, `E`, `f`, `g` or `G` conversion type to apply to a `long double` argument. If `L` appears with any other conversion type, the behavior is undefined.
- *`[type]`*
  `type` specifies what kind of conversion `printf` performs. The following discussion describes the corresponding arguments.

  `%`
  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`).

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

COMPLIANCE The ANSI standard for C specifies that implementations must support

formatted output of up to 509 characters.
Supporting OS subroutines required: `close`, `fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`.

putc

[write a character (macro)]

SYNOPSIS

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

DESCRIPTION

`putc` is a macro, defined in `stdio.h`. `putc` writes the argument, *`ch`*, to the file or stream identified by *`fp`*, after converting it from an `int` to an `unsigned char`.

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

For a subroutine version of this macro, see “fputc” on page 67.

RETURNS

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

COMPLIANCE

ANSI C requires by `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`.

putchar

[write a character (macro)]

SYNOPSIS

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

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

DESCRIPTION

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

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

RETURNS

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

COMPLIANCE

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`.

puts

[write a character string]

SYNOPSIS

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

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

DESCRIPTION

`puts` writes the string at *`s`* (followed by a newline, instead of the trailing `NULL`) to the standard output stream.

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

RETURNS

If successful, the result is a nonnegative integer; otherwise, the result is `EOF`.

COMPLIANCE

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`.

remove

[delete a file's name]

SYNOPSIS

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

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

DESCRIPTION

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

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

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

RETURNS

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

COMPLIANCE

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`.

rename

[rename a file]

SYNOPSIS

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

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

DESCRIPTION

Use `rename` to establish a new name (the whole string at *`new`*) for a file now known by the string at *`old`*. After a successful rename, the file is no longer accessible by the string at *`old`*.

If `rename` fails, the file named `*`*`old`* is unaffected. The conditions for failure depend on the host operating system.

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

RETURNS

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

COMPLIANCE

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

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

rewind

[reinitialize a file or stream]

SYNOPSIS

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

DESCRIPTION

`rewind` returns the file position indicator (if any) for the file or stream, identified by *fp*, to the beginning of the file. It also clears any error indicator and flushes any pending output.

RETURNS

`rewind` does not return a result.

COMPLIANCE

ANSI C requires `rewind`.

No supporting OS subroutines are required.

scanf, fscanf, sscanf

[scan and format input]

SYNOPSIS

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

int fscanf(FILE *fd, const char *format [, arg, ...]);

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

DESCRIPTION

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

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

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

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

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

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

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

Format specifications tell `scanf` to read and convert characters from the input field into specific types of values, and store them 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 use the following example's form.

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

Each format specification begins with the percent character (`%`). The other fields are described in the following discussions.

- *`[*]`*
  An optional marker; if present, *`[*]`* suppresses interpretation and assignment of this input field.
- *`[width]`*
  An optional maximum field *`[width]`* specifier: a decimal integer, which controls the maximum number of characters that will be read before converting the current input field. If the input field has fewer than *`[width]`* characters, `scanf` reads all the characters in the field, and then proceeds with the next field and its format specification. If a whitespace or a non-convertible character occurs before a *`[width]`* character is 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. See Table 1 for more details on *`size`* characters.

  Table 1: *`size`* characters

Modifier	***Type(s)***	
`h`	`d, i, o, u, x`	Convert input to `short`, store in `short` object.
`h`	`D, I, O, U, X, e, f, c, s, n, p`	No effect.
`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, c, s, n, p`	No effect.
`L`	`d, i, o, u, x`	Convert to `long double`, store in `long double`.
`L`	*All others*	No effect.

- *`[type]`*
  `[type]`, a character that specifies what kind of conversion `scanf` performs. Usage of the *`[type]`* field is described in the following discussions.

  `%`
  No conversion is done; the percent character (`%`) is stored.

`c`
Scans one character. Corresponding argument: `char *arg`.

`s`
Reads a character string into the array supplied. Corresponding argument: `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. Corresponding argument: `char *arg`.

A *`pattern`* character 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.

See Table 2 for some *`[pattern]`* examples.

Table 2: *`[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`, or `G`) must correspond to the following general form. Objects enclosed in square brackets are optional, and *`ddd`* represents decimal, octal, or hexadecimal digits.

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

`d`
Reads a decimal integer into the corresponding argument: `int *arg`.

`D`
Reads a decimal integer into the corresponding argument: `long *arg`.

`o`
Reads an octal integer into the corresponding argument: `int *arg`.

`O`
Reads an octal integer into the corresponding argument: `long *arg`.

`u`
Reads an unsigned decimal integer into the corresponding argument: `unsigned int *arg`.

`U`
Reads an unsigned decimal integer into the corresponding argument: `unsigned long *arg`.

`x`, `X`
Read a hexadecimal integer into the corresponding argument: `int *arg`.

`e`, `f`, `g`
Read a floating point number into the corresponding argument: `float *arg`.

`E`, `F`, `G`
Read a floating point number into the corresponding argument: `double *arg`.

`i`
Reads a decimal, octal or hexadecimal integer into the corresponding argument: `int *arg`.

`I`
Reads a decimal, octal or hexadecimal integer into the corresponding argument: `long *arg`.

`n`
Stores the number of characters read in the corresponding argument: `int *arg`.

`p`
Stores a scanned pointer. ANSI C leaves the de-tails to each implementation; this implementation treats `%p` exactly the same as `%U`. Corresponding argument: `void **arg`.

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

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

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

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

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

- The assignment suppressing character (`*`) appears after the `%` in the format specification; the current input field is scanned but not stored.

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

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

`scanf` will terminate under the following circumstances.

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

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

COMPLIANCE `scanf` is ANSI C.

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

setbuf

[specify full buffering for a file or stream]

SYNOPSIS

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

DESCRIPTION

`setbuf` specifies that output to the file or stream identified by *`fp`* should be fully buffered. All output for this file will go to a buffer (of size, `BUFSIZ`, specified in `stdio.h`). Output will be passed on to the host system only when the buffer is full, or when an input operation intervenes.

You may, if you wish, supply your own buffer by passing a pointer to it as the argument, *`buf`*. It must have size, `BUFSIZ`. You can also use `NULL` as the value of *`buf`*, to signal that the `setbuf` function is to allocate the buffer.

WARNING!

You may only use `setbuf` before performing any file operation other than opening the file. If you supply a non-null *`buf`*, you must ensure that the associated storage continues to be available until you close the stream identified by *`fp`*.

RETURNS

`setbuf` does not return a result.

COMPLIANCE

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 System V Interface Definition (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`.

setvbuf

[specify file or stream buffering]

SYNOPSIS

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

DESCRIPTION

Use `setvbuf` to specify what kind of buffering you want for the file or stream identified by *`fp`*, 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.

WARNING!

You may only use `setvbuf` before performing any file operation other than opening the file. If you supply a non-null *`buf`*, you must ensure that the associated storage continues to be available until you close the stream identified by *`fp`*.

RETURNS

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

COMPLIANCE

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 System V Interface Definition (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`.

siprintf

[write formatted output (integer only)]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

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

COMPLIANCE

`siprintf` is not required by ANSI C.

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

tmpfile

[create a temporary file]

SYNOPSIS

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

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

DESCRIPTION

`tmpfile` creates a temporary file (a file which will be deleted automatically), using a name generated by `tmpnam`. The temporary file is opened with the mode, `wb+`, permitting you to read and write anywhere in it as a *binary* file (without any data transformations the host system may perform for text files). The alternate function , `_tmpfile_r`, is a reentrant version.

The argument, *`reent`*, is a pointer to a reentrancy structure.

RETURNS

`tmpfile` normally returns a pointer to the temporary file. If no temporary file could be created, the result is `NULL`, and `errno` records the reason for failure.

COMPLIANCE

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`.

tmpnam, tempnam

[name for a temporary file]

SYNOPSIS

```
#include <stdio.h>
char *tmpnam(char *s);

char *tempnam(char *dir, char *pfx);

char *_tmpnam_r(void *reent, char *s);

char *_tempnam_r(void *reent, char *dir, char *pfx);
```

DESCRIPTION

Use either of these functions, `tmpnam` or `tempnam`, 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.

DANGER!!!

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

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

RETURNS

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

COMPLIANCE

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.

vprintf, vfprintf, vsprintf

[format argument list]

SYNOPSIS

```
#include <stdio.h>
#include <stdarg.h>
int vprintf(const char *fmt, va_list list);
int vfprintf(FILE *fp, const char *fmt, va_list list);
int vsprintf(char *str, const char *fmt, va_list list);

int _vprintf_r(void *reent, const char *fmt,
                                        va_list list);
int _vfprintf_r(void *reent, FILE *fp, const char *fmt,
                                        va_list list);
int _vsprintf_r(void *reent, char *str,
                            const char *fmt, va_list list);
```

DESCRIPTION

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

RETURNS

The return values are consistent with the corresponding functions: `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`.

COMPLIANCE

ANSI C requires all three functions.

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

4

Strings and Memory (`string.h`)

The following documentation describes string-handling functions and functions for managing areas of memory. The corresponding declarations are in `string.h`.

bcmp

[compare two memory areas]

SYNOPSIS

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

DESCRIPTION

The function, bcmp, compares not more than *n* characters of the object pointed to by *s1* with the object pointed to by *s2*. This function is identical to memcmp.

RETURNS

The function returns an integer greater than, equal to or less than zero, according to whether the object pointed to by *s1* is greater than, equal to or less than the object pointed to by *s2*.

COMPLIANCE

`bcmp` requires no supporting OS subroutines.

bcopy

[copy memory regions]

SYNOPSIS

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

DESCRIPTION The function, `bcopy`, 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`.

RETURNS `bcopy` does not return a result.

COMPLIANCE `bcopy` requires no supporting OS subroutines.

bzero

[initialize memory to zero]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

`bzero` does not return a result.

COMPLIANCE

`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.

index

[search for character in string]

SYNOPSIS

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

DESCRIPTION

The function, `index`, finds the first occurrence of *`c`* (converted to a `char`) in the string pointed to by *`string`* (including the terminating null character).

This function is identical to `strchr`.

RETURNS

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

COMPLIANCE

`index` requires no supporting OS subroutines.

memchr

[find character in memory]

SYNOPSIS

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

DESCRIPTION

The function, `memchr`, searches memory starting at `*src` for the character, `c`. The search only ends with the first occurrence of `c`, or after `length` characters; in particular, `NULL` does not terminate the search.

RETURNS

If the character, `c`, is found within `length` characters of `*src`, a pointer to the character is returned. If `c` is not found, then `NULL` is returned.

COMPLIANCE

`memchr>` is ANSI C.

`memchr` requires no supporting OS subroutines.

memcmp

[compare two memory areas]

SYNOPSIS

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

DESCRIPTION

The function, `memcmp`, compares not more than *`n`* characters of the object pointed to by *`s1`* with the object pointed to by *`s2`*.

RETURNS

The function returns an integer greater than, equal to or less than zero according to whether the object pointed to by *`s1`* is greater than, equal to or less than the object pointed to by *`s2`*.

COMPLIANCE

`memcmp` is ANSI C.

`memcmp` requires no supporting OS subroutines.

memcpy

[copy memory regions]

SYNOPSIS

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

DESCRIPTION

The function, memcpy, copies n bytes from the memory region pointed to by in to the memory region pointed to by out.

If the regions overlap, the behavior is undefined.

RETURNS

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

COMPLIANCE

`memcpy` is ANSI C.

`memcpy` requires no supporting OS subroutines.

memmove

[move possibly overlapping memory]

SYNOPSIS

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

DESCRIPTION

The function, `memmove`, moves *`length`* characters from the block of memory starting at *`*src`* to the memory starting at *`*dst`*. `memmove` reproduces the characters correctly at *`*dst`* even if the two areas overlap.

RETURNS

The function returns *`dst`* as passed.

COMPLIANCE

`memmove` is ANSI C.

`memmove` requires no supporting OS subroutines.

memset

[set an area of memory]

SYNOPSIS

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

DESCRIPTION The function, `memset`, converts the argument, *`c`*, into an `unsigned char` and fills the first *`length`* characters of the array pointed to by *`dst`* to the value.

RETURNS `memset` returns the value of *`m`*.

COMPLIANCE `memset` is ANSI C.

`memset` requires no supporting OS subroutines.

rindex

[reverse search for character in string]

SYNOPSIS

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

DESCRIPTION

The function, `rindex`, finds the last occurrence of *`c`* (converted to `char`) in the string pointed to by *`string`* (including the terminating null character).

This function is identical to `strrchr`.

RETURNS

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

COMPLIANCE

`rindex` requires no supporting OS subroutines.

strcat

[concatenate strings]

SYNOPSIS

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

DESCRIPTION

`strcat` appends a copy of the string pointed to by *`src`* (including the terminating null character) to the end of the string pointed to by *`dst`*. The initial character of *`src`* overwrites the null character at the end of *`dst`*.

RETURNS

`strcat` returns the initial value of *`dst`*.

COMPLIANCE

`strcat` is ANSI C.

`strcat` requires no supporting OS subroutines.

strchr

[search for character in string]

SYNOPSIS

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

DESCRIPTION

The function, `strchr`, finds the first occurrence of *`c`* (converted to `char`) in the string pointed to by *`string`* (including the terminating null character).

RETURNS

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

COMPLIANCE

`strchr` is ANSI C.

`strchr` requires no supporting OS subroutines.

strcmp

[character string compare]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

If `*a` sorts lexicographically after `*b`, `strcmp` returns a number greater than zero. If the two strings match, `strcmp` returns zero. If `*a` sorts lexicographically before `*b`, `strcmp` returns a number less than zero.

COMPLIANCE

`strcmp` is ANSI C.

`strcmp` requires no supporting OS subroutines.

strcoll

[locale specific character string compare]

SYNOPSIS

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

DESCRIPTION

`strcoll` compares the string pointed to by *`stra`* to the string pointed to by *`strb`*, using an interpretation appropriate to the current `LC_COLLATE` state.

RETURNS

If the first string is greater than the second string, `strcoll` returns a number greater than zero. If the two strings are equivalent, `strcoll` returns zero. If the first string is less than the second string, `strcoll` returns a number less than zero.

COMPLIANCE

`strcoll` is ANSI C.

`strcoll` requires no supporting OS subroutines.

strcpy

[copy string]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

`strcpy` returns the initial value of *`dst`*.

COMPLIANCE

`strcpy` is ANSI C.

`strcpy` requires no supporting OS subroutines.

strcspn

[count chars not in string]

SYNOPSIS

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

DESCRIPTION

The function, `strcspn`, computes the length of the initial part of the string pointed to by *`s1`* which consists entirely of characters not from the string pointed to by *`s2`* (excluding the terminating null character).

RETURNS

`strcspn` returns the length of the substring found.

COMPLIANCE

`strcspn` is ANSI C.

`strcspn` requires no supporting OS subroutines.

strerror

[convert error number to string]

SYNOPSIS

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

DESCRIPTION

`strerror` converts the error number, *`errnum`*, into a string. The value of `errnum` is usually a copy of `errno`. If *`errnum`* is not a known error number, the result points to an empty string.

This implementation of `strerror` prints out the strings for each of the values defined in `errno.h`, using the conversions in Table 3.

Table 3: Strings for values defined by `errno.h`

`E2BIG`	*`arg list`* too long
`EACCES`	Permission denied
`EADV`	Advertise error
`EAGAIN`	No more processes
`EBADF`	Bad file number
`EBADMSG`	Bad message
`EBUSY`	Device or resource busy
`ECHILD`	No children
`ECOMM`	Communication error
`EDEADLK`	Deadlock
`EEXIST`	File exists
`EDOM`	Math argument
`EFAULT`	Bad address
`EFBIG`	File too large
`EIDRM`	Identifier removed
`EINTR`	Interrupted system call
`EINVAL`	Invalid argument
`EIO`	I/O error
`EISDIR`	Is a directory
`ELIBACC`	Cannot access a needed shared library
`ELIBBAD`	Accessing a corrupted shared library
`ELIBEXEC`	Cannot exec a shared library directly

ELIBMAX	Attempting to link in more shared libraries than system limit
ELIBSCN	`.lib` section in `a.out` corrupted
EMFILE	Too many open files
EMLINK	Too many links
EMULTIHOP	Multihop attempted
ENAMETOOLONG	File or path name too long
ENFILE	Too many open files in system
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	`exec format` error
ENOLCK	No lock
ENOLINK	Virtual circuit is gone
ENOMEM	Not enough space
ENOMSG	No message of desired type
ENONET	Machine is not on the network
ENOPKG	No package
ENOSPC	No space left on device
ENOSR	No stream resources
ENOSTR	Not a stream
ENOSYS	Function not implemented
ENOTBLK	Block device required
ENOTDIR	Not a directory
ENOTEMPTY	Directory not empty
ENOTTY	Not a character device
ENXIO	No such device or address
EPERM	Not owner
EPIPE	Broken pipe
EPROTO	Protocol error
ERANGE	Result too large
EREMOTE	Resource is remote
EROFS	Read-only file system
ESPIPE	Illegal seek

ESRCH	No such process
ESRMNT	`srmount` error
ETIME	Stream `ioctl` timeout
ETXTBSY	Text file busy
EXDEV	Cross-device link

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

COMPLIANCE ANSI C requires `strerror`, but does not specify the strings used for each error number.

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

This implementation of `strerror` provides for user-defined extensibility. `errno.h` defines `__ELASTERROR`, which can be used as a base for user-defined error values. If the user supplies a routine named `_user_strerror`, and *`errnum`* passed to `strerror` does not match any of the supported values, _user_strerror is called with *`errnum`* as its argument.

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

`strerror` requires no supporting OS subroutines.

strlen

[character string length]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

`strlen` returns the character count.

COMPLIANCE

`strlen` is ANSI C.

`strlen` requires no supporting OS subroutines.

strlwr

[force string to lower case]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

`strlwr` returns its argument, *a*.

COMPLIANCE

`strlwr` is not widely portable.

`strlwr` requires no supporting OS subroutines.

strupr

[force string to uppercase]

SYNOPSIS

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

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

RETURNS `strupr` returns its argument, *a*.

COMPLIANCE `strupr` is not widely portable.

`strupr` requires no supporting OS subroutines.

strncat

[concatenate strings]

SYNOPSIS

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

DESCRIPTION

`strncat` appends not more than *`length`* characters from the string pointed to by *`src`* (including the terminating null character) to the end of the string pointed to by *`dst`*. The initial character of *`src`* overwrites the null character at the end of *`dst`*. A terminating null character is always appended to the result.

WARNING!

A null is always appended, so that if the copy is limited by the *`length`* argument, the number of characters appended to *`dst`* is `n +1`.

RETURNS

`strncat` returns the initial value of *`dst`*.

COMPLIANCE

`strncat` is ANSI C.

`strncat` requires no supporting OS subroutines.

strncmp

[character string compare]

SYNOPSIS

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

DESCRIPTION

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

RETURNS

If *`*a`* sorts lexicographically after *`*b`*, `strncmp` returns a number greater than zero. If the two strings are equivalent, `strncmp` returns zero. If *`*a`* sorts lexicographically before *`*b`*, `strncmp` returns a number less than zero.

COMPLIANCE

`strncmp` is ANSI C.

`strncmp` requires no supporting OS subroutines.

strncpy

[counted copy string]

SYNOPSIS

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

DESCRIPTION

`strncpy` copies not more than *`length`* characters from the string pointed to by *`src`* (including the terminating null character) to the array pointed to by *`dst`*. If the string pointed to by *`src`* is shorter than *`length`* characters, null characters are appended to the destination array until a total of *`length`* characters have been written.

RETURNS

`strncpy` returns the initial value of *`dst`*.

COMPLIANCE

`strncpy` is ANSI C.

`strncpy` requires no supporting OS subroutines.

strpbrk

[find chars in string]

SYNOPSIS

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

DESCRIPTION `strpbrk` locates the first occurrence in the string pointed to by *`s1`* of any character in string pointed to by *`s2`* (excluding the terminating null character).

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

COMPLIANCE `strpbrk` requires no supporting OS subroutines.

strrchr

[reverse search for character in string]

SYNOPSIS

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

DESCRIPTION

`strrchr` finds the last occurrence of *`c`* (converted to `char`) in the string pointed to by *`string`* (including the terminating null character).

RETURNS

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

COMPLIANCE

`strrchr` is ANSI C.

`strrchr` requires no supporting OS subroutines.

strspn

[find initial match]

SYNOPSIS

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

DESCRIPTION

`strspn` computes the length of the initial segment of the string pointed to by *`s1`*, consisting entirely of characters from the string pointed to by *`s2`* (excluding the terminating null character).

RETURNS

`strspn` returns the length of the segment found.

COMPLIANCE

`strspn` is ANSI C.

`strspn` requires no supporting OS subroutines.

strstr

[find string segment]

SYNOPSIS

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

DESCRIPTION

`strstr` locates the first occurrence in the string pointed to by *`s1`* of the sequence of characters in the string pointed to by *`s2`* (excluding the terminating null character).

RETURNS

`strstr` 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.

COMPLIANCE

`strstr` is ANSI C.

`strstr` requires no supporting OS subroutines.

strtok

[get next token from a string]

SYNOPSIS

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

DESCRIPTION

A series of calls to `strtok` breaks 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.

RETURNS

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

COMPLIANCE

`strtok` is ANSI C.

`strtok` requires no supporting OS subroutines.

strxfrm

[transform string]

SYNOPSIS

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

DESCRIPTION

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

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

With a C locale, this function just copies.

RETURNS

The `strxfrm` function returns the length of the transformed string (not including the terminating null character). If the value returned is *n* or more, the contents of the array pointed to by *s1* are indeterminate.

COMPLIANCE

`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 signals in Table 4.

Table 4: Signals

`SIGABRT`	Abnormal termination of a program; raised by the `abort` function (see "abort" on page 5).
`SIGFPE`	A domain error in arithmetic, such as overflow, or division by zero.
`SIGILL`	Attempt to execute as unexecutable function data.
`SIGINT`	Interrupt; an interactive attention signal.
`SIGSEGV`	An attempt to access an unavailable memory location.
`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 (called *raising* a signal; see "raise" on page 141), and one to specify subroutines (called *handlers*; see "signal" on page 142) 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 the three macros in Table 5 on page 140.

Table 5: Asynchronous signals

`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.)

raise

[send a signal]

SYNOPSIS

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

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

DESCRIPTION

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

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

RETURNS

The result is 0 if *`sig`* was successfully raised, 1 otherwise. However, the return value (since it depends on the normal flow of execution) may not be visible, unless the signal handler for *`sig`* terminates with a return or unless `SIG_IGN` is in effect for this signal.

COMPLIANCE

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

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

signal

[specify handler subroutine for a signal]

SYNOPSIS

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

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

int raise (int sig);

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

DESCRIPTION

`signal` and `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` (for selecting system default handling) or `SIG_IGN` (for ignoring 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` and `_raise_r` , are the reentrant versions. The extra argument, *`reent`*, is a pointer to a reentrancy structure.

RETURNS

If your request for a signal handler cannot be honored, the result is `SIG_ERR`; a specific error number is also recorded in `errno`.

Otherwise, the result is the previous handler (a function pointer or one of the predefined macros).

COMPLIANCE ANSI C requires `raise` and `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`)

The following documentation includes 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 fields in Table 1.

Figure 1: Field representations for `time.h`

`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.

asctime

[format time as string]

SYNOPSIS

```
#include <time.h>
char *asctime(const struct tm *clock);
char *asctime_r(const struct tm *clock, char *buf);
```

DESCRIPTION

`asctime` formats the time value at *`clock`* into a string of the following form.

```
Wed Jun 15 11:38:07 1988\n\0
```

The string is generated in a static buffer; each call to `asctime` overwrites the string generated by previous calls.

RETURNS

A pointer to the string containing a formatted timestamp.

COMPLIANCE

ANSI C requires `asctime`.

`asctime` requires no supporting OS subroutines.

clock

[cumulative processor time]

SYNOPSIS

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

DESCRIPTION

`clock` calculates the best available approximation of the cumulative amount of time used by your program since it started. To convert the result into seconds, divide by the macro, `CLOCKS_PER_SEC`.

RETURNS

The amount of processor time used so far by your program, in units defined by the machine-dependent macro, `CLOCKS_PER_SEC`. If no measurement is available, the result is -1.

COMPLIANCE

ANSI C requires `clock` and `CLOCKS_PER_SEC`.

Supporting OS subroutine required: `times`.

ctime

[convert time to local and format as string]

SYNOPSIS

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

DESCRIPTION

`ctime` converts the time value at *`clock`* to local time (like `localtime`) and formats it into a string of the following form (like `asctime`).

```
Wed Jun 15 11:38:07 1988\n\0
```

RETURNS

A pointer to the string containing a formatted timestamp.

COMPLIANCE

ANSI C requires `ctime`.

`ctime` requires no supporting OS subroutines.

difftime

[subtract two times]

SYNOPSIS

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

DESCRIPTION

`difftime` subtracts the two times in the arguments : *`tim2`* from *`tim1`*.

RETURNS

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

COMPLIANCE

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

`difftime` requires no supporting OS subroutines.

gmtime

[convert time to UTC traditional form]

SYNOPSIS

```
#include <time.h>
struct tm *gmtime(const time_t *clock);
struct tm *gmtime_r(const time_t *clock, struct tm *res);
```

DESCRIPTION

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

`gmtime` constructs the traditional time representation in static storage; each call to `gmtime` or `localtime` will overwrite the information generated by previous calls to either function.

RETURNS

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

COMPLIANCE

ANSI C requires `gmtime`.

`gmtime` requires no supporting OS subroutines.

localtime

[convert time to local representation]

SYNOPSIS

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

DESCRIPTION

`localtime` converts the time at *`clock`* into local time, then converts its representation from the arithmetic representation to the traditional representation defined by `struct tm`.

`localtime` constructs the traditional time representation in static storage; each call to `gmtime` or `localtime` will overwrite the information generated by previous calls to either function.

`mktime` is the inverse of `localtime`.

RETURNS

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

COMPLIANCE

ANSI C requires `localtime`.

`localtime` requires no supporting OS subroutines.

mktime

[convert time to arithmetic representation]

SYNOPSIS

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

DESCRIPTION

`mktime` assumes the time at *`timp`* is a local time, and converts its representation from the traditional representation defined by `struct tm` into a representation suitable for arithmetic.

`localtime` is the inverse of `mktime`.

RETURNS

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

COMPLIANCE

ANSI C requires `mktime`.

`mktime` requires no supporting OS subroutines.

strftime

[flexible calendar time formatter]

SYNOPSIS

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

DESCRIPTION

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

You control the format of the output using the string at *`format`*. *`*format`* can contain two kinds of specifications: text to be copied literally into the formatted string, and time conversion specifications.

Time conversion specifications are two-character sequences beginning with `%` (use `%%` to include a percent sign in the output). Each defined conversion specification selects a field of calendar time data from *`*timp`*, and converts it to a string; see Table 2 for more details of the character sequences for conversion.

Figure 2: Time conversion character sequences

%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, as in the following example:
	`Mon Apr 01 13:13:13 1992`
	`Mon Apr 01 1992`
	`13:13:13`

RETURNS

When the formatted time takes up no more than *`maxsize`* characters, the result is the length of the formatted string. Otherwise, if the formatting operation was abandoned due to lack of room, the result is 0, and the string starting at *`s`* corresponds to just those parts of *`*format`* that could be completely filled in within the *`maxsize`* limit.

COMPLIANCE

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.

Figure 3: Representations of time

`%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” on page 155.
`%w`	A single digit representing the day of the week, Sunday being day 0.
`%W`	Another version of the week number: like `%U`, but counting week 1 as beginning with the first Monday in a year.
`%x`	A string representing the complete date, as in the following example.

Figure 4: Strings for time

`%X`	A string representing the full time of day (hours, minutes, and seconds), as in the following example.

Figure 5: Special time requirements

`%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, `%`.

time

[get current calendar time (as single number)]

SYNOPSIS

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

DESCRIPTION

`time` looks up the best available representation of the current time and returns it, encoded as a `time_t`. It stores the same value at *t* unless the argument is `NULL`.

RETURNS

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

COMPLIANCE

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, equivalent to `"C"`.

`locale.h` defines the structure, `lconv`, to collect the information on a locale, using the following fields. See "setlocale, localeconv" on page 160 for more specific discussion.

`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 specified value. 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 representative 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 that the local currency symbol is used *before* a *positive or zero* formatted amount of money; `0` indicates that the currency symbol is placed *after* the formatted number, `CHAR_MAX` in the C locale.

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

`char n_sep_by_space`
`1` indicates that 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; `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.

setlocale, localeconv

[select or query locale]

SYNOPSIS

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

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

DESCRIPTION

`setlocale` is the facility defined by ANSI C to condition the execution environment for international collating and formatting information; `localeconv` reports on the settings of the current locale.

This is a minimal implementation, supporting only the required `"C"` value for *`locale`*; since strings representing other locales are not honored. `" "` is also accepted, representing a ***default*** locale for an implementation, 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"`, although 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`) that describes the locale-specific conventions currently in effect. `_localeconv_r` and `_setlocale_r` are reentrant versions of `localeconv` and `setlocale`, respectively. The extra argument, *`reent`*, is a pointer to a reentrancy structure.

RETURNS

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

`localeconv` returns a pointer to a structure of type, `lconv`, describing the formatting and collating conventions in effect (in this implementation, always those of the C locale).

COMPLIANCE

ANSI C requires `setlocale`, although 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 allowing multiple processes to use the same address space with assurance that the values stored in those spaces will remain constant between calls. Cygnus implements the library functions to ensure that, whenever possible, these library functions are reentrant.

However, there are some functions that cannot *trivially* be made reentrant. Hooks have been provided to allow for using these functions in a fully reentrant fashion. These hooks use the structure, `_reent`, defined in `reent.h`. All functions which must manipulate global information are available in the following two versions.

- The first version has the usual name, using a single global instance of the reentrancy structure.
- The second has a different name, normally formed by prepending '_' and `appending _r`, taking 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 argument, `struct_reent`, which is a pointer to an instance of the reentrancy structure, *`file`* and *`mode`*.

Each function that uses the global reentrancy structure uses the global variable, `_impure_ptr`, which points to a reentrancy structure.

This means that you have the following two ways to achieve reentrancy, with *both* requiring that *each* thread of execution control initialize a *unique global variable* of type, `struct _reent`.

- Using 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.
- Ensuring that *each* thread of execution control has a pointer to its own unique reentrancy structure in the global variable, `_impure_ ptr`, which calls the standard library subroutines.

The following functions are provided in both reentrant and non-reentrant versions.

```
_asctime_r      _read_r
_close_r        _raise_r
_dtoa_r         _rand_r
_errno_r        _setlocale_r
_fdopen_r       _stdin_r
_free_r         _stdout_r
_fork_r         _stderr_r
_fopen_r        _tempnam_r
_fstat_r        _tmpnam_r
_getchar_r      _tmpfile_r
_gets_r         _signal_r
_iprintf_r      _realloc_r
_localeconv_r   _strtoul_r
_lseek_r        _srand_r
_link_r         _system_r
_mkstemp_r      _strtod_r
_mktemp_r       _strtol_r
_malloc_r       _strtok_r
_open_r         _sbrk_r
_perror_r       _stat_r
_putchar_r      _unlink_r
_puts_r         _wait_r
_remove_r       _write_r
_rename_r
```

9

Miscellaneous Macros and Functions

The following documentation usually describes miscellaneous functions not discussed elsewhere. However, now, many use other header files.

One macro remains to discuss, “unctrl” on page 164.

unctrl

[translate characters to upper case]

SYNOPSIS

```
#include <unctrl.h>
char *unctrl(int c);
int unctrllen(int c);
```

DESCRIPTION

`unctrl` is a macro that returns the printable representation of *c* as a string. `unctrllen` is a macro that returns the length of the printable representation of *c*.

RETURNS

`unctrl` returns a string of the printable representation of *c*.

`unctrllen` returns the length of the string that is the printable representation of *c*.

COMPLIANCE

unctrl and unctrllen are not ANSI C.

No supporting OS subroutines are required.

10

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 the following 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). Providing stubs will allow your programs to link with the subroutines in `libc.a`.

Definitions for OS interface

The following discussions describe the complete set of system definitions (primarily subroutines) required. The accompanying examples implement the minimal functionality required to allow `libc` to link, failing gracefully where OS services are not available.

Graceful failure is permitted by returning an error code. A minor complication arises since 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 GNUPro C library provides a macro definition for `errno` in the header file, `errno.h`, serving to support reentrant routines (see “Reentrancy” on page 161). The bridge between these two interpretations of `errno` is straightforward: the C library routines with OS interface calls capture the `errno` values returned globally, recording 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`, and *then* disable the macro, as in the following example.

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

The examples in the following documentation describe the subroutines and their corresponding treatment of `errno`.

`_exit`
: Exits a program without cleaning up files. If your system doesn’t provide this routine, it is best to avoid linking with subroutines that require it (such as `exit` or `system`).

`close`
: Closes a file. Minimal implementation is shown in the following example (in which *`file`* stands for the *filename* to substitute).

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

`environ`
: Points to a list of environment variables and their values. For a minimal environment, the following empty list is adequate.

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

`execve`
Transfers control to a new process. Minimal implementation (for a system *without* processes) is shown in the following example (in which *`name`* stands for the *process name* to substitute, *`argv`* stands for the *argument value* to subtitute, and *`env`* stands for the *environment* to substitute).

```
#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) is shown in the following example.

```
#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 is shown in the following example.

```
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`, the minimal implementation is shown in the following example.

```
int isatty(int file){
   return 1;
```

```
}
```

`kill`
Send a signal. Minimal implementation is shown in the following example.

```
#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 is shown in the following example.

```
#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 is shown in the following example.

```
int lseek(int file, int ptr, int dir){
   return 0;
}
```

`read`
Read from a file. Minimal implementation is shown in the following example.

```
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, **`ld`**.

```
caddr_t sbrk(int incr){
   extern char end;
/* Defined by the linker.   */
   static char *heap_end;
   char *prev_heap_end;

   if (heap_end == 0) {
      heap_end = &end;
   }
   prev_heap_end = heap_end;

   if (heap_end + incr > stack_ptr)
```

```
            {
              _write (1, "Heap and stack collision\n", 25);
              abort ();
            }

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

`stat`
Status of a file (by name). Minimal implementation is shown in the following example.

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

`times`
Timing information for current process. Minimal implementation is shown in the following example.

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

`unlink`
Remove a file's directory entry. Minimal implementation is shown in the following example.

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

`wait`
Wait for a child process. Minimal implementation is shown in the following example.

```
#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 instance, to a serial port for debugging), you should make your minimal write capable of accomplishing this objective. The following minimal implementation is an incomplete example; it relies on a `writechar` subroutine to actually perform the output (this subroutine is not provided here since it is usually in assembler form from examples provided by your hardware manufacturer).

```
int write(int file, char *ptr, int len){
   int todo;

   for (todo = 0; todo < len; todo++) {
   writechar(*ptr++);
   }
   return len;
}
```

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 the GNUPro library, and achieve reentrancy by using a *reserved global data block* (see “Reentrancy” on page 161).

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

```
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`, as shown in the following example.

```
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`, as shown in the following example.

```
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`, as shown in the following example.

```
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`, as shown in the following example.

```
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`, as shown in the following example.

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

`_wait_r`
A reentrant version of `wait`. It takes a pointer to the global data block, which holds `errno`, as shown in the following example.

```
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`, as shown in the following example.

```
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`, as shown in the following example.

```
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`, as shown in the following example.

```
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`, as shown in the following example.

```
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`, as shown in the following example.

```
char *_sbrk_r(void *reent, size_t incr);
```

11

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 standards for C) or from `varargs.h` (for compatibility with a popular convention prior to meeting ANSI standard requirements for C). The following documentation describes in further detail the variable argument lists.

ANSI-standard macros (stdarg.h)

By ANSI standards for 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 standards for C define 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.

va_start

[initialize variable argument list]

SYNOPSIS

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

DESCRIPTION

Use `va_start` to initialize the variable argument list *`ap`*, so that `va_arg` can extract values from it. *`rightmost`* is the name of the last explicit argument in the parameter list (the argument immediately preceding the ellipsis, `...`, that flags variable arguments in an ANSI C function header). You can only use `va_start` in a function declared using this ellipsis notation (not, for example, in one of its subfunctions).

RETURNS

`va_start` does not return a result.

COMPLIANCE

ANSI C requires `va_start`.

va_arg

[extract a value from argument list]

SYNOPSIS

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

DESCRIPTION

`va_arg` returns the next unprocessed value from a variable argument list *`ap`* (which you must previously create with *`va_start`*). Specify the type for the value as the second parameter to the macro, type.

You may pass a `va_list` object ap to a subfunction, and use `va_arg` from the subfunction rather than from the function actually declared with an ellipsis in the header; however, in that case you may only use `va_arg` from the subfunction. ANSI C does not permit extracting successive values from a single variable-argument list from different levels of the calling stack.

There is no mechanism for testing whether there is actually a next argument available; you might instead pass an argument count (or some other data that implies an argument count) as one of the fixed arguments in your function call.

RETURNS

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

COMPLIANCE

ANSI C requires `va_arg`.

va_end

[abandon a variable argument list]

SYNOPSIS

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

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

RETURNS `va_end` does not return a result.

COMPLIANCE ANSI C requires `va_end`.

Traditional macros (varargs.h)

If your C compiler predates requirements set by ANSI standards for 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.

va_dcl

[declare variable arguments]

SYNOPSIS

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

DESCRIPTION To use the `varargs.h` version of variable argument lists, you must declare your function with a call to the macro `va_alist` as its argument list, and use `va_dcl` as the declaration.

WARNING! Do not use a semicolon after `va_dcl`.

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

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

va_start

[initialize variable argument list]

SYNOPSIS

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

DESCRIPTION

With the `varargs.h` macros, use `va_start` to initialize a data structure *ap* to permit manipulating a variable argument list. *ap* must have the type `va_alist`.

RETURNS

`va_start` does not return a result.

COMPLIANCE

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

va_arg

[extract a value from argument list]

SYNOPSIS

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

DESCRIPTION

`va_arg` returns the next unprocessed value from a variable argument list *`ap`* (which you must previously create with *`va_start`*). Specify the type for the value as the second parameter to the macro, *`type`*.

RETURNS

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

COMPLIANCE

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

va_end

[abandon a variable argument list]

SYNOPSIS

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

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

RETURNS `va_end` does not return a result.

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

GNUPRO™ TOOLKIT

GNUPro Math Library

June, 1998
98r1

CYGNUS

This documentation has been prepared by Cygnus Technical Publications; contact the Cygnus Technical Publications staff: **doc@cygnus.com**.

1 Mathematical Functions (`math.h`)

The following documentation groups a wide variety of mathematical functions. The corresponding definitions and declarations are in `math.h`.

- “floor, floorf, ceil, ceilf” on page 204
- “fmod, fmodf” on page 205
- “frexp, frexpf” on page 206
- “gamma, gammaf, lgamma, lgammaf, gamma_r, gammaf_r, lgamma_r, lgammaf_r” on page 207
- “hypot, hypotf” on page 209
- “ilogb, ilogbf” on page 210
- “infinity, infinityf” on page 211
- “isnan, isnanf, isinf, isinff, finite, finitef” on page 212
- “ldexp, ldexpf” on page 213
- “log, logf” on page 214
- “log10, log10f” on page 215
- “log1p, log1pf” on page 216
- “matherr” on page 217
- “modf, modff” on page 219
- “nan, nanf” on page 220
- “nextafter, nextafterf” on page 221
- “pow, powf” on page 222
- “rint, rintf, remainder, remainderf” on page 223
- “scalbn, scalbnf” on page 224
- “sqrt, sqrtf” on page 225
- “sin, sinf, cos, cosf” on page 226
- “sinh, sinhf” on page 227
- “tan, tanf” on page 228
- “tanh, tanhf” on page 229

Two definitions from `math.h` are of particular interest.

- The representation of infinity as a `double` is defined as `HUGE_VAL`; this number being returned on overflow by many functions.
- The structure, `exception`, is used when you write customized error handlers for the mathematical functions. You can customize error handling for most of these functions by defining your own version of `matherr`; see the discussion with “nan, nanf” on page 220 for specific details.

Since the error handling code calls `fputs`, the mathematical subroutines require *stubs* or minimal implementations for the same list of OS subroutines as `fputs`: `close`,

`fstat`, `isatty`, `lseek`, `read`, `sbrk`, `write`. See “Reentrant covers for OS subroutines” on page 171 for specific discussion of subroutine calls, and for sample minimal implementations of these support subroutines.

Alternative declarations of the mathematical functions, which exploit specific machine capabilities to operate faster—although, generally, they have less error checking and may reflect additional limitations on some machines—are available when you include `fastmath.h` instead of `math.h`.

See also “Reentrancy Properties of libm” on page 230.

Version of math library

There are four different versions of the math library routines: IEEE, POSIX, X/Open, or SVID.

The version may be selected at runtime by setting the global variable, `_LIB_VERSION`, defined in `math.h`. It may be set to one of the following constants defined in math.h: `_IEEE_`, `_POSIX_`, `_XOPEN_`, or `_SVID_`.

The `_LIB_VERSION` variable is not specific to any thread, and changing it will affect all threads.

The versions of the library differ only in how errors are handled.

In IEEE mode, the `matherr` function is never called, no warning messages are printed, and `errno` is never set.

In POSIX mode, `errno` is set correctly, but the `matherr` function is never called and no warning messages are printed.

In X/Open mode, `errno` is set correctly, and `matherr` is called, but warning messages are not printed. In SVID mode, functions that overflow return 3.40282346638528860e+38, the maximum single precision floating point value, rather than infinity. Also, `errno` is set correctly, `matherr` is called, and, if `matherr` returns `0`, warning messages are printed for some errors. For example, by default '`log(-1.0)`' writes the following message on standard error output.

```
log: DOMAIN error.
```

The library is set to X/Open mode by default.

acos, acosf

[arc cosine]

SYNOPSIS

```
#include <math.h>
double acos(double x);
float acosf(float x);
```

DESCRIPTION

`acos` computes the inverse cosine (arc cosine) of the input value. Arguments to `acos` must be in the range of -1 to 1.

`acosf` is identical to `acos`, except that it performs its calculations on floats.

RETURNS

`acos` and `acosf` return values in radians, in the range of 0 to π.

If x is not between -1 and 1, the returned value is `NaN` (not a number), the global variable, `errno`, is set to `EDOM`, and a `DOMAIN error` message is sent as standard error output.

You can modify error handling for these functions using `matherr`.

acosh, acoshf

[inverse hyperbolic cosine]

SYNOPSIS

```
#include <math.h>
double acosh(double x);
float acoshf(float x);
```

DESCRIPTION

`acosh` calculates the inverse hyperbolic cosine of *x*.

`acosh` is defined as the following equation shows.

$$ln(x + \sqrt{x^2 - 1})$$

x in the synopsis is the same as *x* in the equation and must be a number greater than or equal to 1.

`acoshf` is identical, other than taking and returning floats.

RETURNS

`acosh` and `acoshf` return the calculated value. If *x* is less than 1, the return value is `NaN` and `errno` is set to `EDOM`.

You can change the error-handling behavior with the non-ANSI `matherr` function.

COMPLIANCE

Neither `acosh` nor `acoshf` are ANSI C.

They are not recommended for portable programs.

asin, asinf

[arc sine]

SYNOPSIS

```
#include <math.h>
double asin(double x);
float asinf(float x);
```

DESCRIPTION

`asin` computes the inverse sine (arc sine) of the argument, *x*. Arguments to `asin` must be in the range -1 to 1.

`asinf` is identical to `asin`, other than taking and returning floats.

You can modify error handling for these routines using `matherr`.

RETURNS

`asin` returns values in radians, in the range of $-\pi/2$ to $\pi/2$.

If *x* is not in the range -1 to 1, `asin` and `asinf` return `NaN` (not a number), set the global variable, `errno`, to `EDOM`, and issue a `DOMAIN error` message.

You can change this error treatment using `matherr`.

asinh, asinhf

[inverse hyperbolic sine]

SYNOPSIS

```
# include <math.h>
double asinh(double x);
float asinhf(float x);
```

DESCRIPTION

`asinh` calculates the inverse hyperbolic sine of *x*.

`asinh` is defined as in the following calculation.

$$sign(x) \times ln(|x| + \sqrt{1 + x^2})$$

`asinhf` is identical, other than taking and returning floats.

RETURNS

`asinh` and `asinhf` return the calculated value.

COMPLIANCE

Neither `asinh` nor `asinhf` are ANSI C.

atan, atanf

[arc tangent]

SYNOPSIS

```
#include <math.h>
double atan(double x);
float atanf(float x);
```

DESCRIPTION

`atan` computes the inverse tangent (arc tangent) of the input value.

`atanf` is identical to `atan`, save that it operates on floats.

RETURNS

`atan` returns a value in radians, in the range of $-\pi/2$ to $\pi/2$.

COMPLIANCE

`atan` is ANSI C.

`atanf` is an extension.

atan2, atan2f

[arc tangent of y/x]

SYNOPSIS

```
#include <math.h>
double atan2(double y,double x);
float atan2f(float y,float x);
```

DESCRIPTION

`atan2` computes the inverse tangent (arc tangent) of `y/x`. `atan2` produces the correct result even for angles near $-\pi/2$ or $\pi/2$. (that is, when x is near 0).

`atan2f` is identical to `atan2`, save that it takes and returns float.

RETURNS

atan2 and atan2f return a value in radians, in the range of $-\pi$ to π. If both `x` and `y` are 0.0, `atan2` causes a `DOMAIN error`. You can modify error handling for these functions using `matherr`.

COMPLIANCE

`atan2` is ANSI C.

`atan2f` is an extension.

atanh, atanhf

[inverse hyperbolic tangent]

SYNOPSIS

```
#include <math.h>
double atanh(double x);
float atanhf(float x);
```

DESCRIPTION

`atanh` calculates the inverse hyperbolic tangent of *x*.

`atanhf` is identical, other than taking and returning float values.

RETURNS

`atanh` and `atanhf` return the calculated value.

If $|x|$ is greater than 1, the global, `errno`, is set to `EDOM` and the result is a `NaN`. A `DOMAIN error` is reported.

If $|x|$ is 1, the global, `errno`, is set to `EDOM`; and the result is infinity with the same sign as *x*. A `SING error` is reported.

You can modify the error handling for these routines using `matherr`.

COMPLIANCE

Neither `atanh` nor `atanhf` are ANSI C.

jN, jNf, yN, yNf

[Bessel functions]

SYNOPSIS

```
#include <math.h>
double j0(double x);
float j0f(float x);
double j1(double x);
float j1f(float x);
double jn(int n, double x);
float jnf(int n, float x);
double y0(double x);
float y0f(float x);
double y1(double x);
float y1f(float x);
double yn(int n, double x);
float ynf(int n, float x);
```

DESCRIPTION The Bessel functions are a family of functions that solve the following differential equation.

$$x^2\frac{x^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - p^2)y = 0$$

These functions have many applications in engineering and physics.

`jn` calculates the Bessel function of the first kind of order, n. `j0` and `j1` are special cases for order, 0, and order, 1, respectively. Similarly, `yn` calculates the Bessel function of the second kind of order, n, and `y0` and `y1` are special cases for order, 0 and 1, respectively.

`jnf`, `j0f`, `j1f`, `ynf`, `y0f`, and `y1f` perform the same calculations, but on `float` rather than `double` values.

RETURNS The value of each Bessel function at *x* is returned.

COMPLIANCE None of the Bessel functions are in ANSI C.

cbrt, cbrtf

[cube root]

SYNOPSIS

```
#include <math.h>
double cbrt(double x);
float cbrtf(float x);
```

DESCRIPTION `cbrt` computes the cube root of the argument.

RETURNS The cube root is returned.

COMPLIANCE `cbrt` is in System V release 4.

`cbrtf` is an extension.

copysign, copysignf

[sign of *y*, magnitude of *x*]

SYNOPSIS

```
#include <math.h>
double copysign (double x, double y);
float copysignf (float x, float y);
```

DESCRIPTION

`copysign` constructs a number with the magnitude (absolute value) of its first argument, *x*, and the sign of its second argument, *y*.

`copysignf` does the same thing; the two functions differ only in the type of their arguments and result.

RETURNS

`copysign` returns a double with the magnitude of *x* and the sign of *y*. `copysignf` returns a float with the magnitude of *x* and the sign of *y*.

COMPLIANCE

`copysign` is not required by either ANSI C or the System V Interface Definition (Issue 2).

cosh, coshf

[hyperbolic cosine]

SYNOPSIS

```
#include <math.h>
double cosh(double x);
float coshf(float x)
```

DESCRIPTION

cosh computes the hyperbolic cosine of the argument x.

cosh(x) is defined as the following equation.

$$\frac{(e^2 + e^{-x})}{2}$$

Angles are specified in radians. `coshf` is identical, save that it takes and returns float.

RETURNS

The computed value is returned. When the correct value would create an overflow, `cosh` returns the value, `HUGE_VAL`, with the appropriate sign, and the global value, `errno`, is set to `ERANGE`.

You can modify error handling for these functions using the function, `matherr`.

COMPLIANCE

`cosh` is ANSI.

`coshf` is an extension.

erf, erff, erfc, erfcf

[error function]

SYNOPSIS

```
#include <math.h>
double erf(double x);
float erff(float x);
double erfc(double x);
float erfcf(float x);
```

DESCRIPTION

`erf` calculates an approximation to the ***error function*** which estimates the probability that an observation will fall within *x* standard deviations of the mean (assuming a normal distribution).

The error function is defined as the following differential equation.

$$\frac{2}{\sqrt{\pi}} \times \int_0^x e^{-t^2} dt$$

`erfc` calculates the complementary probability; that is, `erfc(`*x*`)` is `1-erf(`*x*`)`. `erfc` is computed directly, so that you can use it to avoid the loss of precision that would result from subtracting large probabilities (on large *x*) from 1.

`erff` and `erfcf` differ from `erf` and `erfc` only in the argument and result types.

RETURNS

For positive arguments, `erf` and all its variants return a probability—a number between 0 and 1.

COMPLIANCE

None of the variants of `erf` are ANSI C.

exp, expf

[exponential]

SYNOPSIS

```
#include <math.h>
double exp(double x);
float expf(float x);
```

DESCRIPTION

`exp` and `expf` calculate the exponential of x, that is, e^x (where e is the base of the natural system of logarithms, approximately 2.71828).

You can use the (non-ANSI) function, `matherr`, to specify error handling for these functions.

RETURNS

On success, `exp` and `expf` return the calculated value. If the result underflows, the returned value is 0. If the result overflows, the returned value is `HUGE_VAL`. In either case, `errno` is set to `ERANGE`.

COMPLIANCE

`exp` is ANSI C.

`expf` is an extension.

expm1, expm1f

[exponential minus 1]

SYNOPSIS

```
#include <math.h>
double expm1(double x);
float expm1f(float x);
```

DESCRIPTION

`expm1` and `expm1f` calculate the exponential of *x* and subtract 1, that is, $e^x - 1$ (where e is the base of the natural system of logarithms, approximately 2.71828).

The result is accurate even for small values of *x*, where using `exp(x)-1` would lose many significant digits.

RETURNS

$e^x - 1$.

COMPLIANCE

Neither `expm1` nor `expm1f` is required by ANSI C or by the System V Interface Definition (Issue 2).

fabs, fabsf

[absolute value (magnitude)]

SYNOPSIS

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

DESCRIPTION

`fabs` and `fabsf` calculate $|x|$, the absolute value (magnitude) of the argument, *x*, by direct manipulation of the bit representation of *x*.

RETURNS

The calculated value is returned. No errors are detected.

COMPLIANCE

`fabs` is ANSI.

`fabsf` is an extension.

floor, floorf, ceil, ceilf

[floor and ceiling]

SYNOPSIS

```
#include <math.h>
double floor(double x);
float floorf(float x);
double ceil(double x);
float ceilf(float x);
```

DESCRIPTION

`floor` and `floorf` find $\lfloor x \rfloor$, the nearest integer less than or equal to *x*. `ceil` and `ceilf` find $\lceil x \rceil$, the nearest integer greater than or equal to *x*.

RETURNS

`floor` and `ceil` return the integer result as a `double`.

`floorf` and `ceilf` return the integer result as a `float`.

COMPLIANCE

`floor` and `ceil` are ANSI.

`floorf` and `ceilf` are extensions.

fmod, fmodf

[floating-point remainder (modulo)]

SYNOPSIS

```
#include <math.h>
double fmod(double x, double y)
float fmodf(float x, float y)
```

DESCRIPTION

The `fmod` and `fmodf` functions compute the floating-point remainder of x/y (*x modulo y*).

RETURNS

The `fmod` function returns the value, $x - i \times y$, for the largest integer, i, such that, if y is nonzero, the result has the same sign as x and magnitude less than the magnitude of y.

`fmod(`x`,0)` returns `NaN`, and sets `errno` to `EDOM`.

You can modify error treatment for these functions using `matherr`.

COMPLIANCE

`fmod` is ANSI C.

`fmodf` is an extension.

frexp, frexpf

[split floating-point number]

SYNOPSIS

```
#include <math.h>
double frexp(double val, int *exp);
float frexpf(float val, int *exp);
```

DESCRIPTION

All *non-zero*, *normal numbers* can be described as `m * 2**p`.

`frexp` represents the double, `val`, as a *mantissa*, `m`, and a power of 2^p.

The resulting mantissa will always be greater than or equal to 0.5, and less than 1.0 (as long as `val` is non-zero).

The power of two will be stored in `*exp`.

`m` and `p` are calculated so that $val = m \times 2^p$.

`frexpf` is identical, other than taking and returning floats rather than doubles.

RETURNS

`frexp` returns the mantissa, `m`. If `val` is 0, infinity, or `NaN`, `frexp` will set `*exp` to 0 and return `val`.

COMPLIANCE

`frexp` is ANSI.

`frexpf` is an extension.

gamma, gammaf, lgamma, lgammaf, gamma_r, gammaf_r, lgamma_r, lgammaf_r

[logarithmic gamma function]

SYNOPSIS

```
#include <math.h>
double gamma(double x);
float gammaf(float x);

double lgamma(double x);
float lgammaf(float x);

double gamma_r(double x, int *signgamp);
float gammaf_r(float x, int *signgamp);

double lgamma_r(double x, int *signgamp);
float lgammaf_r(float x, int *signgamp);
```

DESCRIPTION

`gamma` calculates $ln(\Gamma(x))$, the natural logarithm of the gamma function of *x*. The gamma function (`exp(gamma(x))`) is a generalization of factorial, and retains the property that $\Gamma(N) \equiv N \times \Gamma(N_{-1})$. Accordingly, the results of the `gamma` function itself grow *very quickly*. `gamma` is defined as $ln(\Gamma(x))$ rather than simply $\Gamma(x)$, to extend the useful range of results representable.

The sign of the result is returned in the global variable, `signgam`, which is declared in `math.h`.

`gammaf` performs the same calculation as `gamma`, although using and returning `float` values.

`lgamma` and `lgammaf` are alternate names for `gamma` and `gammaf`. The use of `lgamma` instead of `gamma` is a reminder that these functions compute the log of the `gamma` function, rather than the `gamma` function itself.

The functions, `gamma_r`, `gammaf_r`, `lgamma_r`, and `lgammaf_r` are just like `gamma`, `gammaf`, `lgamma`, and `lgammaf`, respectively, although they take an *additional* argument. This additional argument is a pointer to an integer. As an additional argument, it is used to return the sign of the result, and the global variable, `signgam`, is not used. These functions may be used for reentrant calls (although they will still set the global variable, `errno`, *if an error occurs*).

RETURNS

Normally, the computed result is returned.

When *x* is a nonpositive integer, `gamma` returns `HUGE_VAL`, and `errno` is set to `EDOM`. If the result overflows, `gamma` returns `HUGE_VAL`, and `errno` is set to

ERANGE. You can modify this error treatment using matherr.

COMPLIANCE Neither gamma nor gammaf is ANSI C.

hypot, hypotf

[distance from origin]

SYNOPSIS

```
#include <math.h>
double hypot(double x, double y);
float hypotf(float x, float y);
```

DESCRIPTION

`hypot` calculates the Euclidean distance: $\sqrt{x^2+y^2}$ between the origin (`0`,`0`) and a point represented by the Cartesian coordinates (`x`,`y`). `hypotf` differs only in the type of its arguments and result.

RETURNS

Normally, the distance value is returned. On overflow, `hypot` returns `HUGE_VAL` and sets `errno` to `ERANGE`.

You can change the error treatment with `matherr`.

COMPLIANCE

`hypot` and `hypotf` are not ANSI C.

ilogb, ilogbf

[get exponent of floating point number]

SYNOPSIS

```
#include <math.h>
int ilogb(double val);
int ilogbf(float val);
```

DESCRIPTION

All non zero, normal numbers can be described as *m*`* 2**`*p*. `ilogb` and `ilogbf` examine the argument, *val*, and return *p*. The functions, `frexp` and `frexpf`, are similar to `ilogb` and `ilogbf`, but also return *m*.

RETURNS

`ilogb` and `ilogbf` return the power of two used to form the floating point argument. If *val* is `0`, they return `-INT_MAX` (`INT_MAX` is defined in `limits.h`). If *val* is infinite, or `NaN`, they return `INT_MAX`.

COMPLIANCE

Neither `ilogb` nor `ilogbf` is required by ANSI C or by the System V Interface Definition (Issue 2).

infinity, infinityf

[representation of infinity]

SYNOPSIS

```
#include <math.h>
double infinity(void);
float infinityf(void);
```

DESCRIPTION

`infinity` and `infinityf` return the special number IEEE, `infinity`, in, respectively, double and single precision arithmetic.

isnan, isnanf, isinf, isinff, finite, finitef

[test for exceptional numbers]

SYNOPSIS

```
#include <ieeefp.h>
int isnan(double arg);
int isinf(double arg);
int finite(double arg);
int isnanf(float arg);
int isinff(float arg);
int finitef(float arg);
```

DESCRIPTION

These functions provide information on the floating point argument supplied.

The following are five major number formats.

`zero`
A number which contains all zero bits.

`subnormal`
Used to represent number with a zero exponent, but a non-zero fraction.

`normal`
A number with an exponent, and a fraction.

`infinity`
A number with an all 1's exponent and a zero fraction.

`NAN`
A number with an all 1's exponent and a non-zero fraction.

RETURNS

`isnan` returns 1 if the argument is a `NaN`.

`isinf` returns 1 if the argument is infinity.

`finite` returns 1 if the argument is zero, subnormal or normal.

The `isnanf`, `isinff` and `finitef` perform the same operations as their `isnan`, `isinf` and `finite` counterparts, but on single precision floating point numbers.

ldexp, ldexpf

[load exponent]

SYNOPSIS

```
#include <math.h>
double ldexp(double val, int exp);
float ldexpf(float val, int exp);
```

DESCRIPTION

`ldexp` calculates the value, $val \times 2^{exp}$. `ldexpf` is identical, save that it takes and returns `float` rather than `double` values.

RETURNS

`ldexp` returns the calculated value. Underflow and overflow both set `errno` to `ERANGE`. On underflow, `ldexp` and `ldexpf` return `0.0`. On overflow, `ldexp` returns plus or minus `HUGE_VAL`.

COMPLIANCE

`ldexp` is ANSI; `ldexpf` is an extension.

log, logf

[natural logarithms]

SYNOPSIS

```
#include <math.h>
double log(double x);
float logf(float x);
```

DESCRIPTION

Return the natural logarithm of *x*, that is, its logarithm base, *e*, (where *e* is the base of the natural system of logarithms, 2.71828...). `log` and `logf` are identical save for the return and argument types.

You can use the (non-ANSI) function, `matherr`, to specify error handling for these functions.

RETURNS

Normally, returns the calculated value. When *x* is zero, the returned value is `-HUGE_VAL` and `errno` is set to `ERANGE`. When *x* is negative, the returned value is `-HUGE_VAL` and `errno` is set to `EDOM`. You can control the error behavior, using `matherr`.

COMPLIANCE

`log` is ANSI, `logf` is an extension.

log10, log10f

[base 10 logarithms]

SYNOPSIS

```
#include <math.h>
double log10(double x);
float log10f(float x);
```

DESCRIPTION

`log10` returns the base 10 logarithm of *x*. It is implemented as `log(x)/log(10)`.

`log10f` is identical, save that it takes and returns `float` values.

RETURNS

`log10` and `log10f` return the calculated value. See the description for “log, logf” on page 214 for information on errors.

COMPLIANCE

`log10` is ANSI C. `log10f` is an extension.

log1p, log1pf

[log of 1 + *x*]

SYNOPSIS

```
#include <math.h>
double log1p(double x);
float log1pf(float x);
```

DESCRIPTION

`log1p` calculates *ln(1+x)*, the natural logarithm of `1+`*x*. You can use `log1p` rather than `log(1+`*x*`)` for greater precision when *x* is very small.

`log1pf` calculates the same thing, but accepts and returns `float` values rather than `double`.

RETURNS

`log1p` returns a `double`, the natural log of `1+`*x*. `log1pf` returns a `float`, the natural log of `1+`*x*.

COMPLIANCE

Neither `log1p` nor `log1pf` is required by ANSI C or by the System V Interface Definition (Issue 2).

matherr

[modifiable math error handler]

SYNOPSIS

```
#include <math.h>
int matherr(struct exception *e);
```

DESCRIPTION

`matherr` is called whenever a math library function generates an error. You can replace `matherr` by your own subroutine to customize error treatment. The customized `matherr` must return 0 if it fails to resolve the error, and non-zero if the error is resolved.

When `matherr` returns a nonzero value, no error message is printed and the value of `errno` is not modified.

You can accomplish either or both of these things in your own `matherr` using the information passed in the structure, `*e`. The following example shows the `exception` structure (defined in `math.h`).

```
struct exception {
            int type;
            char *name;
            double arg1, arg2, retval;
int err;
};
```

The members of the exception structure have the following meanings.

`type`
The type of mathematical error that occurred; macros encoding error types are also defined in `math.h`.

`name`
A pointer to a null-terminated string holding the name of the math library function where the error occurred.

`arg1`, **`arg2`**
The arguments which caused the error.

`retval`
The error return value (what the calling function will return).

`err`
If set to be non-zero, this is the new value assigned to `errno`.

The error types defined in `math.h` represent possible mathematical errors as follows.

`DOMAIN`
An argument was not in the domain of the function; e.g., `log(-1.0)`.

SING
The requested calculation would result in a singularity; e.g., `pow(0.0,-2.0)`.

OVERFLOW
A calculation would produce a result too large to represent; e.g., `exp(1000.0)`.

UNDERFLOW
A calculation would produce a result too small to represent; e.g., `exp(-1000.0)`.

TLOSS
Total loss of precision. The result would have no significant digits; e.g., `sin(10e70)`.

PLOSS
Partial loss of precision.

RETURNS The library definition for `matherr` returns `0` in all cases. You can change the calling function's result from a customized `matherr` by modifying `e->retval`, which propagates backs to the caller. If `matherr` returns `0` (indicating that it was not able to resolve the error) the caller sets `errno` to an appropriate value, and prints an error message.

COMPLIANCE `matherr` is not ANSI C.

modf, modff

[split fractional and integer parts]

SYNOPSIS

```
#include <math.h>
double modf(double val, double *ipart);
float modff(float val, float *ipart);
```

DESCRIPTION

`modf` splits the double *`val`* apart into an integer part and a fractional part, returning the fractional part and storing the integer part in `*`*`ipart`*. No rounding whatsoever is done; the sum of the integer and fractional parts is guaranteed to be exactly equal to *`val`*.

That is, if `.`*`realpart`*`=modf(`*`val`*`,&`*`intpart`*`)`; then *`realpart`*`+`*`intpart`* is the same as *`val`*.

`modff` is identical, save that it takes and returns `float` rather than `double` values.

RETURNS

The fractional part is returned. Each result has the same sign as the supplied argument, *`val`*.

COMPLIANCE

`modf` is ANSI C. `modff` is an extension.

nan, nanf

[representation of infinity]

SYNOPSIS

```
#include <math.h>
double nan(void);
float nanf(void);
```

DESCRIPTION

`nan` and `nanf` return an IEEE **`NaN`** (Not a Number) in double and single precision arithmetic respectively.

nextafter, nextafterf

[get next number]

SYNOPSIS

```
#include <math.h>
double nextafter(double val, double dir);
float nextafterf(float val, float dir);
```

DESCRIPTION

`nextafter` returns the double precision floating point number closest to *`val`* in the direction toward *`dir`*.

`nextafterf` performs the same operation in single precision. For example, `nextafter(0.0,1.0)` returns the smallest positive number, which is representable in double precision.

RETURNS

Returns the next closest number to *`val`* in the direction toward *`dir`*.

COMPLIANCE

Neither `nextafter` nor `nextafterf` is required by ANSI C or by the System V Interface Definition (Issue 2).

pow, powf

[*x* to the power *y*]

SYNOPSIS

```
#include <math.h>
double pow(double x, double y);
float pow(float x, float y);
```

DESCRIPTION

`pow` and `powf` calculate *x* raised to the exp1.0nty. (That is, x^y.)

RETURNS

On success, `pow` and `powf` return the value calculated.

When the argument values would produce overflow, `pow` returns `HUGE_VAL` and sets `errno` to `ERANGE`. If the argument *x* passed to `pow` or `powf` is a negative noninteger, and *y* is also not an integer, then `errno` is set to `EDOM`. If *x* and *y* are both 0, then `pow` and `powf` return `1`.

You can modify error handling for these functions using `matherr`.

COMPLIANCE

`pow` is ANSI C. `powf` is an extension.

rint, rintf, remainder, remainderf

[round and remainder]

SYNOPSIS

```
#include <math.h>
double rint(double x);
float rintf(float x);
double remainder(double x, double y);
float remainderf(float x, float y);
```

DESCRIPTION

`rint` and `rintf` returns their argument rounded to the nearest integer. `remainder` and `remainderf` find the remainder of *x*/*y*; this value is in the range -*y*/2 ... +*y*/2.

RETURNS

`rint` and `remainder` return the integer result as a double.

COMPLIANCE

`rint` and `remainder` are System Vr4. `rintf` and `remainderf` are extensions.

scalbn, scalbnf

[scale by integer]

SYNOPSIS

```
#include <math.h>
double scalbn(double x, int y);
float scalbnf(float x, int y);
```

DESCRIPTION

`scalbn` and `scalbnf` scale *x* by *n*, returning *x* times 2 to the power *n*. The result is computed by manipulating the exponent, rather than by actually performing an exponentiation or multiplication.

RETURNS

x times 2 to the power *n*.

COMPLIANCE

Neither `scalbn` nor `scalbnf` is required by ANSI C or by the System V Interface Definition (Issue 2).

sqrt, sqrtf

[positive square root]

SYNOPSIS

```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
```

DESCRIPTION

`sqrt` computes the positive square root of the argument. You can modify error handling for this function with `matherr`.

RETURNS

On success, the square root is returned. If *x* is real and positive, then the result is positive. If *x* is real and negative, the global value `errno` is set to `EDOM` (domain error).

COMPLIANCE

`sqrt` is ANSI C. `sqrtf` is an extension.

sin, sinf, cos, cosf

[sine or cosine]

SYNOPSIS

```
#include <math.h>
double sin(double x);
float sinf(float x);
double cos(double x);
float cosf(float x);
```

DESCRIPTION

`sin` and `cos` compute (respectively) the sine and cosine of the argument *x*. Angles are specified in radians.

`sinf` and `cosf` are identical, save that they take and return `float` values.

RETURNS

The sine or cosine of *x* is returned.

COMPLIANCE

`sin` and `cos` are ANSI C. `sinf` and `cosf` are extensions.

sinh, sinhf

[hyperbolic sine]

SYNOPSIS

```
#include <math.h>
double sinh(double x);
float sinhf(float x);
```

DESCRIPTION

`sinh` computes the hyperbolic sine of the argument *x*. Angles are specified in radians. `sinh( x)` is defined as:

$$\frac{e^x - e^{-x}}{2}$$

`sinhf` is identical, save that it takes and returns `float` values.

RETURNS

The hyperbolic sine of *x* is returned. When the correct result is too large to be representable (an overflow), `sinh` returns `HUGE_VAL` with the appropriate sign, and sets the global value `errno` to `ERANGE`.

You can modify error handling for these functions with `matherr`.

COMPLIANCE

`sinh` is ANSI C. `sinhf` is an extension.

tan, tanf

[tangent]

SYNOPSIS

```
#include <math.h>
double tan(double x);
float tanf(float x);
```

DESCRIPTION `tan` computes the tangent of the argument *x*. Angles are specified in radians. `tanf` is identical, save that it takes and returns `float` values.

RETURNS The tangent of *x* is returned.

COMPLIANCE `tan` is ANSI. `tanf` is an extension.

tanh, tanhf

[hyperbolic tangent]

SYNOPSIS

```
#include <math.h>
double tanh(double x);
float tanhf(float x);
```

DESCRIPTION

`tanh` computes the hyperbolic tangent of the argument *x*. Angles are specified in radians.

`tanh(x)` is defined as the following input.

```
sinh(x)/cosh(x)
```

`tanhf` is identical, save that it takes and returns `float` values.

RETURNS

The hyperbolic tangent of *x* is returned.

COMPLIANCE

`tanh` is ANSI C. `tanhf` is an extension.

Reentrancy Properties of libm

When a libm function detects an exceptional case, errno may be set, the matherr function may be called, and a error message may be written to the standard error stream. This behavior may not be reentrant.

With reentrant C libraries like the GNUPro C library, errno is a macro which expands to the per-thread error value. This makes it thread safe.

When the user provides his own matherr function it must be reentrant for the math library as a whole to be reentrant.

In normal debugged programs, there are usually no math subroutine errors—and therefore no assignments to errno and no matherr calls; in that situation, the math functions behave reentrantly.

GNUPRO™ TOOLKIT

GNU C++ Iostreams Library

June, 1998
98r1

CYGNUS

This documentation has been prepared by Cygnus Technical Publications; contact the Cygnus Technical Publications staff: **doc@cygnus.com**.

1

Introduction to Iostreams

Iostream classes implement most of the features of AT&T version 2.0 `iostream` library classes, and most of the features of the ANSI X3J16 library draft (based on the AT&T design). However they only support streams of type, `char`, rather than using a template.

The following documentation is meant as a reference. For tutorial material on iostreams, see the corresponding section of any popular introduction to C++.

Licensing terms for libio

Since the iostream classes are so fundamental to standard C++, the Free Software Foundation has agreed to a special exception to its standard license, in order to link programs with libio.a.

As a special exception, in order to link this library with files compiled with a GNU compiler to produce an executable, the resulting executable does not have the coverage of the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might have the coverage of the GNU General Public License.

The code is under the GNU General Public License (version 2) for all purposes other than linking with this library, meaning that you can modify and redistribute the code as usual, although, if you do, your modifications, and anything you link with the modified code, must be available to others on the same terms.

Acknowledgments

Per Bothner wrote most of the `iostream` library, although some portions have their origins elsewhere in the free software community.

Heinz Seidl wrote the IO manipulators.

The floating-point conversion software is by David M. Gay of AT&T.

Some code was derived from parts of BSD 4.4, written at the University of California, Berkeley.

The `iostream` classes are found in the `libio` library. An early version was originally distributed in `libg++`. Doug Lea was the original author of `libg++`, and some of the file management code still in `libio` is his property.

Various people found bugs or offered suggestions. Hongjiu Lu worked hard to use the library as the default `stdio` implementation for Linux, and has provided much stress-testing of the library.

2

Operators and Default Streams

The GNU `iostream` library, `libio`, implements the standard input and output facilities for C++. These facilities are roughly analogous (in their purpose and ubiquity, at least) with those defined by the C `stdio` functions. Although these definitions come from a library, rather than being part of the core language, they are sufficiently central to be specified in the latest draft standard for C++. The following documentation discusses operators and default streams in more detail.

Input and Output Operators

You can use two operators defined in this library for basic input and output operations. They are familiar from any C++ introductory textbook: **<<** for ***output***, and **>>** for ***input***. (Think of data flowing in the direction of the *arrows*.) The **<<** (output) and **>>** (input) operators are often used in conjunction with the following three streams that are open by default.

`ostream` **cout**
(Variable)

The standard output stream, analogous to the C `stdout`.

`ostream` **cin**
(Variable)

The standard input stream, analogous to the C `stdin`.

`ostream` **cerr**
(Variable)

An alternative output stream for errors, analogous to the C `stderr`. The barebones C++ version of the traditional "`hello`" program uses **<<** and `cout`, as the following example shows.

```
#include <iostream.h>

int main(int argc, char **argv)
{
     out << "Well, hi there.\n";
     return 0;
}
```

Managing operators for input and output

Casual use of these operators may be seductive, but—other than in writing throwaway code for your own use—it is not necessarily simpler than managing input and output in any other language. For example, robust code should check the state of the input and output streams between operations (for example, using the method, `good`). See "Checking the state of a stream" on page 244. You may also need to adjust maximum input or output field widths, using manipulators like `setw` or `setprecision`.

<< on **ostream**

(Operator)

Write output to an open output stream of class `ostream`. Defined by this library on any *`object`* of a C++ primitive type, and on other classes of the library. You can overload the definition for any of your own applications' classes.

Returns a reference to the implied argument, `*this` (the open stream it writes on), permitting multiple inputs like the following statement.

```
cout << "The value of i is " << i << "\n";
```

>> on **istream**

(Operator)

Read input from an open input stream of class `istream`. Defined by this library on primitive *numeric*, *pointer*, and *string* types, you can extend the definition for any of your own applications' classes.

Returns a reference to the implied argument, `*this` (the open stream it reads), permitting multiple inputs in one statement.

3

Stream Classes

In the documentation for “Input and Output Operators” on page 238, there is a discussion of the classes, `ostream` and `istream`, for output and input, respectively. These classes share certain properties, captured in their base class, `ios`.

The following documentation discusses the properties and functionality of the stream classes.

Shared properties: class ios

The base class ios provides methods to test and manage the state of input or output streams.

ios delegates the job of actually reading and writing bytes to the abstract class, streambuf, which is designed to provide buffered streams (compatible with C, in the GNU implementation). See “Using the streambuf Layer” on page 269 for information on the facilities available at the streambuf level.

ios::ios (streambuf * *sb* [, ostream * *tie*])

(Constructor)

The ios constructor by default initializes a new ios, and if you supply a streambuf sb to associate with it, sets the state good in the new ios object. It also sets the default properties of the new object. You can also supply an optional second argument, *tie*, to the constructor; if present, it is an initial value for **ios::tie**, to associate the new ios object with another stream.

ios::~ios ()

(Destructor)

The **ios** destructor is virtual, permitting application-specific behavior when a stream is closed—typically, the destructor frees any storage associated with the stream and releases any other associated objects.

Checking the state of a stream

Use this collection of methods to test for (or signal) errors and other exceptional conditions of streams:

ios::operator `void* () const`

(Method)

You can do a quick check on the state of the most recent operation on a stream by examining a pointer to the stream itself. The pointer is arbitrary except for its truth value; it is true if no failures have occurred (**ios::fail** is not true). For instance, you might ask for input on **cin** only if all prior output operations succeeded, as in the following example.

```
if (cout)
{
     // Everything OK so far
     cin >> new_value;
     ...
}
```

ios::operator `! () const`

(Method)

In case it is more convenient to check whether something has failed, the operator, `!`, returns true if **ios::fail** is true (signifying that an operation has failed).

For instance, you might issue an error message if input failed, as in the following example.

```
if (!cin)
{
     // Oops
     cerr << "Eh?\n";
}
```

`iostate` ios::rdstate `()const`

(Method)

Return the state flags for this stream. The value is from the enumeration `iostate`. You can test for any combination of the following flags.

`ios::goodbit`
There are no indications of exceptional states on this stream.

`ios::eofbit`
End of file.

`ios::failbit`
An operation has failed on this stream; this usually indicates bad format of input.

`ios::badbit`
The stream is unusable.

`void ios::setstate (iostate state)`

(Method)

Set the state flag for this stream to *`state`* in addition to any state flags already set. Synonym (for upward compatibility): `ios::set`.

See `ios::clear` to set the stream state without regard to existing state flags.

See `ios::good`, `ios::eof`, `ios::fail`, and `ios::bad`, to test the state.

`int ios::good ()const`

(Method)

Test the state flags associated with this stream; true if no error indicators are set.

`int ios::bad ()const`

(Method)

Test whether a stream is marked as unusable. (Whether `ios::badbit` is set.)

`int` ios::eof `()const`

(Method)

True if end of file was reached on this stream. (If `ios::eofbit` is set.)

`int ios::fail ()const`

(Method)

Test for any kind of failure on this stream: either some operation failed, or the stream is marked as bad. (If either `ios::failbit` or `ios::badbit` is set.)

`void ios::clear (iostate state)`

(Method)

Set the state indication for this stream to the argument *`state`*. You may call `ios::clear` with no argument, in which case the state is set to `good` (no errors pending).

See `ios::good`, `ios::eof`, `ios::fail`, and `ios::bad`, to test the state; see `ios::set` or `ios::setstate` for an alternative way of setting the state.

Choices in formatting

The following methods control (or report on) settings for some details of controlling streams, primarily to do with formatting output.

`char` ios::fill `()const`

(Method)

Returns the current *`padding`* character.

`char` ios::fill `(char` *`padding`*`)`

(Method)

Sets the padding character for fill output requirements. You can also use the manipulator, `setfill`. See“Changing stream properties using manipulators” on page 249.

Default: space.

`int` ios::precision `()const`

(Method)

Report the number of significant digits currently in use for output of floating point numbers.

Default: 6.

`int` ios::precision `(int` *`signif`*`)`

(Method)

Set the number of significant digits (for input and output numeric conversions) to *`signif`*. You can also use the manipulator `setprecision` for this purpose. See “Changing stream properties using manipulators” on page 249.

`int` ios::width `()const`

(Method)

Report the current output field width setting (the number of characters to write on the next **`<<`** output operation).

Default: `0`, which means to use as many characters as necessary.

`int` ios::width `(int` *`num`*`)`

(Method)

Set the input field width setting to *`num`*. Return the previous value for this stream.

This value resets to zero (the default) every time you use **`<<`**; it is essentially an additional implicit argument to that operator.

You can also use the manipulator `setw` for this purpose. See “Changing stream properties using manipulators” on page 249.

`fmtflags` ios::flags `()const`

(Method)

Returns the current value of the complete collection of flags controlling the format state. The following documentation describes the flags and their meanings when set.

ios::dec
ios::oct
ios::hex

Each of these flags is for a numeric base to use in converting integers from internal to display representation, or vice versa: **ios::dec**, decimal, **ios::oct**, octal, or **ios::hex**, hexadecimal, respectively. (You can change the base using the manipulator `setbase`, or any of the manipulators: **dec**, **oct**, or **hex**; see “Changing stream properties using manipulators” on page 249.)

On input, if none of these flags is set, reads numeric constants according to the prefix: decimal, (if no prefix, or a ‘`.`’ suffix), octal (if a `0` prefix is present), or hexadecimal (if a `0`*`x`* prefix is present).

Default: `dec`.

ios::fixed

Avoid scientific notation, and always show a fixed number of digits after the decimal point, according to the output precision in effect. Use **ios::precision** to set precision.

ios::left
ios::right
ios::internal

Where output is to appear in a fixed-width field: **ios::left** sets as left-justified, **ios::right** sets as right-justified, and **ios::internal** sets with padding in the middle (such as between a numeric sign and the associated value).

ios::scientific

Uses scientific (exponential) notation to display numbers.

ios::showbase

Displays the conventional prefix as a visual indicator of the conversion base: no prefix for decimal, `0` for octal, `0`*`x`* for hexadecimal.

ios::showpoint

Displays a decimal point followed by trailing zeros to fill out numeric fields, even when redundant.

ios::showpos
Displays a positive sign on display of positive numbers.

ios::skipws
Skips white space. (On by default).

ios::stdio
Flushes the C `stdio` streams, `stdout` and `stderr`, after each output operation (for programs that mix C and C++ output conventions).

ios::unitbuf
Flushes after each output operation.

ios::uppercase
Uses uppercase rather than lowercase characters in numeric displays; for instance, `0X7A` rather than `0x7a`, or `3.14E+09` rather than `3.14e+09`.

`fmtflags` ios::flags (`fmtflags` *`value`*)

(Method)

Sets a value as the complete collection of flags controlling the format state. See the descriptions for the flag values with "fmtflags ios::flags ()const" on page 247.

Use **ios::setf** or **ios::unsetf** to change one property at a time.

`fmtflags` ios::setf (`fmtflags` *`flag`*)

(Method)

Sets one particular flag (of those described for **ios::flags** `()`; returns the complete collection of flags *previously* in effect. (Use **ios::unsetf** to cancel.)

`fmtflags` ios::setf (`fmtflags` *`flag`*, `fmtflags` *`mask`*)

(Method)

Clears the flag values indicated by *`mask`*, then sets any of them that are also in *`flag`*. See the descriptions for flag values for "fmtflags ios::flags ()const" on page 247. Returns the complete collection of flags *previously* in effect. (See "fmtflags ios::unsetf (fmtflags flag)" on page 248 for another way of clearing flags.)

`fmtflags` ios::unsetf (`fmtflags` *`flag`*)

(Method)

The converse of **ios::setf**, returning the old values of those flags. Makes certain *`flag`* is not set for this stream (*`flag`* signifies a combination of flag values; see the discussions with "fmtflags ios::flags ()const" on page 247).

Changing stream properties using manipulators

For convenience, manipulators provide a way to change certain properties of streams, or otherwise affect them, in the middle of expressions involving << or >>. For example, you might use the following input statement to produce |**234| as output.

```
cout << "|" << setfill('*') << setw(5) << 234 << "|";
```

Manipulators that take an argument require `#include <iomanip.h>`.

ws

(Manipulator)

Skips whitespace.

flush

(Manipulator)

Flushes an output stream. For instance, the input, `cout<<...<<flush;`, has the same effect as the input, `cout<<...; cout.flush();`.

endl

(Manipulator)

Writes an end of line character, `\n`, then flushes the output stream.

ends

(Manipulator)

Writes the string terminator character, `\0`.

setprecision (`int` *`signif`*)

(Manipulator)

Changes the value of **ios::precision** in **<<** expressions with the manipulator, `setprecision(`*`signif`*`)` with, for instance, the use of the following input to print **`4.6`**.

Manipulators such as `setprecision(`*`signif`*`)` that take an argument require `#include <iomanip.h>`.

```
cout << setprecision(2) << 4.567;
```

setw (`int` *`n`*)

(Manipulator)

Changes the value of **ios::width** in << expressions with the manipulator, **`setw(`** *`n`***`)`**; use the following input statement, for example.

```
cout << setw(5) << 234;
```

This input prints **`234`** with two leading spaces.

Requires `#include <iomanip.h>`.

setbase (int *base***)**

(Manipulator)

Changes the base value for numeric representations, where *base* is one of `10` (decimal), `8` (octal), or `16` (hexadecimal).

Requires `#include <iomanip.h>`.

dec

(Manipulator)

Selects decimal base; equivalent to `setbase(10)`.

hex

(Manipulator)

Select hexadecimal base; equivalent to `setbase(16)`.

oct

(Manipulator)

Selects octal base; equivalent to `setbase(8)`.

setfill (char *padding***)**

(Manipulator)

Sets the *padding* character, in the same way as **ios::fill**.

Requires `#include <iomanip.h>`.

Extended data fields

A related collection of methods allows you to extend the collection of flags and parameters for many applications, without risk of conflict between them.

static fmtflags ios::bitalloc ()

(Method)

Reserves a bit (the single bit on in the result) to use as a flag. Using `bitalloc` guards against conflict between two packages that use **ios** objects for different purposes.

This method is available for upward compatibility, but is not in the ANSI working paper. The number of bits available is limited; a return value of **0** means no bit is available.

static int ios::xalloc ()

(Method)

Reserves space for a long integer or pointer parameter. The result is a unique non-negative integer. You can use it as an index to **ios::iword** or **ios::pword**. Use

xalloc to arrange for arbitrary special-purpose data in your **ios** objects, with-out risk of conflict between packages designed for different purposes.

`long&` ios::iword `(int` *`index`*`)`

(Method)

Returns a reference to arbitrary data, of long integer type, stored in an **ios** instance. *`index`*, conventionally returned from **ios::xalloc**, identifies the particular data you need.

`long` ios::iword `(int` *`index`*`) const`

(Method)

Returns the actual value of a long integer stored in an **ios**.

`void*&` ios::pword `(int` *`index`*`)`

(Method)

Returns a reference to an arbitrary pointer, stored in an **ios** instance. *`index`*, originally returned from **ios::xalloc**, identifies a particular pointer you need.

`void*` ios::pword `(int` *`index`*`)const`

(Method)

Returns the actual value of a pointer stored in an **ios**.

Synchronizing related streams

You can use the following methods to synchronize related streams so that they correspond:

`ostream*` ios::tie `() const`

(Method)

Report on what output stream, if any, is to be flushed before accessing this one. A pointer value of `0` means no stream is tied.

`ostream*` ios::tie `(ostream*` *`assoc`*`)`

(Method)

Declare that an output stream, *`assoc`*, must be flushed before accessing this stream.

`int` ios::sync_with_stdio `([int` *`switch`*`])`

(Method)

Unless iostreams and C `stdio` are designed to work together, you may have to choose between efficient C++ streams output and output which is compatible with C `stdio`. Use **ios::sync_with_stdio**`()` to select C compatibility. The argument,

`switch`, is a GNU extension; since the default value for *`switch`* is usually `1`, use `0` as the argument to choose output that is not necessarily compatible with C `stdio`.

If you install the `stdio` implementation that comes with GNU `libio`, there are compatible input/output facilities for both C and C++. In that situation, this method is unnecessary— but you may still want to write programs that call it, for portability.

Reaching the underlying `streambuf`

Finally, you can use the following method to access the underlying object:

`streambuf*` ios::rdbuf `()const`
(Method)

Return a pointer to the `streambuf` object that underlies this **ios**.

Managing output streams: class `ostream`

Objects of the `ostream` class inherit the generic methods from **ios**, and in addition have the following methods available. Declarations for this class come from `iostream.h`.

***ostream::ostream* `()`**

(Constructor)

The simplest form of the constructor for an `ostream` simply initializes a new **ios** object.

***ostream::ostream* `(streambuf*` *`sb`* `[, ostream` *`tie`*`])`**

(Constructor)

This alternative constructor requires a first argument, *`sb`*, (of type, `streambuf*`) to use an existing open stream for output. It also accepts an optional second argument, *`tie`*, to specify a related `ostream*` as the initial value for **ios::tie**.

If you use this constructor, the argument, *`sb`*, is not destroyed (or deleted or closed) when the **`ostream`** is destroyed.

Writing on an `ostream`

These methods write on an `ostream` . You may also use the operator, **<<**; see “” on page 237.

`ostream&` *ostream::put* `(char` *`c`*`)`

(Method)

Write the single character, *`c`*.

`ostream&` *ostream::write* `(`*`string`*`, int` *`length`*`)`

(Method)

Write *`length`* characters of a string to this ostream, beginning at the pointer, *`string`*. *`string`* may have any of these types: `char*`, `unsigned char*`, `signed char*`.

`ostream&` *ostream::form* `(const char*` *`format`*`, ...)`

(Method)

A GNU extension, similar to `fprintf (`*`file`*`,` *`format`*`, ...)`.

`format` is a `printf`-style format control string, which is used to format the (variable number of) arguments, printing the result on this `ostream`. See

ostream::vform for a version that uses an argument list rather than a variable number of arguments.

ostream& ostream::vform (const char *format*, **va_list** *args*)

(Method)

A GNU extension, similar to `vfprintf(`*`file`*, *`format`*, *`args`*`)`. *`format`* is a `printf`-style format control string, which is used to format the argument list, *`args`*, printing the result on this `ostream`. See **ostream::form** for a version that uses a variable number of arguments rather than an argument list.

Repositioning an `ostream`

You can control the output position (on output streams that actually support positions, typically files) with the following methods.

streampos ostream::tellp ()

(Method)

Returns the current write position in the stream.

ostream& ostream::seekp (streampos *loc*)

(Method)

Resets the output position to *`loc`* (which is usually the result of a previous call to **ostream::tellp**). *`loc`* specifies an absolute position in the output stream.

ostream& ostream::seekp (streamoff *loc*, *rel*)

(Method)

Resets the output position to *`loc`*, relative to the beginning, end, or current output position in the stream, as indicated by *`rel`* (a value from the enumeration of **ios::seekdir**):

- **beg**
  Interpret *`loc`* as an absolute offset from the beginning of the file.
- **cur**
  Interpret *`loc`* as an offset relative to the current output position.
- **end**
  Interpret *`loc`* as an offset from the current end of the output stream.

Miscellaneous `ostream` utilities

You may need to use the following `ostream` methods for housekeeping.

`ostream&` flush `()`

(Method)

Deliver any pending buffered output for this `ostream`.

`int` ostream::opfx `()`

(Method)

opfx is a prefix method for operations on `ostream` objects; it is designed to be called before any further processing. See the following method, **ostream::osfx**, for the converse of **opfx** functionality.

opfx tests that the stream is in state `good`, and if so flushes any stream tied to this one. The result is **`1`** when **opfx** succeeds; else (if the stream state is not `good`), the result is **`0`**.

`void` ostream::osfx `()`

(Method)

osfx is a suffix method for operations on `ostream` objects; it is designed to be called at the conclusion of any processing. All the `ostream` methods end by calling **osfx**. See the previous method, **ostream::opfx**, for the converse of **osfx** functionality. If the `unitbuf` flag is set for this stream, **osfx** flushes any buffered output for it.

If the `stdio` flag is set for this stream, **osfx** flushes any output buffered for the C output streams, `stdout` and `stderr`.

Managing input streams: class `istream`

Class `istream` objects are specialized for input; as for `ostream`, they are derived from **ios**, so you can use any of the general-purpose methods from that base class. Declarations for this class also come from `iostream.h`.

istream::istream ()
(Constructor)

When used without arguments, the `istream` constructor initializes the `ios` object and initializes the input counter (the value reported by **istream::gcount**) to `0`.

istream::istream (streambuf* *sb* **[, ostream** *tie*`]`**)**
(Constructor)

Calls the constructor with one or two arguments. The first argument, *sb*, is a `streambuf*`; with this pointer, the constructor uses that `streambuf` for input. The second optional argument, *tie*, specifies a related output stream as the initial value for **ios::tie**.

Using this constructor, the argument, *sb*, is *not* destroyed (or deleted or closed) when the `ostream` is destroyed.

Reading one character

Use the following methods to read a single character from the input stream.

int istream::get ()

(Method)

Reads a single character (or `EOF`) from the input stream, returning it (coerced to an `unsigned char`) as the result.

istream& istream::get (char *&c*`)`

(Method)

Reads a single character from the input stream into *&c*.

int istream::peek ()

(Method)

Returns the next available input character, but *without* changing the current input position.

Reading strings

Use the following methods to read strings (for example, a line at a time) from the input stream.

istream& istream::get (char* *c*, int *len* [, char *delim*])

(Method)

Reads a string from the input stream into the array at *c*. The remaining arguments limit how much to read: up to `len-1` characters, or up to (but not including) the first occurrence in the input of a particular delimiter character, *delim*—newline (`\n`), by default. (Naturally, if the stream reaches end of file first, that too will terminate reading.) If *delim* was present in the input, it remains available as if unread; to discard it instead, see **iostream::getline**.

get writes `\0` at the end of the string, regardless of which condition terminates the read.

istream& istream::get (streambuf& *sb* [, char *delim*])

(Method)

Reads characters from the input stream and copies them on the `streambuf` object, *sb*. Copying ends either just before the next instance of the delimiter character, *delim*—newline (`\n`), by default, or when either stream ends. If *delim* was present in the input, it remains available as if unread.

istream& istream::getline (*charptr*, int *len* [,char *delim*])

(Method)

Reads a line from the input stream, into the array at *charptr*. *charptr* may be any of three kinds of pointer: `char*`, `unsigned char*`, or `signed char*`.

The remaining arguments limit how much to read: up to (but not including) the first occurrence in the input of a line delimiter character, *delim*—newline (`\n`), by default, or up to `len-1` characters (or to end of file, if that happens sooner).

If **getline** succeeds in reading a full line, it also discards the trailing delimiter character from the input stream. (To preserve it as available input, see the similar form of **iostream::get**.) If *delim* was not found before *len* characters or end of file, **getline** sets the **ios::fail** flag, as well as the **ios::eof** flag if appropriate. **getline** writes a null character at the end of the string, regardless of which condition terminates the read.

istream& istream::read (*pointer*, int *len*)

(Method)

Read *len* bytes into the location at *pointer*, unless the input ends first.

pointer may be of type `char*`, `void*`, `unsigned char*`, or `signed char*`. If the

istream ends before reading *`len`* bytes, **read** sets the **ios::fail** flag.

`istream& istream::gets (char ** s [, char delim])`

(Method)

A GNU extension, reads an arbitrarily long string from the current input position to the next instance of the character, *`delim`*—newline (`\n`), by default.

To permit reading a string of arbitrary length, **gets** allocates whatever memory is required.

NOTE: The first argument, *`s`*, is an address to record a character pointer, rather than the pointer itself.

`istream& istream::scan (const char *format, ...)`

(Method)

A GNU extension, similar to `fscanf (file, format, ...)`. *`format`* is a `scanf`-style format control string, which is used to read the variables in the remainder of the argument list from the **istream**.

`istream& istream::vscan (const char *format, va_list args)`

(Method)

Like **istream::scan**, although only taking a single `va_list` argument.

Repositioning an `istream`

Use the following methods to control the current input position.

`streampos istream::tellg ()`

(Method)

Returns the current read position, in order to save it and return to it later with **istream::seekg**.

`istream& istream::seekg (streampos p)`

(Method)

Resets the input pointer (if the input device permits it) to *`p`*, usually the result of an earlier call to **istream::tellg**.

`istream& istream::seekg (streamoff offset, ios::seek_dir ref)`

(Method)

Resets the input pointer (if the input device permits it) to *`offset`* characters from the beginning of the input, the current position, or the end of input. Specifies how to interpret *`offset`* with one of the following values for the second argument, *`ref`*.

- Interprets *`loc`* as an absolute offset from the beginning of the file.

- Interprets *`loc`* as an offset relative to the current output position.
- Interprets *`loc`* as an offset from the current end of the output stream.

Miscellaneous `istream` utilities

Use the following methods for housekeeping on **istream** objects.

`int` istream::gcount `()`

(Method)

Reports how many characters were read from this **istream** in the last unformatted input operation.

`int` istream::ipfx `(int` *`keepwhite`*`)`

(Method)

Ensures that the **istream** object is ready for reading; checks for errors and end of file and flushes any tied stream. **ipfx** skips whitespace if you specify `0` as the *`keepwhite`* argument, *and* if **ios::skipws** is set for this stream.

To avoid skipping whitespace (regardless of the `skipws` setting on the stream), use `1` as the argument.

Call **istream::ipfx** to simplify writing non-standardized methods for reading **istream** objects.

`void` istream::isfx `()`

(Method)

A placeholder for compliance with the draft ANSI standard; this method does nothing whatsoever.

In order to write portable standard-conforming code on **istream** objects, call **isfx** after any operation that reads from an **istream**; if **istream::ipfx** has any special effects that must be canceled when done, **istream::isfx** will cancel them.

`istream&` istream::ignore `([int` *`n`*`][,int` *`delim`*`])`

(Method)

Discards some number of characters pending input. The first optional argument, *`n`*, specifies how many characters to skip.

The second optional argument, *`delim`*, specifies a “boundary” character: ignore returns immediately if this character appears in the input.

By default, *`delim`* is `EOF`; that is, if you do not specify a second argument, only the count, *`n`*, restricts how much to ignore (while input is still available).

If you do not specify how many characters to ignore, ignore returns after discarding only one character.

`istream&` istream::putback `(char ch)`

(Method)

Attempts to back up one character, replacing the character backed-up over by *ch*. Returns `EOF` if this is not allowed. Putting back the most recently read character is always allowed. (This method corresponds to the C function, `ungetc`.)

`istream&` istream::unget `()`

(Method)

Attempts to back up one character.

Input and output together: class `iostream`

In order to use the same stream for input and output, use an object of the class, **iostream**, derived from both **istream** and **ostream**.

The constructors for **iostream** behave just like the constructors for **istream**.

iostream::iostream `()`

(Constructor)

When used without arguments, `iostream` constructs the `ios` object, and initializes the input counter (the value reported by `istream::gcount`) to `0`.

iostream::iostream `(streambuf*` *`sb`* `[,ostream*` *`tie`*`])`

(Constructor)

You can also call a constructor with one or two arguments. The first argument, *`sb`*, is a `streambuf*`; if you supply this pointer, the constructor uses that `streambuf` for input and output. You can use the optional second argument, *`tie`* (an `ostream*`) to specify a related output stream as the initial value for **ios::tie**.

As for **ostream** and **istream**, **iostream** simply uses the **ios** destructor. However, an **iostream** is not deleted by its destructor.

You can use all the **istream**, **ostream**, and **ios** methods with an **iostream** object.

4

Classes for Files and Strings

There are two very common special cases of input and output: using files, and using strings in memory. **`libio`** defines the following four specialized classes for such cases.

- **`ifstream`**
  Methods for reading files.
- **`ofstream`**
  Methods for writing files.
- **`istrstream`**
  Methods for reading strings from memory.
- **`ostrstream`**
  Methods for writing strings in memory.

The following documentation discusses in more detail the classes for files and strings.

Reading and writing files

The following methods are declared in **`fstream.h`**. You can read data from class **`ifstream`** with any operation from class **`istream`**.

There are also a few specialized facilities, as in the following methods.

ifstream::ifstream `()`
(Constructor)

Make an **`ifstream`** associated with a new file for input. (If you use this version of the constructor, you need to call **ifstream::open** before actually reading anything)

ifstream::ifstream `(int` *`fd`*`)`
(Constructor)

Make an **`ifstream`** for reading from a file that was already open, using file descriptor, *`fd`*. (This constructor is compatible with other versions of **`iostreams`** for POSIX systems, but is not part of the ANSI working paper.)

ifstream::ifstream `(const char*` *`fname`* `[, int` *`mode`* `[, int` *`prot`*`]])`
(Constructor)

Open a file, *`*fname`*, for this **`ifstream`** object.

By default, the file is opened for input (with **ios::in** as *`mode`*).

If you use this constructor, the file will be closed when the **`ifstream`** is destroyed. You can use the optional argument *`mode`* to specify how to open the file, by combining these enumerated values (with '**`|`**'—the bitwise 'or' signifier). These values are actually defined in class **ios**, so that all file-related streams may inherit them. Only some of these modes are defined in the latest draft ANSI specification; if portability is important, you may wish to avoid the others.

- **ios::in**
  Open for input. (Included in ANSI draft.)
- **ios::out**
  Open for output. (Included in ANSI draft.)
- **ios::ate**
  Set the initial input (or output) position to the end of the file.
- **ios::app**
  Seek to end of file before each write. (Included in ANSI draft.)
- **ios::trunc**
  Guarantee a fresh file; discard any contents that were previously associated with it.

- **ios::nocreate**
  Guarantee an existing file; fail if the specified file did not already exist.
- **ios::noreplace**
  Guarantee a new file; fail if the specified file al-ready existed.
- **ios::binary**
  Open as a binary file (on systems where binary and text files have different properties, which is typically how '`\n`' is mapped; included in ANSI draft).

The last optional argument *`prot`* is specific to Unix-like systems; it specifies the file protection (by default, `644`).

`void ifstream::open (const char *`*`fname`***`[, int `***`mode`***`[, int `***`prot`***`]])`**
(Method)

Open a file explicitly after the associated **`ifstream`** object already exists (for instance, after using the default constructor). The arguments, options and defaults all have the same meanings as in the fully specified **`ifstream`** constructor.

You can write data to class **`ofstream`** with any operation from class **`ostream`**. The following documentation describes a few specialized facilities

`ofstream::ofstream ()`
(Constructor)

Make an **`ofstream`** associated with a new file for output.

`ofstream::ofstream (int `*`fd`***`)`**
(Constructor)

Make an **`ofstream`** for writing to a file that was already open, using file descriptor, *`fd`*.

`ofstream::ofstream (const char * `*`fname`***`[,int `***`mode`***`[,int `***`prot`***`]])`**
(Constructor)

Open a file, *`*fname`*, for this **`ofstream`** object.

By default, the file is opened for output (with **ios::out** as mode). You can use the optional argument, *`mode`*, to specify how to open the file, just as described for **ifstream::ifstream**.

The last optional argument, *`prot`*, specifies the file protection, which is, by default, **`644`**).

`ofstream::~ofstream ()`
(Destructor)

The files associated with **`ofstream`** objects are closed when the corresponding object is destroyed.

`void` ofstream::open `(const char*` *`fname`* `[,int` *`mode`* `[,int` *`prot`*`]])`
(Method)

Open a file explicitly after the associated **`ofstream`** object already exists (for instance, after using the default constructor). The arguments, options and defaults all have the same meanings as in the fully specified **`ofstream`** constructor.

The class **`fstream`** combines the facilities of **`ifstream`** and **`ofstream`**, just as **`iostream`** combines **`istream`** and **`ostream`**.

The class **`fstreambase`** underlies both **`ifstream`** and **`ofstream`**. They both inherit this additional method:

`void` fstreambase::close `()`
(Method)

Close the file associated with this object, and set **ios::fail** in this object to mark the event.

Reading and writing in memory

The classes, **istrstream**, **ostrstream**, and **strstream**, provide some additional features for reading and writing strings in memory—both static strings, and dynamically allocated strings. The underlying class, **strstreambase**, provides some features common to all three; **strstreambuf** underlies that in turn.

istrstream::istrstream (const char *_str_ [, int _size_])
(Constructor)

Associate the new input string class, **istrstream**, with an existing static string starting at *str*, of size, *size*. If you do not specify *size*, the string is treated as a NULL terminated string.

ostrstream::ostrstream ()
(Constructor)

Create a new stream for output to a dynamically managed string, which will grow as needed.

ostrstream::ostrstream (char *_str_, int _size_ [,int _mode_])
(Constructor)

A new stream for output to a statically defined string of length *size*, starting at *str*. You may optionally specify one of the modes described for **ifstream::ifstream**; if you do not specify one, the new stream is simply open for output, with mode **ios::out**.

int ostrstream::pcount ()
(Method)

Report the current length of the string associated with this **ostrstream**.

char * ostrstream::str ()
(Method)

A pointer to the string managed by this `ostrstream`. Implies `ostrstream::freeze()`.

NOTE: If you want the string to be NULL terminated, you must do that yourself (perhaps by writing **ends** to the stream).

void ostrstream::freeze ([int _n_])
(Method)

If *n* is nonzero (the default), declare that the string associated with this **ostrstream** is not to change dynamically; while frozen, it will not be reallocated if it needs more space, and it will not be de-allocated when the **ostrstream** is destroyed. Use **freeze(1)** if you refer to the string as a pointer after creating it via **ostrstream** facilities. **freeze(0)** cancels this declaration, allowing a

dynamically allocated string to be freed when its **`ostrstream`** is destroyed. If this **`ostrstream`** is already static—that is, if it was created to manage an existing statically allocated string—**`freeze`** is unnecessary, and has no effect.

`int` ostrstream::frozen `()`
(Method)

Test whether **`freeze(1)`** is in effect for this string.

`strstreambuf *` strstreambase::rdbuf `()`
(Method)

A pointer to the underlying **`strstreambuf`**.

5

Using the `streambuf` Layer

The `istream` and `ostream` classes are meant to handle conversion between objects in your program and their textual representation.

By contrast, the underlying `streambuf` class is for transferring raw bytes between your program, and input sources or output sinks. Different `streambuf` subclasses connect to different kinds of sources and sinks.

The following documentation discusses the streambuf layer with more details.

Areas of a streambuf

streambuf buffer management is fairly sophisticated (or complicated).

The standard protocol has the following ***areas***.

- The ***put area*** contains characters waiting for output.
- The ***get area*** contains characters available for reading.

The following methods are used to manipulate these areas. These are all protected methods, which are intended to be used by virtual function in classes derived from streambuf. They are also all ANSI/ISO-standard, and the ugly names are traditional.

NOTE: If a pointer points to the 'end' of an area, it means that it points to the character *after* the area.

char * streambuf::pbase ()const

(Method)

Returns a pointer to the start of the *put area*.

char * streambuf::epptr ()const
(Method)

Returns a pointer to the end of the *put area*.

char * **streambuf::pptr**()const
(Method)

If pptr() < epptr (), the pptr() returns a pointer to the current *put* position. (In that case, the next write will overwrite *pptr(), and increment pptr().) Otherwise, there is no *put* position available (and the next character written will cause **streambuf::overflow** to be called).

void streambuf::pbump (int *N*)
(Method)

Add *N* to the current *put pointer*. No error checking is done.

void streambuf::setp (char * *P*, char * *E*)
(Method)

Sets the start of the *put area* to *P*, the end of the *put area* to *E*, and the current *put pointer* also to *P*.

char * streambuf::eback ()const
(Method)

Returns a pointer to the start of the *get area*.

`char *` streambuf::egptr `()const`
(Method)

Returns a pointer to the end of the *get area*.

`char *` streambuf::gptr`()const`
(Method)

If `gptr() < egptr ()`, then `gptr()` returns a pointer to the current *get position*. (In that case, the next read will read `*gptr()`, and possibly increment `gptr()`.) Otherwise, there is no read position available (and the next read will cause **streambuf::underflow** to be called).

`void` streambuf:gbump `(int` *`N`*`)`
(Method)

Add *`N`* to the current *get pointer*. No error checking is done.

`void` streambuf::setg `(char *` *`B`*`, char *` *`P`*`, char *` *`E`*`)`
(Method)

Sets the start of the *get area* to *`B`*, the end of the *get area* to *`E`*, and the current *put pointer* to *`P`*.

Simple output re-direction by redefining `overflow`

Suppose you have a function, `write_to_window`, that writes characters to a window object. If you want to use the `ostream` function to write to it, what follows is one (portable) way to do it (remembering that this process depends on the default buffering, if any exists).

```
#include <iostream.h>
/* Returns number of characters successfully written to win.*/
extern int write_to_window (window* win, char* text, int
length);

class windowbuf : public streambuf {
      window* win;
   public:
      windowbuf (window* w) { win = w; }
      int sync ();
      int overflow (int ch);
      // Defining xsputn is an optional optimization.
      // (streamsize was recently added to ANSI C++, not
portable yet.)
      streamsize xsputn (char* text, streamsize n);
};

int windowbuf::sync ()
{ streamsize n = pptr () - pbase ();
```

```
      return (n && write_to_window (win, pbase (), n) != n) ?
EOF : 0;
}
int windowbuf::overflow (int ch)
{ streamsize n = pptr () - pbase ();
      if (n && sync ())
        return EOF;
      if (ch != EOF)
{
        char cbuf[1];
        cbuf[0] = ch;
        if (write_to_window (win, cbuf, 1) != 1)
          return EOF;
        }
      pbump (-n); // Reset pptr().
      return 0;
}
streamsize windowbuf::xsputn (char* text, streamsize n)
{ return sync () == EOF ? 0 : write_to_window (win, text, n); }

int
main (int argc, char**argv)
{
      window *win = ...;
      windowbuf wbuf(win);
      ostream wstr(&wbuf);
      wstr << "Hello world!\n";
}
```

C-style formatting for `streambuf` objects

The GNU `streambuf` class supports `printf`-like formatting and scanning.

`int` streambuf::vform `(const char *` *`format`*`, ...)`
(Method)

Similar to `fprintf (`*`file`*`, `*`format`*`, ...)`. The *`format`* is a `printf`-style format control string, which is used to format the (variable number of) arguments, printing the result on the `this` **streambuf**. The result is the number of characters printed.

`int` streambuf::vform `(const char *` *`format`*`, va_list` *`args`*`)`
(Method)

Similar to `vfprintf (`*`file`*`, `*`format`*`, `*`args`*`)`. The *`format`* is a `printf`-style format control string, which is used to format the argument list, *`args`*, printing the result on the `this` **streambuf**. The result is the number of characters printed.

int streambuf::scan (const char * *format*, ...)
(Method)

Similar to `fscanf (file, format, ...)`. The *format* is a `scanf`-style format control string, which is used to read the (variable number of) arguments from the `this` **streambuf**. The result is the number of items assigned, or `EOF` in case of input failure before any conversion.

int streambuf::vscan (const char * *format*, va_list *args*)
(Method)

Like **streambuf::scan**, but takes a single `va_list` argument.

Wrappers for C stdio

A `stdiobuf` is a **streambuf** object that points to a `FILE` object (as defined by `stdio.h`). All **streambuf** operations on the `stdiobuf` are forwarded to the `FILE`. Thus the `stdiobuf` object provides a wrapper around a `FILE`, allowing use of **streambuf** operations on a `FILE`. This can be useful when mixing C code with C++ code.

The pre-defined streams, `cin`, `cout`, and `cerr`, are normally implemented as `stdiobuf` objects that point to, respectively, `stdin`, `stdout`, and `stderr`. This is convenient, but it does cost some extra overhead.

If you set things up to use the implementation of `stdio` provided with this library, then `cin`, `cout`, and `cerr` will be set up to use `stdiobuf` objects, since you get their benefits for free. See “C Input and Output” on page 277.

Reading/writing from/to a pipe

The `procbuf` class is a GNU extension. It is derived from `streambuf`. A `procbuf` can be closed (in which case it does nothing), or open (in which case it allows communicating through a pipe with some other program).

procbuf::procbuf `()`
(*Constructor)*

Creates a `procbuf` in a *closed* state.

`procbuf *` procbuf::open `(const char *` *`command`*`, int` *`mode`*`)`
(*Method)*

Uses the shell (`/bin/sh`) to run a program specified by command.

If *`mode`* is **ios::in**, standard output from the program is sent to a pipe; you can read from the pipe by reading from the **procbuf**.

This is similar to `popen (`*`command`*`, "r")`.

If *`mode`* is **ios::out**, output written to the **procbuf** is written to a pipe; the program is set up to read its standard input from (the other end of) the pipe.

This is similar to `popen (`*`command`*`, "w")`.

The **procbuf** must start out in the closed state. Returns `* this` on success, and NULL on failure.

procbuf::procbuf `(const char *` *`command`*`, int` *`mode`*`)`
(*Constructor)*

Calls **procbuf::open** `(`*`command`*`, `*`mode`*`)`.

`procbuf *` procbuf::close `()`
(***Method)***

Waits for the program to finish executing, and then cleans up the resources used. Returns `* this` on success, and `NULL` on failure.

procbuf::~procbuf `()`
(***Destructor)***

Calls **procbuf::close**.

Backing up

The GNU iostream library allows you to ask a `streambuf` to remember the current position back up. This allows you to go back to this position later, after reading further. You can back up arbitrary amounts, even on unbuffered files or multiple buffers' worth, as long as you tell the library in advance. This unbounded backup is very useful for scanning and parsing applications. The following example shows a typical scenario.

```
// Read either "dog", "hound", or "hounddog".
// If "dog" is found, return 1.
// If "hound" is found, return 2.
// If "hounddog" is found, return 3.
// If none of these are found, return -1.
int my_scan(streambuf* sb)
{
        streammarker fence(sb);
        char buffer[20];
        // Try reading "hounddog":
        if (sb->sgetn(buffer, 8) == 8
                        && strncmp(buffer, "hounddog", 8) == 0)
                return 3;
        // No, no "hounddog": Back up to 'fence'
        sb->seekmark(fence); //
        // ... and try reading "dog":
        if (sb->sgetn(buffer, 3) == 3
                        && strncmp(buffer, "dog", 3) == 0)
                return 1;
        // No, no "dog" either: Back up to 'fence'
        sb->seekmark(fence); //
        // ... and try reading "hound":
        if (sb->sgetn(buffer, 5) == 5
                        && strncmp(buffer, "hound", 5) == 0)
                return 2;
        // No, no "hound" either: Back up and signal failure.
        sb->seekmark(fence); // Backup to 'fence'
        return -1;
}
```

streammarker::streammarker `(streambuf *` *`sbuf`*`)`

(Constructor)

Create a **streammarker** associated with *`sbuf`* that remembers the current position of the *get pointer*.

`int` streammarker::delta `(streammarker&` *`mark2`*`)`

(Method)

Return the difference between the *get positions* corresponding to `* this` and

mark2, which must point into the same **streambuffer** as `this`.

`int` streammarker::delta `()`
(Method)

Return the position relative to the **streambuffer**'s current *get position*.

`int` streambuf::seekmark `(``streammarker&` *mark*`)`
(Method)

Move the *get pointer* to where it (logically) was when *mark* was constructed.

Forwarding I/O activity

An ***indirectbuf*** is one that forwards all of its I/O requests to another streambuf.

An `indirectbuf` can be used to implement Common Lisp synonym-streams and two-way-streams, as with the following example.

```
class synonymbuf : public indirectbuf
       { Symbol *sym;
       synonymbuf(Symbol *s) { sym = s; }
       virtual streambuf *lookup_stream(int mode) {
              return coerce_to_streambuf(lookup_value(sym));
}
};
```

6

C Input and Output

`libio` is distributed with a complete implementation of the ANSI C `stdio` facility. It is implemented using `streambuf` objects. See “Wrappers for C stdio” on page 273.

The `stdio` package is intended as a replacement for whatever `stdio` is in the C library. Since `stdio` works best when you build `libc` to contain it, and that may be inconvenient, it is not installed by default.

The following extensions are beyond ANSI.

- A `stdio` `FILE` is identical to a **streambuf**. So, there is no need to worry about synchronizing C and C++ input/output—they are by definition always synchronized.
- If you create a new **streambuf** sub-class (in C++), you can use it as a `FILE` from C. Thus the system is extensible using the standard **streambuf** protocol.
- You can arbitrarily mix reading and writing, without having to seek between the two processes.
- Unbounded `ungetc()` buffer.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y