Qualcomm Technologies, Inc.

# Qualcomm® Hexagon™ C Library

## User Guide

80-N2040-13 Rev. D

March 4, 2019



Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

# Contents

# Figures

# Tables

# 1 Introduction

This document describes the Qualcomm® Hexagon™ C Library. It also provides the following library-related information:

- How to use the library, including what happens at program startup and at program termination
- How to write character constants and string literals, and how to convert between multibyte characters and wide characters
- How to read and write data between the program and files
- How to generate text under control of a format string
- How to scan and parse text under control of a format string

As much as possible, this reference indicates any extensions to standard-conforming behavior particular to this implementation.

## 1.1 Conventions

`Courier` font is used for computer text and code samples:

```
hexagon-profiler --packet_analyze --elf=<file>.
```

The following notation is used to define command syntax:

- Square brackets enclose optional items, for example, **`[label]`**.
- **Bold** indicates literal symbols for example, **`[`***`comment`***`]`**.
- The vertical bar character, `|`, indicates a choice of items.
- Parentheses enclose a choice of items for example, **`(add|del)`**.
- An ellipsis, `...`, follows items that can appear more than once.

## 1.2 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

# 2 Overview

C programs on the Hexagon processor can access the following C libraries:

- The Dinkum C99 library[1]
- Additional APIs to improve POSIX compliance

The functions defined in these libraries perform essential services such as input/output and storage allocation. They also provide efficient implementations of frequently-used operations. Numerous macro and type definitions accompany the functions to help you make better use of the library.

## 2.1 Dinkum C99 library

The Dinkum C99 library implements the traditional Standard C library headers:

- <assert.h> – For enforcing assertions when functions execute
- <complex.h> – For performing complex arithmetic
- <ctype.h> – For classifying characters
- <errno.h> – For testing error codes reported by library functions
- <fenv.h> – For controlling IEEE-style floating point arithmetic
- <float.h> – For testing floating point type properties
- <inttypes.h> – For converting various integer types
- <iohw.h> [Added with TR18015/TR18037] – For writing portable I/O hardware drivers in C
- <iso646.h> – For programming in ISO 646 variant character sets
- <limits.h> – For testing integer type properties
- <locale.h> – For adapting to different cultural conventions
- <math.h> – For computing common mathematical functions
- <setjmp.h> – For executing non-local goto statements
- <signal.h> – For controlling various exceptional conditions
- <stdarg.h> – For accessing a varying number of arguments
- <stdbool.h> – For defining a convenient Boolean type name and constants

[1] An ISO-conforming implementation of the Standard C Library, as revised in 1999, corrected through 2003, and extended by several (non-normative) Technical Reports.

- <stddef.h> – For defining several useful types and macros
- <stdint.h> – For defining various integer types with size constraints
- <stdio.h> – For performing input and output
- <stdlib.h> – For performing a variety of operations
- <string.h> – For manipulating several kinds of strings
- <tgmath.h> – For declaring various type-generic math functions
- <time.h> – For converting between various time and date formats
- <uchar.h> [Added with TR19769] – For manipulating 16-bit and 32-bit UNICODE wide characters
- <wchar.h> – For manipulating wide streams and several kinds of strings
- <wctype.h> – For classifying wide characters.

## 2.2 Posix APIs

The following APIs are provided with the C99 library for improved POSIX compliance:

- <search.h> – For manipulating search tables
- <strings.h> – For manipulating strings
- <sys/stat.h> – For file query operations
- <sys/time.h> – For defining the `timeval` structure and `gettimeofday` function
- <sys/times.h> – For defining the `tms` structure and the `times` function prototype
- <unistd.h> – For defining standard symbolic constants and types

For more information on these libraries, including library declarations and definitions, see Chapter 11.

# 3 Using the C library

All Standard C library entities are declared or defined in one or more standard headers. To make use of a library entity in a program, write an `include` directive that names the relevant standard header.

The full set of Standard C headers constitutes a hosted implementation: `<assert.h>`, `<complex.h>`, `<ctype.h>`, `<errno.h>`, `<fenv.h>`, `<float.h>`, `<inttypes.h>`, `<iso646.h>`, `<limits.h>`, `<locale.h>`, `<math.h>`, `<setjmp.h>`, `<signal.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, `<tgmath.h>`, `<time.h>`, `<wchar.h>`, and `<wctype.h>`.

The headers `<iso646.h>`, `<wchar.h>`, and `<wctype.h>` are added with Amendment 1, an addition to the C Standard published in 1995.

The headers `<complex.h>`, `<fenv.h>`, `<inttypes.h>`, `<stdbool.h>`, `<stdint.h>`, and `<tgmath.h>` are added with C99, a revision to the C Standard published in 1999.

A freestanding implementation of Standard C provides only a subset of these standard headers: `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, and `<stdint.h>`. Each freestanding implementation defines:

- How it starts the program
- What happens when the program terminates
- What library functions (if any) it provides

## 3.1 Using standard C headers

You include the contents of a standard header by naming it in an `include directive`, as in:

```
#include <stdio.h>  /* include I/O facilities */
```

You can include the standard headers in any order, a standard header more than once, or two or more standard headers that define the same macro or the same type. Do not include a standard header within a declaration. Do not define macros that have the same names as keywords before you include a standard header.

A standard header never includes another standard header. A standard header declares or defines only the entities described for it in this document.

Every function in the library is declared in a standard header. The standard header can also provide a masking macro, with the same name as the function, that masks the function declaration and achieves the same effect. The macro typically expands to an expression that executes faster than a call to the function of the same name. The macro can, however, cause confusion when you are tracing or debugging the program. So you can use a standard header in two ways to declare or define a library function. To take advantage of any macro version, include the standard header so that each apparent call to the function can be replaced by a macro expansion.

For example:

```
#include <ctype.h>
char *skip_space(char *p)
    {
    while (isspace(*p))         // can be a macro
        ++p;
    return (p);
    }
```

To ensure that the program calls the actual library function, include the standard header and remove any macro definition with an `undef` directive.

For example:

```
#include <ctype.h>
#undef isspace                  remove any macro definition
int f(char *p) {
    while (isspace(*p))         must be a function
        ++p;
```

You can use many functions in the library without including a standard header (although this practice is no longer permitted in C99 and is generally not recommended). If you do not need defined macros or types to declare and call the function, you can simply declare the function as it appears in this chapter. Again, you have two choices. You can declare the function explicitly.

For example:

```
double sin(double x);       declared in <math.h>
y = rho * sin(theta);
```

Or you can declare the function implicitly if it is a function returning int with a fixed number of arguments, as in:

```
n = atoi(str);              declared in <stdlib.h>
```

If the function has a varying number of arguments, such as `printf`, you must declare it explicitly: Either include the standard header that declares it or write an explicit declaration.

You cannot define a macro or type definition without including its standard header because each of these typically varies among implementations.

## 3.2 C library conventions

A library macro that masks a function declaration expands to an expression that evaluates each of its arguments once (and only once). Arguments that have side effects evaluate the same way whether the expression executes the macro expansion or calls the function. Macros for the functions `getc` and `putc` are explicit exceptions to this rule. Their `stream` arguments can be evaluated more than once. Avoid argument expressions that have side effects with these macros.

A library function that alters a value stored in memory assumes that the function accesses no other objects that overlap the object whose stored value it alters. You cannot depend on consistent behavior from a library function that accesses and alters the same storage via different arguments. The function `memmove` is an explicit exception to this rule. Its arguments can point at objects that overlap.

An implementation has a set of reserved names that it can use for its own purposes. All the library names described in this document are, of course, reserved for the library. Do not define macros with the same names. Do not try to supply your own definition of a library function, unless this document explicitly says you can (only in C++). An unauthorized replacement may be successful on some implementations and not on others. Names that begin with two underscores (or contain two successive underscores, in C++), such as `__STDIO`, and names that begin with an underscore followed by an upper case letter, such as `_Entry`, can be used as macro names, whether or not a translation unit explicitly includes any standard headers. Names that begin with an underscore can be defined with external linkage. Avoid writing such names in a program that you wish to keep maximally portable.

Some library functions operate on C strings, or pointers to NULL-terminated strings. You designate a C string that can be altered by an argument expression that has type *pointer to char* (or type *array of char*, which converts to *pointer to char* in an argument expression). You designate a C string that cannot be altered by an argument expression that has type *pointer to const char* (or type *const array of char*). In any case, the value of the expression is the address of the first byte in an array object. The first successive element of the array that has a NULL character stored in it marks the end of the C string.

- A filename is a string whose contents meet the requirements of the target environment for naming files.
- A multibyte string is composed of zero or more multibyte characters, followed by a NULL character.
- A wide-character string is composed of zero or more wide characters (stored in an array of `wchar_t`), followed by a NULL wide character.

If an argument to a library function has a pointer type, the value of the argument expression must be a valid address for an object of its type. This is true even if the library function has no need to access an object by using the pointer argument. An explicit exception is when the description of the library function spells out what happens when you use a NULL pointer. Some examples are:

```
strcpy(s1, 0)              is INVALID
memcpy(s1, 0, 0)           is UNSAFE
realloc(0, 50)             is the same as malloc(50)
```

## 3.3 Program startup and termination

The target environment controls the execution of the program (in contrast to the translator part of the implementation, which prepares the parts of the program for execution). The target environment passes control to the program at program startup by calling the `main` function that you define as part of the program. Program arguments are C strings that the target environment provides, such as text from the command line that you type to invoke the program. If the program does not need to access program arguments, you can define main as:

```
extern int main(void)
{ <body of main> }
```

If the program uses program arguments, you define main as:

```
extern int main(int argc, char **argv)
{ <body of main> }
```

You can omit either or both of `extern int` because these are the default storage class and type for a function definition. For program arguments:

- `argc` is a value (always greater than zero) that specifies the number of program arguments.
- `argv[0]` designates the first element of an array of C strings.
- `argv[argc]` designates the last element of the array, whose stored value is a NULL pointer.

For example, if you invoke a program by entering the following:

```
echo hello
```

A target environment can call `main` with:

- The value 2 for `argc`
- The address of an array object containing `"echo"` stored in `argv[0]`
- The address of an array object containing `"hello"` stored in `argv[1]`
- A NULL pointer stored in `argv[2]`

`argv[0]` is the name used to invoke the program. The target environment can replace this name with a NULL string(`""`). The program can alter the values stored in `argc`, in `argv`, and in the array objects whose addresses are stored in `argv`.

Before the target environment calls main, it stores the initial values you specify in all objects that have static duration. It also opens three standard streams, controlled by the text-stream objects designated by the macros:

- stdin – For standard input
- stdout – For standard output
- stderr – For standard error output

If `main` returns to its caller, the target environment calls `exit` with the value returned from `main` as the status argument to `exit`. If the return statement that the program executes has no expression, the status argument is undefined. This is the case if the program executes the implied return statement at the end of the function definition.

You can also call `exit` directly from any expression within the program. In both cases, `exit` calls all functions registered with `atexit` in reverse order of registry and then begins program termination. At program termination, the target environment closes all open files, removes any temporary files that you created by calling `tmpfile`, and then returns control to the invoker, using the status argument value to determine the termination status to report for the program.

The program can terminate abnormally by calling `abort`, for example. Each implementation defines whether it closes files, whether it removes temporary files, and what termination status it reports when a program terminates abnormally.

# 4 Characters

Characters play a central role in Standard C. You represent a C program as one or more source files. The translator reads a source file as a text stream consisting of characters that you can read when you display the stream on a terminal screen or produce hard copy with a printer. You often manipulate text when a C program executes. The program might produce a text stream that people can read, or it might read a text stream entered by someone typing at a keyboard or from a file modified using a text editor.

This document describes the characters that you use to write C source files and that you manipulate as streams when executing C programs.

## 4.1 Character sets

When you write a program, you express C source files as text lines containing characters from the source character set. When a program executes in the target environment, it uses characters from the target character set. These character sets are related, but need not have the same encoding or all the same members.

Every character set contains a distinct code value for each character in the basic C character set. A character set can also contain additional characters with other code values. For example:

- The character constant `'x'` becomes the value of the code for the character corresponding to x in the target character set.
- The string literal `"xyz"` becomes a sequence of character constants stored in successive bytes of memory, followed by a byte containing the value zero:

  ```
  {'x', 'y', 'z', '\0'}
  ```

A string literal is one way to specify a NULL-terminated string, an array of zero or more bytes followed by a byte containing the value zero.

**Table 4-1 Visible characters in C character set**

| Form | Members |
|---|---|
| letter | 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 Z | |
| 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 z | |
| digit | 0 1 2 3 4 5 6 7 8 9 |
| underscore | _ |

**Table 4-1 Visible characters in C character set**

| Form | Members |
|---|---|
| punctuation | ! " # % & ' ( ) * + , - . / |
| : ; < = > ? [ \ ] ^ { \| } ~ | |

**Table 4-2 Additional characters in C character set**

| Form | Members |
|---|---|
| space | Leave blank space |
| BEL | Signal an alert (BELI) |
| BS | Go back one position (BackSpace) |
| FF | Go to top of page (FormFeed) |
| NL | Go to start of next line (NewLine) |
| CR | Go to start of this line (Carriage Return) |
| HT | Go to next Horizontal Tab stop |
| VT | Go to next Vertical Tab stop |

The code value `zero` is reserved for the NULL character that is always in the target character set. Code values for the basic C character set are positive when stored in an object of type `char`. Code values for the digits are contiguous, with increasing value. For example, `'0' + 5 equals '5'`. Code values for any two letters are not necessarily contiguous.

### 4.1.1 Character sets and locales

An implementation can support multiple locales, each with a different character set. A locale summarizes conventions particular to a given culture, such as how to format dates or how to sort names. To change locales and, therefore, target character sets while the program is running, use the function `setlocale`. The translator encodes character constants and string literals for the `"C"` locale, which is the locale in effect at program startup.

## 4.2 Escape sequences

Within character constants and string literals, you can write a variety of escape sequences. Each escape sequence determines the code value for a single character. Use escape sequences to represent character codes:

- That you cannot otherwise write (such as `\n`)
- That can be difficult to read properly (such as `\t`)
- That might change value in different target character sets (such as `\a`)
- That must not change in value among different target environments (such as `\0`)

Figure 4-1 shows the syntax for an escape sequence.

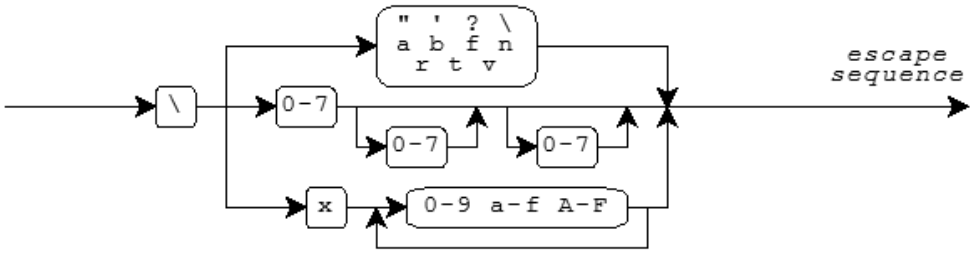


**Figure 4-1 Escape sequences**

Mnemonic escape sequences help you remember the characters they represent.

**Table 4-3 Mnemonic escape sequences**

| Character | Escape Sequence |
|---|---|
| | "\" |
| | '\' |
| ? | \? |
| \ | \\ |
| BEL | \a |
| BS | \b |
| FF | \f |
| NL | \n |
| CR | \r |
| HT | \t |
| VT | \v |

### 4.2.1 Numeric escape sequences

You can also write numeric escape sequences using either octal or hexadecimal digits. An octal escape sequence takes one of the forms: `\d` or `\dd` or `\ddd`.

The escape sequence yields a code value that is the numeric value of the 1-, 2-, or 3-digit octal number following the backslash (`\`). Each `d` can be any digit in the range 0 to 7. A hexadecimal escape sequence takes one of the forms: `\xh` or `\xhh` or `....`

The escape sequence yields a code value that is the numeric value of the arbitrary-length hexadecimal number following the backslash (`\`). Each `h` can be any decimal digit 0 to 9, or any of the letters a to f or A to F. The letters represent the digit values 10 to15, where either a or A has the value 10.

A numeric escape sequence terminates with the first character that does not fit the digit pattern. For example:

- You can write the NULL character as `'\0'`.
- You can write a newline character (`NL`) within a string literal by writing:

  ```
  "hi\n"
  ```

  Which becomes the array:

  ```
  {'h', 'i', '\n', 0}
  ```

- You can write a string literal that begins with a specific numeric value:

  ```
  "\3abc"
  ```

  Which becomes the array:

  ```
  {3, 'a', 'b', 'c', 0}
  ```

- You can write a string literal that contains the hexadecimal escape sequence `\xF` followed by the digit `3` by writing two string literals:

  ```
  "\xF" "3"
  ```

  Which becomes the array:

  ```
  {0xF, '3', 0}
  ```

# 4.3 Trigraphs

A trigraph is a sequence of three characters that begins with two question marks (??). You use trigraphs to write C source files with a character set that does not contain convenient graphic representations for some punctuation characters. (The resultant C source file is not necessarily more readable, but it is unambiguous.)

**Table 4-4 Defined trigraphs**

| Character | Trigraph |
|---|---|
| [ | ??( |
| \ | ??/ |
| ] | ??) |
| ^ | ??' |
| { | ??< |
| \| | ??! |
| } | ??> |
| ~ | ??- |
| # | ??= |

These are the only trigraphs. The translator does not alter any other sequence that begins with two question marks.

For example, the expression statements:

```
printf("Case ??=3 is done??/n");
printf("You said what????/n");
```

are equivalent to:

```
printf("Case #3 is done\n");
printf("You said what??\n");
```

The translator replaces each trigraph with its equivalent single character representation in an early phase of translation. You can always treat a trigraph as a single source character.

## 4.4 Multibyte characters

A source character set or target character set can also contain multibyte characters (sequences of one or more bytes). Each sequence represents a single character in the extended character set. You use multibyte characters to represent large sets of characters, such as Kanji. A multibyte character can be a one-byte sequence that is a character from the basic C character set, an additional one-byte sequence that is implementation defined, or an additional sequence of two or more bytes that is implementation defined.

Any multibyte encoding that contains sequences of two or more bytes depends, for its interpretation between bytes, on a conversion state determined by bytes earlier in the sequence of characters. In the initial conversion state if the byte immediately following matches one of the characters in the basic C character set, the byte must represent that character.

For example, the EUC encoding is a superset of ASCII. A byte value in the interval [0xA1, 0xFE] is the first of a two-byte sequence (whose second byte value is in the interval [0x80, 0xFF]). All other byte values are one-byte sequences. Because all members of the basic C character set have byte values in the range [0x00, 0x7F] in ASCII, EUC meets the requirements for a multibyte encoding in Standard C. Such a sequence is not in the initial conversion state immediately after a byte value in the interval [0xA1, 0xFe]. It is ill-formed if a second byte value is not in the interval [0x80, 0xFF].

Multibyte characters can also have a state-dependent encoding. How you interpret a byte in such an encoding depends on a conversion state that involves both a parse state, as before, and a shift state, determined by bytes earlier in the sequence of characters. The initial shift state, at the beginning of a new multibyte character, is also the initial conversion state. A subsequent shift sequence can determine an alternate shift state, after which all byte sequences (including one-byte sequences) can have a different interpretation. A byte containing the value zero, however, always represents the NULL character. It cannot occur as any of the bytes of another multibyte character.

For example, the JIS encoding is another superset of ASCII. In the initial shift state, each byte represents a single character, except for two three-byte shift sequences:

- The three-byte sequence `"\x1B$B"` shifts to two-byte mode. Subsequently, two successive bytes (both with values in the range [0x21, 0x7E]) constitute a single multibyte character.
- The three-byte sequence `"\x1B(B"` shifts back to the initial shift state.

JIS also meets the requirements for a multibyte encoding in Standard C. Such a sequence is not in the initial conversion state when partway through a three-byte shift sequence or when in two-byte mode.

(Amendment 1 adds the type `mbstate_t`, which describes an object that can store a conversion state. It also relaxes the above rules for generalized multibyte characters, which describe the encoding rules for a broad range of wide streams.)

You can write multibyte characters in C source text as part of a comment, a character constant, a string literal, or a filename in an `include` directive. How such characters print is implementation defined. Each sequence of multibyte characters that you write must begin and end in the initial shift state. The program can also include multibyte characters in NULL-terminated C strings used by several library functions, including the format strings for `printf` and `scanf`. Each such character string must begin and end in the initial shift state.

### 4.4.1 Wide-character encoding

Each character in the extended character set also has an integer representation, called a wide-character encoding. Each extended character has a unique wide-character value. The value zero always corresponds to the NULL wide character. The type definition `wchar_t` specifies the integer type that represents wide characters.

Write a wide-character constant as `L'mbc'`, where `mbc` represents a single multibyte character. Write a wide-character string literal `asL"mbs"`, where `mbs` represents a sequence of zero or more multibyte characters. The wide-character string literal `L"xyz"` becomes a sequence of wide-character constants stored in successive bytes of memory, followed by a NULL wide character:

```
{L'x', L'y', L'z', L'\0'}
```

The following library functions help you convert between the multibyte and wide-character representations of extended characters: `btowc`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, and `wctomb`.

The macro `MB_LEN_MAX` specifies the length of the longest possible multibyte sequence required to represent a single character defined by the implementation across supported locales. And the macro `MB_CUR_MAX` specifies the length of the longest possible multibyte sequence required to represent a single character defined for the current locale.

For example, the string literal `"hello"` becomes an array of six `char`:

```
{'h', 'e', 'l', 'l', 'o', 0}
```

While the wide-character string literal `L"hello"` becomes an array of six integers of type `wchar_t`:

```
{L'h', L'e', L'l', L'l', L'o', 0}
```

# 5 Files and streams

A program communicates with the target environment by reading and writing files (ordered sequences of bytes). A file can be, for example, a data set that you can read and write repeatedly (such as a disk file), a stream of bytes generated by a program (such as a pipeline), or a stream of bytes received from or sent to a peripheral device (such as the keyboard or display). The latter two are interactive files. Files are typically the principal means by which to interact with a program.

You manipulate all these kinds of files in much the same way—by calling library functions. You include the standard header `<stdio.h>` to declare most of these functions.

Before you can perform many of the operations on a file, the file must be opened. Opening a file associates it with a stream, a data structure within the Standard C library that glosses over many differences among files of various kinds. The library maintains the state of each stream in an object of type `FILE`.

The target environment opens three files prior to program startup. You can open a file by calling the library function `fopen` with two arguments. The first argument is a filename, a multibyte string that the target environment uses to identify which file you want to read or write. The second argument is a C string that specifies:

- Whether you intend to read data from the file or write data to it or both
- Whether you intend to generate new contents for the file (or create a file if it did not previously exist) or leave the existing contents in place
- Whether writes to a file can alter existing contents or should only append bytes at the end of the file
- Whether you want to manipulate a text stream or a binary stream

Once the file is successfully opened, you can then determine whether the stream is byte oriented (a byte stream) or wide oriented (a wide stream). Wide-oriented streams are supported only with Amendment 1. A stream is initially unbound. Calling certain functions to operate on the stream makes it byte oriented, while certain other functions make it wide oriented. Once established, a stream maintains its orientation until it is closed by a call to `fclose` or `freopen`.

## 5.1 Text and binary streams

A text stream consists of one or more lines of text that can be written to a text-oriented display so that they can be read. When reading from a text stream, the program reads an NL (newline) at the end of each line. When writing to a text stream, the program writes an NL to signal the end of a line. To match differing conventions among target environments for representing text in files, the library functions can alter the number and representations of characters transmitted between the program and a text stream.

Thus, positioning within a text stream is limited. You can obtain the current file-position indicator by calling `fgetpos` or `ftell`. You can position a text stream at a position obtained this way, or at the beginning or end of the stream, by calling `fsetpos` or `fseek`. Any other change of position might well be not supported.

For maximum portability, the program should not write:

- Empty files
- Space characters at the end of a line
- Partial lines (by omitting the `NL` at the end of a file)
- Characters other than the printable characters, `NL`, and `HT` (horizontal tab)

If you follow these rules, the sequence of characters you read from a text stream (either as byte or multibyte characters) will match the sequence of characters you wrote to the text stream when you created the file. Otherwise, the library functions can remove a file you create if the file is empty when you close it. Or they can alter or delete characters you write to the file.

A binary stream consists of one or more bytes of arbitrary information. You can write the value stored in an arbitrary object to a (byte-oriented) binary stream and read exactly what was stored in the object when you wrote it. The library functions do not alter the bytes you transmit between the program and a binary stream. They can, however, append an arbitrary number of NULL bytes to the file that you write with a binary stream. The program must deal with these additional NULL bytes at the end of any binary stream.

Thus, positioning within a binary stream is well defined, except for positioning relative to the end of the stream. You can obtain and alter the current file-position indicator the same as for a text stream. Moreover, the offsets used by `ftell` and `fseek` count bytes from the beginning of the stream (which is byte zero), so integer arithmetic on these offsets yields predictable results.

## 5.2 Byte and wide streams

A byte stream treats a file as a sequence of bytes. Within the program, the stream looks like the same sequence of bytes, except for the possible alterations described above.

By contrast, a wide stream treats a file as a sequence of generalized multibyte characters, which can have a broad range of encoding rules. (Text and binary files are still read and written as described above.) Within the program, the stream looks like the corresponding sequence of wide characters. Conversions between the two representations occur within the Standard C library. The conversion rules can, in principle, be altered by a call to `setlocale` that alters the category `LC_CTYPE`. Each wide stream determines its conversion rules at the time it becomes wide oriented, and retains these rules even if the category `LC_CTYPE` subsequently changes.

Positioning within a wide stream suffers the same limitations as for text streams. Moreover, the file-position indicator may well have to deal with a state-dependent encoding. Typically, it includes both a byte offset within the stream and an object of type `mbstate_t`. Thus, the only reliable way to obtain a file position within a wide stream is by `callingfgetpos`, and the only reliable way to restore a position obtained this way is by calling `fsetpos`.

## 5.3 Controlling streams

`fopen` returns the address of an object of type `FILE`. You use this address as the stream argument to several library functions to perform various operations on an open file. For a byte stream, all input takes place as if each character is read by calling `fgetc`, and all output takes place as if each character is written by calling `fputc`. For a wide stream (with Amendment 1), all input takes place as if each character is read by calling `fgetwc`, and all output takes place as if each character is written by calling `fputwc`.

You can close a file by calling `fclose`, after which the address of the `FILE` object is invalid.

A `FILE` object stores the state of a stream, including:

- An error indicator – Set nonzero by a function that encounters a read or write error
- An end-of-file indicator – Set nonzero by a function that encounters the end of the file while reading
- A file-position indicator – Specifies the next byte in the stream to read or write, if the file can support positioning requests
- A stream state – Specifies whether the stream will accept reads and/or writes and, with Amendment 1, whether the stream is unbound, byte oriented, or wide oriented
- A conversion state – Remembers the state of any partly assembled or generated generalized multibyte character, as well as any shift state for the sequence of bytes in the file)
- A file buffer – Specifies the address and size of an array object that library functions can use to improve the performance of read and write operations to the stream

Do not alter any value stored in a `FILE` object or in a file buffer that you specify for use with that object. You cannot copy a `FILE` object and portably use the address of the copy as a `stream` argument to a library function.

## 5.4 Stream states

Figure 5-1 shows the valid stream states and state transitions.

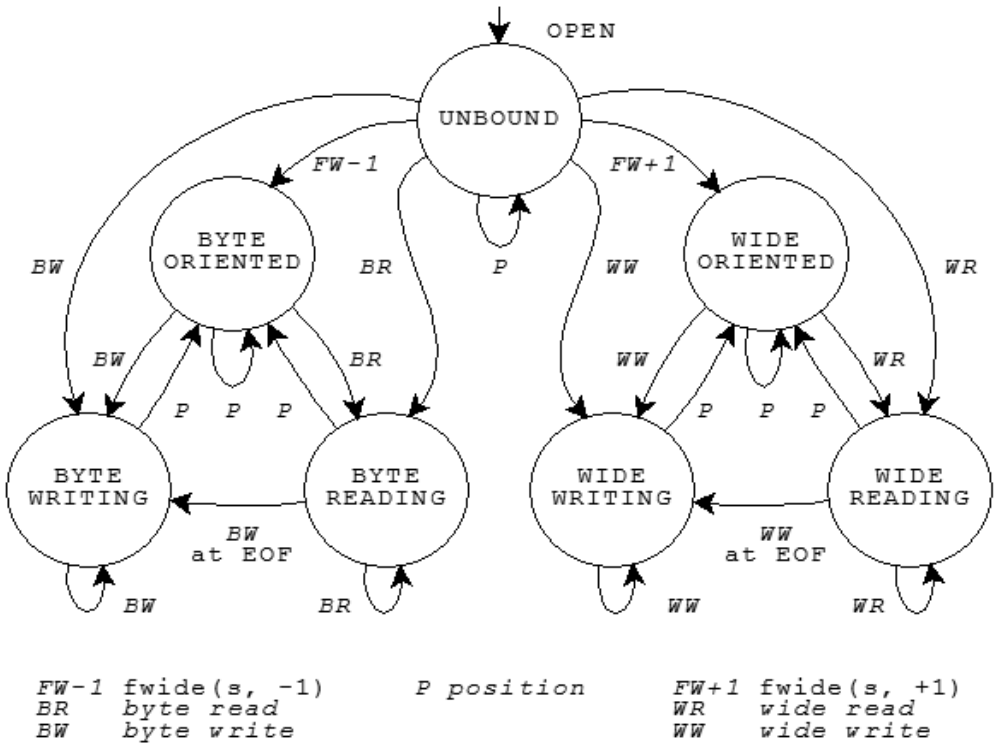


**Figure 5-1 Stream states**

Each of the circles denotes a stable state. Each of the lines denotes a transition that can occur as the result of a function call that operates on the stream. Five groups of functions can cause state transitions.

Functions in the first three groups are declared in `<stdio.h>`:

- The byte read functions – `fgetc`, `fgets`, `fread`, `fscanf`, `getc`, `getchar`, `gets`, `scanf`, `ungetc`, `vfscanf` (added with C99), and `vscanf` (added with C99)
- The byte write functions – `fprintf`, `fputc`, `fputs`, `fwrite`, `printf`, `putc`, `putchar`, `puts`, `vfprintf`, and `vprintf`
- The position functions – `fflush`, `fseek`, `fsetpos`, and `rewind`

Functions in the remaining two groups are declared in <wchar.h>:

- The wide read functions – fgetwc, fgetws, fwscanf, getwc, getwchar, ungetwc, wscanf, vfwscanf (added with C99), and vwscanf (added with C99)
- The wide write functions – fwprintf, fputwc, fputws, putwc, putwchar, vfwprintf, vwprintf, and wprintf,

For the stream s, the call fwide(s, 0) is always valid and never causes a change of state. Any other call to fwide, or to any of the five groups of functions described above, causes the state transition shown in the state diagram. If no such transition is shown, the function call is invalid.

The state diagram shows how to establish the orientation of a stream:

- The call fwide(s, -1), or to a byte read or byte write function, establishes the stream as byte oriented.
- The call fwide(s, 1), or to a wide read or wide write function, establishes the stream as wide oriented.

The state diagram shows that you must call one of the position functions between most write and read operations:

- You cannot call a read function if the last operation on the stream was a write.
- You cannot call a write function if the last operation on the stream was a read, unless that read operation set the end-of-file indicator.

Finally, the state diagram shows that a position operation never decreases the number of valid function calls that can follow.

# 6 Formatted output

Several library functions help you convert data values from encoded internal representations to text sequences that are generally readable by people. You provide a format string as the value of the format argument to each of these functions, hence the term formatted output. The functions fall into two categories.

The byte print functions (declared in `<stdio.h>`) convert internal representations to sequences of type char, and help you compose such sequences for display: `fprintf`, `printf`, `sprintf`, `vfprintf`, `vprintf`, and `vsprintf`. For these function, a format string is a multibyte string that begins and ends in the initial shift state.

The wide print functions (declared in `<wchar.h>` and hence added with Amendment 1) convert internal representations to sequences of type `wchar_t`, and help you compose such sequences for display: `fwprintf`, `swprintf`, `wprintf`, `vfwprintf`, `vswprintf`, and `vwprintf`. For these functions, a format string is a wide-character string. In the descriptions that follow, a wide character `wc` from a format string or a stream is compared to a specific (byte) character `c` as if by evaluating the expression `wctob(wc) == c`.

## 6.1 Print formats

A format string has the same syntax for both the print functions and the scan functions, as shown in Figure 6-1.

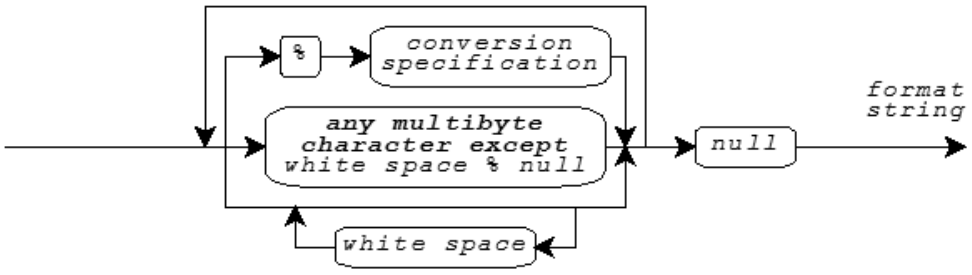


**Figure 6-1 Print format string syntax**

A format string consists of zero or more conversion specifications interspersed with literal text and white space. White space is a sequence of one or more characters c for which the call `isspace(c)` returns nonzero. (The characters defined as white space can change when you change the `LC_CTYPE` locale category.) For the print functions, a conversion specification is one of the print conversion specifications described below.

A print function scans the format string once from beginning to end to determine what conversions to perform. Every print function accepts a varying number of arguments, either directly or under control of an argument of type `va_list`. Some print conversion specifications in the format string use the next argument in the list. A print function uses each successive argument no more than once. Trailing arguments can be left unused.

In the description that follows:

- Integer conversions are the conversion specifiers that end in `d`, `i`, `o`, `u`, `x`, or `X`
- Floating point conversions are the conversion specifiers that end in `e`, `E`, `f`, `F`, `g`, or `G`

## 6.2 Print functions

For the print functions, literal text or white space in a format string generates characters that match the characters in the format string. A print conversion specification typically generates characters by converting the next argument value to a corresponding text sequence. Figure 6-2 shows the syntax for a print conversion specification.

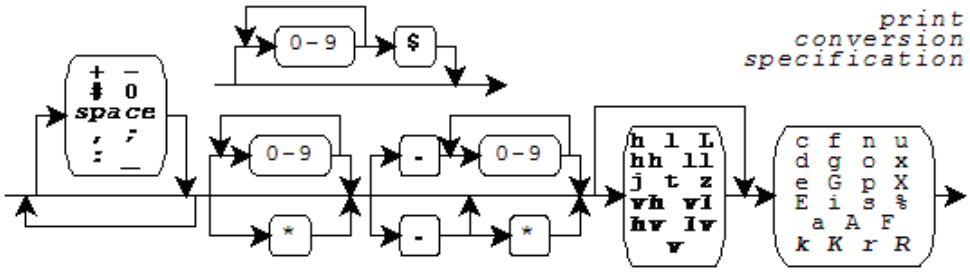


**Figure 6-2 Print conversion specification**

***Boldface Italic*** indicates a feature added after C99. Support for fixed-point conversions is added with the C Technical Report TR18037. These closely match the extensions for the Freescale Signal Processing Engine Auxiliary Processing Unit. This implementation also includes, as a conforming extension, conversions for the vectors supported by the Freescale AltiVec architecture.

Following the percent character (%) in the format string, you can write an optional position, which consists of a nonzero decimal integer followed by a dollar sign ($). If present, an argument position with value `N` indicates that at least `N` arguments to the print function follow, all having the type required for the conversion specifier. Argument number `N`, counting from 1, is the one converted; no arguments are skipped over. If no argument position is present, the next argument to the print function has the type required for the conversion specifier. It is converted and then skipped over. (This implementation supports argument positions as a conforming extension.)

Following any argument position, you can write zero or more format flags:

- `-` to left-justify a conversion
- `+` to generate a plus sign for signed values that are positive

- space to generate a space for signed values that have neither a plus nor a minus sign
- # to prefix 0 on an o conversion, to prefix 0x on an x conversion, to prefix 0X on an X conversion, or to generate a decimal point and fraction digits that are otherwise suppressed on a floating point conversion
- 0 to pad a conversion with leading zeros after any sign or prefix, in the absence of a minus (-) format flag or a specified precision

For AltiVec vector conversions, you can intersperse format flags with an optional separator, the characters comma, semicolon, equal sign, or underscore. The conversion occurs for each of the elements of the vector, and generates a separator between each pair of elements. If you specify no separator, the default is:

- For the c conversion specifier, no character.
- For all other conversion specifications, a space character.

Following any format flags, you can write a field width that specifies the minimum number of characters to generate for the conversion. Unless altered by a format flag, the default behavior is to pad a short conversion on the left with space characters. If you write an asterisk (*) instead of a decimal number for a field width, a print function takes the value of the next argument (which must be of type int) as the field width. If the argument value is negative, it supplies a - format flag and its magnitude is the field width.

Following any field width, you can write a dot (.) followed by a precision that specifies one of the following:

- The minimum number of digits to generate on an integer conversion
- The number of fraction digits to generate on an e, E, or f conversion
- The maximum number of significant digits to generate on a g or G conversion
- The maximum number of characters to generate from a C string on an s conversion

If you write an * instead of a decimal number for a precision, a print function takes the value of the next argument (which must be of type int) as the precision. If the argument value is negative, the default precision applies. If you do not write either an * or a decimal number following the dot, the precision is zero.

## 6.3 Print conversion specifiers

Following any precision, you must write a one-character print conversion specifier, possibly preceded by a one- or two-character qualifier. Each combination determines the type required of the next argument (if any) and how the library functions alter the argument value before converting it to a text sequence. The integer and floating point conversions also determine what base to use for the text representation. If a conversion specifier requires a precision `p` and you do not provide one in the format, the conversion specifier chooses a default value for the precision.

In the descriptions that follow, array declarations for arguments are not to be taken literally. A declaration of the form:

```
short x[8]
```

Is shorthand for:

```
struct {short x[8]; }
```

And a cast of the form:

```
(short x[8])x
```

Is shorthand for:

```
*(struct {short x[8]; } *)&x
```

Table 6-1 lists all defined combinations and their properties.

**Table 6-1 Defined combinations and properties**

| Conversion Specifier | Argument Type | Converted Value | Default Base | Precision | Notes |
|---|---|---|---|---|---|
| `%a` | double x | (double)x | 10 | 6 | Added with C99 |
| `%La` | long double x | (long double)x | 10 | 6 | Added with C99 |
| `%A` | double x | (double)x | 10 | 6 | Added with C99 |
| `%LA` | long double x | (long double)x | 10 | 6 | Added with C99 |
| `%c` | int x | (unsigned char)x | | | |
| `%lc` | wint_t x | (wchar_t)x | | | |
| `%vc` | char x[16] | (char [16])x | | | AltiVec |
| `%d` | int x | (int)x | 10 | 1 | |
| `%hd` | int x | (short)x | 10 | 1 | |
| `%ld` | long x | (long)x | 10 | 1 | |
| `%hhd` | int x | (signed char)x | 10 | 1 | Added with C99 |
| `%jd` | intmax_t x | (intmax_t)x | 10 | 1 | Added with C99 |
| `%lld` | long long x | (long long)x | 10 | 1 | Added with C99 |
| `%td` | ptrdiff_t x | (ptrdiff_t)x | 10 | 1 | Added with C99 |
| `%zd` | size_t x | (ptrdiff_t)x | 10 | 1 | Added with C99 |
| `%vd` | char x[16] | (signed char [16])x | | | AltiVec |

**Table 6-1 Defined combinations and properties**

| Conversion Specifier | Argument Type | Converted Value | Default Base | Precision | Notes |
|---|---|---|---|---|---|
| %hvd | short x[8] | (short [8])x | | | AltiVec |
| %vhd | short x[8] | (short [8])x | | | AltiVec |
| %lvd | int x[4] | (int [4])x | | | AltiVec |
| %vld | int x[4] | (int [4])x | | | AltiVec |
| %e | double x | (double)x | 10 | 6 | |
| %Le | long double x | (long double)x | 10 | 6 | |
| %ve | float x[4] | (float [4])x | 10 | 6 | AltiVec |
| %E | double x | (double)x | 10 | 6 | |
| %LE | long double x | (long double)x | 10 | 6 | |
| %vE | float x[4] | (float [4])x | 10 | 6 | AltiVec |
| %f | double x | (double)x | 10 | 6 | |
| %Lf | long double x | (long double)x | 10 | 6 | |
| %vf | float x[4] | (float [4])x | 10 | 6 | AltiVec |
| %F | double x | (double)x | 10 | 6 | Added with C99 |
| %LF | long double x | (long double)x | 10 | 6 | Added with C99 |
| %vF | float x[4] | (float [4])x | 10 | 6 | AltiVec |
| %g | double x | (double)x | 10 | 6 | |
| %Lg | long double x | (long double)x | 10 | 6 | |
| %vg | float x[4] | (float [4])x | 10 | 6 | AltiVec |
| %G | double x | (double)x | 10 | 6 | |
| %LG | long double x | (long double)x | 10 | 6 | |
| %vG | float x[4] | (float [4])x | 10 | 6 | AltiVec |
| %i | int x | (int)x | 10 | 1 | |
| %hi | int x | (short)x | 10 | 1 | |
| %li | long x | (long)x | 10 | 1 | |
| %hhi | int x | (signed char)x | 10 | 1 | Added with C99 |
| %ji | intmax_t x | (intmax_t)x | 10 | 1 | Added with C99 |
| %lli | long long x | (long long)x | 10 | 1 | Added with C99 |
| %ti | ptrdiff_t x | (ptrdiff_t)x | 10 | 1 | Added with C99 |
| %zi | size_t x | (ptrdiff_t)x | 10 | 1 | Added with C99 |
| %vi | char x[16] | (signed char [16])x | | | AltiVec |
| %hvi | short x[8] | (short [8])x | | | AltiVec |
| %vhi | short x[8] | (short [8])x | | | AltiVec |
| %lvi | int x[4] | (int [4])x | | | AltiVec |
| %vli | int x[4] | (int [4])x | | | AltiVec |

**Table 6-1 Defined combinations and properties**

| Conversion Specifier | Argument Type | Converted Value | Default Base | Precision | Notes |
|---|---|---|---|---|---|
| %k | accum x | (accum)x | 10 | 6 | Added with TR18037 |
| %hk | short accum x | (short accum)x | 10 | 6 | Added with TR18037 |
| %lk | long accum x | (long accum)x | 10 | 6 | Added with TR18037 |
| %K | unsigned accum x | (unsigned accum)x | 10 | 6 | Added with TR18037 |
| %hK | unsigned short accum x | (unsigned short accum)x | 10 | 6 | Added with TR18037 |
| %lK | unsigned long accum x | (unsigned long accum)x | 10 | 6 | Added with TR18037 |
| %n | int *x | | | | |
| %hn | short *x | | | | |
| %ln | long *x | | | | |
| %hhn | int *x | | | | Added with C99 |
| %jn | intmax_t *x | | | | Added with C99 |
| %lln | long long *x | | | | Added with C99 |
| %tn | ptrdiff_t *x | | | | Added with C99 |
| %zn | size_t *x | | | | Added with C99 |
| %o | int x | (unsigned int)x | 8 | 1 | |
| %ho | int x | (unsigned short)x | 8 | 1 | |
| %lo | long x | (unsigned long)x | 8 | 1 | |
| %hho | int x | (unsigned char)x | 8 | 1 | Added with C99 |
| %jo | intmax_t x | (uintmax_t)x | 8 | 1 | Added with C99 |
| %llo | long long x | (unsigned long long)x | 8 | 1 | Added with C99 |
| %to | ptrdiff_t x | (size_t)x | 8 | 1 | Added with C99 |
| %zo | size_t x | (size_t)x | 8 | 1 | Added with C99 |
| %vo | char x[16] | (unsigned char [16])x | | | AltiVec |
| %hvo | short x[8] | (unsigned short [8])x | | | AltiVec |
| %vho | short x[8] | (unsigned short [8])x | | | AltiVec |
| %lvo | int x[4] | (unsigned int [4])x | | | AltiVec |
| %vlo | int x[4] | (unsigned int [4])x | | | AltiVec |
| %p | void *x | (void *)x | | | |
| %r | fract x | (fract)x | 10 | 6 | Added with TR18037 |
| %hr | short fract x | (short fract)x | 10 | 6 | Added with TR18037 |
| %lr | long fract x | (long fract)x | 10 | 6 | Added with TR18037 |
| %R | unsigned fract x | (unsigned fract)x | 10 | 6 | Added with TR18037 |
| %hR | unsigned short fract x | (unsigned short fract)x | 10 | 6 | Added with TR18037 |
| %lR | unsigned long fract x | (unsigned long fract)x | 10 | 6 | Added with TR18037 |
| %s | const char *x | (const char *)x | | large | |

**Table 6-1 Defined combinations and properties**

| Conversion Specifier | Argument Type | Converted Value | Default Base | Precision | Notes |
|---|---|---|---|---|---|
| %ls | const wchar_t *x | (const wchar_t *)x | | large | |
| %u | int x | (unsigned int)x | 10 | 1 | |
| %hu | int x | (unsigned short)x | 10 | 1 | |
| %lu | long x | (unsigned long)x | 10 | 1 | |
| %hhu | int x | (unsigned char)x | 10 | 1 | Added with C99 |
| %ju | intmax_t x | (uintmax_t)x | 10 | 1 | Added with C99 |
| %llu | long long x | (unsigned long long)x | 10 | 1 | Added with C99 |
| %tu | ptrdiff_t x | (size_t)x | 10 | 1 | Added with C99 |
| %zu | size_t x | (size_t)x | 10 | 1 | Added with C99 |
| %vu | char x[16] | (unsigned char [16])x | | | AltiVec |
| %hvu | short x[8] | (unsigned short [8])x | | | AltiVec |
| %vhu | short x[8] | (unsigned short [8])x | | | AltiVec |
| %lvu | int x[4] | (unsigned int [4])x | | | AltiVec |
| %vlu | int x[4] | (unsigned int [4])x | | | AltiVec |
| %x | int x | (unsigned int)x | 16 | 1 | |
| %hx | int x | (unsigned short)x | 16 | 1 | |
| %lx | long x | (unsigned long)x | 16 | 1 | |
| %hhx | int x | (unsigned char)x | 16 | 1 | Added with C99 |
| %jx | intmax_t x | (uintmax_t)x | 16 | 1 | Added with C99 |
| %llx | long long x | (unsigned long long)x | 16 | 1 | Added with C99 |
| %tx | ptrdiff_t x | (size_t)x | 16 | 1 | Added with C99 |
| %zx | size_t x | (size_t)x | 16 | 1 | Added with C99 |
| %vx | char x[16] | (unsigned char [16])x | | | AltiVec |
| %hvx | short x[8] | (unsigned short [8])x | | | AltiVec |
| %vhx | short x[8] | (unsigned short [8])x | | | AltiVec |
| %lvx | int x[4] | (unsigned int [4])x | | | AltiVec |
| %vlx | int x[4] | (unsigned int [4])x | | | AltiVec |
| %X | int x | (unsigned int)x | 16 | 1 | |
| %hX | int x | (unsigned short)x | 16 | 1 | |
| %lX | long x | (unsigned long)x | 16 | 1 | |
| %hhX | int x | (unsigned char)x | 16 | 1 | Added with C99 |
| %jX | intmax_t x | (uintmax_t)x | 16 | 1 | Added with C99 |
| %llX | long long x | (unsigned long long)x | 16 | 1 | Added with C99 |
| %tX | ptrdiff_t x | (size_t)x | 16 | 1 | Added with C99 |
| %zX | size_t x | (size_t)x | 16 | 1 | Added with C99 |

**Table 6-1 Defined combinations and properties**

| Conversion Specifier | Argument Type | Converted Value | Default Base | Precision | Notes |
|---|---|---|---|---|---|
| %vX | char x[16] | (unsigned char [16])x | | | AltiVec |
| %hvX | short x[8] | (unsigned short [8])x | | | AltiVec |
| %vhX | short x[8] | (unsigned short [8])x | | | AltiVec |
| %lvX | int x[4] | (unsigned int [4])x | | | AltiVec |
| %vlX | int x[4] | (unsigned int [4])x | | | AltiVec |
| %% | none | (char)'% | | | ' |

The print conversion specifier determines any behavior not summarized in this table. For all floating point conversions:

- Positive infinity prints as `inf` or `INF`.
- Negative infinity prints as `-inf` or `-INF`.
- Not-a-number (NaN) prints as `nan` or `NAN`.

The upper-case version prints only for an upper-case conversion specifier, such as `%E` but not `%Lg`.

In the following descriptions, p is the precision. Examples follow each of the print conversion specifiers. A single conversion can generate up to 509 characters.

Write `%a` or `%A` to generate a signed hexadecimal fractional representation with a decimal power-of-two exponent. The generated text takes the form `±0Xh.hhhP±dd`, where:

- `±` is either a plus or minus sign, `X` is either `x` (for `%a` conversion) or `X` (for `%A` conversion)
- `h` is a hexadecimal digit
- `d` is a decimal digit
- The hexadecimal point ( . ) is the decimal point for the current locale
- `P` is either `p` (for `%a` conversion) or `P` (for `%A` conversion)

The generated text has one integer digit which is zero only for the value zero, a hexadecimal point if any fraction digits are present or if you specify the `#` format flag, at most `p` fraction digits with no trailing zeros, and at least one exponent digit with no leading zeros. The result is rounded. The value zero has a zero exponent.

```
printf("%a", 30.0)              generates, e.g. 0xfp+1
printf("%.2A", 30.0)            generates, e.g. 0XF.00P+1
```

Write `%c` to generate a single character from the converted value.

```
printf("%c", 'a')               generates a
printf("<%3c|%-3c>", 'a', 'b')  generates <  a|b  >
```

For a wide stream, conversion of the character `x` occurs as if by calling `btowc(x)`.

```
wprintf(L"%c", 'a')             generates btowc(a)
```

Write %lc to generate a single character from the converted value. Conversion of the character x occurs as if it is followed by a NULL character in an array of two elements of type wchar_t converted by the conversion specification ls.

```
printf("%lc", L'a')             generates a
wprintf(L"lc", L'a')            generates L'a'
```

Write %d, %i, %o, %u, %x, or %X to generate a possibly signed integer representation. %d or %i specifies signed decimal representation, %o unsigned octal, %u unsigned decimal, %x unsigned hexadecimal using the digits 0-9 and a-f, and %X unsigned hexadecimal using the digits 0 to 9 and A to F. The conversion generates at least p digits to represent the converted value. If p is zero, a converted value of zero generates no digits.

```
printf("%d %o %x", 31, 31, 31)  generates 31 37 1f
printf("%hu", 0xffff)           generates 65535
printf("%#X %+d", 31, 31)       generates 0X1F +31
```

Write %e or %E to generate a signed decimal fractional representation with a decimal power-of-ten exponent. The generated text takes the form ±d.dddE±dd, where ± is either a plus or minus sign, d is a decimal digit, the decimal point ( . ) is the decimal point for the current locale, and E is either e (for %e conversion) or E (for %E conversion). The generated text has one integer digit, a decimal point if p is nonzero or if you specify the #format flag, p fraction digits, and at least two exponent digits. The result is rounded. The value zero has a zero exponent.

```
printf("%e", 31.4)              generates 3.140000e+01
printf("%.2E", 31.4)            generates 3.14E+01
```

Write %f %F, %k, %K, %r, or %R to generate a signed decimal fractional representation with no exponent. The generated text takes the form ±d.ddd, where ± is either a plus or minus sign, d is a decimal digit, and the decimal point ( . ) is the decimal point for the current locale. The generated text has at least one integer digit, a decimal point if p is nonzero or if you specify the # format flag, and p fraction digits. The result is rounded.

```
printf("%f", 31.4)              generates 31.400000
printf("%.0f %#.0f", 31.0, 31.0)generates 31 31.
```

Write %g or %G to generate a signed decimal fractional representation with or without a decimal power-of-ten exponent, as appropriate. For %g conversion, the generated text takes the same form as either %e or %f conversion. For %G conversion, it takes the same form as either %E or %F conversion. The precision p specifies the number of significant digits generated. (If p is zero, it is changed to 1.) If %e conversion would yield an exponent in the range [-4, p), %f conversion occurs instead. The generated text has no trailing zeros in any fraction and has a decimal point only if there are nonzero fraction digits, unless you specify the # format flag.

```
printf("%.6g", 31.4)            generates 31.4
printf("%.1g", 31.4)            generates 3.14e+01
```

Write %n to store the number of characters generated (up to this point in the format) in an integer object whose address is the value of the next successive argument.

```
printf("abc%n", &x)                   stores 3
```

Write `%p` to generate an external representation of a pointer to void. The conversion is implementation defined.

```
printf("%p", (void *)&x)        generates, e.g. F4C0
```

Write `%s` to generate a sequence of characters from the values stored in the argument C string.

```
printf("%s", "hello")           generates hello
printf("%.2s", "hello")         generates he
```

For a wide stream, conversion occurs as if by repeatedly calling `mbrtowc`, beginning in the initial conversion state. The conversion generates no more than `p` characters, up to but not including the terminating NULL character.

```
wprintf(L"%s", "hello")         generates hello
```

Write `%ls` to generate a sequence of characters from the values stored in the argument wide-character string. For a byte stream, conversion occurs as if by repeatedly calling `wcrtomb`, beginning in the initial conversion state, so long as complete multibyte characters can be generated. The conversion generates no more than `p` characters, up to but not including the terminating NULL character.

```
printf("%ls", L"hello")         generates hello
wprintf(L"%.2s", L"hello")      generates he
```

Write `%%` to generate the percent character (`%`).

```
printf("%%")                    generates %
```

# 7 Formatted input

Several library functions help you convert data values from text sequences that are generally readable by people to encoded internal representations. You provide a format string as the value of the format argument to each of these functions, hence the term formatted input. The functions fall into two categories:

- The byte scan functions (declared in `<stdio.h>`) convert sequences of type char to internal representations, and help you scan such sequences that you read: `fscanf`, `scanf`, `sscanf`, `vfscanf`, `vscanf`, and `vsscanf`. For these function, a format string is a multibyte string that begins and ends in the initial shift state.
- The wide scan functions (declared in <wchar.h> and hence added with Amendment 1) convert sequences of type `wchar_t`, to internal representations, and help you scan such sequences that you read: `fwscanf`, `wscanf`, `swscanf` (added with C99), `vfwscanf` (added with C99), `vwscanf`, and `vswscanf` (added with C99). For these functions, a format string is a wide-character string. In the descriptions that follow, a wide character `wc` from a format string or a stream is compared to a specific (byte) character `c` as if by evaluating the expression `wctob(wc) == c`.

## 7.1 Scan formats

A format string has the same general syntax for the scan functions as for the print functions: zero or more conversion specifications, interspersed with literal text and white space. For the scan functions, however, a conversion specification is one of the scan conversion specifications described below.

A scan function scans the format string once from beginning to end to determine what conversions to perform. Every scan function accepts a varying number of arguments, either directly or under control of an argument of type `va_list`. Some scan conversion specifications in the format string use the next argument in the list. A scan function uses each successive argument no more than once. Trailing arguments can be left unused.

In the description that follows, the integer conversions and floating point conversions are the same as for the print functions.

## 7.2 Scan functions

For the scan functions, literal text in a format string must match the next characters to scan in the input text. White space in a format string must match the longest possible sequence of the next zero or more white-space characters in the input. Except for the scan conversion specifier `%n` (which consumes no input), each scan conversion specification determines a pattern that one or more of the next characters in the input must match. And except for the scan conversion specifiers `c`, `n`, and `[`, every match begins by skipping any white space characters in the input.

A scan function returns when:

- It reaches the terminating NULL in the format string
- It cannot obtain additional input characters to scan (input failure)
- A conversion fails (matching failure)

A scan function returns `EOF` if an input failure occurs before any conversion. Otherwise it returns the number of converted values stored. If one or more characters form a valid prefix but the conversion fails, the valid prefix is consumed before the scan function returns. Thus:

```
scanf("%i", &i)       consumes 0X from field 0XZ
scanf("%f", &f)       consumes 3.2E from field 3.2EZ
```

A scan conversion specification typically converts the matched input characters to a corresponding encoded value. The next argument value must be the address of an object. The conversion converts the encoded representation (as necessary) and stores its value in the object. A scan conversion specification has the format shown in Figure 7-1.

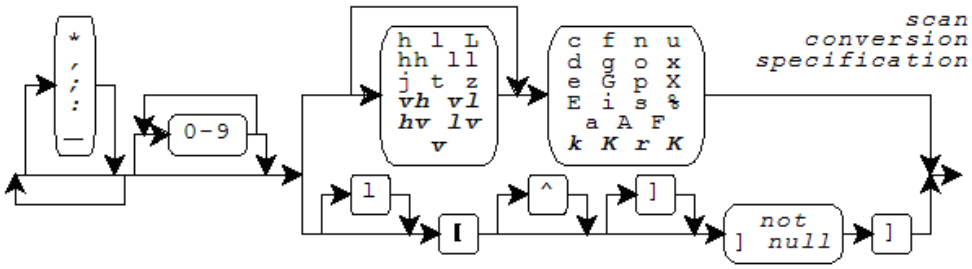


**Figure 7-1 Scan conversion specifications**

***Boldface Italic*** indicates a feature added after C99. Support for fixed-point conversions is added with the C Technical Report TR18037. These closely match the extensions for the Freescale Signal Processing Engine Auxiliary Processing Unit. This implementation also includes, as a conforming extension, conversions for the vectors supported by the Freescale AltiVec architecture.

Following the percent character (%) in the format string, you can write an asterisk (*) to indicate that the conversion should not store the converted value in an object.

For AltiVec vector conversions, you can either precede or follow an asterisk with an optional separator, the characters comma, semicolon, equal sign, or underscore. The conversion occurs for each of the elements of the vector, and matches a separator sequence between each pair of elements. If you specify a separator, the separator sequence is optional white space ending with the separator. If you specify no separator, the default separator sequence is:

- For the `c` conversion specifier, nothing.
- For all other conversion specifications, optional white space ending with a space.

Following any * or separator, you can write a nonzero field width that specifies the maximum number of input characters to match for the conversion (not counting any white space that the pattern can first skip).

## 7.3 Scan conversion specifiers

Following any field width, you must write a one-character scan conversion specifier, either a one-character code or a scan set, possibly preceded by a one- or two-character qualifier. Each combination determines the type required of the next argument (if any) and how the scan functions interpret the text sequence and converts it to an encoded value. The integer and floating point conversions also determine what base to assume for the text representation. (The base is the base argument to the functions `strtol` and `strtoul`.)

Table 7-1 lists the scan conversion specifiers and their properties.

**Table 7-1 Scan conversion specifiers**

| Conversion Specifier | Argument Type | Conversion Function | Base | Notes |
|---|---|---|---|---|
| `%a` | `float *x` | `strtof` | 10 | Added with C99 |
| `%la` | `double *x` | `strtod` | 10 | Added with C99 |
| `%La` | `long double *x` | `strtold` | 10 | Added with C99 |
| `%A` | `float *x` | `strtof` | 10 | Added with C99 |
| `%lA` | `double *x` | `strtod` | 10 | Added with C99 |
| `%LA` | `long double *x` | `strtold` | 10 | Added with C99 |
| `%c` | `char x[]` | | | |
| `%lc` | `wchar_t x[]` | | | |
| `%vc` | `char (*x)[16]` | | | AltiVec |
| `%d` | `int *x` | `strtol` | 10 | |
| `%hd` | `short *x` | `strtol` | 10 | |
| `%ld` | `long *x` | `strtol` | 10 | |
| `%hhd` | `int *x` | `strtol` | 10 | Added with C99 |
| `%jd` | `intmax_t *x` | `strtoimax` | 10 | Added with C99 |
| `%lld` | `long long *x` | `strtoll` | 10 | Added with C99 |
| `%td` | `ptrdiff_t *x` | `strtoimax` | 10 | Added with C99 |
| `%zd` | `size_t *x` | `strtoimax` | 10 | Added with C99 |

**Table 7-1 Scan conversion specifiers**

| Conversion Specifier | Argument Type | Conversion Function | Base | Notes |
|---|---|---|---|---|
| %vd | char (*x)[16] | strtol | 10 | AltiVec |
| %hvd | short (*x)[8] | strtol | 10 | AltiVec |
| %vhd | short (*x)[8] | strtol | 10 | AltiVec |
| %lvd | int (*x)[4]) | strtol | 10 | AltiVec |
| %vld | int (*x)[4]) | strtol | 10 | AltiVec |
| %e | float *x | strtof | 10 | Changed with C99 |
| %le | double *x | strtod | 10 | |
| %Le | long double *x | strtold | 10 | Changed with C99 |
| %ve | float (*x)[4] | strtof | 10 | AltiVec |
| %E | float *x | strtof | 10 | Changed with C99 |
| %lE | double *x | strtod | 10 | |
| %LE | long double *x | strtold | 10 | Changed with C99 |
| %vE | float (*x)[4] | strtof | 10 | AltiVec |
| %f | float *x | strtof | 10 | Changed with C99 |
| %lf | double *x | strtod | 10 | |
| %Lf | long double *x | strtold | 10 | Changed with C99 |
| %vf | float (*x)[4] | strtof | 10 | AltiVec |
| %F | float *x | strtof | 10 | Added with C99 |
| %lF | double *x | strtod | 10 | Added with C99 |
| %LF | long double *x | strtold | 10 | Added with C99 |
| %vF | float (*x)[4] | strtof | 10 | AltiVec |
| %g | float *x | strtof | 10 | Changed with C99 |
| %lg | double *x | strtod | 10 | |
| %Lg | long double *x | strtold | 10 | Changed with C99 |
| %vg | float (*x)[4] | strtof | 10 | AltiVec |
| %G | float *x | strtof | 10 | Changed with C99 |
| %lG | double *x | strtod | 10 | |
| %LG | long double *x | strtold | 10 | Changed with C99 |
| %vG | float (*x)[4] | strtof | 10 | AltiVec |
| %i | int *x | strtol | 0 | |
| %hi | short *x | strtol | 0 | |
| %li | long *x | strtol | 0 | |
| %hhi | int *x | strtol | 0 | Added with C99 |
| %ji | intmax_t *x | strtoimax | 0 | Added with C99 |
| %lli | long long *x | strtoll | 0 | Added with C99 |

**Table 7-1 Scan conversion specifiers**

| Conversion Specifier | Argument Type | Conversion Function | Base | Notes |
|---|---|---|---|---|
| %ti | ptrdiff_t *x | strtoimax | 0 | Added with C99 |
| %zi | size_t *x | strtoimax | 0 | Added with C99 |
| %vi | char (*x)[16] | strtol | 0 | AltiVec |
| %hvi | short (*x)[8] | strtol | 0 | AltiVec |
| %vhi | short (*x)[8] | strtol | 0 | AltiVec |
| %lvi | int (*x)[4]) | strtol | 0 | AltiVec |
| %vli | int (*x)[4]) | strtol | 0 | AltiVec |
| %n | int *x | | | |
| %hn | short *x | | | |
| %ln | long *x | | | |
| %hhn | int *x | | | Added with C99 |
| %jn | intmax_t *x | | | Added with C99 |
| %lln | long long *x | | | Added with C99 |
| %tn | ptrdiff_t *x | | | Added with C99 |
| %zn | size_t *x | | | Added with C99 |
| %o | unsigned int *x | strtoul | 8 | |
| %ho | unsigned short *x | strtoul | 8 | |
| %lo | unsigned long *x | strtoul | 8 | |
| %hho | unsigned int *x | strtoul | 8 | Added with C99 |
| %jo | uintmax_t *x | strtoumax | 8 | Added with C99 |
| %llo | unsigned long long *x | strtoull | 8 | Added with C99 |
| %to | ptrdiff_t *x | strtoumax | 8 | Added with C99 |
| %zo | size_t *x | strtoumax | 8 | Added with C99 |
| %vo | unsigned char (*x)[16] | strtol | 8 | AltiVec |
| %hvo | unsigned short (*x)[8] | strtol | 8 | AltiVec |
| %vho | unsigned short (*x)[8] | strtol | 8 | AltiVec |
| %lvo | unsigned int (*x)[4]) | strtol | 8 | AltiVec |
| %vlo | unsigned int (*x)[4]) | strtol | 8 | AltiVec |
| %p | void **x | | | |
| %s | char x[] | | | |
| %ls | wchar_t x[] | | | |
| %u | unsigned int *x | strtoul | 10 | |
| %hu | unsigned short *x | strtoul | 10 | |
| %lu | unsigned long *x | strtoul | 10 | |
| %hhu | unsigned int *x | strtoul | 10 | Added with C99 |

**Table 7-1 Scan conversion specifiers**

| Conversion Specifier | Argument Type | Conversion Function | Base | Notes |
|---|---|---|---|---|
| %ju | uintmax_t *x | strtoumax | 10 | Added with C99 |
| %llu | unsigned long long *x | strtoull | 10 | Added with C99 |
| %tu | ptrdiff_t *x | strtoumax | 10 | Added with C99 |
| %zu | size_t *x | strtoumax | 10 | Added with C99 |
| %vu | unsigned char (*x)[16] | strtol | 10 | AltiVec |
| %hvu | unsigned short (*x)[8] | strtol | 10 | AltiVec |
| %vhu | unsigned short (*x)[8] | strtol | 10 | AltiVec |
| %lvu | unsigned int (*x)[4]) | strtol | 10 | AltiVec |
| %vlu | unsigned int (*x)[4]) | strtol | 10 | AltiVec |
| %x | unsigned int *x | strtoul | 16 | |
| %hx | unsigned short *x | strtoul | 16 | |
| %lx | unsigned long *x | strtoul | 16 | |
| %hhx | unsigned int *x | strtoul | 16 | Added with C99 |
| %jx | uintmax_t *x | strtoumax | 16 | Added with C99 |
| %llx | unsigned long long *x | strtoull | 16 | Added with C99 |
| %tx | ptrdiff_t *x | strtoumax | 16 | Added with C99 |
| %zx | size_t *x | strtoumax | 16 | Added with C99 |
| %vx | unsigned char (*x)[16] | strtol | 16 | AltiVec |
| %hvx | unsigned short (*x)[8] | strtol | 16 | AltiVec |
| %vhx | unsigned short (*x)[8] | strtol | 16 | AltiVec |
| %lvx | unsigned int (*x)[4]) | strtol | 16 | AltiVec |
| %vlx | unsigned int (*x)[4]) | strtol | 16 | AltiVec |
| %X | unsigned int *x | strtoul | 16 | |
| %hX | unsigned short *x | strtoul | 16 | |
| %lX | unsigned long *x | strtoul | 16 | |
| %hhX | unsigned int *x | strtoul | 16 | Added with C99 |
| %jX | uintmax_t *x | strtoumax | 16 | Added with C99 |
| %llX | unsigned long long *x | strtoull | 16 | Added with C99 |
| %tX | ptrdiff_t *x | strtoumax | 16 | Added with C99 |
| %zX | size_t *x | strtoumax | 16 | Added with C99 |
| %vX | unsigned char (*x)[16] | strtol | 16 | AltiVec |
| %hvX | unsigned short (*x)[8] | strtol | 16 | AltiVec |
| %vhX | unsigned short (*x)[8] | strtol | 16 | AltiVec |
| %lvX | unsigned int (*x)[4]) | strtol | 16 | AltiVec |
| %vlX | unsigned int (*x)[4]) | strtol | 16 | AltiVec |

**Table 7-1 Scan conversion specifiers**

| Conversion Specifier | Argument Type | Conversion Function | Base | Notes |
|---|---|---|---|---|
| %[...] | char x[] | | | |
| %l[...] | wchar_t x[] | | | |
| %% | none | | | |

The scan conversion specifier (or scan set) determines any behavior not summarized in this table. In the following descriptions, examples follow each of the scan conversion specifiers. In each example, the function sscanf matches the bold characters.

Write %c to store the matched input characters in an array object. If you specify no field width w, then w has the value of 1. The match does not skip leading white space. Any sequence of w characters matches the conversion pattern.

```
sscanf("1129E-2", "%c", &c)             stores '1'
sscanf("1229E-2", "%2c", &c[0])         stores '1', '2'
```

For a wide stream, conversion occurs as if by repeatedly calling wcrtomb, beginning in the initial conversion state.

```
swscanf(L"1129E-2", L"%c", &c)          stores '1'
```

Write %lc to store the matched input characters in an array object, with elements of type wchar_t. If you specify no field width w, then w has the value of 1. The match does not skip leading white space. Any sequence of w characters matches the conversion pattern. For a byte stream, conversion occurs as if by repeatedly calling mbrtowc, beginning in the initial conversion state.

```
sscanf("1129E-2", "%lc", &c)            stores L'1'
sscanf("1229E-2", "%2lc", &c)           stores L'1', L'2'
swscanf(L"1129E-2", L"%lc", &c)         stores L'1'
```

Write %d, %i, %o, %u, %x, or %X to convert the matched input characters as a signed integer and store the result in an integer object.

```
sscanf("129E-2", "%o%d%x", &i, &j, &k) stores 10, 9, 14
```

Write %a, %A, %e, %E, %f, %F, %g, or %G to convert the matched input characters as a signed fraction, with an optional exponent, and store the result in a floating point object.

```
sscanf("129E-2", "%e", &f)              stores 1.29
```

Write %n to store the number of characters matched (up to this point in the format) in an integer object. The match does not skip leading white space and does not match any input characters.

```
sscanf("1229E-2", "12%n", &i)           stores 2
```

Write `%p` to convert the matched input characters as an external representation of a pointer to void and store the result in an object of type pointer to void. The input characters must match the form generated by the `%p` print conversion specification.

```
sscanf("129E-2", "%p", &p)              stores, e.g. 0x129E
```

Write `%s` to store the matched input characters in an array object, followed by a terminating NULL character. If you do not specify a field width `w`, then `w` has a large value. Any sequence of up to `w` non-white space characters matches the conversion pattern.

```
sscanf("129E-2", "%s", &s[0])           stores "129E-2"
```

For a wide stream, conversion occurs as if by repeatedly calling `wcrtomb` beginning in the initial conversion state.

```
swscanf(L"129E-2", L"%s", &s[0])        stores "129E-2"
```

Write `%ls` to store the matched input characters in an array object, with elements of type `wchar_t`, followed by a terminating NULL wide character. If you do not specify a field width `w`, then `w` has a large value. Any sequence of up to `w` non-white space characters matches the conversion pattern. For a byte stream, conversion occurs as if by repeatedly calling `mbrtowc`, beginning in the initial conversion state.

```
sscanf("129E-2", "%ls", &s[0])          stores L"129E-2"
swscanf(L"129E-2", L"%ls", &s[0])       stores L"129E-2"
```

Write `%[` to store the matched input characters in an array object, followed by a terminating NULL character. If you do not specify a field width `w`, then `w` has a large value. The match does not skip leading white space. A sequence of up to `w` characters matches the conversion pattern in the scan set that follows. To complete the scan set, you follow the left bracket (`[`) in the conversion specification with a sequence of zero or more match characters, terminated by a right bracket (`]`).

If you do not write a caret (`^`) immediately after the `[`, then each input character must match one of the match characters. Otherwise, each input character must not match any of the match characters, which begin with the character following the `^`. If you write a `]` immediately after the `[` or `[^`, then the `]` is the first match character, not the terminating `]`.

If you write a minus (`-`) as other than the first or last match character, an implementation can give it special meaning. It usually indicates a range of characters, in conjunction with the characters immediately preceding or following, as in 0 to 9 for all the digits.) You cannot specify a NULL match character.

```
sscanf("129E-2", "%[54321]", &s[0])     stores "12"
```

For a wide stream, conversion occurs as if by repeatedly calling `wcrtomb`, beginning in the initial conversion state.

```
swscanf(L"129E-2", L"%[54321]", &s[0])  stores "12"
```

Write `%l[` to store the matched input characters in an array object, with elements of type `wchar_t`, followed by a terminating NULL wide character. If you do not specify a field width `w`, then `w` has a large value. The match does not skip leading white space. A sequence of up to two characters matches the conversion pattern in the scan set that follows.

For a byte stream, conversion occurs as if by repeatedly calling `mbrtowc`, beginning in the initial conversion state.

```
sscanf("12 9E-2", "%l[54321]", &s[0])    stores L"12"
swscanf(L"12 9E-2", L"%l[54321]", &s[0]) stores L"12"
```

Write `%%` to match the percent character (%). The function does not store a value.

```
sscanf("%  0XA", "%% %i", &i)          stores 10
```

# 8 Functions

Write functions to specify all the actions that a program performs when it executes. The type of a function tells you the type of result it returns (if any). It can also tell you the types of any arguments that the function expects when you call it from within an expression.

This chapter briefly describes just those aspects of functions that are most relevant to the use of the Standard C library.

## 8.1 Argument promotion

Argument promotion occurs when the type of the function fails to provide any information about an argument. Promotion occurs if the function declaration is not a function prototype or if the argument is one of the unnamed arguments in a varying number of arguments. In this instance, the argument must be an `rvalue` expression. Hence:

- An integer argument type is promoted.
- An `lvalue` of type *array of Ty* becomes `an rvalue` of type *pointer to Ty.*
- A function designator of type *function returning Ty* becomes an `rvalue` of type *pointer* to *function returning Ty.*
- An argument of type `float` is converted to `double`.

## 8.2 Expressions

In a test-context expression the value of an expression causes control to flow one way within the statement if the computed value is nonzero or another way if the computed value is zero. You can write only an expression that has a scalar `rvalue` result, because only scalars can be compared with zero.

An expression that occurs in a side effects context specifies no value and designates no object or function. Hence, it can have type `void`. You typically evaluate such an expression for its side effects—any change in the state of the program that occurs when evaluating an expression. Side effects occur when the program stores a value in an object, accesses a value from an object of `volatile` qualified type, or alters the state of a file.

# 8.3 Statements

### do statement

A `do` statement executes a statement one or more times, while its `test-context` expression has a nonzero value:

```
do
    statement
    while (test);
```

### expression statement

An `expression` statement evaluates an expression in a side effects context:

```
printf("hello\n");            call a function
y = m * x + b;                store a value
++count;                      alter a stored value
```

### while statement

A `while` statement executes a statement zero or more times, while the `test-context` expression has a nonzero value:

```
while (test)
     statement
```

### for statement

A `for` statement executes a statement zero or more times, while the optional `test-context` expression test has a nonzero value. You can also write two expressions, `se-1` and `se-2`, in a for statement that are each in a `side-effects` context:

```
for (se-1; test; se-2)
     statement
```

### if statement

An `if` statement executes a statement only if the `test-context` expression has a nonzero value:

```
if (test)
     statement
```

### if-else statement

An `if-else` statement executes one of two statements, depending on whether the test-context expression has a nonzero value:

```
if (test)
     statement-1
else
     statement-2
```

### return statement

A `return` statement terminates execution of the function and transfers control to the expression that called the function. If you write the optional `rvalue` expression within the return statement, the result must be assignment-compatible with the type returned by the function. The program converts the value of the expression to the type returned and returns it as the value of the function call:

```
return expression;
```

### switch statement

A `switch` statement jumps to a place within a controlled statement, depending on the value of an integer expression:

```
switch (expr)
{
case val-1:
     stat-1;
     break;
case val-2:
     stat-2;        falls through to next
default:
     stat-n
}
```

# 9 Expressions

Write expressions to determine values, to alter values stored in objects, and to call functions that perform input and output. In fact, you express all computations in the program by writing expressions. The translator must evaluate some of the expressions you write to determine properties of the program. The translator or the target environment must evaluate other expressions prior to program startup to determine the initial values stored in objects with static duration. The program evaluates the remaining expressions when it executes.

This chapter describes briefly just those aspect of expressions most relevant to the use of the Standard C library.

## 9.1 Expression types

### address constant expression

An address constant expression specifies a value that has a pointer type and that the translator or target environment can determine prior to program startup.

### constant expression

A constant expression specifies a value that the translator or target environment can determine prior to program startup.

### integer constant expression

An integer constant expression specifies a value that has an integer type and that the translator can determine at the point in the program where you write the expression. (You cannot write a function call, assigning operator, or comma operator except as part of the operand of a `sizeof` operator.) In addition, you must write only subexpressions that have integer type. You can, however, write a floating point constant expression as the operand of an integer type `cast` operator.

### floating point constant expression

A floating point constant expression specifies a value that has a floating point type and that the translator can determine at the point in the program where you write the expression. (You cannot write a function call, assigning operator, or comma operator except as part of the operand of a `sizeof` operator.) In addition, you must write only subexpressions that have integer or floating point type.

### lvalue expression

An `lvalue` expression designates an object that has an object type other than an array type. Hence, you can access the value stored in the object. A modifiable `lvalue` expression designates an object that has an object type other than an array type or a `const` type. Hence, you can alter the value stored in the object. You can also designate objects with an `lvalue` expression that has an array type or an incomplete type, but you can only take the address of such an expression.

### rvalue expression

An `rvalue` expression is an expression whose value can be determined only when the program executes. The term also applies to expressions which need not be determined until program execution.

### sizeof expression

Use the `sizeof` operator, as in the expression `sizeof X` to determine the size in bytes of an object whose type is the type of `X`. The translator uses the expression you write for `X` only to determine a type; it is not evaluated.

### void expression

A `void` expression has type `void`.

## 9.2 Promoting

Promoting occurs for an expression whose integer type is not one of the computational' types. Except when it is the operand of the `sizeof` operator, an integer `rvalue` expression has one of four types: `int`, `unsigned int`, `long`, or `unsigned long`.

When you write an expression in an `rvalue` context and the expression has an integer type that is not one of these types, the translator promotes its type to one of these. If all the values representable in the original type are also representable as type `int`, the promoted type is `int`. Otherwise, the promoted type is `unsigned int`.

Thus, for `signed char`, `short`, and any signed `bitfield` type, the promoted type is `int`. For each of the remaining integer types (`char`, `unsigned char`, `unsigned short`, any plain `bitfield` type, or any `unsigned bitfield` type), the effect of these rules is to favor promoting to `int` wherever possible, but to promote to `unsigned int` if necessary to preserve the original value in all possible cases.

# 10 Preprocessing

The translator processes each source file in a series of phases. Preprocessing constitutes the earliest phases, which produce a translation unit. Preprocessing treats a source file as a sequence of text lines. You can specify directives and macros that insert, delete, and alter source text.

This chapter describes briefly just those aspects of preprocessing most relevant to the use of the Standard C library.

## 10.1 Macros

The macro `__FILE__` expands to a string literal that gives the remembered filename of the current source file. You can alter the value of this macro by writing a `line` directive.

The macro `__LINE__` expands to a decimal integer constant that gives the remembered line number within the current source file. You can alter the value of this macro by writing a `line` directive.

## 10.2 Directives

The name `__func__` (added with C99) is effectively declared at the beginning of each function body as:

```
static const char __func__[] = "func_name";
```

Where `func_name` is the name of the function.

### define directive

A `define` directive defines a name as a macro. Following the directive name define, you write one of two forms:

- A name not immediately followed by a left parenthesis, followed by any sequence of preprocessing tokens—to define a macro without parameters
- A name immediately followed by a left parenthesis with no intervening white space, followed by zero or more distinct parameter names separated by commas, followed by a right parenthesis, followed by any sequence of preprocessing tokens—to define a macro with as many parameters as names that you write inside the parentheses

You can selectively skip groups of lines within source files by writing an `if` directive, or one of the other conditional directives, `ifdef` or `ifndef`. You follow the conditional directive by the first group of lines that you want to selectively skip. Zero or more `elif` directives follow this first group of lines, each followed by a group of lines that you want to selectively skip. An optional `else` directive follows all groups of lines controlled by `elif` directives, followed by the last group of lines you want to selectively skip. The last group of lines ends with an `endif` directive.

At most one group of lines is retained in the translation unit—the one immediately preceded by a directive whose `#if` expression has a nonzero value. For the following directive, this expression is `defined (X)`:

```
#ifdef X
```

And for the following directive, this expression is `!defined (X)`.

```
#ifndef X
```

A `#if` expression is a conditional expression that the preprocessor evaluates. You can write only integer constant expressions, with the following additional considerations:

- The expression defined `x`, or defined (`x`), is replaced by 1 if `x` is defined as a macro; otherwise, it is replaced by 0.
- You cannot write the `sizeof` or type `cast` operators. (The translator expands all macro names and then replaces each remaining name with 0, before it recognizes keywords.)
- The translator may be able to represent a broader range of integers than the target environment.
- The translator represents `type int` the same as `long`, and `unsigned int` the same as `unsigned long`.
- The translator can translate character constants to a set of code values different from the set for the target environment.

### include directive

An `include` directive includes the contents of a standard header or another source file in a translation unit. The contents of the specified standard header or source file replace the include directive. Following the directive name include, write one of the following:

- A standard header name between angle brackets
- A filename between double quotes
- Any other form that expands to one of the two previous forms after macro replacement

### line directive

A `line` directive alters the source line number and filename used by the predefined macros `__LINE__` and `__FILE__`. Following the directive name `line`, write one of the following:

- A decimal integer (giving the new line number of the line following)
- A decimal integer as before, followed by a string literal (giving the new line number and the new source filename)
- Any other form that expands to one of the two previous forms after macro replacement

### undef directive

An `undef` directive removes a macro definition. You might want to remove a macro definition so that you can define it differently with a `define` directive or to unmask any other meaning given to the name. The name whose definition you want to remove follows the directive name `undef`. If the name is not currently defined as a macro, the `undef` directive has no effect.

## 10.3 Preprocessing phases

Preprocessing translates each source file in a series of distinct phases. The first few phases of translation:

- Terminate each line with a newline character (`NL`)
- Convert trigraphs to their single-character equivalents
- Concatenate each line ending in a backslash (`\`) with the line following

Later phases process include directives, expand macros, and so on to produce a translation unit. The translator combines separate translation units, with contributions as needed from the Standard C library, at link time, to form the executable program.

# 11 Header files

This chapter describes the header files in the Hexagon C Library in alphabetical order. Each header file is described, and in turn, all of its function definitions are described in alphabetical order.

## 11.1 <assert.h>

Include the standard header `<assert.h>` to define the macro assert, which is useful for diagnosing logic errors in the program. You can eliminate the testing code produced by the macro assert without removing the macro references from the program by defining the macro `NDEBUG` in the program before you include `<assert.h>`. Each time the program includes this header, it redetermines the definition of the macro assert.

```
#undef assert
#if defined NDEBUG
#define assert(test) (void)0
#else
#define assert(test) <void expression>
#endif
```

### 11.1.1 assert

```
#undef assert
#if defined NDEBUG
#define assert(test) (void)0
#else
#define assert(test) <void expression>
#endif
```

If the `int` expression test equals zero, the macro writes to `stderr` a diagnostic message that includes:

- The text of test
- The source filename (the predefined macro `__FILE__`)
- The source line number (the predefined macro `__LINE__`)
- The function name (the predefined object `__func__`, added with C99)

It then calls `abort`.

You can write the macro `assert` in the program in any side-effects context.

## 11.2 <complex.h>

[Added with C99]

Include the standard header `<complex.h>` to define several macros and a host of functions for use with the three complex arithmetic types, `float _Complex`, `double _Complex`, and `long double _Complex`. (If you include this header in a C++ program, these three types are effectively replaced by `complex<float>`, `complex<double>`, and `complex<long double>`, respectively.)

Unless otherwise specified, functions that can return multiple values return an imaginary part in the half-open interval `(-pi, pi]`.

The following pragma controls the behavior of complex multiply, divide, and magnitude:

```
#pragma STD CX_LIMITED_RANGE [ON|OFF|DEFAULT]
```

If the parameter is ON, the translator is permitted to use the conventional expressions without regard to possible intermediate overflow:

```
(x + I * y) * (u + I * v) =
    (x * u - y * v) + I * (y * u + x * v)
(x + I * y) / (u + I * v) =
    ((x * u + y * v) + I * (y * u - x * v))
        / (u * u + v * v)
abs(x + I * y) = sqrt(x * x + y * y)
```

The parameter `OFF` has the same effect as `DEFAULT`: it restores the original state where such latitude is not permitted. If the pragma occurs outside an external declaration, it remains in effect until overridden by another such pragma. If the pragma occurs inside an external declaration, it must precede all explicit declarations and statements within a compound statement. It remains in effect until overridden by another such pragma or until the end of the compound statement.

Many of the functions declared in this header have additional overloads in C++, which behave much like the generic functions defined in `<tgmath.h>`. The following functions have such additional overloads:

```
acos        conj        pow
acosh       cos         real
arg         cosh        sin
asin        cproj       sinh
asinh       creal       sqrt
atan        exp         tan
atanh       fabs        tanh
carg        imag
cimag       log
```

```
// MACROS
#define complex _Complex [Not in C++]
#define _Complex_I (float _Complex){0, 1}
#define imaginary _Imaginary [Optional]

#ifdef imaginary
  #define _Imaginary_I ((float _Imaginary)1)
#endif

#ifdef imaginary
  #define I _Imaginary_I
#else
  #define I _Complex_I
#endif

// FUNCTIONS double abs(double _Complex left); [C++ only]
float abs(float _Complex left); [C++ only]
long double abs(long double _Complex left); [C++ only]
double fabs(double _Complex left); [C++ only]
float fabs(float _Complex left); [C++ only]
long double fabs(long double _Complex left); [C++ only]
double cabs(double _Complex left);
float cabsf(float _Complex left);
long double cabsl(long double _Complex left);

double _Complex acos(double _Complex left); [C++ only]
float _Complex acos(float _Complex left); [C++ only]
long double _Complex acos(long double _Complex left); [C++ only]
double _Complex cacos(double _Complex left);
float _Complex cacosf(float _Complex left);
long double _Complex cacosl(long double _Complex left);

double _Complex acosh(double _Complex left); [C++ only]
float _Complex acosh(float _Complex left); [C++ only]
long double _Complex acosh(long double _Complex left); [C++ only]
double _Complex cacosh(double _Complex left);
float _Complex cacoshf(float _Complex left);
long double _Complex cacoshl(long double _Complex left);

double arg(double _Complex left); [C++ only]
float arg(float _Complex left); [C++ only]
long double arg(long double _Complex left); [C++ only]
double carg(double _Complex left);
float carg(float _Complex left); [C++ only]
long double carg(long double _Complex left); [C++ only]
float cargf(float _Complex left);
long double cargl(long double _Complex left);

double _Complex asin(double _Complex left); [C++ only]
float _Complex asin(float _Complex left); [C++ only]
long double _Complex asin(long double _Complex left); [C++ only]
double _Complex casin(double _Complex left);
float _Complex casinf(float _Complex left);
long double _Complex casinl(long double _Complex left);
```

```
double _Complex asinh(double _Complex left); [C++ only]
float _Complex asinh(float _Complex left); [C++ only]
long double _Complex asinh(long double _Complex left); [C++ only]
double _Complex casinh(double _Complex left);
float _Complex casinhf(float _Complex left);
long double _Complex casinhl(long double _Complex left);

double _Complex atan(double _Complex left); [C++ only]
float _Complex atan(float _Complex left); [C++ only]
long double _Complex atan(long double _Complex left); [C++ only]
double _Complex catan(double _Complex left);
float _Complex catanf(float _Complex left);
long double _Complex catanl(long double _Complex left);
double _Complex atanh(double _Complex left); [C++ only]
float _Complex atanh(float _Complex left); [C++ only]
long double _Complex atanh(long double _Complex left); [C++ only]
double _Complex catanh(double _Complex left);
float _Complex catanhf(float _Complex left);
long double _Complex catanhl(long double _Complex left);

double _Complex conj(double _Complex left);
float _Complex conj(float _Complex left); [C++ only]
long double _Complex conj(long double _Complex left); [C++ only]
float _Complex conjf(float _Complex left);
long double _Complex conjl(long double _Complex left);

double _Complex cos(double _Complex left); [C++ only]
float _Complex cos(float _Complex left); [C++ only]
long double _Complex cos(long double _Complex left); [C++ only]
double _Complex ccos(double _Complex left);
float _Complex ccosf(float _Complex left);
long double _Complex ccosl(long double _Complex left);

double _Complex cosh(double _Complex left); [C++ only]
float _Complex cosh(float _Complex left); [C++ only]
long double _Complex cosh(long double _Complex left); [C++ only]
double _Complex ccosh(double _Complex left);
float _Complex ccoshf(float _Complex left);
long double _Complex ccoshl(long double _Complex left);

double _Complex cproj(double _Complex left);
float _Complex cproj(float _Complex left); [C++ only]
long double _Complex cproj(long double _Complex left); [C++ only]
float _Complex cprojf(float _Complex left);
long double _Complex cprojl(long double _Complex left);

double _Complex exp(double _Complex left); [C++ only]
float _Complex exp(float _Complex left); [C++ only]
long double _Complex exp(long double _Complex left); [C++ only]
double _Complex cexp(double _Complex left);
float _Complex cexpf(float _Complex left);
long double _Complex cexpl(long double _Complex left);

double imag(double _Complex left); [C++ only]
```

```
float imag(float _Complex left); [C++ only]
long double imag(long double _Complex left); [C++ only]
double cimag(double _Complex left);
float cimag(float _Complex left); [C++ only]
long double cimag(long double _Complex left); [C++ only]
float cimagf(float _Complex left);
long double cimagl(long double _Complex left);

double _Complex log(double _Complex left); [C++ only]
float _Complex log(float _Complex left); [C++ only]
long double _Complex log(long double _Complex left); [C++ only]
double _Complex clog(double _Complex left);
float _Complex clogf(float _Complex left);
long double _Complex clogl(long double _Complex left);

double _Complex pow(double _Complex left, double _Complex right); [C++
only]
float _Complex pow(float _Complex left, float _Complex right); [C++
only]
long double _Complex pow(long double _Complex left, long double
_Complex right); [C++ only]
double _Complex cpow(double _Complex left, double _Complex right);
float _Complex cpowf(float _Complex left, float _Complex right);
long double _Complex cpowl(long double _Complex left, long double
_Complex right);

double real(double _Complex left); [C++ only]
float real(float _Complex left); [C++ only]
long double real(long double _Complex left); [C++ only]
double creal(double _Complex left);
float creal(float _Complex left); [C++ only]
long double creal(long double _Complex left); [C++ only]
float crealf(float _Complex left);
long double creall(long double _Complex left);

double _Complex sin(double _Complex left); [C++ only]
float _Complex sin(float _Complex left); [C++ only]
long double _Complex sin(long double _Complex left); [C++ only]
double _Complex csin(double _Complex left);
float _Complex csinf(float _Complex left);
long double _Complex csinl(long double _Complex left);

double _Complex sinh(double _Complex left); [C++ only]
float _Complex sinh(float _Complex left); [C++ only]
long double _Complex sinh(long double _Complex left); [C++ only]
double _Complex csinh(double _Complex left);
float _Complex csinhf(float _Complex left);
long double _Complex csinhl(long double _Complex left);

double _Complex sqrt(double _Complex left); [C++ only]
float _Complex sqrt(float _Complex left); [C++ only]
long double _Complex sqrt(long double _Complex left); [C++ only]
double _Complex csqrt(double _Complex left);
float _Complex csqrtf(float _Complex left);
long double _Complex csqrtl(long double _Complex left);
```

```
double _Complex tan(double _Complex left); [C++ only]
float _Complex tan(float _Complex left); [C++ only]
long double _Complex tan(long double _Complex left); [C++ only]
double _Complex ctan(double _Complex left);
float _Complex ctanf(float _Complex left);
long double _Complex ctanl(long double _Complex left);

double _Complex tanh(double _Complex left); [C++ only]
float _Complex tanh(float _Complex left); [C++ only]
long double _Complex tanh(long double _Complex left); [C++ only]
double _Complex ctanh(double _Complex left);
float _Complex ctanhf(float _Complex left);
long double _Complex ctanhl(long double _Complex left);
```

### 11.2.1 abs, fabs, cabs, cabsf, cabsl

```
double abs(double _Complex left); [C++ only]
float abs(float _Complex left); [C++ only]
long double abs(long double _Complex left); [C++ only]
double fabs(double _Complex left); [C++ only]
float fabs(float _Complex left); [C++ only]
long double fabs(long double _Complex left); [C++ only]
double cabs(double _Complex left);
float cabsf(float _Complex left);
long double cabsl(long double _Complex left);
```

The function returns the magnitude of `left`, `|left|`.

### 11.2.2 acos, cacos, cacosf, cacosl

```
double _Complex acos(double _Complex left); [C++ only]
float _Complex acos(float _Complex left); [C++ only]
long double _Complex acos(long double _Complex left); [C++ only]
double _Complex cacos(double _Complex left);
float _Complex cacosf(float _Complex left);
long double _Complex cacosl(long double _Complex left);
```

The function returns the arccosine of `left`.

### 11.2.3 acosh, cacosh, cacoshf, cacoshl

```
double _Complex acosh(double _Complex left); [C++ only]
float _Complex acosh(float _Complex left); [C++ only]
long double _Complex acosh(long double _Complex left); [C++ only]
double _Complex cacosh(double _Complex left);
float _Complex cacoshf(float _Complex left);
long double _Complex cacoshl(long double _Complex left);
```

The function returns the hyperbolic arccosine of `left`.

### 11.2.4 arg, carg, cargf, cargl

```
double arg(double _Complex left); [C++ only]
float arg(float _Complex left); [C++ only]
long double arg(long double _Complex left); [C++ only]
double carg(double _Complex left);
float carg(float _Complex left); [C++ only]
long double carg(long double _Complex left); [C++ only]
float cargf(float _Complex left);
long double cargl(long double _Complex left);
```

The function returns the phase angle of `left`.

### 11.2.5 asin, casin, casinf, casinl

```
double _Complex asin(double _Complex left); [C++ only]
float _Complex asin(float _Complex left); [C++ only]
long double _Complex asin(long double _Complex left); [C++ only]
double _Complex casin(double _Complex left);
float _Complex casinf(float _Complex left);
long double _Complex casinl(long double _Complex left);
```

The function returns the arcsine of `left`.

### 11.2.6 asinh, casinh, casinhf, casinhl

```
double _Complex asinh(double _Complex left); [C++ only]
float _Complex asinh(float _Complex left); [C++ only]
long double _Complex asinh(long double _Complex left); [C++ only]
double _Complex casinh(double _Complex left);
float _Complex casinhf(float _Complex left);
long double _Complex casinhl(long double _Complex left);
```

The function returns the hyperbolic arcsine of `left`.

### 11.2.7 atan, catan, catanf, catanl

```
double _Complex atan(double _Complex left); [C++ only]
float _Complex atan(float _Complex left); [C++ only]
long double _Complex atan(long double _Complex left); [C++ only]
double _Complex catan(double _Complex left);
float _Complex catanf(float _Complex left);
long double _Complex catanl(long double _Complex left);
```

The function returns the arctangent of `left`.

### 11.2.8 atanh, catanh, catanhf, catanhl

```
double _Complex atanh(double _Complex left); [C++ only]
float _Complex atanh(float _Complex left); [C++ only]
long double _Complex atanh(long double _Complex left); [C++ only]
double _Complex catanh(double _Complex left);
float _Complex catanhf(float _Complex left);
long double _Complex catanhl(long double _Complex left);
```

The function returns the hyperbolic arctangent of `left`.

### 11.2.9 complex

```
#define complex _Complex [Not in C++]
```

The macro expands to the keyword `_Complex`.

### 11.2.10 _Complex_I

```
#define _Complex_I (float _Complex){0, 1}
```

The macro expands to an expression of type `const float _Complex` whose real component is zero and whose imaginary component is one.

### 11.2.11 conj, conjf, conjl

```
double _Complex conj(double _Complex left); [C++ only]
float _Complex conj(float _Complex left); [C++ only]
long double _Complex conj(long double _Complex left); [C++ only]
float _Complex conjf(float _Complex left);
long double _Complex conjl(long double _Complex left);
```

The function returns the conjugate of `left`.

### 11.2.12 cos, ccos, ccosf, ccosl

```
double _Complex cos(double _Complex left); [C++ only]
float _Complex cos(float _Complex left); [C++ only]
long double _Complex cos(long double _Complex left); [C++ only]
double _Complex ccos(double _Complex left);
float _Complex ccosf(float _Complex left);
long double _Complex ccosl(long double _Complex left);
```

The function returns the cosine of `left`.

### 11.2.13 cosh, ccosh, ccoshf, ccoshl

```
double _Complex cosh(double _Complex left); [C++ only]
float _Complex cosh(float _Complex left); [C++ only]
long double _Complex cosh(long double _Complex left); [C++ only]
double _Complex ccosh(double _Complex left);
float _Complex ccoshf(float _Complex left);
long double _Complex ccoshl(long double _Complex left);
```

The function returns the hyperbolic cosine of `left`.

### 11.2.14 cproj, cprojf, cprojl

```
double _Complex cproj(double _Complex left);
float _Complex cproj(float _Complex left); [C++ only]
long double _Complex cproj(long double _Complex left); [C++ only]
float _Complex cprojf(float _Complex left);
long double _Complex cprojl(long double _Complex left);
```

The function returns a projection of `left` onto the Riemann sphere. Specifically, if either component of left is an infinity of either sign, the function returns a value whose real part is positive infinity and whose imaginary part is zero with the same sign as the imaginary part of left. Otherwise, the function returns `left`.

### 11.2.15 exp, cexp, cexpf, cexpl

```
double _Complex exp(double _Complex left); [C++ only]
float _Complex exp(float _Complex left); [C++ only]
long double _Complex exp(long double _Complex left); [C++ only]
double _Complex cexp(double _Complex left);
float _Complex cexpf(float _Complex left);
long double _Complex cexpl(long double _Complex left);
```

The function returns the exponential of `left`.

### 11.2.16 I

```
#ifdef imaginary
  #define I _Imaginary_I
#else
  #define I _Complex_I
#endif
```

The macro expands to `_Imaginary_I` if `imaginary` is defined; otherwise it expands to `_Complex_I`.

## 11.2.17 imag, cimag, cimagf, cimagl

```
double imag(double _Complex left); [C++ only]
float imag(float _Complex left); [C++ only]
long double imag(long double _Complex left); [C++ only]
double cimag(double _Complex left);
float cimag(float _Complex left); [C++ only]
long double cimag(long double _Complex left); [C++ only]
float cimagf(float _Complex left);
long double cimagl(long double _Complex left);
```

The function returns the imaginary part of `left`.

## 11.2.18 imaginary

```
#define imaginary _Imaginary [Optional]
```

The macro expands to the optional keyword `_Imaginary`, if that keyword is defined by the implementation.

## 11.2.19 _Imaginary_I

```
#ifdef imaginary  #define _Imaginary_I ((float _Imaginary)1) #endif
```

The macro expands to an expression of type `const float _Imaginary)` with value one, but only if `imaginary` is defined.

## 11.2.20 log, clog, clogf, clogl

```
double _Complex log(double _Complex left); [C++ only]
float _Complex log(float _Complex left); [C++ only]
long double _Complex log(long double _Complex left); [C++ only]
double _Complex clog(double _Complex left);
float _Complex clogf(float _Complex left);
long double _Complex clogl(long double _Complex left);
```

The function returns the logarithm of `left`. The branch cuts are along the negative real axis.

In C++, `clog` is not defined in namespace `std`, to avoid collisions with the standard error logging stream object `clog`.

### 11.2.21 pow, cpow, cpowf, cpowl

```
double _Complex pow(double _Complex left, double _Complex right); [C++
only]
float _Complex pow(float _Complex left, float _Complex right); [C++
only]
long double _Complex pow(long double _Complex left, long double
_Complex right); [C++ only]
double _Complex cpow(double _Complex left, double _Complex right);
float _Complex cpowf(float _Complex left, float _Complex right);
long double _Complex cpowl(long double _Complex left, long double
_Complex right);
```

The function returns `left` to the power `right, left^right`. The branch cut for `left` is along the negative real axis.

### 11.2.22 real, creal, crealf, creall

```
double real(double _Complex left); [C++ only]
float real(float _Complex left); [C++ only]
long double real(long double _Complex left); [C++ only]
double creal(double _Complex left);
float creal(float _Complex left); [C++ only]
long double creal(long double _Complex left); [C++ only]
float crealf(float _Complex left);
long double creall(long double _Complex left);
```

The function returns the real part of `left`.

### 11.2.23 sin, csin, csinf, csinl

```
double _Complex sin(double _Complex left); [C++ only]
float _Complex sin(float _Complex left); [C++ only]
long double _Complex sin(long double _Complex left); [C++ only]
double _Complex csin(double _Complex left);
float _Complex csinf(float _Complex left);
long double _Complex csinl(long double _Complex left);
```

The function returns the sine of `left`.

### 11.2.24 sinh, csinh, csinhf, csinhl

```
double _Complex sinh(double _Complex left); [C++ only]
float _Complex sinh(float _Complex left); [C++ only]
long double _Complex sinh(long double _Complex left); [C++ only]
double _Complex csinh(double _Complex left);
float _Complex csinhf(float _Complex left);
long double _Complex csinhl(long double _Complex left);
```

The function returns the hyperbolic sine of `left`.

## 11.2.25 sqrt, csqrt, csqrtf, csqrtl

```
double _Complex sqrt(double _Complex left); [C++ only]
float _Complex sqrt(float _Complex left); [C++ only]
long double _Complex sqrt(long double _Complex left); [C++ only]
double _Complex csqrt(double _Complex left);
float _Complex csqrtf(float _Complex left);
long double _Complex csqrtl(long double _Complex left);
```

The function returns the square root of `left,left^(1/2)`, with phase angle in the half-open `interval (-pi/2, pi/2]`. The branch cuts are along the negative real axis.

## 11.2.26 tan, ctan, ctanf, ctanl

```
double _Complex tan(double _Complex left); [C++ only]
float _Complex tan(float _Complex left); [C++ only]
long double _Complex tan(long double _Complex left); [C++ only]
double _Complex ctan(double _Complex left);
float _Complex ctanf(float _Complex left);
long double _Complex ctanl(long double _Complex left);
```

The function returns the tangent of `left`.

## 11.2.27 tanh, ctanh, ctanhf, ctanhl

```
double _Complex tanh(double _Complex left); [C++ only]
float _Complex tanh(float _Complex left); [C++ only]
long double _Complex tanh(long double _Complex left); [C++ only]
double _Complex ctanh(double _Complex left);
float _Complex ctanhf(float _Complex left);
long double _Complex ctanhl(long double _Complex left);
```

The function returns the hyperbolic tangent of `left`.

# 11.3 <ctype.h>

Include the standard header `<ctype.h>` to declare several functions that are useful for classifying and mapping codes from the target character set. Every function that has a parameter of type `int` can accept the value of the macro EOF or any value representable as type unsigned char. Thus, the argument can be the value returned by any of the functions `fgetc`, `fputc`, `getc`, `getchar`, `putc`, `putchar`, `tolower`, `toupper`, and `ungetc`. You must not call these functions with other argument values.

Other library functions use these functions. The function `scanf`, for example, uses the function `isspace` to determine valid white space within an input field.

The character classification functions are strongly interrelated. Many are defined in terms of other functions. For characters in the basic C character set, Figure 11-1 shows the dependencies between these functions.

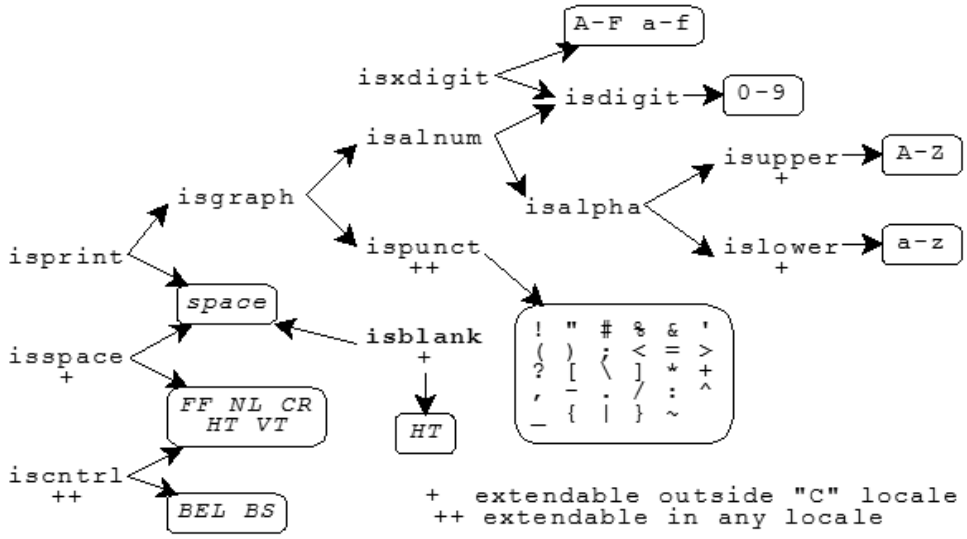


**Figure 11-1 Character classification functions**

The symbol + indicates functions that can define additional characters in locales other than the "C" locale. Boldface indicates a feature added with C99.

Figure 11-1 shows that the function `isprint` returns nonzero for space or for any character for which the function `isgraph` returns nonzero. The function `isgraph`, in turn, returns nonzero for any character for which either the function `isalnum` or the function `ispunct` returns nonzero. The function `isdigit`, on the other hand, returns nonzero only for the digits 0-9.

An implementation can define additional characters that return nonzero for some of these functions. Any character set can contain additional characters that return nonzero for:

- `iscntrl` (provided the characters cause `isprint` to return zero)
- `ispunct` (provided the characters cause `isalnum` to return zero)

The diagram indicates with ++ those functions that can define additional characters in any character set. Moreover, locales other than the “C” locale can define additional characters that return nonzero for:

- `isalpha`, `isupper`, and `islower` (provided the characters cause `iscntrl`, `isdigit`, `ispunct`, and `isspace` to return zero)
- `isblank` (provided the characters cause `isalnum` to return zero)
- `isspace` (provided the characters cause `isprint` to return zero)

An implementation can define locales other than the “C” locale in which a character can cause `isalpha` (and hence `isalnum`) to return nonzero, yet still cause `isupper` and `islower` to return zero.

```
int isalnum(int c);
int isalpha(int c);
int isascii(int c); [POSIX]
int isblank(int c); [Added with C99]
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int toascii(int c); [POSIX]
int tolower(int c);
int _tolower(int c); [POSIX]
int toupper(int c);
int _toupper(int c); [POSIX]
```

## 11.3.1 isalnum

```
int isalnum(int c);
```

The function returns nonzero if `c` is any of:

```
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 z
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 Z
0 1 2 3 4 5 6 7 8 9
```

Or any other locale-specific alphabetic character.

### 11.3.2 isalpha

```
int isalpha(int c);
```

The function returns nonzero if `c` is any of:

```
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 z
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 Z
```

Or any other locale-specific alphabetic character.

### 11.3.3 isascii

```
int isascii(int c);
```
[POSIX]

The function tests for an ASCII character, which is any character with a value in the range from 0 to 127, inclusive.

The function is defined on all integer values.

### 11.3.4 isblank

```
int isblank(int c);
```
[Added with C99]

The function returns nonzero if `c` is any of:

```
HT space
```

Or any other locale-specific blank character.

### 11.3.5 iscntrl

```
int iscntrl(int c);
```

The function returns nonzero if `c` is any of:

```
BEL BS CR FF HT NL VT
```

Or any other implementation-defined control character.

### 11.3.6 isdigit

```
int isdigit(int c);
```

The function returns nonzero if `c` is any of:

```
0 1 2 3 4 5 6 7 8 9
```

### 11.3.7 isgraph

```
int isgraph(int c);
```

The function returns nonzero if `c` is any character for which either `isalnum` or `ispunct` returns nonzero.

### 11.3.8 islower

```
int islower(int c);
```

The function returns nonzero if `c` is any of:

```
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 z
```

Or any other locale-specific lowercase character.

### 11.3.9 isprint

```
int isprint(int c);
```

The function returns nonzero if `c` is space or a character for which `isgraph` returns nonzero.

### 11.3.10 ispunct

```
int ispunct(int c);
```

The function returns nonzero if `c` is any of:

```
! " # % & ' ( ) ; < = > ? [ \ ] * + , -  . / : ^ _ { | } ~
```

Or any other implementation-defined punctuation character.

### 11.3.11 isspace

```
int isspace(int c);
```

The function returns nonzero if `c` is any of:

```
CR FF HT NL VT space
```

Or any other locale-specific space character.

### 11.3.12 isupper

```
int isupper(int c);
```

The function returns nonzero if `c` is any of:

```
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 Z
```

Or any other locale-specific uppercase character.

### 11.3.13 isxdigit

```
int isxdigit(int c);
```

The function returns nonzero if `c` is any of:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

### 11.3.14 toascii

```
int toascii(int c); [POSIX]
```

The function returns the argument with all but the lower 7 bits cleared.

The function always returns a valid ASCII character. The result is a non-negative integer in the range from 0 to 127, inclusive.

### 11.3.15 tolower

```
int tolower(int c);
```

The function returns the corresponding lowercase letter if one exists and if `isupper(c);` otherwise, it returns `c`.

### 11.3.16 _tolower

```
int _tolower(int c); [POSIX]
```

The function returns the corresponding lowercase letter if one exists and if `isupper(c);` otherwise, it returns `c`.

### 11.3.17 toupper

```
int toupper(int c);
```

The function returns the corresponding uppercase letter if one exists and if `islower(c);` otherwise, it returns `c`.

### 11.3.18 _toupper

```
int _toupper(int c); [POSIX]
```

The function returns the corresponding uppercase letter if one exists and if `islower(c);` otherwise, it returns `c`.

## 11.4 <errno.h>

Include the standard header `<errno.h>` to test the value stored in `errno` by certain library functions. At program startup, the value stored is zero. Library functions store only values greater than zero. Any library function can alter the value stored, but only those cases where a library function is explicitly required to store a value are documented here.

To test whether a library function stores a value in `errno`, the program should store the value zero there immediately before it calls the library function.

```
#define EDOM <#if expression>
#define EILSEQ <#if expression> [Added with Amendment 1]
#define ERANGE <#if expression>
#define errno <int modifiable lvalue>
```

An implementation can define additional macros in this standard header that you can test for equality with the value stored. All these additional macros have names that begin with E. The following macros, for example, are added by POSIX:

```
#define E2BIG [argument list too long]
#define EACCES [process lacks access permission]
#define EAGAIN [resource temporarily unavailable]
#define EBADF [bad file descriptor]
#define EBADMSG [bad message]
#define EBUSY [resource is busy]
#define ECANCELED [operation canceled]
#define ECHILD [no child process present]
#define EDEADLK [resource deadlock was avoided]
#define EEXIST [file already exists]
#define EFAULT [bad memory address]
#define EFBIG [file would become too big]
#define EINPROGRESS [asynchronous operation not completed]
#define EINTR [operation interrupted by a signal]
#define EINVAL [invalid argument]
#define EIO [input/output error]
#define EISDIR [can't write to a directory]
#define EMFILE [process has too many files open]
#define EMLINK [too many links to a file]
#define EMSGSIZE [bad message buffer size]
#define ENAMETOOLONG [file name too long]
#define ENFILE [system has too many files open]
#define ENODEV [invalid device operation]
#define ENOENT [bad file or directory name]
#define ENOEXEC [can't execute file]
#define ENOLCK [too many locks on a file or record]
#define ENOMEM [insufficient memory]
#define ENOSPC [insufficient space on a device]
#define ENOSYS [unimplemented function]
#define ENOTDIR [invalid directory name]
#define ENOTEMPTY [directory not empty]
#define ENOTSUP [unsupported feature[
#define ENOTTY [bad I/O control operation]
#define ENXIO [bad device specifier]
#define EPERM [process lacks permission]
#define EPIPE [write to a broken pipe]
#define EROFS [write to a read-only file system]
```

```
#define ESPIPE [seek on a pipe]
#define ESRCH [process search failed]
#define ETIMEDOUT [time limit expired]
#define EXDEV [link across file systems]
```

### 11.4.1 EDOM

```
#define EDOM <#if expression>
```

The macro yields the value stored in `errno` on a domain error.

### 11.4.2 EILSEQ

```
#define EILSEQ <#if expression>
```
[Added with Amendment 1]

The macro yields the value stored in `errno` on an invalid multibyte sequence.

### 11.4.3 ERANGE

```
#define ERANGE <#if expression>
```

The macro yields the value stored in `errno` on a range error.

### 11.4.4 errno

```
#define errno <int modifiable lvalue>
```

The macro designates an object that is assigned a value greater than zero on certain library errors.

# 11.5 <fcntl.h>

This file contains the function for file descriptor control.

```
int fcntl(int fd, int cmd, ...); [POSIX]
fcntl
int fcntl(int fd, int cmd, ...); [POSIX]
```

`fcntl` provides for control over descriptors. The argument fd is a descriptor to be operated on by cmd as described below. The third parameter is called arg and is technically a pointer to void, but it is interpreted as an int by some commands and ignored by others.

Table 11-1 lists the `fcntl` commands.

**Table 11-1 fcntl commands**

| Command | Description |
|---|---|
| `F_DUPFD` | Return a new descriptor as follows:<br>■ Lowest numbered available descriptor greater than or equal to `arg`, which is interpreted as an `int`.<br>■ Same object references as the original descriptor.<br>■ New descriptor shares the same file offset if the object was a file.<br>■ Same access mode (read, write or read/write).<br>■ Same file status flags *(*both file descriptors share the same file status flags).<br>■ The close-on-exec flag associated with the new file descriptor is cleared to remain open across `execve` system calls. |
| `F_GETFD` | Get the close-on-exec flag associated with the file descriptor `fd` as `FD_CLOEXEC`.<br>If the returned value ANDed with `FD_CLOEXEC` is `0`, the file will remain open across `exec`, otherwise the file will be closed upon execution of `exec` (`arg` is ignored). |
| `F_SETFD` | Set the close-on-exec flag associated with `fd` to `arg`, where `arg` is either `0` or `FD_CLOEXEC`, as described above. |
| `F_GETFL` | Get descriptor status flags, as described below (`arg` is ignored). |
| `F_SETFL` | Set descriptor status flags to `arg`, which is interpreted as an `int`. |
| `F_GETOWN` | Get the process ID or process group currently receiving `SIGIO` and `SIGURG` signals; process groups are returned as negative values (`arg` is ignored). |
| `F_SETOWN` | Set the process or process group to receive `SIGIO` and `SIGURG` signals.<br>Process groups are specified by supplying `arg` as negative, otherwise `arg` is interpreted as a process ID. The argument `arg` is interpreted as an int. |
| `F_CLOSEM` | Close all file descriptors greater than or equal to `fd`. |
| `F_MAXFD` | Return the maximum file descriptor number currently open by the process. |

Table 11-2 lists the flags for F_GETFL and F_SETFL.

**Table 11-2 fcntl flags**

| Flag | Description |
|---|---|
| O_NONBLOCK | Non-blocking I/O; if no data is available to a read call, or if a write operation would block, the read or write call returns -1 with the error EAGAIN. |
| O_APPEND | Force each write to append at the end of file; corresponds to the O_APPEND flag of open. |
| O_ASYNC | Enable the SIGIO signal to be sent to the process group when I/O is possible, such as, upon availability of data to be read. |

Several commands are available for doing advisory file locking; they all operate on the following structure:

```
struct flock {
   off_t   l_start;        /* starting offset */
   off_t   l_len;          /* len = 0 means until end of file */
   pid_t   l_pid;          /* lock owner */
   short   l_type;         /* lock type: read/write, etc. */
   short   l_whence;       /* type of l_start */
};
```

Table 11-3 lists the commands available for advisory record locking.

**Table 11-3 Commands for advisory record locking**

| Command | Description |
|---|---|
| F_GETLK | Get the first lock that blocks the lock description pointed to by the third argument, arg, taken as a pointer to a struct flock (see above).<br>The information retrieved overwrites the information passed to fcntl in the flock structure.<br>If no lock is found that would prevent this lock from being created, the structure is left unchanged by this function call except for the lock type l_type, which is set to F_UNLCK. |
| F_SETLK | Set or clear a file segment lock according to the lock description pointed to by the third argument, arg, taken as a pointer to a struct flock (see above).<br>As specified by the value of l_type, F_SETLK is used to establish shared (or read) locks (F_RDLCK) or exclusive (or write) locks, (F_WRLCK), as well as remove either type of lock (F_UNLCK).<br>If a shared or exclusive lock cannot be set, fcntl returns immediately with EAGAIN. |
| F_SETLKW | This command is the same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the process waits until the request can be satisfied.<br>If a signal that is to be caught is received while fcntl is waiting for a region, the fcntl will be interrupted if the signal handler has not specified the SA_RESTART. |

When a shared lock has been set on a segment of a file, other processes can set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock fails if the file descriptor was not opened with read access.

An exclusive lock prevents any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock fails if the file was not opened with write access.

The value of `l_whence` is `SEEK_SET`, `SEEK_CUR`, or `SEEK_END` to indicate that the relative offset, `l_start` bytes, will be measured from the start of the file, current position, or end of the file, respectively. The value of `l_len` is the number of consecutive bytes to be locked. If `l_len` is negative, the result is undefined. The `l_pid` field is only used with `F_GETLK` to return the process ID of the process holding a blocking lock. After a successful `F_GETLK` request, the value of `l_whence` is `SEEK_SET`.

Locks may start and extend beyond the current end of a file, but may not start or extend before the beginning of the file. A lock is set to extend to the largest possible value of the file offset for that file if `l_len` is set to zero. If `l_whence` and `l_start` point to the beginning of the file, and `l_len` is zero, the entire file is locked. If an application wishes only to do entire file locking, the `flock` system call is much more efficient.

There is at most one type of lock set for each byte in the file. Before a successful return from an `F_SETLK` or an `F_SETLKW` request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region is replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, an `F_SETLK` or an `F_SETLKW` request fails or blocks respectively when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

This interface follows the completely stupid semantics of AT&T System V UNIX and IEEE Std. 1003.1-1988 (“POSIX.1”) that require that all locks associated with a file for a given process are removed when any file descriptor for that file is closed by that process. This semantic means that applications must be aware of any files that a subroutine library may access.

For example, if an application for updating the password file locks the password file database while making the update, and then calls `getpwnam` to retrieve a record, the lock will be lost because `getpwnam` opens, reads, and closes the password database. The database close will release all locks that the process has associated with the database, even if the library routine never requested a lock on the database.

Another minor semantic problem with this interface is that locks are not inherited by a child process created using the `fork` function. The `flock` interface has much more rational last close semantics and allows locks to be inherited by child processes. Calling `flock` is recommended for applications that want to ensure the integrity of their locks when using library routines or wish to pass locks to their children. The `flock` and `fcntl` locks can be safely used concurrently.

All locks associated with a file for a given process are removed when the process terminates.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock the locked region of another process. This implementation detects that sleeping until a locked region is unlocked would cause a deadlock and fails with an `EDEADLK` error.

On successful completion, the return value depends on cmd. Table 11-4 lists the possible return values.

**Table 11-4 fcntl return values**

| Value | Description |
|---|---|
| F_DUPFD | A new file descriptor. |
| F_GETFD | Value of flag (only the low-order bit is defined). |
| F_GETFL | Value of flags. |
| F_GETOWN | Value of file descriptor owner. |
| F_MAXFD | Value of the highest file descriptor open by the process. |
| Other | Value other than -1. |

If the operation fails, the return value is `-1`, and `errno` is set to indicate the error.

## 11.6 <fenv.h>

[Added with C99]

Include the standard header `<fenv.h>` to define two types, several macros, and several functions that test and control floating point status, if the implementation permits. The functionality matches IEC 60559, but it can be applied to similar floating point hardware. Floating Point status can be represented in an object of type `fexcept_t`. It forms part of the floating point control, which determines the behavior of floating point arithmetic. A copy of the floating point control can be represented in an object of type `fenv_t`.

Another part of the floating point control is the rounding mode, representable as a value of type `int`, which determines how floating point values convert to integers. The rounding modes are:

- **Downward**, toward the nearest more negative integer
- **To nearest**, toward the nearest integer with the closer value, or toward the nearest even integer if two integers are equally near
- **Toward zero**, toward the nearest integer closer to zero (also called truncation)
- **Upward**, toward the nearest more positive integer

An implementation might define additional rounding modes.

By convention, a C function does not alter the floating point control, nor assume other than the default settings for the floating point control, without explicitly documenting the fact. Any C function can change the floating point status by reporting one of several floating point exceptions:

- An **inexact floating point exception** can occur when a finite floating point result cannot be exactly represented, as in 2.0 / 3.0.
- An **invalid floating point exception** can occur when a floating point operation involves an invalid combination of operators and operands, as in 0.0 / 0.0.
- An **overflow floating point exception** can occur when the magnitude of a finite floating point result is too large to represent, as in `DBL_MAX` or `DBL_MIN`.
- An **underflow floating point exception** can occur when the magnitude of a finite floating point result is too small to represent, as in `DBL_MIN` or `DBL_MAX`.
- A **zero-divide floating point exception** can occur when a floating point divide has a finite dividend and a zero divisor, as in 1.0 / 0.0.

An implementation may define additional floating point exceptions.

Reporting an exception sets a corresponding indicator in the floating point status. It can also raise a floating point exception, which can result in a hardware trap or the raising of a signal.

The following pragma informs the translator whether the program intends to control and test floating point status:

```
#pragma STD FENV_ACCESS [ON|OFF|DEFAULT]
```

If the parameter is ON, the program may use the functions declared in this header to control and test floating point status. If the parameter is OFF, the use of these functions is disallowed. The parameter DEFAULT restores the original state, which is implementation defined. If the pragma occurs outside an external declaration, it remains in effect until overridden by another such pragma. If the pragma occurs inside an external declaration, it must precede all explicit declarations and statements within a compound statement. It remains in effect until overridden by another such pragma or until the end of the compound statement. On a transition from OFF to ON, floating point status flags are unspecified and the floating point control is in its default state.

```
/* MACROS */
#define FE_DIVBYZERO <integer constant expression> [Optional]
#define FE_INEXACT <integer constant expression> [Optional]
#define FE_INVALID <integer constant expression> [Optional]
#define FE_OVERFLOW <integer constant expression> [Optional]
#define FE_UNDERFLOW <integer constant expression> [Optional]
#define FE_ALL_EXCEPT <integer constant expression> [Optional]
#define FE_DOWNWARD <integer constant expression> [Optional]
#define FE_TONEAREST <integer constant expression> [Optional]
#define FE_TOWARDZERO <integer constant expression> [Optional]
#define FE_UPWARD <integer constant expression> [Optional]
#define FE_DFL_ENV <const *fenv_t rvalue>

/* TYPES */ typedef o-type fenv_t;
typedef i-type fexcept_t;

/* FUNCTIONS */
int feclearexcept(int except);
int fegetexceptflag(fexcept_t *pflag, int except);
int feraiseexcept(int except);
int fesetexceptflag(const fexcept_t *pflag, int except);
int fetestexcept(int except);
int fegetround(void);
int fesetround(int mode);
int fegetenv(fenv_t *penv);
int feholdexcept(fenv_t *penv);
int fesetenv(const fenv_t *penv);
int feupdateenv(const fenv_t *penv);
fexcept_t fegettrapenable(void); [non-standard]
int fesettrapenable(fexcept_t enables); [non-standard]
```

### 11.6.1 FE_ALL_EXCEPT

```
#define FE_ALL_EXCEPT <integer constant expression> [Optional]
```

The macro expands to an integer value that, when ANDed with a value of type `fexcept_t`, yields a nonzero value only if the indicator is set for one or more floating point exceptions. The macro is not defined if the functions declared in this header cannot control floating point exceptions.

### 11.6.2 FE_DFL_ENV

```
#define FE_DFL_ENV <const *fenv_t rvalue>
```

The macro expands to a pointer to an object that describes the settings for the floating point control at program startup.

### 11.6.3 FE_DIVBYZERO

```
#define FE_DIVBYZERO <integer constant expression> [Optional]
```

The macro expands to an integer value that, when ANDed with a value of type `fexcept_t`, yields a nonzero value only if the indicator is set for a zero-divide floating point exception. The macro is not defined if the functions declared in this header cannot control floating point exceptions.

### 11.6.4 FE_DOWNWARD

```
#define FE_DOWNWARD <integer constant expression> [Optional]
```

The macro expands to an integer value accepted as an argument to `fesetround` and returned by `fegetround` to indicate the downward rounding mode. The macro is not defined if the functions declared in this header cannot control the rounding mode.

### 11.6.5 FE_INEXACT

```
#define FE_INEXACT <integer constant expression> [Optional]
```

The macro expands to an integer value that, when ANDed with a value of type `fexcept_t`, yields a nonzero value only if the indicator is set for an inexact floating point exception. The macro is not defined if the functions declared in this header cannot control floating point exceptions.

### 11.6.6 FE_INVALID

```
#define FE_INVALID <integer constant expression> [Optional]
```

The macro expands to an integer value that, when ANDed with a value of type `fexcept_t` yields a nonzero value only if the indicator is set for an invalid floating point exception. The macro is not defined if the functions declared in this header cannot control floating point exceptions.

### 11.6.7 FE_TONEAREST

```
#define FE_TONEAREST <integer constant expression> [Optional]
```

The macro expands to an integer value accepted as an argument to `fesetround` and returned by `fegetround` to indicate the to nearest rounding mode. The macro is not defined if the functions declared in this header cannot control the rounding mode.

### 11.6.8 FE_TOWARDZERO

```
#define FE_TOWARDZERO <integer constant expression> [Optional]
```

The macro expands to an integer value accepted as an argument to `fesetround` and returned by `fegetround` to indicate the toward zero rounding mode. The macro is not defined if the functions declared in this header cannot control the rounding mode.

### 11.6.9 FE_OVERFLOW

```
#define FE_OVERFLOW <integer constant expression> [Optional]
```

The macro expands to an integer value that, when ANDed with a value of type `fexcept_t`, yields a nonzero value only if the indicator is set for an overflow floating point exception. The macro is not defined if the functions declared in this header cannot control floating point exceptions.

### 11.6.10 FE_UNDERFLOW

```
#define FE_UNDERFLOW <integer constant expression> [Optional]
```

The macro expands to an integer value that, when ANDed with a value of type `fexcept_t`, yields a nonzero value only if the indicator is set for an underflow floating point exception. The macro is not defined if the functions declared in this header cannot control floating point exceptions.

### 11.6.11 FE_UPWARD

```
#define FE_UPWARD <integer constant expression> [Optional]
```

The macro expands to an integer value accepted as an argument to `fesetround` and returned by `fegetround` to indicate the upward rounding mode. The macro is not defined if the functions declared in this header cannot control the rounding mode.

### 11.6.12 fenv_t

```
typedef o-type fenv_t;
```

The type is an object type `o-type` that can represent the settings stored in the floating point control.

### 11.6.13 feclearexcept

```
int feclearexcept(int except);
```

The function attempts to clear the exceptions selected by `except` in the floating point status portion of the floating point control. It returns zero only if `except` is zero or all the exceptions selected by `except` are successfully cleared.

### 11.6.14 fegettrapenable

```
fexcept_t fegettrapenable(void); [non-standard]
```

The function returns the current floating point enable mask from the floating point control, or `(fexcept_t)(-1)` if it cannot be determined. For an exception selected by `except` (such as `FE_OVERFLOW`), an operation that raises the exception results in a hardware trap or the raising of a signal only if `fegettrapenable() & except` is nonzero. At program startup, `fegettrapenable()` returns zero.

### 11.6.15 fegetenv

```
int fegetenv(fenv_t *penv);
```

The function attempts to store the settings in the floating point control at `*penv`. It returns zero only if the store succeeds.

### 11.6.16 fegetexceptflag

```
int fegetexceptflag(fexcept_t *pflag, int except);
```

The function attempts to store in `*pflag` a representation of the exceptions selected by `except` from the floating point status portion of the floating point control. It returns zero only if `except` is zero or all the exceptions selected by `except` are successfully stored.

## 11.6.17 fegetround

```
int fegetround(void);
```

The function returns the current rounding mode from the floating point control, or a negative value if it cannot be determined.

## 11.6.18 feholdexcept

```
int feholdexcept(fenv_t *penv);
```

The function stores the settings in the floating point control at `*penv`. It also clears all exceptions in the floating point status portion of the floating point control and endeavors to establish settings that will not raise any exceptions.

The effect is equivalent to calling `fegetenv(penv)` followed by `feclearexcept(FE_ALL_EXCEPT)` and `fesettrapenable(0)`. The function returns zero only if it succeeds in establishing such settings.

You can use this function in conjunction with `feupdateenv` to defer the raising of exceptions until spurious ones are cleared, as in:

```
fenv_t env;
feholdexcept(&env);            // save environment
<evaluate expressions>         // may accumulate exceptions
feclearexcept(FE_INEXACT);     // clear unwanted exception
feupdateenv(&env);             // raise any remaining exceptions
```

## 11.6.19 feraiseexcept

```
int feraiseexcept(int except);
```

The function attempts to raise the floating point exceptions specified by except. Whether it raises an inexact floating point exception after an overflow floating point exception or an underflow floating point exception is implementation-defined. It returns zero only if `except` is zero or all the exceptions selected by `except` are successfully raised.

## 11.6.20 fesetenv

```
int fesetenv(const fenv_t *penv);
```

The function attempts to restore the settings in the floating point control from`*penv`. It returns zero only if the settings are successfully restored.

The settings must be determined by `FE_DFL_ENV` or by an earlier call to `fegetenv` or `feholdexcept`. Otherwise, if `fetestexcept( fegettrapenable())` is nonzero for the restored settings, it is unspecified whether the function evaluation results in a hardware trap or the raising of a signal.

## 11.6.21 fesetexceptflag

```
int fesetexceptflag(const fexcept_t *pflag, int except);
```

The function attempts to set the exceptions selected by `except` in the floating point status portion of the floating point control to the values of the corresponding bits selected by `except & *pflag`. It returns zero only if `except` is zero or all the exceptions selected by `except & *pflag` are successfully set.

The value stored in `*pflag` must be determined by an earlier call to `fegetexceptflag`, without an intervening call to `fesettrapenable`. Otherwise, if `fegettrapenable() & except & *pflag` is nonzero, it is unspecified whether the function evaluation results in a hardware trap or the raising of a signal.

## 11.6.22 fesetround

```
int fesetround(int mode);
```

The function sets the current rounding mode from mode in the floating point control. An invalid value of mode leaves the rounding mode unchanged. The function returns zero only if the rounding mode is successfully set to mode.

## 11.6.23 fesettrapenable

```
int fesettrapenable(fexcept_t enables); [non-standard]
```

The function sets the current floating point enable mask from enables. An invalid value of enables leaves the floating point enable mask unchanged. The function returns zero only if the floating point enable mask is successfully set to enables.

If `fetestexcept(enables)` is nonzero, it is unspecified whether `fesettrapenable(enables)` results in a hardware trap or the raising of a signal.

## 11.6.24 fetestexcept

```
int fetestexcept(int except);
```

The function returns a nonzero value only if one or more of the exceptions selected by `except` are set in the floating point status portion of the floating point control.

## 11.6.25 feupdateenv

```
void feupdateenv(const fenv_t *penv);
```

The function effectively executes the following:

```
int except = fetestexcept(FE_ALL_EXCEPT); fesetenv(penv);
feraiseexcept(except);
```

Thus, it restores the settings in the floating point control from `*penv`, after first saving the exceptions selected by the current floating point status stored in the floating point control. The function then raises the saved exceptions. It returns zero only if the settings are successfully restored.

The restored settings must be determined by `FE_DFL_ENV` or by an earlier call to `fegetenv` or `feholdexcept`. Otherwise, it is unspecified whether the call `fesetenv(penv)` results in a hardware trap or the raising of a signal.

## 11.6.26 fexcept_t

```
typedef i-type fexcept_t;
```

The type is an integer type `i-type` that can represent the floating point status.

# 11.7 <float.h>

Include the standard header `<float.h>` to determine various properties of floating point type representations. This header is available even in a freestanding implementation.

You can test the values of any of the integer macros except `FLT_ROUNDS` in an `if` directive. (The macros expand to `#if` expressions.) All other macros defined in this header expand to floating point constant expressions.

Some target environments can change the rounding and error-reporting properties of floating point type representations while the program is running.

```
#define FLT_RADIX <#if expression >= 2>
#define FLT_ROUNDS <integer rvalue>
#define FLT_EVAL_METHOD <#if expression> [Added with C99]
#define DECIMAL_DIG <#if expression> >= 10 [Added with C99]
#define DBL_DIG <#if expression >= 10>
#define DBL_EPSILON <double constant <= 10^(-9)>
#define DBL_MANT_DIG <#if expression>
#define DBL_MAX <double constant >= 10^37>
#define DBL_MAX_10_EXP <#if expression >= 37>
#define DBL_MAX_EXP <#if expression>
#define DBL_MIN <double constant <= 10^(-37)>
#define DBL_MIN_10_EXP <#if expression <= -37>
#define DBL_MIN_EXP <#if expression>
#define FLT_DIG <#if expression >= 6>
#define FLT_EPSILON <float constant <= 10^(-5)>
#define FLT_MANT_DIG <#if expression>
#define FLT_MAX <float constant >= 10^37>
#define FLT_MAX_10_EXP <#if expression >= 37>
#define FLT_MAX_EXP <#if expression>
#define FLT_MIN <float constant <= 10^(-37)>
#define FLT_MIN_10_EXP <#if expression <= -37>
#define FLT_MIN_EXP <#if expression>
#define LDBL_DIG <#if expression >= 10>
#define LDBL_EPSILON <long double constant <= 10^(-9)>
#define LDBL_MANT_DIG <#if expression>
#define LDBL_MAX <long double constant >= 10^37>
#define LDBL_MAX_10_EXP <#if expression >= 37>
#define LDBL_MAX_EXP <#if expression>
#define LDBL_MIN <long double constant <= 10^(-37)>
#define LDBL_MIN_10_EXP <#if expression <= -37>
#define LDBL_MIN_EXP <#if expression>
```

## 11.7.1 DBL_DIG

```
#define DBL_DIG <#if expression >= 10>
```

The macro yields the precision in decimal digits for type `double`.

### 11.7.2 DBL_EPSILON

```
#define DBL_EPSILON <double constant <= 10^(-9)>
```

The macro yields the smallest `X` of type `double` such that `1.0 +X != 1.0`.

### 11.7.3 DBL_MANT_DIG

```
#define DBL_MANT_DIG <#if expression>
```

The macro yields the number of mantissa digits, base `FLT_RADIX`, for type `double`.

### 11.7.4 DBL_MAX

```
#define DBL_MAX <double constant >= 10^37>
```

The macro yields the largest finite representable value of type `double`.

### 11.7.5 DBL_MAX_10_EXP

```
#define DBL_MAX_10_EXP <#if expression >= 37>
```

The macro yields the maximum integer `X`, such that `10^X` is a finite representable value of type `double`.

### 11.7.6 DBL_MAX_EXP

```
#define DBL_MAX_EXP <#if expression>
```

The macro yields the maximum integer `X`, such that `FLT_RADIX^(X - 1)` is a finite representable value of type `double`.

### 11.7.7 DBL_MIN

```
#define DBL_MIN <double constant <= 10^(-37)>
```

The macro yields the smallest normalized, finite representable value of type `double`.

### 11.7.8 DBL_MIN_10_EXP

```
#define DBL_MIN_10_EXP <#if expression <= -37>
```

The macro yields the minimum integer `X` such that `10^X` is a normalized, finite representable value of type `double`.

### 11.7.9 DBL_MIN_EXP

```
#define DBL_MIN_EXP <#if expression>
```

The macro yields the minimum integer `X` such that `FLT_RADIX^(X - 1)` is a normalized, finite representable value of type `double`.

### 11.7.10 DECIMAL_DIG

```
#define DECIMAL_DIG <#if expression >= 10>
```
[Added with C99]

The macro yields the minimum number of decimal digits needed to represent all the significant digits for type `long double`.

### 11.7.11 FLT_DIG

```
#define FLT_DIG <#if expression >= 6>
```

The macro yields the precision in decimal digits for type `float`.

### 11.7.12 FLT_EPSILON

```
#define FLT_EPSILON <float constant <= 10^(-5)>
```

The macro yields the smallest `X` of type `float` such that `1.0 +X != 1.0`.

### 11.7.13 FLT_EVAL_METHOD

```
#define FLT_EVAL_METHOD <#if expression>
```
[Added with C99]

The macro yields a value that describes the evaluation mode for floating point operations. The values are:

- `-1` if the mode is indeterminate
- `0` if no promotions occur
- `1` if `float` values promote to `double`
- `2` if `float` and `double` values promote to `long double`

An implementation can define additional negative values for this macro.

### 11.7.14 FLT_MANT_DIG

```
#define FLT_MANT_DIG <#if expression>
```

The macro yields the number of mantissa digits, base `FLT_RADIX`, for type `float`.

### 11.7.15 FLT_MAX

```
#define FLT_MAX <float constant >= 10^37>
```

The macro yields the largest finite representable value of type `float`.

### 11.7.16 FLT_MAX_10_EXP

```
#define FLT_MAX_10_EXP <#if expression >= 37>
```

The macro yields the maximum integer `X`, such that `10^X` is a finite representable value of type `float`.

### 11.7.17 FLT_MAX_EXP

```
#define FLT_MAX_EXP <#if expression>
```

The macro yields the maximum integer `X`, such that `FLT_RADIX^(X - 1)` is a finite representable value of type `float`.

### 11.7.18 FLT_MIN

```
#define FLT_MIN <float constant <= 10^(-37)>
```

The macro yields the smallest normalized, finite representable value of type `float`.

### 11.7.19 FLT_MIN_10_EXP

```
#define FLT_MIN_10_EXP <#if expression <= -37>
```

The macro yields the minimum integer `X`, such that `10^X` is a normalized, finite representable value of type `float`.

### 11.7.20 FLT_MIN_EXP

```
#define FLT_MIN_EXP <#if expression>
```

The macro yields the minimum integer `X`, such that `FLT_RADIX^(X - 1)` is a normalized, finite representable value of type `float`.

### 11.7.21 FLT_RADIX

```
#define FLT_RADIX <#if expression >= 2>
```

The macro yields the radix of all floating point representations.

## 11.7.22 FLT_ROUNDS

```
#define FLT_ROUNDS <integer rvalue>
```

The macro yields a value that describes the current rounding mode for floating point operations. The target environment can change the rounding mode while the program executes. How it does so, however, is not specified.

The values are:

- `-1` if the mode is indeterminate
- `0` if rounding is toward zero
- `1` if rounding is to nearest representable value
- `2` if rounding is toward positive infinity
- `3` if rounding is toward negative infinity

An implementation can define additional values for this macro.

## 11.7.23 LDBL_DIG

```
#define LDBL_DIG <#if expression >= 10>
```

The macro yields the precision in decimal digits for type `long double`.

## 11.7.24 LDBL_EPSILON

```
#define LDBL_EPSILON <long double constant <= 10^(-9)>
```

The macro yields the smallest X of type `long double` such that `1.0 + X != 1.0`.

## 11.7.25 LDBL_MANT_DIG

```
#define LDBL_MANT_DIG <#if expression>
```

The macro yields the number of mantissa digits, base `FLT_RADIX`, for type `long double`.

## 11.7.26 LDBL_MAX

```
#define LDBL_MAX <long double constant >= 10^37>
```

The macro yields the largest finite representable value of type `long double`.

## 11.7.27 LDBL_MAX_10_EXP

```
#define LDBL_MAX_10_EXP <#if expression >= 37>
```

The macro yields the maximum integer `X`, such that `10^X` is a finite representable value of type `long double`.

## 11.7.28 LDBL_MAX_EXP

```
#define LDBL_MAX_EXP <#if expression>
```

The macro yields the maximum integer `X`, such that `FLT_RADIX^(X - 1)` is a finite representable value of type `long double`.

## 11.7.29 LDBL_MIN

```
#define LDBL_MIN <long double constant <= 10^(-37)>
```

The macro yields the smallest normalized, finite representable value of type `long double`.

## 11.7.30 LDBL_MIN_10_EXP

```
#define LDBL_MIN_10_EXP <#if expression <= -37>
```

The macro yields the minimum integer `X`, such that `10^X` is a normalized, finite representable value of type `long double`.

## 11.7.31 LDBL_MIN_EXP

```
#define LDBL_MIN_EXP <#if expression>
```

The macro yields the minimum integer `X`, such that `FLT_RADIX^(X - 1)` is a normalized, finite representable value of type `long double`.

# 11.8 <inttypes.h>

[Added with C99]

Include the standard header <inttypes.h> to include the standard header <stdint.h> and to define a type, several functions, and numerous macros for fine control over the conversion of integers.

The definitions shown for the macros are merely representative—they can vary among implementations.

```
/* TYPE DEFINITIONS */
typedef struct {
    intmax_t quot, rem;
} imaxdiv_t;

/* FUNCTION DECLARATIONS */
intmax_t imaxabs(intmax_t i);
intmax_t abs(intmax_t i); [C++ only]
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
imaxdiv_t div(intmax_t numer, intmax_t denom); [C++ only]
intmax_t strtoimax(const char *restrict s,
     char **restrict endptr, int base);
 uintmax_t strtoumax(const char *restrict s,
     char **restrict endptr, int base);
 intmax_t wcstoimax(const wchar_t *restrict s,
     wchar_t **restrict endptr, int base);
 uintmax_t wcstoumax(const wchar_t *restrict s,
     wchar_t **restrict endptr, int base);
         /* PRINT FORMAT MACROS */
#define PRId8     "hhd"
#define PRId16    "hd"
#define PRId32    "ld"
#define PRId64    "lld"
#define PRIdFAST8    "hhd"
#define PRIdFAST16    "hd"
#define PRIdFAST32    "ld"
#define PRIdFAST64   "lld"
#define PRIdLEAST8   "hhd"
#define PRIdLEAST16  "hd"
#define PRIdLEAST32  "ld"
#define PRIdLEAST64  "lld"
#define PRIdMAX   "lld"
#define PRIdPTR   "lld"
#define PRIi8     "hhi"
#define PRIi16    "hi"
#define PRIi32    "li"
#define PRIi64    "lli"
#define PRIiFAST8    "hhi"
#define PRIiFAST16    "hi"
#define PRIiFAST32    "li"
#define PRIiFAST64    "lli"
#define PRIiLEAST8   "hhi"
#define PRIiLEAST16  "hi"
#define PRIiLEAST32  "li"
#define PRIiLEAST64  "lli"
```

```
#define PRIiMAX   "lli"
#define PRIiPTR   "lli"
#define PRIo8     "hho"
#define PRIo16    "ho"
#define PRIo32    "lo"
#define PRIo64    "llo"
#define PRIoFAST8    "hho"
#define PRIoFAST16    "ho"
#define PRIoFAST32    "lo"
#define PRIoFAST64   "llo"
#define PRIoLEAST8   "hho"
#define PRIoLEAST16  "ho"
#define PRIoLEAST32  "lo"
#define PRIoLEAST64  "llo"
#define PRIoMAX   "llo"
#define PRIoPTR   "llo"
#define PRIu8     "hhu"
#define PRIu16    "hu"
#define PRIu32    "lu"
#define PRIu64    "llu"
#define PRIuFAST8    "hhu"
#define PRIuFAST16    "hu"
#define PRIuFAST32    "lu"
#define PRIuFAST64   "llu"
#define PRIuLEAST8   "hhu"
#define PRIuLEAST16  "hu"
#define PRIuLEAST32  "lu"
#define PRIuLEAST64  "llu"
#define PRIuMAX   "llu"
#define PRIuPTR   "llu"
#define PRIx8     "hhx"
#define PRIx16    "hx"
#define PRIx32    "lx"
#define PRIx64    "llx"
#define PRIxFAST8    "hhx"
#define PRIxFAST16    "hx"
#define PRIxFAST32    "lx"
#define PRIxFAST64   "llx"
#define PRIxLEAST8   "hhx"
#define PRIxLEAST16  "hx"
#define PRIxLEAST32  "lx"
#define PRIxLEAST64  "llx"
#define PRIxMAX   "llx"
#define PRIxPTR   "llx"
#define PRIX8     "hhX"
#define PRIX16    "hX"
#define PRIX32    "lX"
#define PRIX64    "llX"
#define PRIXFAST8    "hhX"
#define PRIXFAST16    "hX"
#define PRIXFAST32    "lX"
#define PRIXFAST64   "llX"
#define PRIXLEAST8   "hhX"
#define PRIXLEAST16  "hX"
#define PRIXLEAST32  "lX"
#define PRIXLEAST64  "llX"
```

```
#define PRIXMAX   "llX"
#define PRIXPTR   "llX"         /* SCAN FORMAT MACROS */
#define SCNd8     "hhd"
#define SCNd16    "hd"
#define SCNd32    "ld"
#define SCNd64    "lld"
#define SCNdFAST8    "hhd"
#define SCNdFAST16    "hd"
#define SCNdFAST32    "ld"
#define SCNdFAST64   "lld"
#define SCNdLEAST8   "hhd"
#define SCNdLEAST16  "hd"
#define SCNdLEAST32  "ld"
#define SCNdLEAST64  "lld"
#define SCNdMAX   "lld"
#define SCNdPTR   "lld"
#define SCNi8     "hhi"
#define SCNi16    "hi"
#define SCNi32    "li"
#define SCNi64    "lli"
#define SCNiFAST8    "hhi"
#define SCNiFAST16    "hi"
#define SCNiFAST32    "li"
#define SCNiFAST64   "lli"
#define SCNiLEAST8   "hhi"
#define SCNiLEAST16  "hi"
#define SCNiLEAST32  "li"
#define SCNiLEAST64  "lli"
#define SCNiMAX   "lli"
#define SCNiPTR   "lli"
#define SCNo8     "hho"
#define SCNo16    "ho"
#define SCNo32    "lo"
#define SCNo64    "llo"
#define SCNoFAST8    "hho"
#define SCNoFAST16    "ho"
#define SCNoFAST32    "lo"
#define SCNoFAST64   "llo"
#define SCNoLEAST8   "hho"
#define SCNoLEAST16  "ho"
#define SCNoLEAST32  "lo"
#define SCNoLEAST64  "llo"
#define SCNoMAX   "llo"
#define SCNoPTR   "llo"
#define SCNu8     "hhu"
#define SCNu16    "hu"
#define SCNu32    "lu"
#define SCNu64    "llu"
#define SCNuFAST8    "hhu"
#define SCNuFAST16    "hu"
#define SCNuFAST32    "lu"
#define SCNuFAST64   "llu"
#define SCNuLEAST8   "hhu"
#define SCNuLEAST16  "hu"
#define SCNuLEAST32  "lu"
#define SCNuLEAST64  "llu"
```

```
#define SCNuMAX   "llu"
#define SCNuPTR   "llu"
#define SCNx8     "hhx"
#define SCNx16    "hx"
#define SCNx32    "lx"
#define SCNx64    "llx"
#define SCNxFAST8    "hhx"
#define SCNxFAST16    "hx"
#define SCNxFAST32    "lx"
#define SCNxFAST64   "llx"
#define SCNxLEAST8   "hhx"
#define SCNxLEAST16  "hx"
#define SCNxLEAST32  "lx"
#define SCNxLEAST64  "llx"
#define SCNxMAX   "llx"
#define SCNxPTR   "llx"
```

## 11.8.1 imaxabs, abs

```
intmax_t imaxabs(intmax_t i);
intmax_t abs(intmax_t i); [C++ only]
```

The function returns the absolute value of `i`, `|i|`.

## 11.8.2 imaxdiv, div

```
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
 imaxdiv_t div(intmax_t numer, intmax_t denom); [C++ only]
```

The function divides `numer` by `denom` and returns both quotient and remainder in the structure result `x`, if the quotient can be represented. The structure member `x.quot` is the algebraic quotient truncated toward zero. The structure `memberx.rem` is the remainder, such that `numer == x.quot*denom + x.rem`.

## 11.8.3 imaxdiv_t

```
typedef struct {
     intmax_t quot, rem;
     } imaxdiv_t;
```

The type is the structure type returned by the function `imaxdiv`. The structure contains members that represent the quotient (`quot`) and remainder (`rem`) of a signed integer division with operands of type `intmax_t`. The members shown above can occur in either order.

### 11.8.4 PRId8, PRId16, PRId32, PRId64

```
#define PRId8     "hhd"
#define PRId16    "hd"
#define PRId32    "ld"
#define PRId64    "lld"
```

The macros each expand to a string literal suitable for use as ad print conversion specifier, plus any needed qualifiers, to convert values of the types `int8_t`, `int16_t`, `int32_t`, or `int64_t`, respectively.

The definitions shown here are merely representative.

### 11.8.5 PRIdFAST8, PRIdFAST16, PRIdFAST32, PRIdFAST64

```
#define PRIdFAST8    "hhd"
#define PRIdFAST16    "hd"
#define PRIdFAST32    "ld"
#define PRIdFAST64   "lld"
```

The macros each expand to a string literal suitable for use as ad print conversion specifier, plus any needed qualifiers, to convert values of the types `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, `or` int_fast64_t, respectively.

The definitions shown here are merely representative.

### 11.8.6 PRIdLEAST8, PRIdLEAST16, PRIdLEAST32, PRIdLEAST64

```
#define PRIdLEAST8   "hhd"
#define PRIdLEAST16  "hd"
#define PRIdLEAST32  "ld"
#define PRIdLEAST64  "lld"
```

The macros each expand to a string literal suitable for use as a `d` print conversion specifier, plus any needed qualifiers, to convert values of the types `int_least8_t`, `int_least16_t`, `int_least32_t`, or `int_least64_t`, respectively.

The definitions shown here are merely representative.

### 11.8.7 PRIdMAX

```
#define PRIdMAX   "lld"
```

The macro expands to a string literal suitable for use as a `d` print conversion specifier, plus any needed qualifiers, to convert values of the types `intmax_t`.

The definition shown here is merely representative.

### 11.8.8 PRIdPTR

```
#define PRIdPTR   "lld"
```

The macro expands to a string literal suitable for use as a `d` print conversion specifier, plus any needed qualifiers, to convert values of the type's `intptr_t`.

The definition shown here is merely representative.

### 11.8.9 PRIi8, PRIi16, PRIi32, PRIi64

```
#define PRIi8     "hhi"
#define PRIi16    "hi"
#define PRIi32    "li"
#define PRIi64    "lli"
```

The macros each expand to a string literal suitable for use as an `i` print conversion specifier, plus any needed qualifiers, to convert values of the types `int8_t`, `int16_t`, `int32_t`, or `int64_t`, respectively.

The definitions shown here are merely representative.

### 11.8.10 PRIiFAST8, PRIiFAST16, PRIiFAST32, PRIiFAST64

```
#define PRIiFAST8    "hhi"
#define PRIiFAST16    "hi"
#define PRIiFAST32    "li"
#define PRIiFAST64   "lli"
```

The macros each expand to a string literal suitable for use as an `i` print conversion specifier, plus any needed qualifiers, to convert values of the types `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, `or` int_fast64_t, respectively.

The definitions shown here are merely representative.

### 11.8.11 PRIiLEAST8, PRIiLEAST16, PRIiLEAST32, PRIiLEAST64

```
#define PRIiLEAST8   "hhi"
#define PRIiLEAST16  "hi"
#define PRIiLEAST32  "li"
#define PRIiLEAST64  "lli"
```

The macros each expand to a string literal suitable for use as an `i` print conversion specifier, plus any needed qualifiers, to convert values of the types `int_least8_t`, `int_least16_t`, `int_least32_t`, or `int_least64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.12 PRIiMAX

```
#define PRIiMAX   "lli"
```

The macro expands to a string literal suitable for use as an `i` print conversion specifier, plus any needed qualifiers, to convert values of the types `intmax_t`.

The definition shown here is merely representative.

## 11.8.13 PRIiPTR

```
#define PRIiPTR   "lli"
```

The macro expands to a string literal suitable for use as an `i` print conversion specifier, plus any needed qualifiers, to convert values of the type's `intptr_t`.

The definition shown here is merely representative.

## 11.8.14 PRIo8, PRIo16, PRIo32, PRIo64

```
#define PRIo8     "hho"
#define PRIo16    "ho"
#define PRIo32    "lo"
#define PRIo64    "llo"
```

The macros each expand to a string literal suitable for use as an `o` print conversion specifier, plus any needed qualifiers, to convert values of the types `uint8_t`, `uint16_t`, `uint32_t`, or `uint64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.15 PRIoFAST8, PRIoFAST16, PRIoFAST32, PRIoFAST64

```
#define PRIoFAST8    "hho"
#define PRIoFAST16    "ho"
#define PRIoFAST32    "lo"
#define PRIoFAST64   "llo"
```

The macros each expand to a string literal suitable for use as an `o` print conversion specifier, plus any needed qualifiers, to convert values of the types `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, or `uint_fast64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.16 PRIoLEAST8, PRIoLEAST16, PRIoLEAST32, PRIoLEAST64

```
#define PRIoLEAST8   "hho"
#define PRIoLEAST16  "ho"
#define PRIoLEAST32  "lo"
#define PRIoLEAST64  "llo"
```

The macros each expand to a string literal suitable for use as an `o` print conversion specifier, plus any needed qualifiers, to convert values of the types `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, or `uint_least64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.17 PRIoMAX

```
#define PRIoMAX   "llo"
```

The macro expands to a string literal suitable for use as an `o` print conversion specifier, plus any needed qualifiers, to convert values of the type's `uintmax_t`.

The definition shown here is merely representative.

## 11.8.18 PRIoPTR

```
#define PRIoPTR   "llo"
```

The macro expands to a string literal suitable for use as an `o` print conversion specifier, plus any needed qualifiers, to convert values of the type's `uintptr_t`.

The definition shown here is merely representative.

## 11.8.19 PRIu8, PRIu16, PRIu32, PRIu64

```
#define PRIu8     "hhu"
#define PRIu16    "hu"
#define PRIu32    "lu"
#define PRIu64    "llu"
```

The macros each expand to a string literal suitable for use as a `u` print conversion specifier, plus any needed qualifiers, to convert values of the types `uint8_t`, `uint16_t`, `uint32_t`, or `uint64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.20 PRIuFAST8, PRIuFAST16, PRIuFAST32, PRIuFAST64

```
#define PRIuFAST8    "hhu"
#define PRIuFAST16    "hu"
#define PRIuFAST32    "lu"
#define PRIuFAST64   "llu"
```

The macros each expand to a string literal suitable for use as a `u` print conversion specifier, plus any needed qualifiers, to convert values of the types `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, or `uint_fast64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.21 PRIuLEAST8, PRIuLEAST16, PRIuLEAST32, PRIuLEAST64

```
#define PRIuLEAST8   "hhu"
#define PRIuLEAST16  "hu"
#define PRIuLEAST32  "lu"
#define PRIuLEAST64  "llu"
```

The macros each expand to a string literal suitable for use as a `u` print conversion specifier, plus any needed qualifiers, to convert values of the types `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, or `uint_least64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.22 PRIuMAX

```
#define PRIuMAX   "llu"
```

The macro expands to a string literal suitable for use as a `u` print conversion specifier, plus any needed qualifiers, to convert values of the type's `uintmax_t`.

The definition shown here is merely representative.

## 11.8.23 PRIuPTR

```
#define PRIuPTR   "llu"
```

The macro expands to a string literal suitable for use as a `u` print conversion specifier, plus any needed qualifiers, to convert values of the type's `uintptr_t`.

The definition shown here is merely representative.

## 11.8.24 PRIx8, PRIx16, PRIx32, PRIx64

```
#define PRIx8     "hhx"
#define PRIx16    "hx"
#define PRIx32    "lx"
#define PRIx64    "llx"
```

The macros each expand to a string literal suitable for use as an x print conversion specifier, plus any needed qualifiers, to convert values of the types `uint8_t`, `uint16_t`, `uint32_t`, or `uint64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.25 PRIxFAST8, PRIxFAST16, PRIxFAST32, PRIxFAST64

```
#define PRIxFAST8    "hhx"
#define PRIxFAST16    "hx"
#define PRIxFAST32    "lx"
#define PRIxFAST64   "llx"
```

The macros each expand to a string literal suitable for use as an x print conversion specifier, plus any needed qualifiers, to convert values of the types `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, or `uint_fast64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.26 PRIxLEAST8, PRIxLEAST16, PRIxLEAST32, PRIxLEAST64

```
#define PRIxLEAST8   "hhx"
#define PRIxLEAST16  "hx"
#define PRIxLEAST32  "lx"
#define PRIxLEAST64  "llx"
```

The macros each expand to a string literal suitable for use as an x print conversion specifier, plus any needed qualifiers, to convert values of the types `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, or `uint_least64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.27 PRIxMAX

```
#define PRIxMAX   "llx"
```

The macro expands to a string literal suitable for use as an x print conversion specifier, plus any needed qualifiers, to convert values of the type's `uintmax_t`.

The definition shown here is merely representative.

## 11.8.28 PRIxPTR

```
#define PRIxPTR   "llx"
```

The macro expands to a string literal suitable for use as an x print conversion specifier, plus any needed qualifiers, to convert values of the type's `uintptr_t`.

The definition shown here is merely representative.

## 11.8.29 PRIX8, PRIX16, PRIX32, PRIX64

```
#define PRIX8     "hhx"
#define PRIX16    "hx"
#define PRIX32    "lx"
#define PRIX64    "llx"
```

The macros each expand to a string literal suitable for use as an x print conversion specifier, plus any needed qualifiers, to convert values of the types `uint8_t`, `uint16_t`, `uint32_t`, or `uint64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.30 PRIXFAST8, PRIXFAST16, PRIXFAST32, PRIXFAST64

```
#define PRIXFAST8    "hhx"
#define PRIXFAST16    "hx"
#define PRIXFAST32    "lx"
#define PRIXFAST64   "llx"
```

The macros each expand to a string literal suitable for use as an x print conversion specifier, plus any needed qualifiers, to convert values of the types `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, or `uint_fast64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.31 PRIXLEAST8, PRIXLEAST16, PRIXLEAST32, PRIXLEAST64

```
#define PRIXLEAST8   "hhx"
#define PRIXLEAST16  "hx"
#define PRIXLEAST32  "lx"
#define PRIXLEAST64  "llx"
```

The macros each expand to a string literal suitable for use as an x print conversion specifier, plus any needed qualifiers, to convert values of the types `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, or `uint_least64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.32 PRIXMAX

```
#define PRIXMAX   "llx"
```

The macro expands to a string literal suitable for use as an x print conversion specifier, plus any needed qualifiers, to convert values of the type's `uintmax_t`.

The definition shown here is merely representative.

## 11.8.33 PRIXPTR

```
#define PRIXPTR   "llx"
```

The macro expands to a string literal suitable for use as an x print conversion specifier, plus any needed qualifiers, to convert values of the type's `uintptr_t`.

The definition shown here is merely representative.

## 11.8.34 SCNd8, SCNd16, SCNd32, SCNd64

```
#define SCNd8     "hhd"
#define SCNd16    "hd"
#define SCNd32    "ld"
#define SCNd64    "lld"
```

The macros each expand to a string literal suitable for use as a d scan conversion specifier, plus any needed qualifiers, to convert values of the types `int8_t`, `int16_t`, `int32_t`, or `int64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.35 SCNdFAST8, SCNdFAST16, SCNdFAST32, SCNdFAST64

```
#define SCNdFAST8    "hhd"
#define SCNdFAST16    "hd"
#define SCNdFAST32    "ld"
#define SCNdFAST64   "lld"
```

The macros each expand to a string literal suitable for use as a d scan conversion specifier, plus any needed qualifiers, to convert values of the types `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, `or` int_fast64_t, respectively.

The definitions shown here are merely representative.

## 11.8.36 SCNdLEAST8, SCNdLEAST16, SCNdLEAST32, SCNdLEAST64

```
#define SCNdLEAST8   "hhd"
#define SCNdLEAST16  "hd"
#define SCNdLEAST32  "ld"
#define SCNdLEAST64  "lld"
```

The macros each expand to a string literal suitable for use as a d scan conversion specifier, plus any needed qualifiers, to convert values of the types `int_least8_t`, `int_least16_t`, `int_least32_t`, or `int_least64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.37 SCNdMAX

```
#define SCNdMAX   "lld"
```

The macro expands to a string literal suitable for use as a d scan conversion specifier, plus any needed qualifiers, to convert values of the types intmax_t.

The definition shown here is merely representative.

## 11.8.38 SCNdPTR

```
#define SCNdPTR   "lld"
```

The macro expands to a string literal suitable for use as a d scan conversion specifier, plus any needed qualifiers, to convert values of the type's `intptr_t`.

The definition shown here is merely representative.

## 11.8.39 SCNi8, SCNi16, SCNi32, SCNi64

```
#define SCNi8     "hhi"
#define SCNi16    "hi"
#define SCNi32    "li"
#define SCNi64    "lli"
```

The macros each expand to a string literal suitable for use as an `i` scan conversion specifier, plus any needed qualifiers, to convert values of the types `int8_t`, `int16_t`, `int32_t`, or `int64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.40 SCNiFAST8, SCNiFAST16, SCNiFAST32, SCNiFAST64

```
#define SCNiFAST8    "hhi"
#define SCNiFAST16    "hi"
#define SCNiFAST32    "li"
#define SCNiFAST64   "lli"
```

The macros each expand to a string literal suitable for use as an `i` scan conversion specifier, plus any needed qualifiers, to convert values of the types `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, or `int_fast64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.41 SCNiLEAST8, SCNiLEAST16, SCNiLEAST32, SCNiLEAST64

```
#define SCNiLEAST8   "hhi"
#define SCNiLEAST16  "hi"
#define SCNiLEAST32  "li"
#define SCNiLEAST64  "lli"
```

The macros each expand to a string literal suitable for use as an `i` scan conversion specifier, plus any needed qualifiers, to convert values of the types `int_least8_t`, `int_least16_t`, `int_least32_t`, or `int_least64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.42 SCNiMAX

```
#define SCNiMAX   "lli"
```

The macro expands to a string literal suitable for use as an `i` scan conversion specifier, plus any needed qualifiers, to convert values of the types `intmax_t`.

The definition shown here is merely representative.

## 11.8.43 SCNiPTR

```
#define SCNiPTR   "lli"
```

The macro expands to a string literal suitable for use as an `i` scan conversion specifier, plus any needed qualifiers, to convert values of the type's `intptr_t`.

The definition shown here is merely representative.

## 11.8.44 SCNo8, SCNo16, SCNo32, SCNo64

```
#define SCNo8     "hho"
#define SCNo16    "ho"
#define SCNo32    "lo"
#define SCNo64    "llo"
```

The macros each expand to a string literal suitable for use as an `o` scan conversion specifier, plus any needed qualifiers, to convert values of the types `uint8_t`, `uint16_t`, `uint32_t`, or `uint64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.45 SCNoFAST8, SCNoFAST16, SCNoFAST32, SCNoFAST64

```
#define SCNoFAST8    "hho"
#define SCNoFAST16    "ho"
#define SCNoFAST32    "lo"
#define SCNoFAST64   "llo"
```

The macros each expand to a string literal suitable for use as an `o` scan conversion specifier, plus any needed qualifiers, to convert values of the types `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, or `uint_fast64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.46 SCNoLEAST8, SCNoLEAST16, SCNoLEAST32, SCNoLEAST64

```
#define SCNoLEAST8   "hho"
#define SCNoLEAST16  "ho"
#define SCNoLEAST32  "lo"
#define SCNoLEAST64  "llo"
```

The macros each expand to a string literal suitable for use as an `o` scan conversion specifier, plus any needed qualifiers, to convert values of the types `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, or `uint_least64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.47 SCNoMAX

```
#define SCNoMAX   "llo"
```

The macro expands to a string literal suitable for use as an `o` scan conversion specifier, plus any needed qualifiers, to convert values of the type's `uintmax_t`.

The definition shown here is merely representative.

## 11.8.48 SCNoPTR

```
#define SCNoPTR   "llo"
```

The macro expands to a string literal suitable for use as an `o` scan conversion specifier, plus any needed qualifiers, to convert values of the type's `uintptr_t`.

The definition shown here is merely representative.

## 11.8.49 SCNu8, SCNu16, SCNu32, SCNu64

```
#define SCNu8     "hhu"
#define SCNu16    "hu"
#define SCNu32    "lu"
#define SCNu64    "llu"
```

The macros each expand to a string literal suitable for use as a `u` scan conversion specifier, plus any needed qualifiers, to convert values of the types `uint8_t`, `uint16_t`, `uint32_t`, or `uint64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.50 SCNuFAST8, SCNuFAST16, SCNuFAST32, SCNuFAST64

```
#define SCNuFAST8    "hhu"
#define SCNuFAST16    "hu"
#define SCNuFAST32    "lu"
#define SCNuFAST64   "llu"
```

The macros each expand to a string literal suitable for use as a `u` scan conversion specifier, plus any needed qualifiers, to convert values of the types `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, or `uint_fast64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.51 SCNuLEAST8, SCNuLEAST16, SCNuLEAST32, SCNuLEAST64

```
#define SCNuLEAST8   "hhu"
#define SCNuLEAST16  "hu"
#define SCNuLEAST32  "lu"
#define SCNuLEAST64  "llu"
```

The macros each expand to a string literal suitable for use as a `u` scan conversion specifier, plus any needed qualifiers, to convert values of the types `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, or `uint_least64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.52 SCNuMAX

```
#define SCNuMAX   "llu"
```

The macro expands to a string literal suitable for use as a `u` scan conversion specifier, plus any needed qualifiers, to convert values of the type's `uintmax_t`.

The definition shown here is merely representative.

## 11.8.53 SCNuPTR

```
#define SCNuPTR   "llu"
```

The macro expands to a string literal suitable for use as a `u` scan conversion specifier, plus any needed qualifiers, to convert values of the type's `uintptr_t`.

The definition shown here is merely representative.

## 11.8.54 SCNx8, SCNx16, SCNx32, SCNx64

```
#define SCNx8     "hhx"
#define SCNx16    "hx"
#define SCNx32    "lx"
#define SCNx64    "llx"
```

The macros each expand to a string literal suitable for use as an `x` scan conversion specifier, plus any needed qualifiers, to convert values of the types `uint8_t`, `uint16_t`, `uint32_t`, or `uint64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.55 SCNxFAST8, SCNxFAST16, SCNxFAST32, SCNxFAST64

```
#define SCNxFAST8    "hhx"
#define SCNxFAST16    "hx"
#define SCNxFAST32    "lx"
#define SCNxFAST64   "llx"
```

The macros each expand to a string literal suitable for use as an `x` scan conversion specifier, plus any needed qualifiers, to convert values of the types `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t`, or `uint_fast64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.56 SCNxLEAST8, SCNxLEAST16, SCNxLEAST32, SCNxLEAST64

```
#define SCNxLEAST8   "hhx"
#define SCNxLEAST16  "hx"
#define SCNxLEAST32  "lx"
#define SCNxLEAST64  "llx"
```

The macros each expand to a string literal suitable for use as an `x` scan conversion specifier, plus any needed qualifiers, to convert values of the types `uint_least8_t`, `uint_least16_t`, `uint_least32_t`, or `uint_least64_t`, respectively.

The definitions shown here are merely representative.

## 11.8.57 SCNxMAX

```
#define SCNxMAX   "llx"
```

The macro expands to a string literal suitable for use as an `x` scan conversion specifier, plus any needed qualifiers, to convert values of the type's `uintmax_t`.

The definition shown here is merely representative.

## 11.8.58 SCNxPTR

```
#define SCNxPTR   "llx"
```

The macro expands to a string literal suitable for use as an `x` scan conversion specifier, plus any needed qualifiers, to convert values of the type's `uintptr_t`.

The definition shown here is merely representative.

## 11.8.59 strtoimax

```
intmax_t strtoimax(const char *restrict s, char **restrict endptr,
         int base);
```

The function converts the initial characters of string `s` to an equivalent value `x` of type `intmax_t`. If `endptr` is not a NULL pointer, it stores a pointer to the unconverted remainder of the string in `*endptr`. The function then returns `x`. `strtoimax` converts strings exactly as does `strtol`.

If string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If the equivalent value is too large to represent as type `intmax_t`, `strtoimax` stores the value of `ERANGE` in `errno` and returns either `INTMAX_MAX`, if `x` is positive or `INTMAX_MIN` if x is negative.

## 11.8.60 strtoumax

```
uintmax_t strtoumax(const char *restrict s,
          char **restrict endptr, int base);
```

The function converts the initial characters of string `s` to an equivalent value `x` of type `uintmax_t`. If `endptr` is not a NULL pointer, it stores a pointer to the unconverted remainder of the string in `*endptr`. The function then returns `x`. `strtoumax` converts strings exactly as does `strtoul`.

If string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If the equivalent value is too large to represent as type `uintmax_t`, `strtoumax` stores the value of `ERANGE` in `errno` and returns `UINTMAX_MAX`.

## 11.8.61 wcstoimax

```
intmax_t wcstimax(const wchar_t *restrict s,
         wchar_t **restrict endptr, int base);
```

The function converts the initial wide characters of the wide strings to an equivalent value `x` of type `intmax_t`. If `endptr` is not a NULL pointer, the function stores a pointer to the unconverted remainder of the wide string in `*endptr`. The function then returns `x`.

The initial wide characters of the wide string `s` must match the same pattern as recognized by the function `strtol`, with the same base argument, where each wide character `wc` is converted as if by calling `wctob(wc)`).

If the wide string `s` matches this pattern, `wcstoimax` converts strings exactly as does `strtol`, with the same base argument, for the converted sequence. If the wide string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero.

If the equivalent value is too large in magnitude to represent as type `intmax_t`, `wcstoimax` stores the value of `ERANGE` in `errno` and returns either `INTMAX_MAX` if `x` is positive or `INTMAX_MIN` if `x` is negative.

## 11.8.62 wcstoumax

```
uintmax_t wcstoumax(const wchar_t *restrict s,
      wchar_t **restrict endptr, int base);
```

The function converts the initial wide characters of the wide strings to an equivalent value `x` of type `uintmax_t`. If `endptr` is not a NULL pointer, the function stores a pointer to the unconverted remainder of the wide string in `*endptr`. The function then returns `x`.

The initial wide characters of the wide string `s` must match the same pattern as recognized by the function `strtol`, with the same base argument, where each wide character `wc` is converted as if by calling `wctob(wc))`.

If the wide string `s` matches this pattern, `wcstoumax` converts strings exactly as does `strtol`, with the same base argument, for the converted sequence. If the wide string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If the equivalent value is too large to represent as type `uintmax_t`,`wcstoimax` stores the value of `ERANGE` in `errno` and returns `UINTMAX_MAX`.

# 11.9 <iohw.h>

[Added with TR18015 and TR18037]

Include the added header `<iohw.h>` so that you can write low-level I/O hardware drivers in C that are easier to port to different architectures.

The use of this header does not require the additions to the C language mandated by TR18037, which include fixed-point arithmetic and named address spaces.

```
    /* TYPES */ typedef i-type ioindex_t;
typedef i-type ioreg;
    /* FUNCTIONS (all masked by macros) */ unsigned int iord(ioreg
dev);
unsigned long iordl(ioreg dev);
unsigned int iordbuf(ioreg dev, ioindex_t idx);
unsigned long iordbufl(ioreg dev, ioindex_t idx);
void iowr(ioreg dev, unsigned int val);
void iowrl(ioreg dev, unsigned int val);
void iowrbuf(ioreg dev, ioindex_t idx, unsigned int val);
void iowrbufl(ioreg dev, ioindex_t idx, unsigned int val);
void ioor(ioreg dev, unsigned int val);
void ioorl(ioreg dev, unsigned int val);
void ioorbuf(ioreg dev, ioindex_t idx, unsigned int val);
void ioorbufl(ioreg dev, ioindex_t idx, unsigned int val);
void ioand(ioreg dev, unsigned int val);
void ioandl(ioreg dev, unsigned int val);
void ioandbuf(ioreg dev, ioindex_t idx, unsigned int val);
void ioandbufl(ioreg dev, ioindex_t idx, unsigned int val);
void ioxor(ioreg dev, unsigned int val);
void ioxorl(ioreg dev, unsigned int val);
void ioxorbuf(ioreg dev, ioindex_t idx, unsigned int val);
void ioxorbufl(ioreg dev, ioindex_t idx, unsigned int val);
void iogroup_acquire(int group);
void iogroup_release(int group);
void iogroup_map(int group, int direct);
```

The `<iohw.h>` header defines two types and a number of functions, all of which are typically masked as macros. You should view this header as a prototype for defining the atomic operations needed to express a low-level I/O hardware driver (thus the root name `iohw`) that is intended to be reasonably portable C. The facilities in this header are structured around a few basic concepts:

- The type `ioreg` describes the space of all I/O addresses. These can be port addresses, for processors with port I/O instructions, or memory addresses, for processors with memory-mapped I/O hardware. In a simpler implementation, the actual argument corresponding to the parameter name `dev` can also be used to construct the name of a function to call.
- The type `ioindex_t` describes an integer type that can be used to index into a hardware buffer, an array of I/O addresses.

- An argument named group describes the space of all hardware groups, which might be meaningful on an architecture that supports switching among groups of similar I/O addresses by changing a base address dynamically. In a simpler implementation, the actual argument corresponding to the parameter name `dev` can also be used to construct the name of a function to call.

The function names are thus suggestive of specific I/O operations, though they have no required semantics:

- `iord` reads a port and returns as the value of the function.
- `iowr` writes val to a port.
- `ioor` ORs `val` into a port (bitwise inclusive OR).
- `ioand` ANDs `val` into a port (bitwise AND).
- `ioxor` XORs `val` into a port (bitwise exclusive OR).

Moreover:

- The suffix `buf` performs the operation with the element `idx` of a buffer.
- The suffix `l` (lowercase L) takes the type of the port as `unsigned long` instead of `unsigned int`.

Similarly, functions whose name begins with `iogroup` operate on hardware groups:

- The suffix `acquire` establishes group as the active hardware group.
- The suffix `release` disestablished group as the active hardware group.
- The suffix `map` maps the dynamic group into the actual hardware group direct.

In this implementation, all functions are masked by macros that follow the pattern:

```
#define _IOHW_CAT(x, y)    x##_##y /* expand arguments and paste */
#define iordbuf(dev, idx)  _IOHW_CAT(dev, brd)(idx)
```

Thus, the first argument (after macro expansion) is pasted onto a suitable suffix to produce the name of the actual function to call. So you can write code such as:

```
#define KBD         kbd   /* root name of keyboard functions */
#define KBD_STATUS  0     /* first of two adjacent ports */
#define KBD_DATA    1     /* second of two adjacent ports */
#define KBD_DONE    0x80  /* DONE status bit */

extern unsigned int kbd_brd(ioindex_t idx);  /* actual driver */

unsigned int getkbd()
    {   /* read keyboard when ready */
    while ((iordbuf(KBD, KBD_STATUS) & KBD_DONE) == 0)
        ;   /* wait until character is present */
    return (iordbuf(KBD, KBD_DATA)); /* read char and clear DONE */
```

All actual driver calls will be to the function (or macro) `kbd_brd`.

# 11.10 <iso646.h>

[Added with Amendment 1]

Include the standard header `<iso646.h>` to provide readable alternatives to certain operators or punctuators. This header is available even in a freestanding implementation.

```
#define and && [Keyword in C++]
#define and_eq &= [Keyword in C++]
#define bitand & [Keyword in C++]
#define bitor | [Keyword in C++]
#define compl ~ [Keyword in C++]
#define not ! [Keyword in C++]
#define not_eq != [Keyword in C++]
#define or || [Keyword in C++]
#define or_eq |= [Keyword in C++]
#define xor ^ [Keyword in C++]
#define xor_eq ^= [Keyword in C++]
```

## 11.10.1 and

```
#define and && [Keyword in C++]
```

The macro yields the operator `&&`.

## 11.10.2 and_eq

```
#define and_eq &= [Keyword in C++]
```

The macro yields the operator `&=`.

## 11.10.3 bitand

```
#define bitand & [Keyword in C++]
```

The macro yields the operator `&`.

## 11.10.4 bitor

```
#define bitor | [Keyword in C++]
```

The macro yields the operator `|`.

## 11.10.5 compl

```
#define compl ~ [Keyword in C++]
```

The macro yields the operator `~`.

### 11.10.6 not

```
#define not ! [Keyword in C++]
```

The macro yields the operator `!`.

### 11.10.7 not_eq

```
#define not_eq != [Keyword in C++]
```

The macro yields the operator `!=`.

### 11.10.8 or

```
#define or || [Keyword in C++]
```

The macro yields the operator `||`.

### 11.10.9 or_eq

```
#define or_eq |= [Keyword in C++]
```

The macro yields the operator `|=`.

### 11.10.10 xor

```
#define xor ^ [Keyword in C++]
```

The macro yields the operator `^`.

### 11.10.11 xor_eq

```
#define xor_eq ^= [Keyword in C++]
```

The macro yields the operator `^=`.

# 11.11 <limits.h>

Include the standard header <limits.h> to determine various properties of the integer type representations. This header is available even in a freestanding implementation.

You can test the values of all these macros in an if directive. (The macros are #if expressions.)

```
#define CHAR_BIT <#if expression >= 8>
#define CHAR_MAX <#if expression >= 127>
#define CHAR_MIN <#if expression <= 0>
#define SCHAR_MAX <#if expression >= 127>
#define SCHAR_MIN <#if expression <= -127>
#define UCHAR_MAX <#if expression >= 255>
#define MB_LEN_MAX <#if expression >= 1>
#define SHRT_MAX <#if expression >= 32,767>
#define SHRT_MIN <#if expression <= -32,767>
#define USHRT_MAX <#if expression >= 65,535>
#define INT_MAX <#if expression >= 32,767>
#define INT_MIN <#if expression <= -32,767>
#define UINT_MAX <#if expression >= 65,535>
#define LONG_MAX <#if expression >= 2,147,483,647>
#define LONG_MIN <#if expression <= -2,147,483,647>
#define ULONG_MAX <#if expression >= 4,294,967,295>
#define LLONG_MAX <#if expression >= 9,223,372,036,854,775,807> [Added
with C99]
#define LLONG_MIN <#if expression <= -9,223,372,036,854,775,807>
[Added with C99]
#define ULLONG_MAX <#if expression >= 18,446,744,073,709,551,615>
[Added with C99]
```

## 11.11.1 CHAR_BIT

```
#define CHAR_BIT <#if expression >= 8>
```

The macro yields the maximum value for the number of bits used to represent an object of type char.

## 11.11.2 CHAR_MAX

```
#define CHAR_MAX <#if expression >= 127>
```

The macro yields the maximum value for type char. Its value is:

- SCHAR_MAX if char represents negative values
- UCHAR_MAX otherwise

### 11.11.3 CHAR_MIN

```
#define CHAR_MIN <#if expression <= 0>
```

The macro yields the minimum value for type `char`. Its value is:

- `SCHAR_MIN` if `char` represents negative values
- Zero otherwise

### 11.11.4 INT_MAX

```
#define INT_MAX <#if expression >= 32,767>
```

The macro yields the maximum value for type `int`.

### 11.11.5 INT_MIN

```
#define INT_MIN <#if expression <= -32,767>
```

The macro yields the minimum value for type `int`.

### 11.11.6 LLONG_MAX

```
#define LLONG_MAX <#if expression >= 9,223,372,036,854,775,807>
```
[Added with C99]

The macro yields the maximum value for type `long long`.

### 11.11.7 LLONG_MIN

```
#define LLONG_MIN <#if expression <= -9,223,372,036,854,775,807>
```
[Added with C99]

The macro yields the minimum value for type `long long`.

### 11.11.8 LONG_MAX

```
#define LONG_MAX <#if expression >= 2,147,483,647>
```

The macro yields the maximum value for type `long`.

### 11.11.9 LONG_MIN

```
#define LONG_MIN <#if expression <= -2,147,483,647>
```

The macro yields the minimum value for type `long`.

### 11.11.10 MB_LEN_MAX

```
#define MB_LEN_MAX <#if expression >= 1>
```

The macro yields the maximum number of characters that constitute a multibyte character in any supported locale. Its value is [3] `MB_CUR_MAX`.

### 11.11.11 SCHAR_MAX

```
#define SCHAR_MAX <#if expression >= 127>
```

The macro yields the maximum value for type `signed char`.

### 11.11.12 SCHAR_MIN

```
#define SCHAR_MIN <#if expression <= -127>
```

The macro yields the minimum value for type `signed char`.

### 11.11.13 SHRT_MAX

```
#define SHRT_MAX <#if expression >= 32,767>
```

The macro yields the maximum value for type `short`.

### 11.11.14 SHRT_MIN

```
#define SHRT_MIN <#if expression <= -32,767>
```

The macro yields the minimum value for type `short`.

### 11.11.15 UCHAR_MAX

```
#define UCHAR_MAX <#if expression >= 255>
```

The macro yields the maximum value for type `unsigned char`.

### 11.11.16 UINT_MAX

```
#define UINT_MAX <#if expression >= 65,535>
```

The macro yields the maximum value for type `unsigned int`.

## 11.11.17 ULLONG_MAX

```
#define ULLONG_MAX <#if expression >= 18,446,744,073,709,551,615>
```
[Added with C99]

The macro yields the maximum value for type `unsigned long long`.

## 11.11.18 ULONG_MAX

```
#define ULONG_MAX <#if expression >= 4,294,967,295>
```

The macro yields the maximum value for type unsigned `long`.

## 11.11.19 USHRT_MAX

```
#define USHRT_MAX <#if expression >= 65,535>
```

The macro yields the maximum value for type `unsigned short`.

# 11.12 <locale.h>

Include the standard header <locale.h> to alter or access properties of the current locale—a collection of culture-specific information. An implementation can define additional macros in this standard header with names that begin with LC_. You can use any of these macro names as the locale category argument (which selects a cohesive subset of a locale) to setlocale.

```
#define LC_ALL <integer constant expression>
#define LC_COLLATE <integer constant expression>
#define LC_CTYPE <integer constant expression>
#define LC_MONETARY <integer constant expression>
#define LC_NUMERIC <integer constant expression>
#define LC_TIME <integer constant expression>
#define NULL <either 0, 0L, or (void *)0> [0 in C++]

struct lconv;

struct lconv *localeconv(void);
char *setlocale(int category, const char *locname);
```

## 11.12.1 LC_ALL

```
#define LC_ALL <integer constant expression>
```

The macro yields the locale category argument value that affects all locale categories.

## 11.12.2 LC_COLLATE

```
#define LC_COLLATE <integer constant expression>
```

The macro yields the locale category argument value that affects the collation functions strcoll andstrxfrm.

## 11.12.3 LC_CTYPE

```
#define LC_CTYPE <integer constant expression>
```

The macro yields the locale category argument value that affects character classification functions, wide-character classification functions, and various multibyte conversion functions.

## 11.12.4 LC_MONETARY

```
#define LC_MONETARY <integer constant expression>
```

The macro yields the locale category argument value that affects monetary information returned by localeconv.

## 11.12.5 LC_NUMERIC

```
#define LC_NUMERIC <integer constant expression>
```

The macro yields the locale category argument value that affects numeric information returned by `localeconv`, including the decimal point used by numeric conversion, read, and write functions.

## 11.12.6 LC_TIME

```
#define LC_TIME <integer constant expression>
```

The macro yields the locale category argument value that affects the time conversion function `strftime`.

## 11.12.7 lconv

```
struct lconv {
    ELEMENT                   "C" LOCALE   LOCALE CATEGORY
    char *decimal_point;      "."          LC_NUMERIC
    char *grouping;           ""           LC_NUMERIC
    char *thousands_sep;      ""           LC_NUMERIC

    char *mon_decimal_point;  ""           LC_MONETARY
    char *mon_grouping;       ""           LC_MONETARY
    char *mon_thousands_sep;  ""           LC_MONETARY

    char *negative_sign;      ""           LC_MONETARY
    char *positive_sign;      ""           LC_MONETARY

    char *currency_symbol;    ""           LC_MONETARY
    char frac_digits;         CHAR_MAX     LC_MONETARY
    char n_cs_precedes;       CHAR_MAX     LC_MONETARY
    char n_sep_by_space;      CHAR_MAX     LC_MONETARY
    char n_sign_posn;         CHAR_MAX     LC_MONETARY
    char p_cs_precedes;       CHAR_MAX     LC_MONETARY
    char p_sep_by_space;      CHAR_MAX     LC_MONETARY
    char p_sign_posn;         CHAR_MAX     LC_MONETARY

    char *int_curr_symbol;    ""           LC_MONETARY
    char int_frac_digits;     CHAR_MAX     LC_MONETARY
    char int_n_cs_precedes;   CHAR_MAX     LC_MONETARY [Added with C99]
    char int_n_sep_by_space;  CHAR_MAX     LC_MONETARY [Added with C99]
    char int_n_sign_posn;     CHAR_MAX     LC_MONETARY [Added with C99]
    char int_p_cs_precedes;   CHAR_MAX     LC_MONETARY [Added with C99]
    char int_p_sep_by_space;  CHAR_MAX     LC_MONETARY [Added with C99]
    char int_p_sign_posn;     CHAR_MAX     LC_MONETARY [Added with C99]
```

The `lconv` structure contains members that describe how to format numeric and monetary values. Functions in the Standard C library use only the `decimal_point` field. The information is otherwise advisory:

- Members of type `pointer to char` all designate C strings.
- Members of type `char` have non-negative values.
- A `char` value of `CHAR_MAX` indicates that a meaningful value is not available in the current locale.

The members shown above can occur in arbitrary order and can be interspersed with additional members. The comment following each member shows its value for the "C" locale, the locale in effect at program startup, followed by the locale category that can affect its value.

A description of each member follows, with an example in parentheses that would be suitable for a USA locale.

- `currency_symbol` – The local currency symbol (`"$"`)
- `decimal_point` – The decimal point for non-monetary values (`"."`)
- `grouping` – The sizes of digit groups for non-monetary values. An empty string calls for no grouping. Otherwise, successive elements of the string describe groups going away from the decimal point:
  - An element value of zero (the terminating NULL character) calls for the previous element value to be repeated indefinitely.
  - An element value of `CHAR_MAX` ends any further grouping (and hence ends the string).

Thus, the array `{3, 2, CHAR_MAX}` calls for a group of three digits, then two, and then whatever remains, as in `9876,54,321`, while `"\3"` calls for repeated groups of three digits, as in `987,654,321`. `("\3")`

- `int_curr_symbol` – The international currency symbol specified by ISO 4217 ("USD")
- `mon_decimal_point` – The decimal point for monetary values (".")
- `mon_grouping` – The sizes of digit groups for monetary values. Successive elements of the string describe groups going away from the decimal point. The encoding is the same as for grouping.
- `mon_thousands_sep` – The separator for digit groups to the left of the decimal point for monetary values (",")
- `negative_sign` – The negative sign for monetary values ("-")
- `positive_sign` – The positive sign for monetary values ("+")
- `thousands_sep` – The separator for digit groups to the left of the decimal point for non-monetary values (",")
- frac_digits – The number of digits to display to the right of the decimal point for monetary values
- `int_frac_digits` – The number of digits to display to the right of the decimal point for international monetary values

- `int_n_cs_precedes` [Added with C99] – Whether the international currency symbol precedes or follows the value for negative monetary values:
  - A value of `0` indicates that the symbol follows the value.
  - A value of `1` indicates that the symbol precedes the value.
- `int_n_sep_by_space` [Added with C99] – Whether the international currency symbol is separated by a space (defined by int_curr_symbol[3]) or by no space from the value for negative monetary values:
  - A value of `0` indicates that no space separates symbol from value.
  - A value of `1` indicates that a space separates symbol, or adjacent symbol and sign, from value.
  - A value of `2` indicates that a space separates symbol from adjacent sign, otherwise a space separates non-adjacent sign from value.
- `int_n_sign_posn` [Added with C99] – The format for negative international monetary values:
  - A value of `0` indicates that parentheses surround the value and the currency symbol.
  - A value of `1` indicates that the negative sign precedes the value and the currency symbol.
  - A value of `2` indicates that the negative sign follows the value and the currency_symbol.
  - A value of 3 indicates that the negative sign immediately precedes the currency symbol.
  - A value of 4 indicates that the negative sign immediately follows the currency_symbol.
- `int_p_cs_precedes` [Added with C99] – Whether the international currency symbol precedes or follows the value for positive monetary values:
  - A value of `0` indicates that the symbol follows the value.
  - A value of `1` indicates that the symbol precedes the value.
- `int_p_sep_by_space` [Added with C99] – Whether the international currency symbol is separated by a space (defined by int_curr_symbol[3]) or by no space from the value for positive monetary values:
  - A value of `0` indicates that no space separates symbol from value.
  - A value of `1` indicates that a space separates symbol, or adjacent symbol and sign, from value.
  - A value of `2` indicates that a space separates symbol from adjacent sign, otherwise a space separates non-adjacent sign from value.

- `int_p_sign_posn` [Added with C99] – The format for positive international monetary values:
  - A value of `0` indicates that parentheses surround the value and the currency symbol.
  - A value of `1` indicates that the positive sign precedes the value and the currency symbol.
  - A value of `2` indicates that the positive sign follows the value and the currency symbol.
  - A value of 3 indicates that the positive sign immediately precedes the currency symbol.
  - A value of 4 indicates that the positive sign immediately follows the currency symbol.
- `n_cs_precedes` – Whether the currency symbol precedes or follows the value for negative monetary values:
  - A value of `0` indicates that the symbol follows the value.
  - A value of `1` indicates that the symbol precedes the value.
- `n_sep_by_space` – Whether the currency symbol is separated by a space or by no space from the value for negative monetary values:
  - A value of `0` indicates that no space separates symbol from value.
  - A value of `1` indicates that a space separates symbol, or adjacent symbol and sign, from value.
  - [Added with C99] A value of `2` indicates that a space separates symbol from adjacent sign, otherwise a space separates non-adjacent sign from value.
- `n_sign_posn` – The format for negative monetary values:
  - A value of `0` indicates that parentheses surround the value and the currency symbol.
  - A value of `1` indicates that the negative sign precedes the value and the currency symbol.
  - A value of `2` indicates that the negative sign follows the value and the currency_symbol.
  - A value of 3 indicates that the negative sign immediately precedes the currency symbol.
  - A value of 4 indicates that the negative sign immediately follows the currency_symbol.
- `p_cs_precedes` – Whether the currency symbol precedes or follows the value for positive monetary values:
  - A value of `0` indicates that the symbol follows the value.
  - A value of `1` indicates that the symbol precedes the value.

- p_sep_by_space – Whether the currency symbol is separated by a space or by no space from the value for positive monetary values:
  - A value of 0 indicates that no space separates symbol from value.
  - A value of 1 indicates that a space separates symbol, or adjacent symbol and sign, from value.
  - [Added with C99] A value of 2 indicates that a space separates symbol from adjacent sign, otherwise a space separates non-adjacent sign from value.
- p_sign_posn – The format for positive monetary values:
  - A value of 0 indicates that parentheses surround the value and the currency symbol.
  - A value of 1 indicates that the positive sign precedes the value and the currency symbol.
  - A value of 2 indicates that the positive sign follows the value and the currency symbol.
  - A value of 3 indicates that the positive sign immediately precedes the currency symbol.
  - A value of 4 indicates that the positive sign immediately follows the currency symbol.

## 11.12.8 localeconv

```
struct lconv *localeconv(void);
```

The function returns a pointer to a static-duration structure containing numeric formatting information for the current locale. You cannot alter values stored in the static-duration structure. The stored values can change on later calls to localeconv or on calls to setlocale that alter any of the categories LC_ALL, LC_MONETARY, or LC_NUMERIC.

## 11.12.9 NULL

```
#define NULL <either 0, 0L, or (void *)0>
```
[0 in C++]

The macro yields a NULL pointer constant that is usable as an address constant expression.

## 11.12.10setlocale

```
char *setlocale(int category, const char *locname);
```

The function either returns a pointer to a `static-duration` string describing a new locale or returns a NULL pointer (if the new locale cannot be selected). The value of category selects one or more locale categories, each of which must match the value of one of the macros defined in this standard header with names that begin with `LC_`.

If `locname` is a NULL pointer, the locale remains unchanged. If `locname` designates the string "C", the new locale is the “C” locale for the locale category specified. If `locname` designates the string `""`, the new locale is the native locale (a default locale presumably tailored for the local culture) for the locale category specified. `locname` can also designate a string returned on an earlier call to `setlocale` or to other strings that the implementation can define.

At program startup, the target environment calls `setlocale(LC_ALL, "C")` before it calls main.

# 11.13 <math.h>

Include the standard header `<math.h>` to declare a number of functions that perform common mathematical operations on real floating point values (of type `float`, `double`, or `long double`).

A domain error occurs when the function is not defined for its input argument value or values. A function can report a domain error by storing the value of `EDOM` in `errno` and returning a particular value defined for each implementation. Or it can raise an invalid floating point exception. The macro `math_errhandling` specifies whether either or both of these approaches is taken.

A range error occurs when the return value of the function is defined but cannot be represented. A function can report a range error by storing the value of `ERANGE` in `errno` and returning one of several values:

- `HUGE_VAL` – If the value of a function returning double is positive and too large in magnitude to represent
- `HUGE_VALF` – If the value of a function returning float is positive and too large in magnitude to represent
- `HUGE_VALL` – If the value of a function returning long double is positive and too large in magnitude to represent
- `-HUGE_VAL` – If the value of a function returning double is negative and too large in magnitude to represent
- `-HUGE_VAL`F – If the value of a function returning float is negative and too large in magnitude to represent
- `-HUGE_VA`LL – If the value of a function returning long double is negative and too large in magnitude to represent
- Zero – If the value of the function is too small to represent with a finite value

Or it can raise an invalid floating point exception. The macro `math_errhandling` specifies whether either or both of these approaches is taken.

The following pragma (added with C99) controls the behavior of real floating point expression contraction:

```
#pragma STD FP_CONTRACT [ON|OFF|DEFAULT]
```

If the parameter is ON, the translator is permitted to evaluate an expression atomically, possibly omitting rounding errors and the raising of floating point exceptions. If the parameter is OFF, contraction is disallowed. The parameter DEFAULT restores the original state, which is implementation defined.

If the pragma occurs outside an external declaration, it remains in effect until overridden by another such pragma.

If the pragma occurs inside an external declaration, it must precede all explicit declarations and statements within a compound statement. It remains in effect until overridden by another such pragma or until the end of the compound statement.

Many of the functions declared in this header have additional overloads in C++, which behave much like the generic functions defined in <tgmath.h>, which behave much like the generic functions defined in <tgmath.h>. The following functions have such additional overloads:

| | | | |
|---|---|---|---|
| acos | exp | lgamma | remquo |
| acosh | exp2 | llrint | rint |
| asin | expm1 | llround | round |
| asinh | fabs | log | scalbln |
| atan | fdim | log10 | scalbn |
| atan2 | floor | log1p | sin |
| atanh | fma | logb | sinh |
| cbrt | fmax | lrint | sqrt |
| ceil | fmin | lround | tan |
| copysign | fmod | nearbyint | tanh |
| cos | frexp | nextafter | tgamma |
| cosh | hypot | nexttoward | trunc |
| erf | ilogb | pow | |
| erfc | ldexp | remainder | |

```
/* MACROS */
#define HUGE_VAL <double rvalue>
#define HUGE_VALF <float rvalue> [Added with C99]
#define HUGE_VALL <long double rvalue> [Added with C99]

#define INFINITY <float rvalue> [Added with C99]
#define NAN <float rvalue> [Added with C99]

#define FP_FAST_FMA <integer constant expression> [Optional with C99]
#define FP_FAST_FMAF <integer constant expression> [Optional with C99]
#define FP_FAST_FMAL <integer constant expression> [Optional with C99]

#define FP_INFINITE <integer constant expression> [Added with C99]
#define FP_NAN <integer constant expression> [Added with C99]
#define FP_NORMAL <integer constant expression> [Added with C99]
#define FP_SUBNORMAL <integer constant expression> [Added with C99]
#define FP_ZERO <integer constant expression> [Added with C99]

#define FP_ILOGB0 <integer constant expression> [Added with C99]
#define FP_ILOGBNAN <integer constant expression> [Added with C99]

#define MATH_ERRNO 1 [Added with C99]
#define MATH_ERREXCEPT 2 [Added with C99]
#define math_errhandling <int rvalue [0, 4)> [Added with C99]

/* TYPES */
typedef f-type double_t; [Added with C99]
typedef f-type float_t; [Added with C99]
```

```
/* GENERIC FUNCTION MACROS [Macros in C, functions in C++] */
#define signbit(x) <int rvalue> [Added with C99, bool funs in C++]

#define fpclassify(x) <int rvalue> [Added with C99, int fns in C++]
#define isfinite(x) <int rvalue> [Added with C99, bool fns in C++]
#define isinf(x) <int rvalue> [Added with C99, bool fns in C++]
#define isnan(x) <int rvalue> [Added with C99, bool fns in C++]
#define isnormal(x) <int rvalue> [Added with C99, bool fns in C++]

#define isgreater(x, y) <int rvalue> [Added C99, bool fns in C++]
#define isgreaterequal(x, y) <int rvalue> [Added C99, bool in C++]
#define islessequal(x, y) <int rvalue> [Added C99, bool fns in C++]
#define islessgreater(x, y) <int rvalue> [Added C99, bool in C++]
#define isunordered(x, y) <int rvalue> [Added C99, bool fns in C++]

/* FUNCTIONS */
double abs(double x); [C++ only]
float abs(float x); [C++ only]
long double abs(long double x); [C++ only]

double acos(double x);
float acos(float x); [C++ only]
long double acos(long double x); [C++ only]
float acosf(float x); [Required with C99]
long double acosl(long double x); [Required with C99]

double asin(double x);
float asin(float x); [C++ only]
long double asin(long double x); [C++ only]
float asinf(float x); [Required with C99]
long double asinl(long double x); [Required with C99]

double atan(double x);
float atan(float x); [C++ only]
long double atan(long double x); [C++ only]
float atanf(float x); [Required with C99]
long double atanl(long double x); [Required with C99]

double atan2(double y, double x);
float atan2(float y, float x); [C++ only]
long double atan2(long double y, long double x); [C++ only]
float atan2f(float y, float x); [Required with C99]
long double atan2l(long double y, long double x); [Required with C99]

double ceil(double x);
float ceil(float x); [C++ only]
long double ceil(long double x); [C++ only]
float ceilf(float x); [Required with C99]
long double ceill(long double x); [Required with C99]

double cos(double x);
float cos(float x); [C++ only]
long double cos(long double x); [C++ only]
float cosf(float x); [Required with C99]
```

```
long double cosl(long double x); [Required with C99]

double cosh(double x);
float cosh(float x); [C++ only]
long double cosh(long double x); [C++ only]
float coshf(float x); [Required with C99]
long double coshl(long double x); [Required with C99]

double exp(double x);
float exp(float x); [C++ only]
long double exp(long double x); [C++ only]
float expf(float x); [Required with C99]
long double expl(long double x); [Required with C99]

double fabs(double x);
float fabs(float x); [C++ only]
long double fabs(long double x); [C++ only]
float fabsf(float x); [Required with C99]
long double fabsl(long double x); [Required with C99]

double floor(double x);
float floor(float x); [C++ only]
long double floor(long double x); [C++ only]
float floorf(float x); [Required with C99]
long double floorl(long double x); [Required with C99]

double fmod(double x, double y);
float fmod(float x, float y); [C++ only]
long double fmod(long double x, long double y); [C++ only]
float fmodf(float x, float y); [Required with C99]
long double fmodl(long double x, long double y); [Required with C99]

double frexp(double x, int *pexp);
float frexp(float x, int *pexp); [C++ only]
long double frexp(long double x, int *pexp); [C++ only]
float frexpf(float x, int *pexp); [Required with C99]
long double frexpl(long double x, int *pexp); [Required with C99]

double ldexp(double x, int ex);
float ldexp(float x, int ex); [C++ only]
long double ldexp(long double x, int ex); [C++ only]
float ldexpf(float x, int ex); [Required with C99]
long double ldexpl(long double x, int ex); [Required with C99]

double log(double x);
float log(float x); [C++ only]
long double log(long double x); [C++ only]
float logf(float x); [Required with C99]
long double logl(long double x); [Required with C99]

double log10(double x);
float log10(float x); [C++ only]
long double log10(long double x); [C++ only]
float log10f(float x); [Required with C99]
```

```
long double log10l(long double x); [Required with C99]

double modf(double x, double *pint);
float modf(float x, float *pint); [C++ only]
long double modf(long double x, long double *pint); [C++ only]
float modff(float x, float *pint); [Required with C99]
long double modfl(long double x, long double *pint); [Required with C99]


double pow(double x, double y);
float pow(float x, float y); [C++ only]
long double pow(long double x, long double y); [C++ only]
double pow(double x, int y); [C++ only]
float pow(float x, int y); [C++ only]
long double pow(long double x, int y); [C++ only]
float powf(float x, float y); [Required with C99]
long double powl(long double x, long double y); [Required with C99]

double sin(double x);
float sin(float x); [C++ only]
long double sin(long double x); [C++ only]
float sinf(float x); [Required with C99]
long double sinl(long double x); [Required with C99]

double sinh(double x);
float sinh(float x); [C++ only]
long double sinh(long double x); [C++ only]
float sinhf(float x); [Required with C99]
long double sinhl(long double x); [Required with C99]

double sqrt(double x);
float sqrt(float x); [C++ only]
long double sqrt(long double x); [C++ only]
float sqrtf(float x); [Required with C99]
long double sqrtl(long double x); [Required with C99]

double tan(double x);
float tan(float x); [C++ only]
long double tan(long double x); [C++ only]
float tanf(float x); [Required with C99]
long double tanl(long double x); [Required with C99]

double tanh(double x);
float tanh(float x); [C++ only]
long double tanh(long double x); [C++ only]
float tanhf(float x); [Required with C99]
long double tanhl(long double x); [Required with C99]

double acosh(double x); [All added with C99]
float acosh(float x); [C++ only]
long double acosh(long double x); [C++ only]
float acoshf(float x);
long double acoshl(long double x);
```

```
double asinh(double x); [All added with C99]
float asinh(float x); [C++ only]
long double asinh(long double x); [C++ only]
float asinhf(float x);
long double asinhl(long double x);

double atanh(double x); [All added with C99]
float atanh(float x); [C++ only]
long double atanh(long double x); [C++ only]
float atanhf(float x);
long double atanhl(long double x);

double cbrt(double x); [All added with C99]
float cbrt(float x); [C++ only]
long double cbrt(long double x); [C++ only]
float cbrtf(float x);
long double cbrtl(long double x);

double copysign(double x, double y); [All added with C99]
float copysign(float x, float y); [C++ only]
long double copysign(long double x, long double y); [C++ only]
float copysignf(float x, float y);
long double copysignl(long double x, long double y);

double erf(double x); [All added with C99]
float erf(float x); [C++ only]
long double erf(long double x); [C++ only]
float erff(float x);
long double erfl(long double x);

double erfc(double x); [All added with C99]
float erfc(float x); [C++ only]
long double erfc(long double x); [C++ only]
float erfcf(float x);
long double erfcl(long double x);

float exp10f(float x);

double exp2(double x); [All added with C99]
float exp2(float x); [C++ only]
long double exp2(long double x); [C++ only]
float exp2f(float x);
long double exp2l(long double x);

double expm1(double x); [All added with C99]
float expm1(float x); [C++ only]
long double expm1(long double x); [C++ only]
float expm1f(float x);
long double expm1l(long double x);

double fdim(double x, double y); [All added with C99]
float fdim(float x, float y); [C++ only]
long double fdim(long double x, long double y); [C++ only]
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
```

```
double fma(double x, double y, double z); [All added with C99]
float fma(float x, float y, float z); [C++ only]
long double fma(long double x, long double y, long double z); [C++ only]
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);

double fmax(double x, double y); [All added with C99]
float fmax(float x, float y); [C++ only]
long double fmax(long double x, long double y); [C++ only]
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);

double fmin(double x, double y); [All added with C99]
float fmin(float x, float y); [C++ only]
long double fmin(long double x, long double y); [C++ only]
float fminf(float x, float y);
long double fminl(long double x, long double y);

double hypot(double x, double y); [All added with C99]
float hypot(float x, float y); [C++ only]
long double hypot(long double x, long double y); [C++ only]
float hypotf(float x, float y);
long double hypotl(long double x, long double y);

int ilogb(double x); [All added with C99]
int ilogb(float x); [C++ only]
int ilogb(long double x); [C++ only]
int ilogbf(float x);
int ilogbl(long double x);

double j0(double x); [POSIX]
double j1(double x); [POSIX]
double jn(int n, double x); [POSIX]

double lgamma(double x); [All added with C99]
float lgamma(float x); [C++ only]
long double lgamma(long double x); [C++ only]
float lgammaf(float x);
long double lgammal(long double x);

long long llrint(double x); [All added with C99]
long long llrint(float x); [C++ only]
long long llrint(long double x); [C++ only]
long long llrintf(float x);
long long llrintl(long double x);

long long llround(double x); [All added with C99]
long long llround(float x); [C++ only]
long long llround(long double x); [C++ only]
long long llroundf(float x);
long long llroundl(long double x);

double log1p(double x); [All added with C99]
float log1p(float x); [C++ only]
```

```
long double log1p(long double x); [C++ only]
float log1pf(float x);
long double log1pl(long double x);

double log2(double x); [All added with C99]
float log2(float x); [C++ only]
long double log2(long double x); [C++ only]
float log2f(float x);
long double log2l(long double x);

double logb(double x); [All added with C99]
float logb(float x); [C++ only]
long double logb(long double x); [C++ only]
float logbf(float x);
long double logbl(long double x);

long lrint(double x); [All added with C99]
long lrint(float x); [C++ only]
long lrint(long double x); [C++ only]
long lrintf(float x);
long lrintl(long double x);

long lround(double x); [All added with C99]
long lround(float x); [C++ only]
long lround(long double x); [C++ only]
long lroundf(float x);
long lroundl(long double x);

double nan(const char *str); [All added with C99]
float nanf(const char *str);
long double nanl(const char *str);

double nearbyint(double x); [All added with C99]
float nearbyint(float x); [C++ only]
long double nearbyint(long double x); [C++ only]
float nearbyintf(float x);
long double nearbyintl(long double x);

double nextafter(double x, double y); [All added with C99]
float nextafter(float x, float y); [C++ only]
long double nextafter(long double x, long double y); [C++ only]
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);

double nexttoward(double x, long double y); [All added with C99]
float nexttoward(float x, long double y); [C++ only]
long double nexttoward(long double x, long double y); [C++ only]
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);

double remainder(double x, double y); [All added with C99]
float remainder(float x, float y); [C++ only]
long double remainder(long double x, long double y); [C++ only]
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
```

```
double remquo(double x, double y, int *pquo); [All added with C99]
float remquo(float x, float y, int *pquo); [C++ only]
long double remquo(long double x, long double y, int *pquo); [C++ only]
float remquof(float x, float y, int *pquo);
long double remquol(long double x, long double y, int *pquo);

double rint(double x); [All added with C99]
float rint(float x); [C++ only]
long double rint(long double x); [C++ only]
float rintf(float x);
long double rintl(long double x);

double round(double x); [All added with C99]
float round(float x); [C++ only]
long double round(long double x); [C++ only]
float roundf(float x);
long double roundl(long double x);

double scalb(double x, double n); [POSIX]

double scalbln(double x, long ex); [All added with C99]
float scalbln(float x, long ex); [C++ only]
long double scalbln(long double x, long ex); [C++ only]
float scalblnf(float x, long ex);
long double scalblnl(long double x, long ex);

double scalbn(double x, int ex); [All added with C99]
float scalbn(float x, int ex); [C++ only]
long double scalbn(long double x, int ex); [C++ only]
float scalbnf(float x, int ex);
long double scalbnl(long double x, int ex);

double tgamma(double x); [All added with C99]
float tgamma(float x); [C++ only]
long double tgamma(long double x); [C++ only]
float tgammaf(float x);
long double tgammal(long double x);

double trunc(double x); [All added with C99]
float trunc(float x); [C++ only]
long double trunc(long double x); [C++ only]
float truncf(float x);
long double truncl(long double x);

double y0(double x); [POSIX]
double y1(double x); [POSIX]
double yn(int n, double x); [POSIX]
```

### 11.13.1 abs, fabs, fabsf, fabsl

```
double abs(double x); [C++ only]
float abs(float x); [C++ only]
long double abs(long double x); [C++ only]
double fabs(double x);
float fabs(float x); [C++ only]
long double fabs(long double x); [C++ only]
float fabsf(float x); [Required with C99]
long double fabsl(long double x); [Required with C99]
```

The function returns the magnitude of `x`, `|x|`.

### 11.13.2 acos, acosf, acosl

```
double acos(double x);
float acos(float x); [C++ only]
long double acos(long double x); [C++ only]
float acosf(float x); [Required with C99]
long double acosl(long double x); [Required with C99]
```

The function returns the angle whose cosine is `x`, in the range `[0, pi]` radians. A domain error occurs if `1 < |x|`.

### 11.13.3 acosh, acoshf, acoshl

```
double acosh(double x); [All added with C99]
float acosh(float x); [C++ only]
long double acosh(long double x); [C++ only]
float acoshf(float x);
long double acoshl(long double x);
```

The function returns the hyperbolic arccosine of `x`, in the range `[0, infinity]`. A domain error occurs if `x < 1`.

### 11.13.4 asin, asinf, asinl

```
double asin(double x);
float asin(float x); [C++ only]
long double asin(long double x); [C++ only]
float asinf(float x); [Required with C99]
long double asinl(long double x); [Required with C99]
```

The function returns the angle whose sine is `x`, in the range `[-pi/2, +pi/2]` radians. A domain error occurs if `1 < |x|`.

## 11.13.5 asinh, asinhf, asinhl

```
double asinh(double x); [All added with C99]
float asinh(float x); [C++ only]
long double asinh(long double x); [C++ only]
float asinhf(float x);
long double asinhl(long double x);
```

The function returns the hyperbolic arcsine of `x`.

## 11.13.6 atan, atanf, atanl

```
double atan(double x);
float atan(float x); [C++ only]
long double atan(long double x); [C++ only]
float atanf(float x); [Required with C99]
long double atanl(long double x); [Required with C99]
```

The function returns the angle whose tangent is `x`, in the range `[-pi/2, +pi/2]` radians.

## 11.13.7 atan2, atan2f, atan2l

```
double atan2(double y, double x);
float atan2(float y, float x); [C++ only]
long double atan2(long double y, long double x); [C++ only]
float atan2f(float y, float x); [Required with C99]
long double atan2l(long double y, long double x); [Required with C99]
```

The function returns the angle whose tangent is `y/x`, in the full angular range `[-pi, +pi]` radians. A domain error might occur if both `x` and `y` are zero.

## 11.13.8 atanh, atanhf, atanhl

```
double atanh(double x); [All added with C99]
float atanh(float x); [C++ only]
long double atanh(long double x); [C++ only]
float atanhf(float x);
long double atanhl(long double x);
```

The function returns the hyperbolic arctangent of `x`. A domain error occurs if `x < -1` or `+1 < x`.

### 11.13.9 cbrt, cbrtf, cbrtl

```
double cbrt(double x); [All added with C99]
float cbrt(float x); [C++ only]
long double cbrt(long double x); [C++ only]
float cbrtf(float x);
long double cbrtl(long double x);
```

The function returns the real cube root of `x,x^(1/3)`.

### 11.13.10 ceil, ceilf, ceill

```
double ceil(double x);
float ceil(float x); [C++ only]
long double ceil(long double x); [C++ only]
float ceilf(float x); [Required with C99]
long double ceill(long double x); [Required with C99]
```

The function returns the smallest integer value not less than `x`.

### 11.13.11 copysign, copysignf, copysignl

```
double copysign(double x, double y); [All added with C99]
float copysign(float x, float y); [C++ only]
long double copysign(long double x, long double y); [C++ only]
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
```

The function returns x, with its sign bit replaced from y.

### 11.13.12 cos, cosf, cosl

```
double cos(double x);
float cos(float x); [C++ only]
long double cos(long double x); [C++ only]
float cosf(float x); [Required with C99]
long double cosl(long double x); [Required with C99]
```

The function returns the cosine of `x`. If x is large the value returned might not be meaningful, but the function reports no error.

### 11.13.13 cosh, coshf, coshl

```
double cosh(double x);
float cosh(float x); [C++ only]
long double cosh(long double x); [C++ only]
float coshf(float x); [Required with C99]
long double coshl(long double x); [Required with C99]
```

The function returns the hyperbolic cosine of `x`.

## 11.13.14double_t

```
typedef f-type double_t; [Added with C99]
```

The type is a synonym for the floating point type f-type, which is one of:

```
double if FLT_EVAL_METHOD is zero
double if FLT_EVAL_METHOD is 1
long double if FLT_EVAL_METHOD is 2
```

Otherwise, `double_t` is a real floating point type at least as wide as `float_t`.

## 11.13.15erf, erff,erfl

```
double erf(double x); [All added with C99]
float erf(float x); [C++ only]
long double erf(long double x); [C++ only]
float erff(float x);
long double erfl(long double x);
```

The function returns the error function of `x`.

## 11.13.16erfc, erfcf, erfcl

```
double erfc(double x); [All added with C99]
float erfc(float x); [C++ only]
long double erfc(long double x); [C++ only]
float erfcf(float x);
long double erfcl(long double x);
```

The function returns the complementary error function of `x`.

## 11.13.17exp, expf, expl

```
double exp(double x);
float exp(float x); [C++ only]
long double exp(long double x); [C++ only]
float expf(float x); [Required with C99]
long double expl(long double x); [Required with C99]
```

The function returns the exponential of `x, e^x`.

## 11.13.18exp10f

```
float exp10f(float x); [Required with C99]
```

The function returns the base-10 exponential of `x`.

## 11.13.19exp2, exp2f, exp2l

```
double exp2(double x); [All added with C99]
float exp2(float x); [C++ only]
long double exp2(long double x); [C++ only]
float exp2f(float x);
long double exp2l(long double x);
```

The function returns two raised to the power x, 2^x.

## 11.13.20expm1, expm1f, expm1l

```
double expm1(double x); [All added with C99]
float expm1(float x); [C++ only]
long double expm1(long double x); [C++ only]
float expm1f(float x);
long double expm1l(long double x);
```

The function returns one less than the exponential function of `x, e^x - 1`.

## 11.13.21fdim, fdimf, fdiml

```
double fdim(double x, double y); [All added with C99]
float fdim(float x, float y); [C++ only]
long double fdim(long double x, long double y); [C++ only]
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
```

The function returns the larger of `x - y` and zero.

## 11.13.22float_t

```
typedef f-type float_t; [Added with C99]
```

The type is a synonym for the floating point type f-type, which is one of:

```
float if FLT_EVAL_METHOD is zero
double if FLT_EVAL_METHOD is 1
long double if FLT_EVAL_METHOD is 2
```

Otherwise, `float_t` is a real floating point type not wider than `double_t`.

## 11.13.23floor, floorf, floorl

```
double floor(double x);
float floor(float x); [C++ only]
long double floor(long double x); [C++ only]
float floorf(float x); [Required with C99]
long double floorl(long double x); [Required with C99]
```

The function returns the largest integer value not greater than `x`.

## 11.13.24fma, fmaf,fmal

```
double fma(double x, double y, double z); [All added with C99]
float fma(float x, float y, float z); [C++ only]
long double fma(long double x, long double y, long double z); [C++ only]
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);
```

The function returns `x * y + z`, to arbitrary intermediate precision.

## 11.13.25fmax, fmaxf, fmaxl

```
double fmax(double x, double y); [All added with C99]
float fmax(float x, float y); [C++ only]
long double fmax(long double x, long double y); [C++ only]
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
```

The function returns the larger (more positive) of `x` and `y`.

## 11.13.26fmin, fminf, fminl

```
double fmin(double x, double y); [All added with C99]
float fmin(float x, float y); [C++ only]
long double fmin(long double x, long double y); [C++ only]
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

The function returns the smaller (more negative) of `x` and `y`

## 11.13.27fmod, fmodf, fmodl

```
double fmod(double x, double y);
float fmod(float x, float y); [C++ only]
long double fmod(long double x, long double y); [C++ only]
float fmodf(float x, float y); [Required with C99]
long double fmodl(long double x, long double y); [Required with C99]
```

The function returns the remainder of `x/y`, which is defined as follows:

- If `y` is zero, the function either reports a domain error or simply returns zero.
- Otherwise, the function determines the unique signed integer value `i` such that the returned value `x - i * y` has the same sign as x and magnitude less than `|y|`.

## 11.13.28fpclassify

`#define fpclassify(x) <int rvalue>` [Added with C99, int functions in C++]

The generic-function macro accepts an `rvalue` argument `x` of some real floating point type and evaluates to:

- `FP_INFINITE` for an argument that is positive or negative infinity
- `FP_NAN` for an argument that is not-a-number (NaN)
- `FP_NORMAL` for an argument that is finite and normalized
- `FP_SUBNORMAL` for an argument that is finite and denormalized
- `FP_ZERO` for an argument that is positive or negative zero

Or possibly some other implementation-defined value.

## 11.13.29FP_FAST_FMA

`#define FP_FAST_FMA <integer constant expression>` [Optional with C99]

The macro is defined only if the call `fma(x, y, z)` executes as fast as the double expression `x * y + z`.

## 11.13.30FP_FAST_FMAF

`#define FP_FAST_FMAF <integer constant expression>` [Optional with C99]

The macro is defined only if the call `fmaf(x, y, z)` executes as fast as the float expression `x * y + z`.

## 11.13.31FP_FAST_FMAL

`#define FP_FAST_FMAL <integer constant expression>` [Optional with C99]

The macro is defined only if the call `fmal(x, y, z)` executes as fast as the long double expression `x * y + z`.

## 11.13.32FP_ILOGB0

`#define FP_ILOGB0 <integer constant expression>` [Added with C99]

The macro defines the value returned by `ilogb` for an argument that is positive or negative zero. The value of the macro is either `INT_MIN` or `-INT_MAX`.

## 11.13.33FP_ILOGBNAN

`#define FP_ILOGBNAN <integer constant expression>` [Added with C99]

The macro defines the value returned by `ilogb` for an argument that is not-a-number (NaN). The value of the macro is either `INT_MIN` or `INT_MAX`.

## 11.13.34FP_INFINITE

`#define FP_INFINITE <integer constant expression>` [Added with C99]

The macro defines the value of the macro `fpclassify` for an argument that is positive or negative infinity.

## 11.13.35FP_NAN

`#define FP_NAN <integer constant expression>` [Added with C99]

The macro defines the value of the macro `fpclassify` for an argument that is not-a-number (NaN).

## 11.13.36FP_NORMAL

`#define FP_NORMAL <integer constant expression>` [Added with C99]

The macro defines the value of the macro `fpclassify` for an argument that is finite and normalized.

## 11.13.37FP_SUBNORMAL

`#define FP_SUBNORMAL <integer constant expression>` [Added with C99]

The macro defines the value of the macro `fpclassify` for an argument that is finite and denormalized.

## 11.13.38FP_ZERO

`#define FP_ZERO <integer constant expression>` [Added with C99]

The macro defines the value of the macro `fpclassify` for an argument that is positive or negative zero.

## 11.13.39frexp, frexpf, frexpl

```
double frexp(double x, int *pexp);
float frexp(float x, int *pexp); [C++ only]
long double frexp(long double x, int *pexp); [C++ only]
float frexpf(float x, int *pexp); [Required with C99]
long double frexpl(long double x, int *pexp); [Required with C99]
```

The function determines a fraction `frac` and an exponent integer `ex` that represent the value of `x`. It returns the value `frac` and stores the integer `ex` in `*pexp`, such that:

- `|frac|` is in the interval `[1/2, 1)` or is zero
- `x == frac * 2^ex`

If `x` is zero, `*pexp` is also zero.

## 11.13.40HUGE_VAL

```
#define HUGE_VAL <double rvalue>
```

The macro yields the double value returned by some functions on a range error. The value can be a representation of infinity.

## 11.13.41HUGE_VALF

```
#define HUGE_VALF <float rvalue> [Added with C99]
```

The macro yields the float value returned by some functions on a range error. The value can be a representation of infinity.

## 11.13.42HUGE_VALL

```
#define HUGE_VALL <long double rvalue> [Added with C99]
```

The macro yields the long double value returned by some functions on a range error. The value can be a representation of infinity.

## 11.13.43hypot, hypotf, hypotl

```
double hypot(double x, double y); [All added with C99]
float hypot(float x, float y); [C++ only]
long double hypot(long double x, long double y); [C++ only]
float hypotf(float x, float y);
long double hypotl(long double x, long double y);
```

The function returns the square root of `x^2 + y^2`.

## 11.13.44ilogb, ilogbf, ilogbl

```
int ilogb(double x); [All added with C99]
int ilogb(float x); [C++ only]
int ilogb(long double x); [C++ only]
int ilogbf(float x);
int ilogbl(long double x);
```

The function returns:

- For `x` not-a-number (NaN), the value of the macro `FP_ILOGBNAN`
- For `x` equal to zero, the value of the macro `FP_ILOGB0`
- For `x` equal to positive or negative infinity, the value of the macro `INT_MAX`

Otherwise, it returns `(int)logb(x)`.

## 11.13.45INFINITY

```
#define INFINITY <float rvalue> [Added with C99]
```

The macro yields a float value that represents positive infinity.

## 11.13.46isfinite

```
#define isfinite(x) <int rvalue> [Added with C99, bool fns in C++]
```

The generic-function macro accepts an `rvalue` argument `x` of some real floating point type and yields a nonzero value only if `x` is finite.

## 11.13.47isgreater

```
#define isgreater(x, y) <int rvalue> [Added C99, bool fns in C++]
```

The generic-function macro accepts two `rvalue` arguments `x` and `y`, at least one of which is a real floating point type, and yields the value `1` only if `x > y` and neither `x` nor `y` is not-a-number (NaN). Otherwise, it yields the value zero. The macro never raises an invalid floating point exception.

## 11.13.48isgreaterequal

```
#define isgreaterequal(x, y) <int rvalue> [added C99, bool fns C++]
```

The generic-function macro accepts two `rvalue` arguments `x` and `y`, at least one of which is a real floating point type, and yields the value `1` only if `x >= y` and neither `x` nor `y` is not-a-number (NaN). Otherwise, it yields the value zero. The macro never raises an invalid floating point exception.

## 11.13.49isinf

`#define isinf(x) <int rvalue>` [Added with C99, bool fns in C++]

The isinf macro determines whether its argument `x` is an infinity (positive or negative). An argument represented in a format wider than its semantic type is converted to its semantic type first. The determination is then based on the type of the argument.

This macro returns a nonzero value if the value of `x` is an infinity. Otherwise, the macro returns zero.

## 11.13.50isless

`#define isless(x, y) <int rvalue>` [Added with C99, bool functions in C++]

The generic-function macro accepts two `rvalue` arguments `x` and `y`, at least one of which is a real floating point type, and yields the value `1` only if `x < y` and neither `x` nor `y` is not-a-number (NaN). Otherwise, it yields the value zero. The macro never raises an invalid floating point exception.

## 11.13.51islessequal

`#define islessequal(x, y) <int rvalue>` [Added with C99, bool functions in C++]

The generic-function macro accepts two `rvalue` arguments `x` and `y`, at least one of which is a real floating point type, and yields the value `1` only if `x <= y` and neither `x` nor `y` is not-a-number (NaN). Otherwise, it yields the value zero. The macro never raises an invalid floating point exception.

## 11.13.52islessgreater

`#define islessgreater(x, y) <int rvalue>` [Added with C99, bool functions in C++]

The generic-function macro accepts two `rvalue` arguments `x` and `y`, at least one of which is a real floating point type, and yields the value `1` only if `x < y || x > y` and neither `x` nor `y` is not-a-number (NaN). Otherwise, it yields the value zero. The macro never raises an invalid floating point exception.

## 11.13.53isnan

`#define isnan(x) <int rvalue>` [Added with C99, bool fns in C++]

The macro determines whether its argument `x` is not-a-number (NaN). An argument represented in a format wider than its semantic type is converted to its semantic type first. The determination is then based on the type of the argument.

The macro returns a nonzero value if the value of `x` is NaN. Otherwise 0 is returned.

## 11.13.54isnormal

`#define isnormal(x) <int rvalue>` [Added with C99, bool functions in C++]

The generic-function macro accepts an `rvalue` argument `x` of some real floating point type and yields a nonzero value only if x is finite and normalized.

## 11.13.55isunordered

`#define isunordered(x, y) <int rvalue>` [Added with C99, bool functions in C++]

The generic-function macro accepts two `rvalue` arguments `x` and `y`, at least one of which is a real floating point type, and yields the value `1` only if at least one of the two arguments is not-a-number (NaN). Otherwise, it yields the value zero. The macro never raises an invalid floating point exception.

## 11.13.56j0

`double j0(double x);` [POSIX]

The macro computes the Bessel function of the first kind of the order `0` for the real value x;

If successful, the function returns the computed value, otherwise `errno` is set to EDOM and a reserve operand fault is generated.

## 11.13.57j1

`double j1(double x);` [POSIX]

The macro computes the Bessel function of the first kind of the order `1` for the real value x;

If successful, the function returns the computed value, otherwise `errno` is set to EDOM and a reserve operand fault is generated.

## 11.13.58jn

`double jn(int n, double x);` [POSIX]

The macro computes the Bessel function of the first kind of the integer order `n` for the real value `x`.

If successful, the function returns the computed value, otherwise `errno` is set to EDOM and a reserve operand fault is generated.

## 11.13.59 ldexp, ldexpf, ldexpl

```
double ldexp(double x, int ex);
float ldexp(float x, int ex); [C++ only]
long double ldexp(long double x, int ex); [C++ only]
float ldexpf(float x, int ex); [Required with C99]
long double ldexpl(long double x, int ex); [Required with C99]
```

The function returns `x * 2^ex`.

## 11.13.60 lgamma, lgammaf, lgammal

```
double lgamma(double x); [All added with C99]
float lgamma(float x); [C++ only]
long double lgamma(long double x); [C++ only]
float lgammaf(float x);
long double lgammal(long double x);
```

The function returns the natural logarithm of the absolute value of the gamma function of `x`.

## 11.13.61 llrint, llrintf, llrintl

```
long long llrint(double x); [All added with C99]
long long llrint(float x); [C++ only]
long long llrint(long double x); [C++ only]
long long llrintf(float x);
long long llrintl(long double x);
```

The function returns the nearest `long long` integer to `x`, consistent with the current rounding mode.

- It raises an invalid floating point exception if the magnitude of the rounded value is too large to represent.
- It raises an inexact floating point exception if the return value does not equal `x`.

## 11.13.62 llround, llroundf, llroundl

```
long long llround(double x); [All added with C99]
long long llround(float x); [C++ only]
long long llround(long double x); [C++ only]
long long llroundf(float x);
long long llroundl(long double x);
```

The function returns the nearest `long long` integer to `x`, rounding halfway values away from zero, regardless of the current rounding mode.

## 11.13.63log, logf, logl

```
double log(double x);
float log(float x); [C++ only]
long double log(long double x); [C++ only]
float logf(float x); [Required with C99]
long double logl(long double x); [Required with C99]
```

The function returns the natural logarithm of `x`. A domain error occurs if `x < 0`.

## 11.13.64log10, log10f, log10l

```
double log10(double x);
float log10(float x); [C++ only]
long double log10(long double x); [C++ only]
float log10f(float x); [Required with C99]
long double log10l(long double x); [Required with C99]
```

The function returns the base-10 logarithm of `x`. A domain error occurs if `x < 0`.

## 11.13.65log1p, log1pf, log1pl

```
double log1p(double x); [All added with C99]
float log1p(float x); [C++ only]
long double log1p(long double x); [C++ only]
float log1pf(float x);
long double log1pl(long double x);
```

The function returns the natural logarithm of `1 + x`. A domain error occurs if `x < -1`.

## 11.13.66log2, log2f, log2l

```
double log2(double x); [All added with C99]
float log2(float x); [C++ only]
long double log2(long double x); [C++ only]
float log2f(float x);
long double log2l(long double x);
```

The function returns the base-2 logarithm of `x`. A domain error occurs if `x < 0`.

## 11.13.67 logb, logbf, logbl

```
double logb(double x); [All added with C99]
float logb(float x); [C++ only]
long double logb(long double x); [C++ only]
float logbf(float x);
long double logbl(long double x);
```

The function determines an integer exponent ex and a fraction `frac` that represent the value of a finite `x`. It returns the value `ex` such that:

- `x == frac * FLT_RADIX^ex`.
- `|frac|` is in the interval `[1, FLT_RADIX)`.

A domain error might occur if `x` is zero.

## 11.13.68 lrint, lrintf, lrintl

```
long lrint(double x); [All added with C99]
long lrint(float x); [C++ only]
long lrint(long double x); [C++ only]
long lrintf(float x);
long lrintl(long double x);
```

The function returns the nearest long integer to `x`, consistent with the current rounding mode.

- It raises an invalid floating point exception if the magnitude of the rounded value is too large to represent.
- it raises an inexact floating point exception if the return value does not equal `x`.

## 11.13.69 lround, lroundf, lroundl

```
long lround(double x); [All added with C99]
long lround(float x); [C++ only]
long lround(long double x); [C++ only]
long lroundf(float x);
long lroundl(long double x);
```

The function returns the nearest long integer to `x`, rounding halfway values away from zero, regardless of the current rounding mode.

## 11.13.70 MATH_ERRNO

```
#define MATH_ERRNO 1 [Added with C99]
```

The macro yields the value `1`. It is used for testing the value of the macromath_errhandling to determine whether a math function reports an error by storing a nonzero value in `errno`.

## 11.13.71MATH_ERREXCEPT

```
#define MATH_ERREXCEPT 2 [Added with C99]
```

The macro yields the value `2`. It is used for testing the value of the macro `math_errhandling` to determine whether a math function reports an error by raising an invalid floating point exception.

## 11.13.72math_errhandling

```
#define math_errhandling <int rvalue [0, 4)> [Added with C99]
```

The macro specifies how math functions report a domain error or a range error. Specifically:

- If `(math_errhandling & MATH_ERRNO) != 0`, the math function stores a nonzero value in `errno` and returns a specific value that characterizes the error.
- If `(math_errhandling & MATH_ERREXCEPT) != 0`, the math function raises an invalid floating point exception. In this case, the macros `FE_DIVBYZERO`, `FE_INVALID`, and `FE_OVERFLOW` are all defined.

The value of the macro remains unchanged during program execution.

## 11.13.73modf, modff, modfl

```
double modf(double x, double *pint);
float modf(float x, float *pint); [C++ only]
long double modf(long double x, long double *pint); [C++ only]
float modff(float x, float *pint); [Required with C99]
long double modfl(long double x, long double *pint); [Required with C99]
```

The function determines an integer `i` plus a fraction `frac` that represent the value of `x`. It returns the value frac and stores the integer `i` in `*pint`, such that:

- `x == fraC + I`
- `|frac|` is in the interval `[0, 1)`

Both `frac` and `i` have the same sign as `x`.

## 11.13.74NAN

```
#define NAN <float rvalue> [Added with C99]
```

The macro yields a float value that represents not-a-number (NaN).

## 11.13.75nan, nanf,nanl

```
double nan(const char *str); [All added with C99]
float nanf(const char *str);
long double nanl(const char *str);
```

The function converts a NULL-terminated sequence beginning at `str` to a not-a-number (NaN) code. Specifically, the `callnan("n-char-seq")` effectively returns `strtod("NAN(n-char-seq)", (char**)0)` if the conversion succeeds; otherwise it returns `strtod("NAN")`.

## 11.13.76nearbyint, nearbyintf, nearbyintl

```
double nearbyint(double x); [All added with C99]
float nearbyint(float x); [C++ only]
long double nearbyint(long double x); [C++ only]
float nearbyintf(float x);
long double nearbyintl(long double x);
```

The function returns `x` rounded to the nearest integer, consistent with the current rounding mode but without raising an inexact floating point exception.

## 11.13.77nextafter, nextafterf, nextafterl

```
double nextafter(double x, double y); [All added with C99]
float nextafter(float x, float y); [C++ only]
long double nextafter(long double x, long double y); [C++ only]
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
```

The function returns:

- If `x < y`, the next representable value after `x`
- If `x == y`, `y`
- If `x > y`, the next representable value before `x`

## 11.13.78nexttoward, nexttowardf, nexttowardl

```
double nexttoward(double x, long double y); [All added with C99]
float nexttoward(float x, long double y); [C++ only]
long double nexttoward(long double x, long double y); [C++ only]
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

The function returns:

- If `x < y`, the next representable value after `x`
- If `x == y`, `y`
- If `x > y`, the next representable value before `x`

## 11.13.79 pow, powf, powl

```
double pow(double x, double y);
float pow(float x, float y); [C++ only]
long double pow(long double x, long double y); [C++ only]
double pow(double x, int y); [C++ only]
float pow(float x, int y); [C++ only]
long double pow(long double x, int y); [C++ only]
float powf(float x, float y); [Required with C99]
long double powl(long double x, long double y); [Required with C99]
```

The function returns x raised to the power `y, x^y`.

## 11.13.80 remainder, remainderf, remainderl

```
double remainder(double x, double y); [All added with C99]
float remainder(float x, float y); [C++ only]
long double remainder(long double x, long double y); [C++ only]
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
```

The function effectively returns `remquo(x, y, &temp)`, where temp is a temporary object of type `int` local to the function.

## 11.13.81 remquo, remquof, remquol

```
double remquo(double x, double y, int *pquo); [All added with C99]
float remquo(float x, float y, int *pquo); [C++ only]
long double remquo(long double x, long double y, int *pquo); [C++ only]
float remquof(float x, float y, int *pquo);
long double remquol(long double x, long double y, int *pquo);
```

The function computes the remainder `rem == x - n*y`, where `n == x/y` rounded to the nearest integer, or to the nearest even integer if `|n - x/y| == 1/2`. If `rem` is zero, it has the same sign as `x`. A domain error occurs if `y` is zero.

The function stores in `*pquo` at least three of the low-order bits of `|x/y|`, negated if `x/y < 0`. It returns `rem`.

## 11.13.82 rint, rintf, rintl

```
double rint(double x); [All added with C99]
float rint(float x); [C++ only]
long double rint(long double x); [C++ only]
float rintf(float x); long double rintl(long double x);
```

The function returns `x` rounded to the nearest integer, using the current rounding mode. It might raise an inexact floating point exception if the return value does not equal `x`.

## 11.13.83round, roundf, roundl

```
double round(double x); [All added with C99]
float round(float x); [C++ only]
long double round(long double x); [C++ only]
float roundf(float x);
long double roundl(long double x);
```

The function returns `x` rounded to the nearest integer n, or to the value with larger magnitude if `|n - x| == 1/2`.

## 11.13.84scalb

```
double scalb(double x, double n); [POSIX]
```

This function allows users to test conformance to IEEE Std. 754-1985. Its use is not otherwise recommended.

## 11.13.85scalbln, scalblnf, scalblnl

```
double scalbln(double x, long ex); [All added with C99]
float scalbln(float x, long ex); [C++ only]
long double scalbln(long double x, long ex); [C++ only]
float scalblnf(float x, long ex);
long double scalblnl(long double x, long ex);
```

The function returns `x * FLT_RADIX^ex`.

## 11.13.86scalbn, scalbnf, scalbnl

```
double scalbn(double x, int ex); [All added with C99]
float scalbn(float x, int ex); [C++ only]
long double scalbn(long double x, int ex); [C++ only]
float scalbnf(float x, int ex);
long double scalbnl(long double x, int ex);
```

The function returns `x * FLT_RADIX^ex`.

## 11.13.87signbit

```
#define signbit(x) <int rvalue> [Added with C99, bool functions in C++]
```

The generic-function macro accepts an `rvalue` argument `x` of some real floating point type and yields a nonzero value only if the (negative) sign bit of `x` is set. The macro never raises an invalid floating point exception.

## 11.13.88sin, sinf, sinl

```
double sin(double x);
float sin(float x); [C++ only]
long double sin(long double x); [C++ only]
float sinf(float x); [Required with C99]
long double sinl(long double x); [Required with C99]
```

The function returns the sine of x. If x is large the value returned might not be meaningful, but the function reports no error.

## 11.13.89sinh, sinhf, sinhl

```
double sinh(double x);
float sinh(float x); [C++ only]
long double sinh(long double x); [C++ only]
float sinhf(float x); [Required with C99]
long double sinhl(long double x); [Required with C99]
```

The function returns the hyperbolic sine of x.

## 11.13.90sqrt, sqrtf, sqrtl

```
double sqrt(double x);
float sqrt(float x); [C++ only]
long double sqrt(long double x); [C++ only]
float sqrtf(float x); [Required with C99]
long double sqrtl(long double x); [Required with C99]
```

The function returns the real square root of x,x^(1/2). A domain error occurs if x < 0.

## 11.13.91tan, tanf, tanl

```
double tan(double x); float tan(float x); [C++ only]
long double tan(long double x); [C++ only]
float tanf(float x); [Required with C99]
long double tanl(long double x); [Required with C99]
```

The function returns the tangent of x. If x is large the value returned might not be meaningful, but the function reports no error.

## 11.13.92tanh, tanhf, tanhl

```
double tanh(double x);
float tanh(float x); [C++ only]
long double tanh(long double x); [C++ only]
float tanhf(float x); [Required with C99]
long double tanhl(long double x); [Required with C99]
```

The function returns the hyperbolic tangent of x.

## 11.13.93tgamma, tgammaf, tgammal

```
double tgamma(double x); [All added with C99]
float tgamma(float x); [C++ only]
long double tgamma(long double x); [C++ only]
float tgammaf(float x);
long double tgammal(long double x);
```

The function computes the gamma function of x. A domain error occurs if x is a negative integer.

## 11.13.94trunc, truncf, truncl

```
double trunc(double x); [All added with C99]
float trunc(float x); [C++ only]
long double trunc(long double x); [C++ only]
float truncf(float x);
long double truncl(long double x);
```

The function returns x rounded to the nearest integer n not larger in magnitude than x (toward zero).

## 11.13.95y0

```
double y0(double x); [POSIX]
```

The function computes the linearly independent Bessel function of the second kind of the order 0 for the positive integer value x (expressed as a double). If successful, the function returns the computed value, otherwise errno is set to EDOM and a reserve operand fault is generated.

## 11.13.96y1

```
double y1(double x); [POSIX]
```

The function computes the linearly independent Bessel function of the second kind of the order 1 for the positive integer value x (expressed as a double). If successful, the function returns the computed value, otherwise errno is set to EDOM and a reserve operand fault is generated.

## 11.13.97yn

```
double yn(int n, double x); [POSIX]
```

The function computes the Bessel function of the second kind for the integer order n for the positive integer value x (expressed as a double). If successful, the function returns the computed value, otherwise errno is set to EDOM and a reserve operand fault is generated.

# 11.14 <search.h>

The <search.h> header file contains functions to manipulate binary search tables and manage hash search tables.

```
#include <search.h>

int hcreate(size_t nel); [POSIX]
void hdestroy(void); [POSIX]
ENTRY *hsearch(ENTRY item, ACTION action); [POSIX]

void tdelete(const void * restrict key, void ** restrict rootp, int
(*compar) (const void *, const void *)); [POSIX]
void *tfind(const void *key, const void * const *rootp,
    int (*compar) (const void *, const void *)); [POSIX]
void *tsearch(const void *key, void **rootp, int (*compar)
    (const void *, const void *)); [POSIX]
void twalk(const void *root, void (*action) (const void *,
    VISIT, int)); [POSIX]
```

## 11.14.1 hcreate

```
int hcreate(size_t nel); [POSIX]
```

The function allocates and initializes the hash search table. The nel argument specifies an estimate of the maximum number of entries to be held by the table. Unless further memory allocation fails, supplying an insufficient nel value will not result in functional harm, although a performance degradation may occur.

Initialization using the hcreate function is mandatory prior to any access operations using hsearch.

If successful, hcreate returns a nonzero value. Otherwise, a value of 0 is returned and errno is set to indicate the error.

## 11.14.2 hdestroy

```
void hdestroy(void); [POSIX]
```

The function destroys a table previously created using hcreate. After a call to hdestroy, the data can no longer be accessed.

The hdestroy function returns no value.

### 11.14.3 hsearch

```
ENTRY *hsearch(ENTRY item, ACTION action); [POSIX]
```

Use this function to search the hash table. It returns a pointer into the hash table indicating the address of an item. The item argument is of type `ENTRY`, a structural type which contains the following members:

- `char *key` – Comparison key
- `void *data` – Pointer to data associated with the key

The key comparison function used by `hsearch` is `strcmp`

The action argument is of type `ACTION`, an enumeration type which defines the following values:

- `ENTER` – Insert item into the hash table. If an existing item with the same key is found, it is not replaced. The key and data elements of item are used directly by the new table entry. The storage for the key must not be modified during the lifetime of the hash table.
- `FIND` – Search the hash table without inserting item.

If successful, `hsearch` returns a pointer to the hash table entry matching the provided key. If the action is `FIND` and the item was not found, or if the action is `ENTER` and the insertion failed, `NULL` is returned and `errno` is set to indicate the error. If the action is `ENTER` and an entry already exists in the table matching the given key, the existing entry is returned and is not replaced.

### 11.14.4 tdelete

```
void tdelete(const void * restrict key, void ** restrict rootp, int
(*compar) (const void *, const void *)); [POSIX]
```

The function manages binary search trees based on algorithms T and D from Knuth (6.2.2). The comparison function passed in by the user has the same style of return values as `strcmp`.

The `tdelete` function deletes a node from the specified binary search tree and returns a pointer to the parent of the node to be deleted. It takes the same arguments as `tfind` and `tsearch`. If the node to be deleted is the root of the binary search tree, `rootp` will be adjusted.

`tdelete` returns `NULL` if `rootp` is `NULL` or the datum cannot be found.

### 11.14.5 tfind

```
void *tfind(const void *key, const void * const *rootp,
    int (*compar) (const void *, const void *)); [POSIX]
```

The function manages binary search trees based on algorithms T and D from Knuth (6.2.2). The comparison function passed in by the user has the same style of return values as `strcmp`.

The `tfind` function searches for the datum matched by the argument key in the binary tree rooted at `rootp`. It returns a pointer to the datum if it is found and NULL if it is not. It also returns NULL if `rootp` is NULL or the datum cannot be found.

### 11.14.6 tsearch

```
void *tsearch(const void *key, void **rootp, int (*compar)
    (const void *, const void *)); [POSIX]
```

The function manages binary search trees based on algorithms T and D from Knuth (6.2.2). The comparison function passed in by the user has the same style of return values as `strcmp`.

The `tsearch` function is identical to `tfind` except that if no match is found, key is inserted into the tree and a pointer to it is returned. If `rootp` points to a NULL value a new binary search tree is created.

The function returns NULL if allocation of a new node fails (typically due to a lack of free memory). It also returns NULL if `rootp` is NULL or the datum cannot be found.

### 11.14.7 twalk

```
void twalk(const void *root, void (*action) (const void *,
    VISIT, int)); [POSIX]
```

The function manages binary search trees based on algorithms T and D from Knuth (6.2.2). The comparison function passed in by the user has the same style of return values as `strcmp`.

The `twalk` function walks the binary search tree rooted in root and calls the function action on each node. Action is called with three arguments:

- A pointer to the current node
- A value from the `typedef enum {preorder, postorder, endorder, leaf } VISIT;` specifying the traversal type
- A node level (where level zero is the root of the tree)

The function has no return value.

# 11.15 <setjmp.h>

Include the standard header <setjmp.h> to perform control transfers that bypass the normal function call and return protocol.

```
#define setjmp(jmp_buf env) <int rvalue>

typedef a-type jmp_buf;
void longjmp(jmp_buf env, int val);
jmp_buf
typedef a-type jmp_buf;
```

The type is the array type a-type of an object that you declare to hold the context information stored by setjmp and accessed by longjmp.

## 11.15.1 longjmp

```
void longjmp(jmp_buf env, int val);
```

The function causes a second return from the execution of setjmp that stored the current context value in env. If val is nonzero, the return value is val; otherwise, it is 1.

The function that was active when setjmp stored the current context value must not have returned control to its caller. An object with dynamic duration that does not have a volatile type and whose stored value has changed since the current context value was stored will have a stored value that is indeterminate.

## 11.15.2 setjmp

```
#define setjmp(jmp_buf env) <int rvalue>
```

The macro stores the current context value in the array designated by env and returns zero. A later call to longjmp that accesses the same context value causes setjmp to again return, this time with a nonzero value. You can use the macro setjmp only in an expression that:

- Has no operators
- Has only the unary operator, !
- Has one of the relational or equality operators (==, !=, <, <=, >, or >=) with the other operand an integer constant expression

You can write such an expression only as the expression part of a do, expression, for, if, if-else, switch, or while statement.

# 11.16 <signal.h>

Include the standard header `<signal.h>` to specify how the program handles signals while it executes. A signal can report some exceptional behavior within the program, such as division by zero. Or a signal can report some asynchronous event outside the program, such as someone striking an interactive attention key on a keyboard.

You can report any signal by calling `raise`. Each implementation defines what signals it generates (if any) and under what circumstances it generates them. An implementation can define signals other than the ones listed here. The `<signal.h>` header can define additional macros with names beginning with SIG to specify the values of additional signals. All such values are integer constant expressions [3]0.

You can specify a signal handler for each signal. A signal handler is a function that the target environment calls when the corresponding signal occurs. The target environment suspends execution of the program until the signal handler returns or calls `longjmp`. For maximum portability, an asynchronous signal handler should only:

- Make calls (that succeed) to the function `signal`
- Assign values to objects of type `volatile sig_atomic_t`
- Return control to its caller

Furthermore, in C++, a signal handler should:

- Have extern “C” linkage
- Use only language features common to C and C++

If the signal reports an error within the program (and the signal is not asynchronous), the signal handler can terminate by calling `abort`, `exit`, or `longjmp`.

```
/* MACROS */
#define SIGABRT <integer constant expression >= 0>
#define SIGFPE <integer constant expression >= 0>
#define SIGILL <integer constant expression >= 0>
#define SIGINT <integer constant expression >= 0>
#define SIGSEGV <integer constant expression >= 0>
#define SIGTERM <integer constant expression >= 0>
#define SIG_DFL <address constant expression>
#define SIG_ERR <address constant expression>
#define SIG_IGN <address constant expression>

/* TYPES */
typedef i-type sig_atomic_t;

/* FUNCTIONS */
int raise(int sig);
void (*signal(int sig, void (*func)(int)))(int);
```

### 11.16.1 raise

```
int raise(int sig);
```

The function sends the signal `sig` and returns zero if the signal is successfully reported.

### 11.16.2 sig_atomic_t

```
typedef i-type sig_atomic_t;
```

The type is the integer type `i-type` for objects whose stored value is altered by an assigning operator as an atomic operation (an operation that never has its execution suspended while partially completed). Declare such objects to communicate between signal handlers and the rest of the program.

### 11.16.3 SIGABRT

```
#define SIGABRT <integer constant expression >= 0>
```

The macro yields the `sig` argument value for the abort signal.

### 11.16.4 SIGFPE

```
#define SIGFPE <integer constant expression >= 0>
```

The macro yields the `sig` argument value for the arithmetic error signal, such as for division by zero or result out of range.

### 11.16.5 SIGILL

```
#define SIGILL <integer constant expression >= 0>
```

The macro yields the `sig` argument value for the invalid execution signal, such as for a corrupted function image.

### 11.16.6 SIGINT

```
#define SIGINT <integer constant expression >= 0>
```

The macro yields the `sig` argument value for the asynchronous interactive attention signal.

## 11.16.7 signal

```
void (*signal(int sig, void (*func)(int)))(int);
```

The function specifies the new handling for signal `sig` and returns the previous handling, if successful; otherwise, it returns `SIG_ERR`.

- If `func` is `SIG_DFL`, the target environment commences default handling (as defined by the implementation).
- If `func` is `SIG_IGN`, the target environment ignores subsequent reporting of the signal.
- Otherwise, `func` must be the address of a function returning void that the target environment calls with a single int argument. The target environment calls this function to handle the signal when it is next reported, with the value of the signal as its argument.

When the target environment calls a signal handler:

- The target environment can block further occurrences of the corresponding signal until the handler returns, calls `longjmp`, or calls `signal` for that signal.
- The target environment can perform default handling of further occurrences of the corresponding signal.

For signal `SIGILL`, the target environment can leave handling unchanged for that signal.

## 11.16.8 SIGSEGV

```
#define SIGSEGV <integer constant expression >= 0>
```

The macro yields the `sig` argument value for the invalid storage access signal, such as for an erroneous `lvalue` expression.

## 11.16.9 SIGTERM

```
#define SIGTERM <integer constant expression >= 0>
```

The macro yields the `sig` argument value for the asynchronous termination request signal.

## 11.16.10 SIG_DFL

```
#define SIG_DFL <address constant expression>
```

The macro yields the `func` argument value to `signal` to specify default signal handling.

### 11.16.11 SIG_ERR

```
#define SIG_ERR <address constant expression>
```

The macro yields the `signal` return value to specify an erroneous call.

### 11.16.12 SIG_IGN

```
#define SIG_IGN <address constant expression>
```

The macro yields the `func` argument value to `signal` to specify that the target environment is to henceforth ignore the signal.

# 11.17 <stdarg.h>

Include the standard header `<stdarg.h>` to access the unnamed additional arguments (that have no corresponding parameter declarations) in a function that accepts a varying number of arguments. To access the additional arguments:

- The program must first execute the macro `va_start` within the body of the function to initialize an object with context information.
- Subsequent execution of the macro `va_arg`, designating the same context information, yields the values of the additional arguments in order, beginning with the first unnamed argument. You can execute the macro `va_arg` from any function that can access the context information saved by the macro `va_start`.
- If you executed the macro `va_start` in a function, you must execute the macro `va_end` in the same function, designating the same context information, before the function returns.

You can repeat this sequence (as needed) to access the arguments as often as you want.

Declare an object of type `va_list` to store context information. `va_list` can be either an array type or a non-array type. Thus, you cannot reliably assign one such object to another—use the macro `va_copy` instead.

Whether `va_list` is an array type affects how the program shares context information with functions that it calls. The address of the first element of an array is passed, rather than the object itself. So an array type is effectively passed by reference, while a non-array type is passed by value.

For example:

```
#include <stdarg.h>
void va_cat(char *s, ...)
    {
    char *t;
    va_list ap;

    va_start(ap, s);
    while (t = va_arg(ap, char *)) NULL pointer ends list
        {
        s += strlen(s);            skip to end
        strcpy(s, t);             and copy a string
        }
    va_end(ap);
    }
```

The function `va_cat` concatenates an arbitrary number of strings onto the end of an existing string (assuming that the existing string is stored in an object large enough to hold the resulting string).

```
#define va_arg(va_list ap, Ty) <rvalue of type Ty>
#define va_copy(va_list dest, va_list src) <void expression> [Added with
C99]
#define va_end(va_list ap) <void expression>
#define va_start(va_list ap, last-par) <void expression>
typedef do-type va_list;
```

### 11.17.1 va_arg

```
#define va_arg(va_list ap, Ty) <rvalue of type Ty>
```

The macro yields the value of the next argument in order, specified by the context information designated by `ap`. The additional argument must be of object type `Ty` after applying the rules for promoting arguments in the absence of a function prototype.

### 11.17.2 va_copy

```
#define va_copy(va_list dest, va_list src) <void expression>
```
[Added with C99]

The macro copies the context information designated by `src` to the object designated by `dest`. It does so whether `va_list` is an array type.

### 11.17.3 va_end

```
#define va_end(va_list ap) <void expression>
```

The macro performs any cleanup necessary, after processing the context information designated by `ap`, so that the function can return.

### 11.17.4 va_list

```
typedef do-type va_list;
```

The type is the object type `do-type` that you declare to hold the context information initialized by va_start and used by `va_arg` to access additional unnamed arguments.

### 11.17.5 va_start

```
#define va_start(va_list ap, last-par) <void expression>
```

The macro stores initial context information in the object designated by `ap`. `last-par` is the name of the last parameter you declare. For example, `last-par` is `b` for the function declared as `int f(int a, int b, ...)`. The last parameter must not have register storage class, and it must have a type that is not changed by the translator. It cannot have:

- An array type
- A function type
- Type `float`
- Any integer type that changes when promoted
- A reference type [C++ only]

# 11.18 <stdbool.h>

[Added with C99]

Include the standard header `<stdbool.h>` to define a type and several macros suitable for writing Boolean tests. This header is available even in a freestanding implementation.

```
/* MACROS */
#define bool _Bool [Keyword in C++]
#define false 0 [Keyword in C++]
#define true 1 [Keyword in C++]
#define __bool_true_false_are_defined 1

/* TYPES */
typedef i-type bool; [Keyword in C++]
__bool_true_false_are_defined
#define __bool_true_false_are_defined 1
```

The macro yields the decimal constant `1`.

## 11.18.1 bool

```
#define bool _Bool [Keyword in C++]
```

The macro yields the type `_Bool`.

## 11.18.2 false

```
#define false 0 [Keyword in C++]
```

The macro yields the decimal constant `0`.

## 11.18.3 true

```
#define true 1 [Keyword in C++]
```

The macro yields the decimal constant `1`.

# 11.19 <stddef.h>

Include the standard header `<stddef.h>` to define several types and macros that are of general use throughout the program. This header is available even in a freestanding implementation.

```
/* MACROS */
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
#define offsetof(s-type, mbr) <size_t constant expression>

/* TYPES */
typedef si-type ptrdiff_t;
typedef ui-type size_t;
typedef i-type wchar_t; [Keyword in C++]
```

## 11.19.1 NULL

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
```

The macro yields a NULL pointer constant that is usable as an address constant expression.

## 11.19.2 offsetof

```
#define offsetof(s-type, mbr) <size_t constant expression>
```

The macro yields the offset in bytes, of type `size_t`, of member `mbr` from the beginning of structure type `s-type`, where for `X` of type `s-type`, `&X.mbr` is an address constant expression.

## 11.19.3 ptrdiff_t

```
typedef si-type ptrdiff_t;
```

The type is the signed integer type `si-type` of an object that you declare to store the result of subtracting two pointers.

## 11.19.4 size_t

```
typedef ui-type size_t;
```

The type is the unsigned integer type `ui-type` of an object that you declare to store the result of the `sizeof` operator.

## 11.19.5 wchar_t

```
typedef i-type wchar_t; [Keyword in C++]
```

The type is the integer type `i-type` of a wide-character constant, such as `L'X'`. Declare an object of type `wchar_tto` hold a wide character.

# 11.20 <stdint.h>

[Added with C99]

Include the standard header `<stdint.h>` to define various integer types, and related macros, with size constraints.

The definitions shown for the types and macros are merely representative—they can vary among implementations.

```
/* TYPE DEFINITIONS */
typedef signed char int8_t;
typedef short int16_t;
typedef long int32_t;
typedef long long int64_t;
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
typedef unsigned long long uint64_t;
typedef signed char int_least8_t;
typedef short int_least16_t;
typedef long int_least32_t;
typedef long long int_least64_t;
typedef unsigned char uint_least8_t;
typedef unsigned short uint_least16_t;
typedef unsigned long uint_least32_t;
typedef unsigned long long uint_least64_t;
typedef signed char int_fast8_t;
typedef short int_fast16_t;
typedef long int_fast32_t;
typedef long long int_fast64_t;
typedef unsigned char uint_fast8_t;
typedef unsigned short uint_fast16_t;
typedef unsigned long uint_fast32_t;
typedef unsigned long long uint_fast64_t;
typedef long intptr_t;
typedef unsigned long uintptr_t;
typedef long long intmax_t;
typedef unsigned long long uintmax_t;

/* LIMIT MACROS */
#define INT8_MIN (-0x7f - 1)
#define INT16_MIN (-0x7fff - 1)
#define INT32_MIN (-0x7fffffff - 1)
#define INT64_MIN (-0x7fffffffffffffff - 1)
#define INT8_MAX 0x7f [exact]
#define INT16_MAX 0x7fff [exact]
#define INT32_MAX 0x7fffffff [exact]
#define INT64_MAX 0x7fffffffffffffff [exact]
#define UINT8_MAX 0xff [exact]
#define UINT16_MAX 0xffff [exact]
#define UINT32_MAX 0xffffffff [exact]
#define UINT64_MAX 0xffffffffffffffff [exact]
#define INT_LEAST8_MIN (-0x7f - 1)
#define INT_LEAST16_MIN (-0x7fff - 1)
#define INT_LEAST32_MIN (-0x7fffffff - 1)
```

```
#define INT_LEAST64_MIN (-0x7fffffffffffffff - 1)
#define INT_LEAST8_MAX 0x7f
#define INT_LEAST16_MAX 0x7fff
#define INT_LEAST32_MAX 0x7fffffff
#define INT_LEAST64_MAX 0x7fffffffffffffff
#define UINT_LEAST8_MAX 0xff
#define UINT_LEAST16_MAX 0xffff
#define UINT_LEAST32_MAX 0xffffffff
#define UINT_LEAST64_MAX 0xffffffffffffffff
#define INT_FAST8_MIN (-0x7f - 1)
#define INT_FAST16_MIN (-0x7fff - 1)
#define INT_FAST32_MIN (-0x7fffffff - 1)
#define INT_FAST64_MIN (-0x7fffffffffffffff - 1)
#define INT_FAST8_MAX 0x7f
#define INT_FAST16_MAX 0x7fff
#define INT_FAST32_MAX 0x7fffffff
#define INT_FAST64_MAX 0x7fffffffffffffff
#define UINT_FAST8_MAX 0xff
#define UINT_FAST16_MAX 0xffff
#define UINT_FAST32_MAX 0xffffffff
#define UINT_FAST64_MAX 0xffffffffffffffff
#define INTPTR_MIN (-0x7fffffff - 1)
#define INTPTR_MAX 0x7fffffff
#define UINTPTR_MAX 0xffffffff
#define INT8_C(x)(x)
#define INT16_C(x) (x)
#define INT32_C(x) ((x) + (INT32_MAX - INT32_MAX))
#define INT64_C(x) ((x) + (INT64_MAX - INT64_MAX))
#define UINT8_C(x) (x)
#define UINT16_C(x)(x)
#define UINT32_C(x)((x) + (UINT32_MAX - UINT32_MAX))
#define UINT64_C(x)((x) + (UINT64_MAX - UINT64_MAX))
#define INTMAX_C(x)((x) + (INT64_MAX - INT64_MAX))
#define UINTMAX_C(x) ((x) + (UINT64_MAX - UINT64_MAX))
#define PTRDIFF_MIN INT32_MIN
#define PTRDIFF_MAX INT32_MAX
#define SIG_ATOMIC_MIN INT32_MIN
#define SIG_ATOMIC_MAX INT32_MAX
#define SIZE_MAX UINT32_MAX
#define WCHAR_MIN 0
#define WCHAR_MAX UINT16_MAX
#define WINT_MIN 0
#define WINT_MAX UINT16_MAX
#define INTMAX_MIN(-0x7fffffffffffffff - 1)
#define INTMAX_MAX 0x7fffffffffffffff
#define UINTMAX_MAX 0xffffffffffffffff
```

## 11.20.1 INT8_C, INT16_C, INT32_C, INT64_C

```
#define INT8_C(x)(x)
#define INT16_C(x) (x)
#define INT32_C(x) ((x) + (INT32_MAX - INT32_MAX))
#define INT64_C(x) ((x) + (INT64_MAX - INT64_MAX))
```

The macros each convert an integer literal to a signed integer type whose representation has at least 8, 16, 32, or 64 bits, respectively.

The definitions shown here are merely representative.

## 11.20.2 INT8_MAX, INT16_MAX, INT32_MAX, INT64_MAX

```
#define INT8_MAX 0x7f [exact]
#define INT16_MAX 0x7fff [exact]
#define INT32_MAX 0x7fffffff [exact]
#define INT64_MAX 0x7fffffffffffffff [exact]
```

The macros each expand to an `#if` expression that yields the maximum value representable as type `int8_t`, `int16_t`, `int32_t`, or `int64_t`, respectively.

The definitions shown here are exact.

## 11.20.3 INT8_MIN, INT16_MIN, INT32_MIN, INT64_MIN

```
#define INT8_MIN (-0x7f - 1)
#define INT16_MIN (-0x7fff - 1)
#define INT32_MIN (-0x7fffffff - 1)
#define INT64_MIN (-0x7fffffffffffffff - 1)
```

The macros each expand to an `#if` expression that yields the minimum value representable as type `int8_t`, `int16_t`, `int32_t`, or `int64_t`, respectively.

The definitions shown here are merely representative.

## 11.20.4 int8_t, int16_t, int32_t, int64_t

```
typedef signed char int8_t;
typedef short int16_t;
typedef long int32_t;
typedef long long int64_t;
```

The types each specify a signed integer type whose representation has exactly 8, 16, 32, or 64 bits, respectively.

The definitions shown here are merely representative.

## 11.20.5 INT_FAST8_MAX, INT_FAST16_MAX, INT_FAST32_MAX, INT_FAST64_MAX

```
#define INT_FAST8_MAX 0x7f
#define INT_FAST16_MAX 0x7fff
#define INT_FAST32_MAX 0x7fffffff
#define INT_FAST64_MAX 0x7fffffffffffffff
```

The macros each expand to an `#if` expression that yields the maximum value representable as type `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, or int_fast64_t, respectively.

The definitions shown here are merely representative.

## 11.20.6 INT_FAST8_MIN, INT_FAST16_MIN, INT_FAST32_MIN, INT_FAST64_MIN

```
#define INT_FAST8_MIN (-0x7f - 1)
#define INT_FAST16_MIN (-0x7fff - 1)
#define INT_FAST32_MIN (-0x7fffffff - 1)
#define INT_FAST64_MIN (-0x7fffffffffffffff - 1)
```

The macros each expand to an `#if` expression that yields the minimum value representable as type `int_fast8_t`, `int_fast16_t`, `int_fast32_t`, or `int_fast64_t`, respectively.

The definitions shown here are merely representative.

## 11.20.7 int_fast8_t, int_fast16_t, int_fast32_t, int_fast64_t

```
typedef signed char int_fast8_t;
typedef short int_fast16_t;
typedef long int_fast32_t;
typedef long long int_fast64_t;
```

The types each specify a signed integer type that supports the fastest operations among those whose representation has at least 8, 16, 32, or 64 bits, respectively.

The definitions shown here are merely representative.

## 11.20.8 INT_LEAST8_MAX, INT_LEAST16_MAX, INT_LEAST32_MAX, INT_LEAST64_MAX

```
#define INT_LEAST8_MAX 0x7f
#define INT_LEAST16_MAX 0x7fff
#define INT_LEAST32_MAX 0x7fffffff
#define INT_LEAST64_MAX 0x7fffffffffffffff
```

The macros each expand to an `#if` expression that yields the maximum value representable as type `int_least8_t`, `int_least16_t`, `int_least32_t`, or `int_least64_t`, respectively.

The definitions shown here are merely representative.

## 11.20.9 INT_LEAST8_MIN, INT_LEAST16_MIN, INT_LEAST32_MIN, INT_LEAST64_MIN

```
#define INT_LEAST8_MIN (-0x7f - 1)
#define INT_LEAST16_MIN (-0x7fff - 1)
#define INT_LEAST32_MIN (-0x7fffffff - 1)
#define INT_LEAST64_MIN (-0x7fffffffffffffff - 1)
```

The macros each expand to an `#if` expression that yields the minimum value representable as type `int_least8_t`, `int_least16_t`, `int_least32_t`, or `int_least64_t`, respectively.

The definitions shown here are merely representative.

## 11.20.10 int_least8_t, int_least16_t, int_least32_t, int_least64_t

```
typedef signed char int_least8_t;
typedef short int_least16_t;
typedef long int_least32_t;
typedef long long int_least64_t;
```

The types each specify a signed integer type whose representation has at least 8, 16, 32, or 64 bits, respectively.

The definitions shown here are merely representative.

## 11.20.11 INTMAX_C

```
#define INTMAX_C(x)((x) + (INT64_MAX - INT64_MAX))
```

The macro converts an integer literal to the largest signed integer type.

The definition shown here is merely representative.

## 11.20.12 INTMAX_MAX

```
#define INTMAX_MAX0x7fffffffffffffff
```

The macro expands to an `#if` expression that yields the maximum value representable as type `intmax_t`.

The definition shown here is merely representative.

## 11.20.13 INTMAX_MIN

```
#define INTMAX_MIN(-0x7fffffffffffffff - 1)
```

The macro expands to an `#if` expression that yields the minimum value representable as type `intmax_t`.

The definition shown here is merely representative.

## 11.20.14intmax_t

```
typedef long long intmax_t;
```

The type specifies the largest signed integer type.

The definition shown here is merely representative.

## 11.20.15INTPTR_MAX

```
#define INTPTR_MAX0x7fffffff
```

The macro expands to an `#if` expression that yields the maximum value representable as type `intptr_t`.

The definition shown here is merely representative.

## 11.20.16INTPTR_MIN

```
#define INTPTR_MIN(-0x7fffffff - 1)
```

The macro expands to an `#if` expression that yields the minimum value representable as type `intptr_t`.

The definition shown here is merely representative.

## 11.20.17intptr_t

```
typedef long intptr_t;
```

The type specifies a signed integer type large enough to support interconversion with a void pointer. (You can convert a void pointer to `intptr_t` and back, and the result compares equal to the original pointer value.) The definition shown here is merely representative.

## 11.20.18PTRDIFF_MAX

```
#define PTRDIFF_MAX INT32_MAX
```

The macro expands to an `#if` expression that that yields the maximum value representable as type `ptrdiff_t`.

The definition shown here is merely representative.

## 11.20.19 PTRDIFF_MIN

```
#define PTRDIFF_MIN INT32_MIN
```

The macro expands to an `#if` expression that yields the minimum value representable as type `ptrdiff_t`.

The definition shown here is merely representative.

## 11.20.20 SIG_ATOMIC_MAX

```
#define SIG_ATOMIC_MAX INT32_MAX
```

The macro expands to an `#if` expression that yields the maximum value representable as type `sig_atomic_t`.

The definition shown here is merely representative.

## 11.20.21 SIG_ATOMIC_MIN

```
#define SIG_ATOMIC_MIN INT32_MIN
```

The macro expands to an `#if` expression that yields the minimum value representable as type `sig_atomic_t`.

The definition shown here is merely representative.

## 11.20.22 SIZE_MAX

```
#define SIZE_MAX UINT32_MAX
```

The macro expands to an `#if` expression that yields the maximum value representable as type `size_t`.

The definition shown here is merely representative.

## 11.20.23 UINT8_C, UINT16_C, UINT32_C, UINT64_C

```
#define UINT8_C(x) (x)
#define UINT16_C(x)(x)
#define UINT32_C(x)((x) + (UINT32_MAX - UINT32_MAX))
#define UINT64_C(x)((x) + (UINT64_MAX - UINT64_MAX))
```

The macros each convert an integer literal to an unsigned integer type whose representation has at least 8, 16, 32, or 64 bits, respectively.

The definitions shown here are merely representative.

## 11.20.24UINT8_MAX, UINT16_MAX, UINT32_MAX, UINT64_MAX

```
#define UINT8_MAX 0xff [exact]
#define UINT16_MAX 0xffff [exact]
#define UINT32_MAX 0xffffffff [exact]
#define UINT64_MAX 0xffffffffffffffff [exact]
```

The macros each expand to an `#if` expression that yields the maximum value representable as type uint8_t, uint16_t, uint32_t, or uint64_t, respectively.

The definitions shown here are exact.

## 11.20.25uint8_t, uint16_t, uint32_t, uint64_t

```
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
typedef unsigned long long uint64_t;
```

The types each specify an unsigned integer type whose representation has exactly 8, 16, 32, or 64 bits, respectively.

The definitions shown here are merely representative.

## 11.20.26UINT_FAST8_MAX, UINT_FAST16_MAX, UINT_FAST32_MAX, UINT_FAST64_MAX

```
#define UINT_FAST8_MAX 0xff
#define UINT_FAST16_MAX 0xffff
#define UINT_FAST32_MAX 0xffffffff
#define UINT_FAST64_MAX 0xffffffffffffffff
```

The macros each expand to an `#if` expression that yields the maximum value representable as type uint_fast8_t, uint_fast16_t, uint_fast32_t, or uint_fast64_t, respectively.

The definitions shown here are merely representative

## 11.20.27uint_fast8_t, uint_fast16_t, uint_fast32_t, uint_fast64_t

```
typedef unsigned char uint_fast8_t;
typedef unsigned short uint_fast16_t;
typedef unsigned long uint_fast32_t;
typedef unsigned long long uint_fast64_t;
```

The types each specify an unsigned integer type that supports the fastest operations among those whose representation has at least 8, 16, 32, or 64 bits, respectively.

The definitions shown here are merely representative.

## 11.20.28 UINT_LEAST8_MAX, UINT_LEAST16_MAX, UINT_LEAST32_MAX, UINT_LEAST64_MAX

```
#define UINT_LEAST8_MAX 0xff
#define UINT_LEAST16_MAX 0xffff
#define UINT_LEAST32_MAX 0xffffffff
#define UINT_LEAST64_MAX 0xffffffffffffffff
```

The macros each expand to an `#if` expression that yields the maximum value representable as type uint_least8_t, uint_least16_t, uint_least32_t, or uint_least64_t, respectively.

The definitions shown here are merely representative

## 11.20.29 uint_least8_t, uint_least16_t, uint_least32_t, uint_least64_t

```
typedef unsigned char uint_least8_t;
typedef unsigned short uint_least16_t;
typedef unsigned long uint_least32_t;
typedef unsigned long long uint_least64_t;
```

The types each specify an unsigned integer type whose representation has at least 8, 16, 32, or 64 bits, respectively.

The definitions shown here are merely representative.

## 11.20.30 UINTMAX_C

```
#define UINTMAX_C(x) ((x) + (UINT64_MAX - UINT64_MAX))
```

The macro converts an unsuffixed integer literal to the largest unsigned integer type.

The definition shown here is merely representative.

## 11.20.31 UINTMAX_MAX

```
#define UINTMAX_MAX 0xffffffffffffffff
```

The macro expands to an `#if` expression that yields the maximum value representable as type `uintmax_t`.

The definition shown here is merely representative.

## 11.20.32 uintmax_t

```
typedef unsigned long long uintmax_t;
```

The type specifies the largest unsigned integer type.

The definition shown here is merely representative.

## 11.20.33UINTPTR_MAX

```
#define UINTPTR_MAX 0xffffffff
```

The macro expands to an `#if` expression that yields the maximum value representable as type `uintptr_t`.

The definition shown here is merely representative.

## 11.20.34uintptr_t

```
typedef unsigned long uintptr_t;
```

The type specifies an unsigned integer type large enough to support interconversion with a void pointer. (You can convert a void pointer to `uintptr_t` and back, and the result compares equal to the original pointer value.) The definition shown here is merely representative.

## 11.20.35WCHAR_MAX

```
#define WCHAR_MAX UINT16_MAX
```

The macro expands to an `#if` expression that yields the maximum value representable as type `wchar_t`.

The definition shown here is merely representative.

## 11.20.36WCHAR_MIN

```
#define WCHAR_MIN 0
```

The macro expands to an `#if` expression that yields the minimum value representable as type `wchar_t`.

The definition shown here is merely representative.

## 11.20.37WINT_MAX

```
#define WINT_MAX UINT16_MAX
```

The macro expands to an `#if` expression that yields the maximum value representable as type `wint_t`.

The definition shown here is merely representative.

## 11.20.38WINT_MIN

```
#define WINT_MIN 0
```

The macro expands to an `#if` expression that yields the minimum value representable as type `wint_t`.

The definition shown here is merely representative.

# 11.21 <stdio.h>

Include the standard header `<stdio.h>` so that you can perform input and output operations on streams and files.

```
/* MACROS */
#define _IOFBF <integer constant expression>
#define _IOLBF <integer constant expression>
#define _IONBF <integer constant expression>
#define BUFSIZ <integer constant expression >= 256>
#define EOF <integer constant expression < 0>
#define FILENAME_MAX <integer constant expression > 0>
#define FOPEN_MAX <integer constant expression >= 8>
#define L_tmpnam <integer constant expression > 0>
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
#define SEEK_CUR <integer constant expression>
#define SEEK_END <integer constant expression>
#define SEEK_SET <integer constant expression>
#define TMP_MAX <integer constant expression >= 25>
#define stderr <pointer to FILE rvalue>
#define stdin <pointer to FILE rvalue>
#define stdout <pointer to FILE rvalue>

/* TYPES */ typedef o-type FILE;
typedef o-type fpos_t;
typedef ui-type size_t;

/* FUNCTIONS */
void clearerr(FILE *stream);
int fclose(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
int fflush(FILE *stream);
FILE *fopen(const char *restrict filename, const char *restrict mode);
FILE *freopen(const char *restrict filename, const char *restrict
mode, FILE *stream);
int remove(const char *filename);
int rename(const char *old, const char *new);
void rewind(FILE *stream);
void setbuf(FILE *restrict stream, char *restrict buf);
int setvbuf(FILE *restrict stream, char *restrict buf, int mode,
size_t size);
FILE *tmpfile(void);
char *tmpnam(char *s);
int fseek(FILE *stream, long offset, int mode);
int fseeko(FILE *stream, off_t offset, int whence); [POSIX]
off_t ftello(FILE *stream); [POSIX]
int fsetpos(FILE *stream, const fpos_t *pos);
int fgetpos(FILE *restrict stream, fpos_t *restrict pos);
long ftell(FILE *stream);
int fgetc(FILE *stream);
char *fgets(char *restrict s, int n, FILE *restrict stream);
size_t fread(void *restrict ptr, size_t size, size_t nelem, FILE
*restrict stream);
int getc(FILE *stream);
int getchar(void);
```

```
int getc_unlocked(FILE *stream); [POSIX]
int getchar_unlocked(void); [POSIX]
char *gets(char *s);
int ungetc(int c, FILE *stream);
int fputc(int c, FILE *stream);
int fputs(const char *restrict s, FILE *restrict stream);
size_t fwrite(const void *restrict ptr, size_t size, size_t nelem,
FILE *restrict stream);
void perror(const char *s);
int putc(int c, FILE *stream);
int putchar(int c);
int putc_unlocked(int c, FILE *stream); [POSIX]
int putchar_unlocked(int c); [POSIX]
int puts(const char *s);
int fscanf(FILE *restrict stream, const char *restrict format, ...);
int scanf(const char *restrict format, ...);
int sscanf(const char *restrict s, const char *restrict format, ...);
int vfscanf(FILE *restrict stream, const char *restrict format,
va_list ap); [Added with C99]
int vscanf(const char *restrict format, va_list ap); [Added with C99]
int vsscanf(const char *restrict s, const char *restrict format,
va_list ap); [Added with C99]
int fprintf(FILE *restrict stream, const char *restrict format, ...);
int printf(const char *restrict format, ...);
int snprintf(char *restrict s, size_t n, const char *restrict format,
...); [Added with C99]
int sprintf(char *restrict s, const char *restrict format, ...);
int vfprintf(FILE *restrict stream, const char *restrict format,
va_list ap);
int vprintf(const char *restrict format, va_list ap);
int vsnprintf(char *restrict s, size_t n, const char *restrict format,
va_list ap); [Added with C99]
int vsprintf(char *restrict s, const char *restrict format, va_list
ap);
```

## 11.21.1 BUFSIZ

```
#define BUFSIZ <integer constant expression >= 256>
```

The macro yields the size of the stream buffer used by `setbuf`.

## 11.21.2 clearerr

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

The function clears the end-of-file and error indicators for the stream `stream`.

### 11.21.3 EOF

```
#define EOF <integer constant expression < 0>
```

The macro yields the return value used to signal the end of a stream or to report an error condition.

### 11.21.4 fclose

```
int fclose(FILE *stream);
```

The function closes the file associated with the stream `stream`. It returns zero if successful; otherwise, it returns `EOF`.

This function writes any buffered output to the file, deallocates the stream buffer if it was automatically allocated, and removes the association between the stream and the file. Do not use the value of stream in subsequent expressions.

### 11.21.5 feof

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

The function returns a nonzero value if the end-of-file indicator is set for the stream `stream`.

### 11.21.6 ferror

```
int ferror(FILE *stream);
```

The function returns a nonzero value if the error indicator is set for the stream `stream`.

### 11.21.7 fflush

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

The function writes any buffered output to the file associated with the stream `stream` and returns zero if successful; otherwise, it returns `EOF`.

If stream is a NULL pointer, this function writes any buffered output to all files opened for output.

### 11.21.8 fgetc

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

The function reads the next character c (if present) from the input stream `stream`, advances the file-position indicator (if defined), and returns `(int)(unsigned char)c`. If the function sets either the end-of-file indicator or the error indicator, it returns `EOF`.

## 11.21.9 fgetpos

```
int fgetpos(FILE *restrict stream, fpos_t *restrict pos);
```

The function stores the file-position indicator for the stream `stream` in `*pos` and returns zero if successful; otherwise, the function stores a positive value in `errno` and returns a nonzero value.

## 11.21.10 fgets

```
char *fgets(char *restrict s, int n, FILE *restrict stream);
```

The function reads characters from the input stream `stream` and stores them in successive elements of the array beginning at `s` and continuing until it stores `n-1` characters, stores an `NL` character, or sets the end-of-file or error indicators. If this function stores any characters, it concludes by storing a NULL character in the next element of the array.

The function returns `s` if it stores any characters and it has not set the error indicator for the stream; otherwise, it returns a NULL pointer. If it sets the error indicator, the array contents are indeterminate.

## 11.21.11 FILE

```
typedef o-type FILE;
```

The type is an object type `o-type` that stores all control information for a stream. The functions `fopen` and `freopen` allocate all `FILE` objects used by the read and write functions.

## 11.21.12 FILENAME_MAX

```
#define FILENAME_MAX <integer constant expression > 0>
```

The macro yields the maximum size array of characters that you must provide to hold a filename.

## 11.21.13fopen

```
FILE *fopen(const char *restrict filename, const char *restrict mode);
```

The function opens the file with the filename `filename`, associates it with a stream, and returns a pointer to the object controlling the stream. If the open fails, it returns a `NULL pointer`. The initial characters of mode determine how the program manipulates the stream and whether it interprets the stream as text or binary. The initial characters must be one of the following sequences:

- `"r"` – To open an existing text file for reading.
- `"w"` – To create a text file or to open and truncate an existing text file, for writing.
- `"a"` – To create a text file or to open an existing text file, for writing. The file-position indicator is positioned at the end of the file before each write.
- `"rb"` – To open an existing binary file for reading.
- `"wb"` – To create a binary file or to open and truncate an existing binary file, for writing.
- `"ab"` – To create a binary file or to open an existing binary file, for writing. The file-position indicator is positioned at the end of the file (possibly after arbitrary NULL byte padding) before each write.
- `"r+"` – To open an existing text file for reading and writing.
- `"w+"` – To create a text file or to open and truncate an existing text file, for reading and writing.
- `"a+"` – To create a text file or to open an existing text file, for reading and writing. The file-position indicator is positioned at the end of the file before each write.
- `"r+b"` or `"rb+"` – To open an existing binary file for reading and writing.
- `"w+b"` or `"wb+"` – To create a binary file or to open and truncate an existing binary file, for reading and writing.
- `"a+b"` or `"ab+"` – To create a binary file or to open an existing binary file, for reading and writing. The file-position indicator is positioned at the end of the file (possibly after arbitrary NULL byte padding) before each write.

If you open a file for both reading and writing, the target environment can open a binary file instead of a text file. If the file is not interactive, the stream is fully buffered.

## 11.21.14FOPEN_MAX

```
#define FOPEN_MAX <integer constant expression >= 8>
```

The macro yields the maximum number of files that the target environment permits to be simultaneously open (including `stderr`, `stdin`, and `stdout`).

## 11.21.15fpos_t

```
typedef o-type fpos_t;
```

The type is an object type `o-type` of an object that you declare to hold the value of a file-position indicator stored by `fsetpos` and accessed by `fgetpos`.

## 11.21.16fprintf

```
int fprintf(FILE *restrict stream, const char *restrict format, ...);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated character to the stream `stream`. It returns the number of characters generated, or it returns a negative value if the function sets the error indicator for the stream.

## 11.21.17fputc

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

The function writes the character `(unsigned char)c` to the output stream `stream`, advances the file-position indicator (if defined), and returns `(int)(unsigned char)c`. If the function sets the error indicator for the stream, it returns `EOF`.

## 11.21.18fputs

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

The function accesses characters from the C string s and writes them to the output stream `stream`. The function does not write the terminating NULL character. It returns a non-negative value if it has not set the error indicator; otherwise, it returns `EOF`.

## 11.21.19fread

```
size_t fread(void *restrict ptr, size_t size, size_t nelem, FILE
*restrict stream);
```

The function reads characters from the input stream `stream` and stores them in successive elements of the array whose first element has the address `(char *)ptr` until the function stores `size*nelem` characters or sets the end-of-file or error indicator.

It returns `n/size`, where `n` is the number of characters it read. If `n` is not a multiple of size, the value stored in the last element is indeterminate. If the function sets the error indicator, the file-position indicator is indeterminate.

## 11.21.20freopen

```
FILE *freopen(const char *restrict filename, const char *restrict
mode, FILE *stream);
```

The function closes the file associated with the stream `stream` (as if by calling `fclose`); then it opens the file with the filename `filename` and associates the file with the stream `stream` (as if by calling `fopen(filename, mode)`).

The function returns `stream` if the open is successful; otherwise, it returns a NULL pointer.

## 11.21.21fscanf

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

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned character from the stream `stream`.

The function returns the number of input items matched and assigned, or it returns `EOF` if the function does not store values before it sets the end-of-file or error indicator for the stream.

## 11.21.22fseek

```
int fseek(FILE *stream, long offset, int mode);
```

The function sets the file-position indicator for the stream `stream` (as specified by `offset` and `mode`), clears the end-of-file indicator for the stream, and returns zero if successful.

For a binary stream, `offset` is a signed offset in bytes:

- If mode has the value `SEEK_SET`, `fseek` adds offset to the file-position indicator for the beginning of the file.
- If mode has the value `SEEK_CUR`, `fseek` adds offset to the current file-position indicator.
- If mode has the value `SEEK_END`, `fseek` adds offset to the file-position indicator for the end of the file (possibly after arbitrary NULL character padding).

fseek sets the file-position indicator to the result of this addition.

For a text stream:

- If mode has the value `SEEK_SET`, `fseek` sets the file-position indicator to the value encoded in offset, which is either a value returned by an earlier successful call to `ftell` or zero to indicate the beginning of the file.
- If mode has the value `SEEK_CUR` and offset is zero, `fseek` leaves the file-position indicator at its current value.
- If mode has the value `SEEK_END` and offset is zero, `fseek` sets the file-position indicator to indicate the end of the file.

The function defines no other combination of argument values.

## 11.21.23fseeko

```
int fseeko(FILE *stream, off_t offset, int whence); [POSIX]
```

The function is identical to `fseek`, except that the offset argument is of type `off_t`.

## 11.21.24fsetpos

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

The function sets the file-position indicator for the stream `stream` to the value stored in `*pos`, clears the end-of-file indicator for the stream, and returns zero if successful. Otherwise, the function stores a positive value in `errno` and returns a nonzero value.

## 11.21.25ftell

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

The function returns an encoded form of the file-position indicator for the stream `stream` or stores a positive value in `errno` and returns the value `-1`.

- For a binary file, a successful return value gives the number of bytes from the beginning of the file.
- For a text file, target environments can vary on the representation and range of encoded file-position indicator values.

## 11.21.26ftello

```
off_t ftello(FILE *stream); [POSIX]
```

The function is identical to `ftell`, except that the return value type is `off_t`.

## 11.21.27fwrite

```
size_t fwrite(const void *restrict ptr, size_t size, size_t nelem,
FILE *stream);
```

The function writes characters to the output stream `stream`, accessing values from successive elements of the array whose first element has the address `(char *)ptr` until the function writes `size*nelem` characters or sets the error indicator.

It returns `n/size`, where `n` is the number of characters it wrote. If the function sets the error indicator, the file-position indicator is indeterminate.

## 11.21.28getc

```
int getc(FILE *stream);
```

The function has the same effect as `fgetc(stream)` except that a macro version of `getc` can evaluate stream more than once.

## 11.21.29getc_unlocked

```
int getc_unlocked(FILE *stream); [POSIX]
```

The function provide functionality identical to that of `getc` but does not perform implicit locking of the streams it operates on. In multi-threaded programs, it can be used only within a scope in which the stream has been successfully locked by the calling thread.

## 11.21.30 getchar

```
int getchar(void);
```

The function has the same effect as `fgetc(stdin)`, reading a character from the stream `stdin`

## 11.21.31 getchar_unlocked

```
int getchar_unlocked(void); [POSIX]
```

The function provide functionality identical to that of `getchar` but does not perform implicit locking of the streams it operates on. In multi-threaded programs, it can be used only within a scope in which the stream has been successfully locked by the calling thread.

## 11.21.32 gets

```
char *gets(char *s);
```

The function reads characters from the stream `stdin` and stores them in successive elements of the array whose first element has the address `s` until the function reads an `NL` character (which is not stored) or sets the end-of-file or error indicator. If `gets` reads any characters, it concludes by storing a NULL character in the next element of the array.

It returns `s` if it reads any characters and has not set the error indicator for the stream; otherwise, it returns a NULL pointer.

If it sets the error indicator, the array contents are indeterminate. The number of characters that gets reads and stores cannot be limited. Use `fgets` instead.

## 11.21.33 _IOFBF

```
#define _IOFBF <integer constant expression>
```

The macro yields the value of the mode argument to `setvbuf` to indicate full buffering. (Flush the stream buffer only when it fills.)

## 11.21.34 _IOLBF

```
#define _IOLBF <integer constant expression>
```

The macro yields the value of the mode argument to `setvbuf` to indicate line buffering. (Flush the stream buffer at the end of a text line.)

## 11.21.35_IONBF

```
#define _IONBF <integer constant expression>
```

The macro yields the value of the mode argument to `setvbuf` to indicate no buffering. (Flush the stream buffer at the end of each write operation.)

## 11.21.36L_tmpnam

```
#define L_tmpnam <integer constant expression > 0>
```

The macro yields the number of characters that the target environment requires for representing temporary filenames created by `tmpnam`.

## 11.21.37NULL

```
#define NULL <either 0, 0L, or (void *)0>
```
[0 in C++]

The macro yields a NULL pointer constant that is usable as an address constant expression.

## 11.21.38perror

```
void perror(const char *s);
```

The function writes a line of text to the stream `stderr`. If `s` is not a NULL pointer, the function first writes the C string s (as if by calling `fputs(s, stderr)`), followed by a colon (`:`) and a space. It then writes the same message C string that is returned by `strerror(errno)`, converting the value stored in `errno`, followed by an `NL` character.

## 11.21.39printf

```
int printf(const char *restrict format, ...);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated character to the stream `stdout`.

It returns the number of characters generated, or it returns a negative value if the function sets the error indicator for the stream.

## 11.21.40putc

```
int putc(int c, FILE *stream);
```

The function has the same effect as `fputc(c, stream)` except that a macro version of `putc` can evaluate stream more than once.

## 11.21.41 putc_unlocked

```
int putc_unlocked(int c, FILE *stream); [POSIX]
```

The function provides functionality identical to that of `putc`, but does not perform implicit locking of the streams it operates on. In multi-threaded programs, it can be used only within a scope in which the stream has been successfully locked by the calling thread.

## 11.21.42 putchar

```
int putchar(int c);
```

The function has the same effect as `fputc(c, stdout)`, writing a character to the stream `stdout`.

## 11.21.43 putchar_unlocked

```
int putchar_unlocked(int c); [POSIX]
```

The function provides functionality identical to that of `putchar`, but does not perform implicit locking of the streams it operates on. In multi-threaded programs, it can be used only within a scope in which the stream has been successfully locked by the calling thread.

## 11.21.44 puts

```
int puts(const char *s);
```

The function accesses characters from the C string s and writes them to the stream `stdout`. The function writes an `NL` character to the stream in place of the terminating NULL character. It returns a non-negative value if it has not set the error indicator; otherwise, it returns `EOF`.

## 11.21.45 remove

```
int remove(const char *filename);
```

The function removes the file with the filename `filename` and returns zero if successful. If the file is open when you remove it, the result is implementation defined. After you remove it, you cannot open it as an existing file.

## 11.21.46 rename

```
int rename(const char *old, const char *new);
```

The function renames the file with the filename `old` to have the filename `new` and returns zero if successful. If a file with the filename new already exists, the result is implementation defined. After you rename it, you cannot open the file with the filename `old`.

## 11.21.47rewind

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

The function calls `fseek(stream, 0L, SEEK_SET)` and then clears the error indicator for the stream `stream`.

## 11.21.48scanf

```
int scanf(const char *restrict format, ...);
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned character from the stream `stdin`.

It returns the number of input items matched and assigned, or it returns `EOF` if the function does not store values before it sets the end-of-file or error indicators for the stream.

## 11.21.49SEEK_CUR

```
#define SEEK_CUR <integer constant expression>
```

The macro yields the value of the mode argument to `fseek` to indicate seeking relative to the current file-position indicator.

## 11.21.50SEEK_END

```
#define SEEK_END <integer constant expression>
```

The macro yields the value of the mode argument to `fseek` to indicate seeking relative to the end of the file.

## 11.21.51SEEK_SET

```
#define SEEK_SET <integer constant expression>
```

The macro yields the value of the mode argument to `fseek` to indicate seeking relative to the beginning of the file.

## 11.21.52setbuf

```
void setbuf(FILE *restrict stream, char *restrict buf);
```

If `buf` is not a NULL pointer, the function calls `setvbuf(stream, buf, __IOFBF, BUFSIZ)`, specifying full buffering with `_IOFBF` and a buffer size of `BUFSIZ` characters. Otherwise, the function calls `setvbuf(stream, 0, _IONBF, BUFSIZ)`, specifying no buffering with `_IONBF`.

## 11.21.53setvbuf

```
int setvbuf(FILE *restrict stream, char *restrict buf, int mode,
size_t size);
```

The function sets the buffering mode for the stream `stream` according to `buf`, `mode`, and `size`. It returns zero if successful.

If `buf` is not a NULL pointer, `buf` is the address of the first element of an array of `char` of size `size` that can be used as the stream buffer. Otherwise, `setvbuf` can allocate a stream buffer that is freed when the file is closed. For mode, you must supply one of the following values:

- `_IOFBF` – To indicate full buffering
- `_IOLBF` – To indicate line buffering
- `_IONBF` – To indicate no buffering

You must call `setvbuf` after you call `fopen` to associate a file with that stream and before you call a library function that performs any other operation on the stream.

## 11.21.54size_t

```
typedef ui-type size_t;
```

The type is the unsigned integer type `ui-type` of an object that you declare to store the result of the `sizeof` operator.

## 11.21.55snprintf

```
int snprintf(char *restrict s, size_t n, const char *restrict format,
...);
```
[Added with C99]

The function generates formatted text, under the control of the format `format` and any additional arguments, and stores each generated character in successive locations of the array object whose first element has the address `s`.

If `n` is zero, it stores no characters. Otherwise, the function stores up to `n - 1` characters and concludes by storing a NULL character in the next location of the array. It returns the number of characters generated—not including the NULL character.

## 11.21.56sprintf

```
int sprintf(char *restrict s, const char *restrict format, ...);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and stores each generated character in successive locations of the array object whose first element has the address `s`.

The function concludes by storing a NULL character in the next location of the array. It returns the number of characters generated—not including the NULL character.

## 11.21.57sscanf

```
int sscanf(const char *restrict s, const char *restrict format, ...);
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It accesses each scanned character from successive locations of the array object whose first element has the address `s`.

It returns the number of items matched and assigned, or it returns `EOF` if the function does not store values before it accesses a NULL character from the array.

## 11.21.58stderr

```
#define stderr <pointer to FILE rvalue>
```

The macro yields a pointer to the object that controls the standard error output stream.

## 11.21.59stdin

```
#define stdin <pointer to FILE rvalue>
```

The macro yields a pointer to the object that controls the standard input stream.

## 11.21.60stdout

```
#define stdout <pointer to FILE rvalue>
```

The macro yields a pointer to the object that controls the standard output stream.

## 11.21.61tmpfile

```
FILE *tmpfile(void)
```

The function creates a temporary binary file with the filename `temp-name` and then has the same effect as calling `fopen(temp-name, "wb+")`. The file `temp-name` is removed when the program closes it, either by calling `fclose` explicitly or at normal program termination.

The filename `temp-name` does not conflict with any filenames that you create. If the `open` is successful, the function returns a pointer to the object controlling the stream; otherwise, it returns a NULL pointer.

## 11.21.62TMP_MAX

```
#define TMP_MAX <integer constant expression >= 25>
```

The macro yields the minimum number of distinct filenames created by the function `tmpnam`.

## 11.21.63tmpnam

```
char *tmpnam(char *s);
```

The function creates a unique filename `temp-name` and returns a pointer to the filename. If `s` is not a NULL pointer, `s` must be the address of the first element of an array at least of size `L_tmpnam`. The function stores `temp-name` in the array and returns `s`. Otherwise, if `s` is a NULL pointer, the function stores `temp-name` in a `static-duration` array and returns the address of its first element. Subsequent calls to `tmpnam` can alter the values stored in this array.

The function returns unique filenames for each of the first `TMP_MAX` times it is called, after which its behavior is implementation defined. The filename `temp-name` does not conflict with any filenames that you create.

## 11.21.64ungetc

```
int ungetc(int c, FILE *stream);
```

If `c` is not equal to `EOF`, the function stores `(unsigned char)c` in the object whose address is `stream` and clears the end-of-file indicator. If `c` equals `EOF` or the store cannot occur, the function returns `EOF`; otherwise, it returns `(unsigned char)c`. A subsequent library function call that reads a character from the stream `stream` obtains this stored value, which is then forgotten.

Thus, you can effectively push back a character to a stream after reading a character. (You need not push back the same character that you read.) An implementation can let you push back additional characters before you read the first one. You read the characters in reverse order of pushing them back to the stream. You cannot portably:

- Push back more than one character
- Push back a character if the file-position indicator is at the beginning of the file
- Call `ftell` for a text file that has a character currently pushed back

A call to the functions `fseek`, `fsetpos`, or `rewind` for the stream causes the stream to forget any pushed-back characters. For a binary stream, the file-position indicator is decremented for each character that is pushed back.

## 11.21.65vfprintf

```
int vfprintf(FILE *restrict stream, const char *restrict
format,va_list ap);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated character to the stream `stream`.

The function returns the number of characters generated, or it returns a negative value if the function sets the error indicator for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

## 11.21.66vfscanf

```
int vfscanf(FILE *restrict stream, const char *restrict format,
va_list ap);
```
[Added with C99]

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned character from the stream `stream`.

The function returns the number of input items matched and assigned, or it returns `EOF` if the function does not store values before it sets the end-of-file or error indicator for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

## 11.21.67vprintf

```
int vprintf(const char *restrict format, va_list ap);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated character to the stream `stdout`.

The function returns the number of characters generated, or a negative value if the function sets the error indicator for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

## 11.21.68vscanf

```
int vscanf(const char *restrict format, va_list ap); [Added with C99]
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned character from the stream `stdin`.

The function returns the number of input items matched and assigned, or it returns `EOF` if the function does not store values before it sets the end-of-file or error indicators for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

## 11.21.69vsnprintf

```
int vsnprintf(char *restrict s, size_t n, const char *restrict format,
va_list ap); [Added with C99]
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and stores each generated character in successive locations of the array object whose first element has the address `s`. If `n` is zero, it stores no characters. Otherwise, the function stores up to `n - 1` characters and concludes by storing a NULL character in the next location of the array. It returns the number of characters generated—not including the NULL character.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

## 11.21.70vsprintf

```
int vsprintf(char *restrict s, const char *restrict format, va_list
ap);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and stores each generated character in successive locations of the array object whose first element has the address `s`. The function concludes by storing a NULL character in the next location of the array. It returns the number of characters generated—not including the NULL character.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

## 11.21.71vsscanf

```
int vsscanf(const char *restrict s, const char *restrict format,
va_list ap); [Added with C99]
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It accesses each scanned character from successive locations of the array object whose first element has the address `s`. It returns the number of items matched and assigned, or it returns `EOF` if the function does not store values before it accesses a NULL character from the array.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

# 11.22 <stdlib.h>

Include the standard header <stdlib.h> to declare an assortment of useful functions and to define the macros and types that help you use them.

```
/* MACROS */
#define EXIT_FAILURE <rvalue integer expression>
#define EXIT_SUCCESS <rvalue integer expression>
#define MB_CUR_MAX <rvalue integer expression >= 1>
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
#define RAND_MAX <integer constant expression >= 32,767>

/* TYPES */
typedef struct {
    int quot, rem;
    } div_t;
typedef struct {
    long quot, rem;
    } ldiv_t;
typedef struct { [Added with C99]
    long long quot, rem;
    } lldiv_t;

typedef ui-type size_t;
typedef i-type wchar_t; [keyword in C++]

/* FUNCTIONS */ int abs(int i);
long abs(long i); [C++ only]
long long abs(long long i); [C++ only, added with C99]
long long llabs(long long i); [Added with C99]
long labs(long i);

div_t div(int numer, int denom);
ldiv_t div(long numer, long denom); [C++ only]
lldiv_t div(long long numer, long long denom); [C++ only, added with C99]
lldiv_t lldiv(long long numer, long long denom); [Added with C99]
ldiv_t ldiv(long numer, long denom);

int rand(void);
void srand(unsigned int seed);

long a64l(const char *s); [POSIX]
char *l64a(long int l); [POSIX]
double drand48(void); [POSIX]
double erand48(unsigned short xseed[3]); [POSIX]
long lrand48(void); [POSIX]
long nrand48(unsigned short xseed[3]); [POSIX]
long mrand48(void); [POSIX]
long jrand48(unsigned short xseed[3]); [POSIX]
void srand48(long seed); [POSIX]
unsigned short *seed48(unsigned short xseed[3]); [POSIX]
void lcong48(unsigned short p[7]); [POSIX]
int rand_r(unsigned int *seed);
char *ecvt(double value, int ndigit, int *decpt, int *sign); [POSIX]
char *fcvt(double value, int ndigit, int *decpt, int *sign); [POSIX]
```

```
char *gcvt(double value, int ndigit, char *buf); [POSIX]
int getsubopt(char **optionp, char *const *tokens, char **valuep);
[POSIX]
int mkstemp(char *template); [POSIX]
char *tempnam(const char *tmpdir, const char *prefix); [POSIX]

double atof(const char *s);
int atoi(const char *s);
long atol(const char *s);
long long atoll(const char *s); [Added with C99]

double strtod(const char *restrict s, char **restrict endptr);
float strtof(const char *restrict s, char **restrict endptr); [Added
with C99]
long double strtold(const char *restrict s, char **restrict endptr);
[Added with C99]

long long strtoll(const char *restrict s, char **restrict endptr, int
base); [Added with C99]
unsigned long long strtoull(const char *restrict s, char **restrict
endptr, int base); [Added with C99]

long strtol(const char *restrict s, char **restrict endptr, int base);
unsigned long strtoul(const char *restrict s, char **restrict endptr,
int base);

void *calloc(size_t nelem, size_t size);
void free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);

int mblen(const char *s, size_t n);
size_t mbstowcs(wchar_t *restrict wcs, const char *restrict s, size_t
n);
int mbtowc(wchar_t *restrict pwc, const char *restrict s, size_t n);
size_t wcstombs(char *restrict s, const wchar_t *restrict wcs, size_t
n);
int wctomb(char *s, wchar_t wchar);

void _Exit(int status); [Added with C99]
void exit(int status);
void abort(void);
char *getenv(const char *name);
int putenv(const char *string); [POSIX]
int system(const char *s);

extern "C++"
    int atexit(void (*func)(void)); [C++ only]
extern "C" [C++ only]
    int atexit(void (*func)(void));
extern "C++"
    void *bsearch(const void *key, const void *base,
        size_t nelem, size_t size,
            int (*cmp)(const void *ck,
                const void *ce)); [C++ only]
```

```
extern "C" [C++ only]
    void *bsearch(const void *key, const void *base,
        size_t nelem, size_t size,
            int (*cmp)(const void *ck,
                const void *ce));
extern "C++"
    void qsort(void *base, size_t nelem, size_t size,
        int (*cmp)(const void *e1, const void *e2)); [C++ only]
extern "C" [C++ only]
    void qsort(void *base, size_t nelem, size_t size,
        int (*cmp)(const void *e1, const void *e2));

char *mktemp(char *template); [POSIX]
```

## 11.22.1 a64l

```
long a64l(const char *s); [POSIX]
```

The `a64l` and `l64a` functions convert between a long integer and its base-64 ASCII string representation.

Table 11-5lists the characters used to represent digits.

**Table 11-5 a64l character to digit representations**

| Character | Digit |
|---|---|
| . | 0 |
| / | 1 |
| 0-9 | 2-11 |
| A-Z | 12-37 |
| a-z | 38-63 |

`a64l` takes a pointer to a NULL-terminated base-64 ASCII string representation, `s`, and returns the corresponding long integer value. On successful completion, `a64l` returns the long integer value corresponding to the input string. If the string pointed to by s is an empty string, `a64l` returns a value of `0L`.

## 11.22.2 abort

```
void abort(void);
```

The function calls `raise(SIGABRT)`, which reports the abort signal, `SIGABRT`. Default handling for the abort signal is to cause abnormal program termination and report unsuccessful termination to the target environment. Whether the target environment flushes output streams, closes open files, or removes temporary files on abnormal termination is implementation defined.

If you specify handling that causes raise to return control to abort, the function calls `exit(EXIT_FAILURE)`, to report unsuccessful termination with `EXIT_FAILURE`. The `abovt` function never returns control to its caller.

### 11.22.3 abs

```
int abs(int i);
long abs(long i); [C++ only]
long long abs(long long i); [C++ only, added with C99]
```

The function returns the absolute value of `i, |i|`. The version that accepts a long argument behaves the same as `labs`

### 11.22.4 atexit

```
extern "C++"
    int atexit(void (*func)(void)); [C++ only]
extern "C" [C++ only]
    int atexit(void (*func)(void));
```

The function registers the function whose address is `func` to be called by `exit` (or when `main` returns) and returns zero if successful. The functions are called in reverse order of registry. You can register at least 32 functions.

Furthermore, in C++, if control leaves a called function because it fails to handle a thrown exception, `terminate` is called.

### 11.22.5 atof

```
double atof(const char *s);
```

The function converts the initial characters of string `s` to an equivalent value `x` of type `double` and then returns `x`. The conversion is the same as for `strtod(s, 0)`, except that a value is not necessarily stored in `errno` if a conversion error occurs.

### 11.22.6 atoi

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

The function converts the initial characters of string `s` to an equivalent value `x` of type `int` and then returns `x`. The conversion is the same as for `(int)strtol(s, 0, 10)`, except that a value is not necessarily stored in `errno` if a conversion error occurs.

### 11.22.7 atol

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

The function converts the initial characters of string `s` to an equivalent value `x` of type `long` and then returns `x`. The conversion is the same as for `strtol(s, 0, 10)`, except that a value is not necessarily stored in `errno` if a conversion error occurs.

## 11.22.8 atoll

```
long long atoll(const char *s); [Added with C99]
```

The function converts the initial characters of string `s` to an equivalent value `x` of type `long long` and then returns `x`. The conversion is the same as for `strtoll(s, 0, 10)`, except that a value is not necessarily stored in `errno` if a conversion error occurs.

## 11.22.9 bsearch

```
extern "extern "C++"
    void *bsearch(const void *key, const void *base,
        size_t nelem, size_t size,
            int (*cmp)(const void *ck,
                const void *ce)); [C++ only]
extern "C" [C++ only]
    void *bsearch(const void *key, const void *base,
        size_t nelem, size_t size,
            int (*cmp)(const void *ck,
                const void *ce));ce));
```

The function searches an array of ordered values and returns the address of an array element that equals the search key `key` (if one exists); otherwise, it returns a NULL pointer. The array consists of `nelem` elements, each of size `bytes`, beginning with the element whose address is `base`.

`bsearch` calls the comparison function whose address `iscmp` to compare the search key with elements of the array. The comparison function must return:

- A negative value if the search key `ck` is less than the array element `ce`
- Zero if the two are equal
- A positive value if the search key is greater than the array element

This function assumes that the array elements are in non-descending order according to the same comparison rules that are used by the comparison function.

## 11.22.10 calloc

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

The function allocates an array object containing `nelem` elements each of size `size`, stores zeros in all bytes of the array, and returns the address of the first element of the array if successful. Otherwise, it returns a NULL pointer.

You can safely convert the return value to an object pointer of any type whose size in bytes is not greater than `size`.

## 11.22.11 div

```
div_t div(int numer, int denom);
ldiv_t div(long numer, long denom); [C++ only]
lldiv_t div(long long numer, long long denom); [C++ only, added with C99]
```

The function divides `numer` by `denom` and returns both quotient and remainder in the structure result `x`, if the quotient can be represented. The structure member `x.quot` is the algebraic quotient truncated toward zero. The structure `memberx.rem` is the remainder, such that `numer == x.quot*denom + x.rem`.

## 11.22.12 div_t

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

The type is the structure type returned by the function `div`. The structure contains members that represent the quotient (`quot`) and remainder (`rem`) of a signed integer division with operands of type `int`. The members shown above can occur in either order.

## 11.22.13 drand48

```
double drand48(void); [POSIX]
```

The rand48 family of functions generates pseudo-random numbers using a linear congruential algorithm working on 48-bit integers. The formula employed is `r(n+1) = (a * r(n) + c) mod m`, where the default values are for the multiplicand, `a = 0x5deece66d = 25214903917`, and the addend, `c = 0xb = 11`. The modulus is always fixed at `m = 248`. `r(n)` is called the seed of the random number generator.

The first computational step is to perform a single iteration of the algorithm.

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

`drand48` uses an internal buffer to store `r(n)`. For these functions the initial value of `r(0) = 0x1234abcd330e = 20017429951246`.

## 11.22.14 ecvt

```
char *ecvt(double value, int ndigit, int *decpt, int *sign); [POSIX]
```

This function has been marked as legacy in the POSIX standard, and exists only for backwards compatibility. Use `strtod` instead.

## 11.22.15erand48

```
double erand48(unsigned short xseed[3]); [POSIX]
```

The rand48 family of functions generates pseudo-random numbers using a linear congruential algorithm working on 48-bit integers. The formula employed is `r(n+1) = (a * r(n) + c) mod m`, where the default values are for the multiplicand, `a = 0x5deece66d = 25214903917`, and the addend, `c = 0xb = 11`. The modulus is always fixed at `m = 248`. `r(n)` is called the seed of the random number generator.

The first computational step is to perform a single iteration of the algorithm.

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

`erand48` uses a user-supplied buffer to store the seed `r(n)`, which consists of an array of three shorts, where the zeroth member holds the least significant bits.

## 11.22.16exit

```
void exit(int status);
```

The function calls all functions registered by `atexit`, closes all files, and returns control to the target environment. If status is zero or `EXIT_SUCCESS`, the program reports successful termination. If status is `EXIT_FAILURE`, the program reports unsuccessful termination. An implementation can define additional values for status.

## 11.22.17_Exit

```
void _Exit(int status); [Added with C99]
```

The function returns control to the target environment. The value of status has the same effect as for a call to exit. The function does not call functions registered by `atexit`. It may or may not close files.

## 11.22.18EXIT_FAILURE

```
#define EXIT_FAILURE <rvalue integer expression>
```

The macro yields the value of the status argument to exit that reports unsuccessful termination.

## 11.22.19EXIT_SUCCESS

```
#define EXIT_SUCCESS <rvalue integer expression>
```

The macro yields the value of the status argument to exit that reports successful termination.

## 11.22.20fcvt

```
char *fcvt(double value, int ndigit, int *decpt, int *sign); [POSIX]
```

This function has been marked as legacy in the POSIX standard, and exists only for backwards compatibility. Use `strtod`, instead.

## 11.22.21free

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

If `ptr` is not a NULL pointer, the function deallocates the object whose address is `ptr`; otherwise, it does nothing. You can deallocate only objects that you first allocate by calling `calloc`, `malloc`, or `realloc`.

## 11.22.22gcvt

```
char *gcvt(double value, int ndigit, char *buf); [POSIX]
```

This function has been marked as legacy in the POSIX standard, and exists only for backwards compatibility. Use `strtod`, instead.

## 11.22.23getenv

```
char *getenv(const char *name);
```

The function searches an environment list, which each implementation defines, for an entry whose name matches the string name. If the function finds a match, it returns a pointer to a `static-duration` object that holds the definition associated with the target environment name. Otherwise, it returns a NULL pointer.

Do not alter the value stored in the object. If you call `getenv` again, the value stored in the `object` can change. No target environment names are required of all environments.

## 11.22.24getsubopt

```
extern char *suboptarg
int getsubopt(char **optionp, char *const *tokens, char **valuep);
[POSIX]
```

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

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

The getsubopt function returns the zero-based offset of the pointer in the tokens array referencing a string which matches the first token in the string, or `-1` if the string contains no tokens or tokens does not contain a matching string.

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

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

For example:

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

...

extern char *optarg, *suboptarg;
char *options, *value;

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

## 11.22.25jrand48

```
long jrand48(void); [POSIX]
```

The rand48 family of functions generates pseudo-random numbers using a linear congruential algorithm working on 48-bit integers. The formula employed is `r(n+1) = (a * r(n) + c) mod m` where the default values are for the multiplicand, `a = 0x5deece66d = 25214903917`, and the addend, `c = 0xb = 11`. The modulus is always fixed at `m = 248`. `r(n)` is called the seed of the random number generator.

The first computational step is to perform a single iteration of the algorithm.

## 11.22.26l64a

```
char *l64a(long int l); [POSIX]
```

The function takes a long integer value, `l`, and returns a pointer to the corresponding NULL-terminated base-64 ASCII string representation.

Table 11-6 lists the characters used to represent digits.

**Table 11-6 l64a character to digit representations**

| Character | Digit |
|---|---|
| . | 0 |
| / | 1 |
| 0-9 | 2-11 |
| A-Z | 12-37 |
| a-z | 38-63 |

If `l` is `0L`, `l64a` returns a pointer to an empty string.

The `l64a` function is not reentrant.

## 11.22.27labs

```
long labs(long i);
```

The function returns the absolute value of `i, |i|`, the same as `abs`.

## 11.22.28lcong48

```
void lcong48(unsigned short p[7]); [POSIX]
```

The rand48 family of functions generates pseudo-random numbers using a linear congruential algorithm working on 48-bit integers. The formula employed is `r(n+1) = (a * r(n) + c) mod m` where the default values are for the multiplicand, `a = 0x5deece66d = 25214903917`, and the addend, `c = 0xb = 11`. The modulus is always fixed at `m = 248`. `r(n)` is called the seed of the random number generator.

The first computational step is to perform a single iteration of the algorithm.

Finally, `lcong48` allows full control over the multiplicand and addend used in `drand48`, `erand48`, `lrand48`, `nrand48`, `mrand48`, and `jrand48`, and the seed used in `drand48`, `lrand48`, and `mrand48`.

An array of seven shorts is passed as parameter; the first three shorts are used to initialize the seed; the second three are used to initialize the multiplicand; and the last short is used to initialize the addend. It is thus not possible to use values greater than `0xffff` as the addend.

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

## 11.22.29 llabs

```
long long llabs(long long i);
```
[Added with C99]

The function returns the absolute value of `i, |i|`, the same as `abs`.

## 11.22.30 ldiv

```
ldiv_t ldiv(long numer, long denom);
```

The function divides `numer` by `denom` and returns both quotient and remainder in the structure result `x`, if the quotient can be represented. The structure member `x.quot` is the algebraic quotient truncated toward zero. The structure `memberx.rem` is the remainder, such that `numer == x.quot*denom + x.rem`.

## 11.22.31 lldiv

```
lldiv_t lldiv(long long numer, long long denom);
```
[Added with C99]

The function divides `numer` by `denom` and returns both quotient and remainder in the structure result `x`, if the quotient can be represented. The structure member `x.quot` is the algebraic quotient truncated toward zero. The structure `memberx.rem` is the remainder, such that `numer == x.quot*denom + x.rem`.

## 11.22.32ldiv_t

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

The type is the structure type returned by the function `ldiv`. The structure contains members that represent the quotient (`quot`) and remainder (`rem`) of a signed integer division with operands of type `long`. The members shown above can occur in either order.

## 11.22.33lldiv_t

```
typedef struct { [Added with C99]
    long long quot, rem;
    } lldiv_t;
```

The type is the structure type returned by the function lldiv. The structure contains members that represent the quotient (`quot`) and remainder (`rem`) of a signed integer division with operands of type `long long`. The members shown above can occur in either order.

## 11.22.34lrand48

```
long lrand48(void); [POSIX]
```

The rand48 family of functions generates pseudo-random numbers using a linear congruential algorithm working on 48-bit integers. The formula employed is `r(n+1) = (a * r(n) + c) mod m` where the default values are for the multiplicand, `a = 0x5deece66d = 25214903917`, and the addend, `c = 0xb = 11`. The modulus is always fixed at `m = 248`. `r(n)` is called the seed of the random number generator.

The first computational step is to perform a single iteration of the algorithm.

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

`lrand48` uses an internal buffer to store `r(n)`. The initial value of `r(0) = 0x1234abcd330e = 20017429951246`.

## 11.22.35malloc

```
void *malloc(size_t size);
```

The function allocates an object of size `size`, and returns the address of the object if successful; otherwise, it returns a NULL pointer. The values stored in the object are indeterminate. You can safely convert the return value to an object pointer of any type whose size is not greater than size.

## 11.22.36MB_CUR_MAX

```
#define MB_CUR_MAX <rvalue integer expression >= 1>
```

The macro yields the maximum number of characters that constitute a multibyte character in the current locale. Its value is <= MB_LEN_MAX.

## 11.22.37mblen

```
int mblen(const char *s, size_t n);
```

If `s` is not a NULL pointer, the function returns the number of bytes in the multibyte string `s` that constitute the next multibyte character, or it returns `-1` if the next n (or the remaining) bytes do not constitute a valid multibyte character. mblen does not include the terminating NULL in the count of bytes. The function can use a conversion state stored in an internal `static-duration` object to determine how to interpret the multibyte string.

If `s` is a NULL pointer and if multibyte characters have a state-dependent encoding in the current locale, the function stores the initial conversion state in its internal `static-duration` object and returns nonzero; otherwise, it returns zero.

## 11.22.38mbstowcs

```
size_t mbstowcs(wchar_t *restrict wcs, const char *restrict s, size_t
n);
```

The function stores a wide character string, in successive elements of the array whose first element has the address wcs, by converting, in turn, each of the multibyte characters in the multibyte string s. The string begins in the initial conversion state. The function converts each character as if by calling mbtowc (except that the internal conversion state stored for that function is unaffected). It stores at most n wide characters, stopping after it stores a NULL wide character. It returns the number of wide characters it stores, not counting the NULL wide character, if all conversions are successful; otherwise, it returns `-1`.

## 11.22.39mbtowc

```
int mbtowc(wchar_t *restrict pwc, const char *restrict s, size_t n);
```

If `s` is not a NULL pointer, the function determines `x`, the number of bytes in the multibyte string `s` that constitute the next multibyte character. (`x` cannot be greater than MB_CUR_MAX.) If `pwc` is not a NULL pointer, the function converts the next multibyte character to its corresponding wide-character value and stores that value in `*pwc`. It then returns `x`, or it returns `-1` if the next n or the remaining bytes do not constitute a valid multibyte character.mbtowc does not include the terminating NULL in the count of bytes. The function can use a conversion state stored in an internal `static-duration` object to determine how to interpret the multibyte string.

If `s` is a NULL pointer and if multibyte characters have a state-dependent encoding in the current locale, the function stores the initial conversion state in its internal `static-duration` object and returns nonzero; otherwise, it returns zero.

## 11.22.40 mkstemp

```
int mkstemp(char *template); [POSIX]
```

The mkstemp function makes the same replacement to the template as mktemp and creates the template file, mode 0600, returning a file descriptor opened for reading and writing. This avoids the race between testing for a file's existence and opening it for use.

## 11.22.41 mktemp

```
char * mktemp(char *template); [POSIX]
```

The function takes the given file name template and overwrites a portion of it to create a file name. This file name is unique and suitable for use by the application. The template can be any file name with some number of "X"s appended to it, for example, /tmp/temp.XXXXXX. The trailing Xs are replaced with the current process number or a unique letter combination. The number of unique file names mktemp can return depends on the number of Xs provided. Although the NetBSD implementation of the functions will accept any number of trailing Xs, for portability reasons one should only use six. Using six Xs will result in mktemp testing roughly 266 (308,915,776) combinations.

**NOTE:** The permissions of the file or directory being created are subject to the restrictions imposed by the umask system call, so the created file might be unreadable or unwritable.

mktemp returns a pointer to the template on success and NULL on failure. If the call fails an error code is placed in the global variable errno.

Often code uses mktemp very early on, perhaps to globally initialize the template nicely, but the code that calls open or fopen on that filename occurs much later. In almost all cases, the use of fopen means that the flags O_CREAT or O_EXCL are not given to open, and thus a symbolic link race becomes possible, making it necessary to use fdopen. Furthermore, be careful about code that opens, closes, and then re-opens the file in question. Finally, ensure that upon error, the temporary file is removed correctly.

There are also cases where it is better to modify the code to use mktemp, in concert with open using the flags O_CREAT or O_EXCL, as long as the code retries a new template if open fails with an errno of EEXIST.

mktemp can set errno to ENOTDIR if the pathname portion of the template is not an existing directory, and it can also set errno to any value specified by the stat function.

## 11.22.42mrand48

```
long mrand48(void); [POSIX]
```

The rand48 family of functions generates pseudo-random numbers using a linear congruential algorithm working on 48-bit integers. The formula employed is `r(n+1) = (a * r(n) + c) mod m` where the default values are for the multiplicand, `a = 0x5deece66d = 25214903917`, and the addend, `c = 0xb = 11`. The modulus is always fixed at `m = 248`. `r(n)` is called the seed of the random number generator.

The first computational step is to perform a single iteration of the algorithm.

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

`mrand48` uses an internal buffer to store `r(n)`. The initial value of `r(0) = 0x1234abcd330e = 20017429951246`.

## 11.22.43nrand48

```
long nrand48(unsigned short xseed[3]); [POSIX]
```

The rand48 family of functions generates pseudo-random numbers using a linear congruential algorithm working on 48-bit integers. The formula employed is `r(n+1) = (a * r(n) + c) mod m` where the default values are for the multiplicand, `a = 0x5deece66d = 25214903917`, and the addend, `c = 0xb = 11`. The modulus is always fixed at `m = 248`. `r(n)` is called the seed of the random number generator.

The first computational step is to perform a single iteration of the algorithm.

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

`nrand48` uses a user-supplied buffer to store the seed `r(n)`, which consists of an array of three shorts, where the zeroth member holds the least significant bits.

## 11.22.44NULL

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
```

The macro yields a NULL pointer constant that is usable as an address constant expression.

## 11.22.45putenv

```
int putenv(const char *string); [POSIX]
```

The function sets an environment variable from the host environment list. For compatibility with differing environment conventions, the given argument's name and value can be appended and prepended, respectively, with an equal sign, `=`.

This function takes an argument of the form `name=value` and is equivalent to:

```
setenv(name, value, 1);
```

The function returns zero if successful; otherwise the global variable `errno` is set to indicate the error and a `-1` is returned.

## 11.22.46qsort

```
extern "C++"
    void qsort(void *base, size_t nelem, size_t size,
        int (*cmp)(const void *e1, const void *e2));
            [C++ only]
extern "C" [C++ only]
    void qsort(void *base, size_t nelem, size_t size,
        int (*cmp)(const void *e1, const void *e2));
```

The function sorts, in place, an array consisting of `nelem` elements, each of size `bytes`, beginning with the element whose address is `base`. It calls the comparison function whose address `iscmp` to compare pairs of elements. The comparison function must return a negative value if:

- `e1` is less than `e2`
- Zero if the two are equal
- A positive value if `e1` is greater than `e2`

Two array elements that are equal can appear in the sorted array in either order.

## 11.22.47rand

```
int rand(void);
```

The function computes a pseudo-random number x based on a seed value stored in an internal `static-duration` object, alters the stored seed value, and returns `x`. x is in the interval [0, RAND_MAX].

## 11.22.48RAND_MAX

```
#define RAND_MAX <integer constant expression >= 32,767>
```

The macro yields the maximum value returned by rand.

## 11.22.49rand_r

```
int rand_r(unsigned int *seed);
```

The `rand_r` function has the same functionality as rand except that a pointer to a seed must be supplied and maintained by the caller.

## 11.22.50realloc

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

The function allocates an object of size `size`, possibly obtaining initial stored values from the object whose address is ptr. It returns the address of the new object if successful; otherwise, it returns a NULL pointer. You can safely convert the return value to an object pointer of any type whose size is not greater than size.

If `ptr` is not a NULL pointer, it must be the address of an existing object that you first allocate by calling `calloc`, `malloc`, or `realloc`. If the existing object is not larger than the newly allocated object, `realloc` copies the entire existing object to the initial part of the allocated object. (The values stored in the remainder of the object are indeterminate.) Otherwise, the function copies only the initial part of the existing object that fits in the allocated object.

If `realloc` succeeds in allocating a new object, it deallocates the existing object. Otherwise, the existing object is left unchanged.

If `ptr` is a NULL pointer, the function does not store initial values in the newly created object.

## 11.22.51seed48

```
unsigned short *seed48(unsigned short xseed[3]);
```
[POSIX]

The rand48 family of functions generates pseudo-random numbers using a linear congruential algorithm working on 48-bit integers. The formula employed is `r(n+1) = (a * r(n) + c) mod m`, where the default values are for the multiplicand, `a = 0x5deece66d = 25214903917`, and the addend, `c = 0xb = 11`. The modulus is always fixed at `m = 248`. `r(n)` is called the seed of the random number generator.

The first computational step is to perform a single iteration of the algorithm.

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

`seed48` returns a pointer to an array of three shorts which contains the old seed. This array is statically allocated, thus its contents are lost after each new call to `seed48`.

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

## 11.22.52size_t

```
typedef ui-type size_t;
```

The type is the unsigned integer type `ui-type` of an object that you declare to store the result of the `sizeof` operator.

## 11.22.53srand

```
void srand(unsigned int seed);
```

The function stores the seed value seed in a `static-duration` object that rand uses to compute a pseudo-random number. From a given seed value, that function always generates the same sequence of return values. The program behaves as if the target environment calls srand(1) at program startup.

## 11.22.54srand48

`void srand48(long seed);` [POSIX]

The rand48 family of functions generates pseudo-random numbers using a linear congruential algorithm working on 48-bit integers. The formula employed is `r(n+1) = (a * r(n) + c) mod m` where the default values are for the multiplicand, `a = 0x5deece66d = 25214903917`, and the addend, `c = 0xb = 11`. The modulus is always fixed at `m = 248`. `r(n)` is called the seed of the random number generator.

The first computational step is to perform a single iteration of the algorithm.

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

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

## 11.22.55strtod

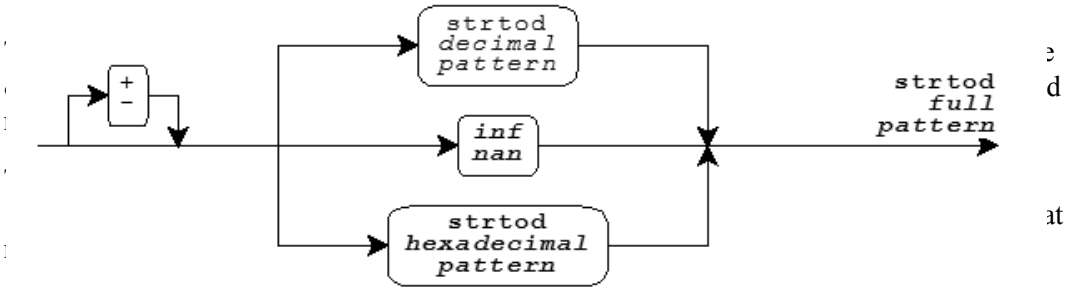


**Figure 11-2 strtod pattern matching**

**`Boldface`** indicates a feature added with C99.

Here, `inf` is the sequence of characters inf or infinity with individual letters in either case, to represent the special value infinity. Similarly, `nan` is the sequence of characters `nan` or `nan(qualifier)` with individual letters in either case, to represent the special value not-a-number (NaN). A qualifier is any sequence of zero or more letters, digits, and underscores. Each implementation defines what effect, if any, a qualifier has on the actual encoding of a NaN.

Figure 11-3 shows the pattern for a `strtod` hexadecimal string.

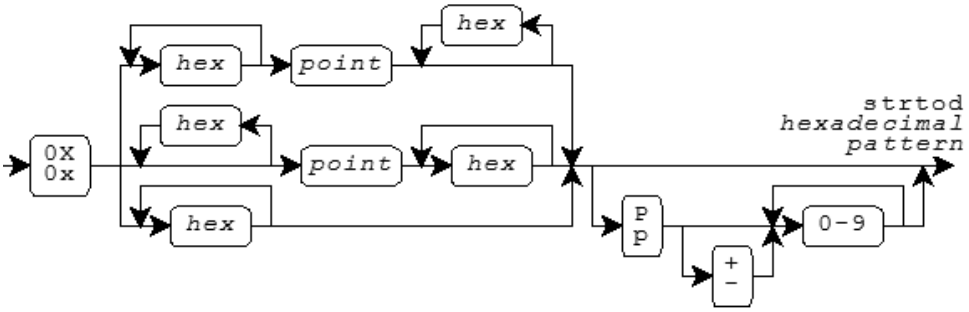


**Figure 11-3 strtod hexadecimal string**

Here, `hex` is a hexadecimal digit in either case, and a point is the decimal-point character for the current locale. (It is the dot (`.`) in the “C” locale.) If the strings matches this pattern, its equivalent value is the hexadecimal integer represented by any digits to the left of the point, plus the hexadecimal fraction represented by any digits to the right of the point, times two raised to the signed decimal integer power that follows an optional `p` or `P`. A leading minus sign negates the value.

Figure 11-4 shows the pattern for a `strtod` decimal string.

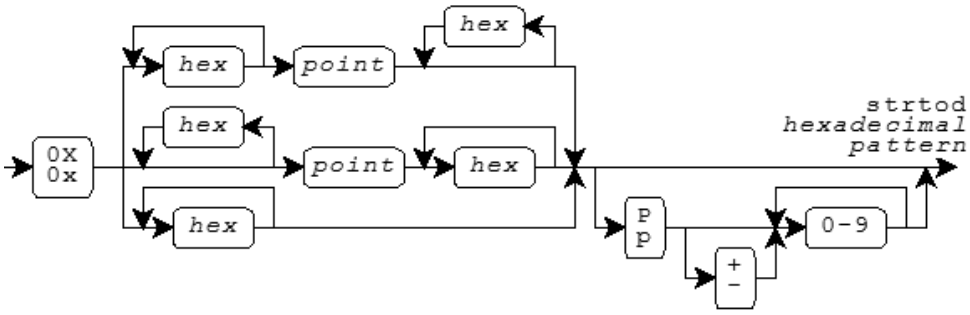


**Figure 11-4 strtod decimal string**

Here, a point is the decimal-point character for the current locale. (It is the dot ( . ) in the “C” locale.) If the strings matches this pattern, its equivalent value is the decimal integer represented by any digits to the left of the point, plus the decimal fraction represented by any digits to the right of the point, times 10 raised to the signed decimal integer power that follows an optional `e` or `E`. A leading minus sign negates the value.

In locales other than the “C” locale, `strtod` can define additional patterns as well.

If string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If a range error occurs, `strtod` behaves exactly as the functions declared in `<math.h>`.

## 11.22.56strtof

`float strtof(const char *restrict s, char **restrict endptr);` [Added with C99]

The function converts the initial characters of string `s` to an equivalent value `x` of type `float`. If `endptr` is not a NULL pointer, the function stores a pointer to the unconverted remainder of the string in `*endptr`. The function then returns `x`. `strtof` converts strings exactly as does `strtod`.

If string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If a range error occurs, `strtod` behaves exactly as the functions declared in `<math.h>`.

## 11.22.57strtol

```
long strtol(const char *restrict s, char **restrict endptr, int base);
```

The function converts the initial characters of string `s` to an equivalent value `x` of type `long`. If `endptr` is not a NULL pointer, it stores a pointer to the unconverted remainder of the string in `*endptr`. The function then returns `x`.

The initial characters of string `s` must consist of zero or more characters for which `isspace` returns nonzero, followed by the longest sequence of one or more characters that match the pattern for `strtol` shown in Figure 11-5.

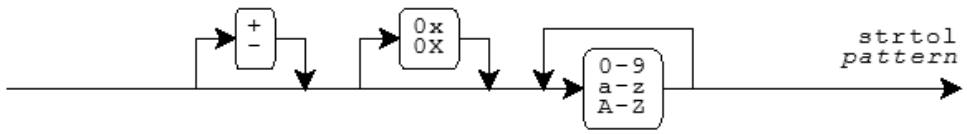


**Figure 11-5 strtol pattern**

The function accepts the sequences `0x` or `0X` only when base equals zero or 16. The letters a-z or A-Z represent digits in the range `[10, 36)`. If base is in the range `[2, 36]`, the function accepts only digits with values less than base. If `base == 0`, a leading `0x` or`0X` (after any sign) indicates a hexadecimal (base 16) integer, a leading 0 indicates an octal (base 8) integer, and any other valid pattern indicates a decimal (base 10) integer.

If string `s` matches this pattern, its equivalent value is the signed integer of the appropriate base represented by the digits that match the pattern. (A leading minus sign negates the value.) In locales other than the “C” locale, `strtol` can define additional patterns as well.

If string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If the equivalent value is too large to represent as type `long`, `strtol` stores the value of `ERANGE` in `errno` and returns either `LONG_MAX` if `x` is positive, or `LONG_MIN` if `x` is negative.

## 11.22.58strtold

```
long double strtold(const char *restrict s, char **restrict endptr);
```
[Added with C99]

The function converts the initial characters of string `s` to an equivalent value `x` of type `long double`. If `endptr` is not a NULL pointer, the function stores a pointer to the unconverted remainder of the string in `*endptr`. The function then returns `x`. `strtold` converts strings exactly as does `strtod`.

If string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If a range error occurs, `strtod` behaves exactly as the functions declared in `<math.h>`.

## 11.22.59strtoll

```
long long strtoll(const char *restrict s, char **restrict endptr, int
base); [Added with C99]
```

The function converts the initial characters of string `s` to an equivalent value `x` of type `long long`. If `endptr` is not a NULL pointer, it stores a pointer to the unconverted remainder of the string in `*endptr`. The function then returns `x`. `strtoll` converts strings exactly as does `strtol`.

If string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If the equivalent value is too large to represent as type long long, strtoll stores the value of `ERANGE` in `errno` and returns either `LLONG_MAX` if `x` is positive, or `LLONG_MIN` if `x` is negative.

## 11.22.60strtoul

```
unsigned long strtoul(const char *restrict s, char **restrict endptr,
int base);
```

The function converts the initial characters of string `s` to an equivalent value `x` of type unsigned `long`. If `endptr` is not a NULL pointer, it stores a pointer to the unconverted remainder of the string in `*endptr`. The function then returns `x`.

strtoul converts strings exactly as does strtol, but reports a range error only if the equivalent value is too large to represent as type unsigned `long`. In this case, `strtoul` stores the value of `ERANGE` in `errno` and returns `ULONG_MAX`.

## 11.22.61strtoull

```
unsigned long long strtoull(const char *restrict s, char **restrict
endptr, int base); [Added with C99]
```

The function converts the initial characters of string `s` to an equivalent value `x` of type `unsigned long long`. If `endptr` is not a NULL pointer, it stores a pointer to the unconverted remainder of the string in `*endptr`. The function then returns `x`. `strtoull` converts strings exactly as does `strtoul`.

If string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If the equivalent value is too large to represent as type unsigned `long long`, `strtoull` stores the value of `ERANGE` in `errno` and returns `ULLONG_MAX`.

## 11.22.62system

```
int system(const char *s);
```

If `s` is not a NULL pointer, the function passes the strings to be executed by a command processor, supplied by the target environment, and returns the status reported by the command processor. If `s` is a NULL pointer, the function returns nonzero only if the target environment supplies a command processor. Each implementation defines what strings its command processor accepts.

## 11.22.63tempnam

```
char *tempnam(const char *tmpdir, const char *prefix); [POSIX]
```

The function is similar to `tmpnam` but provides the ability to specify the directory that will contain the temporary file, `tmpdir`, and the file name prefix. `tempnam` functions return a pointer to a file name on success and a NULL pointer on error.

## 11.22.64wchar_t

```
typedef i-type wchar_t; [keyword in C++]
```

The type is the integer type `i-type` of a wide-character constant, such as `L'X'`. Declare an object of type `wchar_tto` hold a wide character.

## 11.22.65wcstombs

```
size_t wcstombs(char *restrict s, const wchar_t *restrict wcs, size_t
n);
```

The function stores a multibyte string in successive elements of the array whose first element has the address `s`, by converting in turn each of the wide characters in the string `wcs`.

The multibyte string begins in the initial conversion state. The function converts each wide character as if by calling `wctomb` (except that the conversion state stored for that function is unaffected). It stores no more than `n` bytes, stopping after it stores a NULL byte. It returns the number of bytes it stores, not counting the NULL byte, if all conversions are successful; otherwise, it returns `-1`.

## 11.22.66wctomb

```
int wctomb(char *s, wchar_t wchar);
```

If `s` is not a NULL pointer, the function determines `x`, the number of bytes needed to represent the multibyte character corresponding to the wide character `wchar`. `x` cannot exceed `MB_CUR_MAX`.

The function converts `wchar` to its corresponding multibyte character, which it stores in successive elements of the array whose first element has the address `s`. It then returns `x`, or it returns `-1` if `wchar` does not correspond to a valid multibyte character.

This function includes the terminating NULL byte in the count of bytes. The function can use a conversion state stored in a `static-duration` object to determine how to interpret the multibyte character string.

If `s` is a NULL pointer and if multibyte characters have a state-dependent encoding in the current locale, the function stores the initial conversion state in its `static-duration` object and returns nonzero; otherwise, it returns zero.

# 11.23 <string.h>

Include the standard header `<string.h>` to declare a number of functions that help you manipulate C strings and other arrays of characters.

```
/* MACROS */
#define NULL <either 0, 0L, or (void *)0> [0 in C++]

/* TYPES */
typedef ui-type size_t;

/* FUNCTIONS */
void *memccpy(void *dst, const void *src, int c, size_t len); [POSIX]
int memcmp(const void *s1, const void *s2, size_t n);
void *memcpy(void *restrict s1, const void *restrict s2, size_t n);

volatile void *memcpy_v(volatile void *restrict s1,
const volatile void *restrict s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
volatile void *memmove_v(volatile void *s1,
const volatile void *s2, size_t n);

void *memset(void *s, int c, size_t n);
char *strcat(char *restrict s1, const char *restrict s2);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
char *strcpy(char *restrict s1, const char *restrict s2);
size_t strcspn(const char *s1, const char *s2);
char *strerror(int errcode);
int strerror_r(int errnum, char *strerrbuf, size_t buflen); [POSIX]
size_t strlen(const char *s);
:char *strncat(char *_Restrict, const char *_Restrict, size_t)
_NO_THROW;
int strncmp(const char *, const char *, size_t) _NO_THROW;
size_t strlcpy(char *_Restrict, const char *_Restrict, size_t)
_NO_THROW;
size_t strlcat(char *_Restrict, const char *_Restrict, size_t)
_NO_THROW;
char *strncpy(char *_Restrict, const char *_Restrict, size_t)
_NO_THROW;
size_t strspn(const char *s1, const char *s2);
char *strtok(char *restrict s1, const char *restrict s2);
size_t strxfrm(char *restrict s1, const char *restrict s2,
size_t n);
void *memchr(const void *s, int c, size_t n); [Not in C++]
const void *memchr(const void *s, int c, size_t n); [C++ only]
void *memchr(void *s, int c, size_t n); [C++ only]
char *strchr(const char *s, int c); [Not in C++]
const char *strchr(const char *s, int c); [C++ only]
char *strchr(char *s, int c); [C++ only]
char *strpbrk(const char *s1, const char *s2); [Not in C++]
const char *strpbrk(const char *s1, const char *s2); [C++ only]
char *strpbrk(char *s1, const char *s2); [C++ only]
char *strrchr(const char *s, int c); [Not in C++]
const char *strrchr(const char *s, int c); [C++ only]
```

```
char *strrchr(char *s, int c); [C++ only]
char *strstr(const char *s1, const char *s2); [Not in C++]
const char *strstr(const char *s1, const char *s2); [C++ only]
char *strstr(char *s1, const char *s2); [C++ only]
int strcasecmp(const char *s1, const char *s2); [POSIX]
int strncasecmp(const char *s1, const char *s2, size_t len); [POSIX]
char *strdup(const char *str); [POSIX]
char *strtok_r(char *str, const char *sep, char **lasts); [POSIX]
```

### 11.23.1 memccpy

```
void *memccpy(void *dst, const void *src, int c, size_t len); [POSIX]
```

The function copies bytes from string `src` to string `dst`. If the character `c` (as converted to an unsigned `char`) occurs in string `src`, the copy stops and a pointer to the byte after the copy of `c` in the string `dst` is returned. Otherwise, `len` bytes are copied, and a NULL pointer is returned.

### 11.23.2 memchr

```
void *memchr(const void *s, int c, size_t n); [Not in C++]
const void *memchr(const void *s, int c, size_t n); [C++ only]
void *memchr(void *s, int c, size_t n); [C++ only]
```

The function searches for the first element of an array of unsigned `char`, beginning at the address `s` with size `n`, that equals `(unsigned char)c`. If successful, it returns the address of the matching element; otherwise, it returns a NULL pointer.

### 11.23.3 memcmp

```
int memcmp(const void *s1, const void *s2, size_t n);
```

The function compares successive elements from two arrays of unsigned `char`, beginning at the addresses `s1` and `s2` (both of size `n`), until it finds elements that are not equal:

- If all elements are equal, the function returns zero.
- If the differing element from `s1` is greater than the element from `s2`, the function returns a positive number.
- Otherwise, the function returns a negative number.

### 11.23.4 memcpy

```
void *memcpy(void *restrict s1, const void *restrict s2, size_t n);
```

The function copies the array of `char` beginning at the address `s2` to the array of `char` beginning at the address `s1` (both of size `n`). If copying takes place between arrays that overlap, the behavior is undefined. The function returns `s1`.

The order and size of memory accesses used to perform the copy are not specified.

**NOTE:** In certain cases, `memcpy` might generate a Hexagon processor cache exception. If this occurs, use the `memcpy_v` function instead.

### 11.23.5 memcpy_v

```
volatile void *memcpy_v(volatile void *restrict s1,
                const volatile void *restrict s2,
size_t n);
```

The function copies the array of `char` beginning at the address `s2` to the array of `char` beginning at the address `s1` (both of size `n`). If copying takes place between arrays that overlap, the behavior is undefined. The function returns `s1`.

The order and size of memory accesses used to perform the copy are not specified.

**NOTE:** `memcpy_v` (unlike `memcpy`) does not access memory more than once per instruction packet, and therefore it is guaranteed to not generate exception `0x28` in Hexagon processor V2 and V3.

### 11.23.6 memmove

```
void *memmove(void *s1, const void *s2, size_t n);
```

The function copies the array of `char` beginning at `s2` to the array of `char` beginning at `s1` (both of size `n`). It returns `s1`. If copying takes place between arrays that overlap, the function accesses each of the element values from `s2` before it stores a new value in that element, so the copy is not corrupted.

The order and size of memory accesses used to perform the copy are not specified.

**NOTE:** In certain cases, `memmove` might generate a Hexagon processor cache exception. If this occurs use the `memmove_v` function instead.

### 11.23.7 memmove_v

```
volatile void *memmove_v(volatile void *s1,
                         const volatile void *s2, size_t n);
```

The function copies the array of `char` beginning at `s2` to the array of `char` beginning at `s1` (both of size `n`). It returns `s1`. If copying takes place between arrays that overlap, the function accesses each of the element values from `s2` before it stores a new value in that element, so the copy is not corrupted.

The order and size of memory accesses used to perform the copy are not specified.

**NOTE:** `memmove_v` (unlike `memmove`) does not access memory more than once per instruction packet, and therefore is guaranteed to not generate exception `0x28` in Hexagon processor V2 and V3.

### 11.23.8 memset

```
void *memset(void *s, int c, size_t n);
```

The function stores `(unsigned char)c` in each of the elements of the array of unsigned char beginning at `s`, with size `n`. It returns `s`.

### 11.23.9 NULL

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
```

The macro yields a NULL pointer constant that is usable as an address constant expression.

### 11.23.10 size_t

```
typedef ui-type size_t;
```

The type is the unsigned integer type `ui-type` of an object that you declare to store the result of the `sizeof` operator.

### 11.23.11 strcasecmp

```
int strcasecmp(const char *s1, const char *s2); [POSIX]
```

The function compares the NULL-terminated strings `s1` and `s2` and returns an integer greater than, equal to, or less than 0, according to whether `s1` is lexicographically greater than, equal to, or less than `s2` after translation of each corresponding character to lowercase. The strings themselves are not modified. The comparison is done using unsigned characters, so that `\200` is greater than `\0`.

## 11.23.12strcat

```
char *strcat(char *restrict s1, const char *restrict s2);
```

The function copies string `s2`, including its terminating NULL character, to successive elements of the array of `char` that stores string `s1`, beginning with the element that stores the terminating NULL character of `s1`. It returns `s1`.

## 11.23.13strchr

```
char *strchr(const char *s, int c); [Not in C++]
const char *strchr(const char *s, int c); [C++ only]
char *strchr(char *s, int c); [C++ only]
```

The function searches for the first element of string `s` that equals `(char)c`. It considers the terminating NULL character as part of the string. If successful, the function returns the address of the matching element; otherwise, it returns a NULL pointer.

## 11.23.14strcmp

```
int strcmp(const char *s1, const char *s2);
```

The function compares successive elements from two strings, `s1` and `s2`, until it finds elements that are not equal.

- If all elements are equal, the function returns zero.
- If the differing element from `s1` is greater than the element from `s2` (both taken as unsigned `char`), the function returns a positive number.
- Otherwise, the function returns a negative number.

## 11.23.15strcoll

```
int strcoll(const char *s1, const char *s2);
```

The function compares two strings, `s1` and `s2`, using a comparison rule that depends on the current locale. If `s1` compares greater than `s2` by this rule, the function returns a positive number. If the two strings compare equal, it returns zero. Otherwise, it returns a negative number.

## 11.23.16strcpy

```
char *strcpy(char *restrict s1, const char *restrict s2);
```

The function copies string `s2`, including its terminating NULL character, to successive elements of the array of char whose first element has the address `s1`. It returns `s1`.

## 11.23.17strcspn

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

The function searches for the first element `s1[i]` in string `s1` that equals any one of the elements of string `s2` and returns `i`. Each terminating NULL character is considered part of its string.

## 11.23.18strdup

```
char *strdup(const char *str); [POSIX]
```

The function allocates sufficient memory for a copy of string `str`, does the copy, and returns a pointer to it. The pointer may subsequently be used as an argument to the function free.

If insufficient memory is available, NULL is returned.

The following example points `p` to an allocated area of memory containing the NULL-terminated string `"foobar"`:

```
char *p;

if ((p = strdup("foobar")) == NULL) {
        fprintf(stderr, "Out of memory.\n");
        exit(1);
}
```

**NOTE:** The `strdup` function might fail and set the external variable `errno` for any of the errors specified for the library function `malloc`.

## 11.23.19strerror

```
char *strerror(int errcode);
```

The function returns a pointer to an internal `static-duration` object containing the message string corresponding to the error code `errcode`. The program must not alter any of the values stored in this object. A later call to `strerror` can alter the value stored in this object.

## 11.23.20strerror_r

```
int strerror_r(int errnum, char *strerrbuf, size_t buflen); [POSIX]
```

The function is the re-entrant version of `strerror`. It can be re-entered while it is running. It renders the same result into `strerrbuf` for a maximum of `buflen` characters and returns 0 upon success.

## 11.23.21strlcat

```
size_t
    strlcat(char * restrict dst, const  char * restrict  src,
    size_t dstsize);
```

The function concatenates strings with the same input parameters and output result as `snprintf`. This function is a safer, more consistent, and less error prone replacement the easily misused function, `strncat`.

## 11.23.22strlcpy

```
size_t
    strlcpy(char * restrict dst, const  char * restrict  src,
    size_t dstsize);
```

The function copies strings with the same input parameters and output result as `snprintf`. This function is a safer, more consistent, and less error prone replacement the easily misused function, `strncpy`.

## 11.23.23strlen

```
size_t strlen(const char *s);
```

The function returns the number of characters in string `s`, not including its terminating NULL character.

## 11.23.24strncasecmp

```
int strncasecmp(const char *s1, const char *s2, size_t len);
```
[POSIX]

The function compares the NULL-terminated strings `s1` and `s2` and returns an integer greater than, equal to, or less than 0, according to whether `s1` is lexicographically greater than, equal to, or less than `s2` after translation of each corresponding character to lowercase. The strings themselves are not modified. The comparison is done using unsigned characters, so that `\200` is greater than `\0`.

The function compares, at most, `len` characters.

**NOTE:** If `len` is zero, `strncasecmp` always returns 0.

## 11.23.25strncat

```
char *strncat(char *restrict s1, const char *restrict s2, size_t n);
```

The function copies string `s2`, not including its terminating NULL character, to successive elements of the array of char that stores string `s1`, beginning with the element that stores the terminating NULL character of `s1`. The function copies no more than `n` characters from `s2`. It then stores a NULL character, in the next element to be altered in `s1`, and returns `s1`.

## 11.23.26strncmp

```
int strncmp(const char *s1, const char *s2, size_t n);
```

The function compares successive elements from two strings, `s1` and `s2`, until it finds elements that are not equal or until it has compared the first n elements of the two strings.

- If all elements are equal, the function returns zero.
- If the differing element from `s1` is greater than the element from `s2` (both taken as unsigned `char`), the function returns a positive number.
- Otherwise, it returns a negative number.

## 11.23.27strncpy

```
char *strncpy(char *restrict s1, const char *restrict s2, size_t n);
```

The function copies string `s2`, not including its terminating NULL character, to successive elements of the array of `char` whose first element has the address `s1`. It copies no more than `n` characters from `s2`. The function then stores zero or more NULL characters in the next elements to be altered in s1until it stores a total of `n` characters. It returns `s1`.

## 11.23.28strpbrk

```
char *strpbrk(const char *s1, const char *s2); [Not in C++]
const char *strpbrk(const char *s1, const char *s2); [C++ only]
char *strpbrk(char *s1, const char *s2); [C++ only]
```

The function searches for the first element `s1[i]` in string `s1` that equals any one of the elements of string `s2`. It considers each terminating NULL character as part of its string. If `s1[i]` is not the terminating NULL character, the function returns `&s1[i]`; otherwise, it returns a NULL pointer.

## 11.23.29strrchr

```
char *strrchr(const char *s, int c); [Not in C++]
const char *strrchr(const char *s, int c); [C++ only]
char *strrchr(char *s, int c); [C++ only]
```

The function searches for the last element of string `s` that equals `(char)c`. It considers the terminating NULL character as part of the string. If successful, the function returns the address of the matching element; otherwise, it returns a NULL pointer.

## 11.23.30strspn

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

The function searches for the first element `s1[i]` in string `s1` that equals none of the elements of string `s2` and returns i. It considers the terminating NULL character as part of string `s1` only.

## 11.23.31strstr

```
char *strstr(const char *s1, const char *s2); [Not in C++]
const char *strstr(const char *s1, const char *s2); [C++ only]
char *strstr(char *s1, const char *s2); [C++ only]
```

The function searches for the first sequence of elements in string `s1` that matches the sequence of elements in string `s2`, not including its terminating NULL character. If successful, the function returns the address of the matching first element; otherwise, it returns a NULL pointer.

## 11.23.32strtok

```
char *strtok(char *restrict s1, const char *restrict s2);
```

If `s1` is not a NULL pointer, the function begins a search of string `s1`. Otherwise, it begins a search of the string whose address was last stored in an internal `static-duration` object on an earlier call to the function, as described below. The search proceeds as follows:

- The function searches the string for begin, the address of the first element that equals none of the elements of string `s2` (a set of token separators). It considers the terminating NULL character as part of the search string only.
- If the search does not find an element, the function stores the address of the terminating NULL character in the internal `static-duration` object (so that a subsequent search beginning with that address will fail) and returns a NULL pointer. Otherwise, the function searches from begin for end, the address of the first element that equals any one of the elements of string `s2`. It again considers the terminating NULL character as part of the search string only.

- If the search does not find an element, the function stores the address of the terminating NULL character in the internal `static-duration` object. Otherwise, it stores a NULL character in the element whose address is end. Then it stores the address of the next element after end in the internal `static-duration` object (so that a subsequent search beginning with that address will continue with the remaining elements of the string) and returns begin.

## 11.23.33strtok_r

`char *strtok_r(char *str, const char *sep, char **lasts);` [POSIX]

The function is used to isolate sequential tokens in a NULL-terminated string, `str`. These tokens are separated in the string by at least one of the characters in `sep`. The first time that `strtok_r` is called, `str` should be specified; subsequent calls, wishing to obtain further tokens from the same string, should pass a NULL pointer instead. The separator string, `sep`, must be supplied each time, and may change between calls.

The function returns a pointer to the beginning of each subsequent token in the string, after replacing the separator character itself with a NULL character. Separator characters at the beginning of the string or at the continuation point are skipped so that zero length tokens are not returned. When no more tokens remain, a NULL pointer is returned.

The function is also passed the argument `lasts`, which points to a user-provided pointer that is used by `strtok_r` to store any state which needs to be kept between calls to scan the same string; it is not necessary to delineate tokenizing to a single string at a time when using `strtok_r`.

For example, the following will construct an array of pointers to each individual word in string `s`:

```
#define MAXTOKENS       128

char s[512], *p, *tokens[MAXTOKENS];
char *last;
int i = 0;

snprintf(s, sizeof(s), "cat dog horse cow");

for ((p = strtok_r(s, " ", &last)); p;
     (p = strtok_r(NULL, " ", &last)), i++) {
         if (i < MAXTOKENS - 1)
      tokens[i] = p;
}
tokens[i] = NULL;
```

That is:

- `tokens[0]` will point to `"cat"`
- `tokens[1]` will point to `"dog"`
- `tokens[2]` will point to `"horse"`
- `tokens[3]` will point to `"cow"`

## 11.23.34strxfrm

```
size_t strxfrm(char *restrict s1, const char *restrict s2,
               size_t n);
```

The function stores a string in the array of `char` whose first element has the address `s1`. It stores no more than `n` characters, including the terminating NULL character, and returns the number of characters needed to represent the entire string, not including the terminating NULL character. If the value returned is `n` or greater, the values stored in the array are indeterminate. (If `n` is zero, `s1` can be a NULL pointer.)

The function generates the string it stores from string `s2` by using a transformation rule that depends on the current locale. For example, if `x` is a transformation of `s1` and `y` is a transformation `ofs2`, then `strcmp(x, y)` returns the same value as `strcoll(s1, s2)`.

# 11.24 <strings.h>

The <strings.h> header contains functions for manipulating strings.

```
int bcmp(const void *b1, const void *b2, size_t len); [POSIX]
void bcopy(const void *src, void *dst, size_t len); [POSIX]
void bzero(void *b, size_t len); [POSIX]

int ffs(int value); [POSIX]

char *index(const char *s, int c); [POSIX]
char *rindex(const char *s, int c); [POSIX]
```

## 11.24.1 bcmp

```
int bcmp(const void *b1, const void *b2, size_t len); [POSIX]
```

The function compares byte string b1 against byte string b2, returning zero if they are identical or nonzero otherwise. Both strings are assumed to be len bytes long. Zero-length strings are always identical. The strings can overlap.

## 11.24.2 bcopy

```
void bcopy(const void *src, void *dst, size_t len); [POSIX]
```

The function copies len bytes from string src to string dst. The two strings can overlap. If len is zero, no bytes are copied.

## 11.24.3 bzero

```
void bzero(void *b, size_t len); [POSIX]
```

The function writes len zero bytes to the string b. If len is zero, bzero does nothing.

## 11.24.4 ffs

```
int ffs(int value); [POSIX]
```

The function finds the first bit set in value and returns the index of that bit. Bits are numbered starting from 1, starting at the right-most bit. A return value of 0 means that the argument was zero.

### 11.24.5 index

```
char *index(const char *s, int c); [POSIX]
```

The index function locates the first character matching c (converted to a `char`) in the NULL-terminated string s. A pointer to the character is returned if it is found; otherwise NULL is returned. If `c` is `'\0'`, `index` locates the terminating `'\0'`.

### 11.24.6 rindex

```
char *rindex(const char *s, int c); [POSIX]
```

The function locates the last character matching `c` (converted to a `char`) in the NULL-terminated string `s`. A pointer to the character is returned if it is found; otherwise NULL is returned. If `c` is `'\0'`, `rindex` locates the terminating `'\0'`.

# 11.25 <sys/stat.h>

Include the system header <sys/stat.h> to define several macros and a host of functions for use with file query operations.

```
int stat(const char *path, struct stat *sb); [POSIX]
stat
int stat(const char *path, struct stat *sb); [POSIX]
```

The stat() function obtains information about the file pointed to by path. Read, write or execute permission of the named file is not required, but all directories listed in the path name leading to the file must be searchable.

The sb argument is a pointer to a stat structure as defined by <sys/stat.h> (shown below) and into which information is placed concerning the file.

For example:

```
struct stat {
    dev_t     st_dev;     /* device containing the file */
    ino_t     st_ino;     /* file's serial number */
    mode_t    st_mode;    /* file's mode (protection and type) */
    nlink_t   st_nlink;   /* number of hard links to the file */
    uid_t     st_uid;     /* user-id of owner */
    gid_t     st_gid;     /* group-id of owner */
    dev_t     st_rdev;    /* device type, for device special file */
#if defined(_NETBSD_SOURCE)
    struct timespec st_atimespec;  /* time of last access */
    struct timespec st_mtimespec;  /* time of last data modification
*/
    struct timespec st_ctimespec;  /* time of last file status change
*/
#else
    time_t    st_atime;            /* time of last access */
    long      st_atimensec;        /* nsec of last access */
    time_t    st_mtime;           /* time of last data modification */
    long      st_mtimensec;       /* nsec of last data modification */
    time_t    st_ctime;           /* time of last file status change */
    long      st_ctimensec;       /* nsec of last file status change */
#endif
    off_t     st_size;    /* file size, in bytes */
    blkcnt_t  st_blocks;  /* blocks allocated for file */
    blksize_t st_blksize; /* optimal file sys I/O ops blocksize */
    uint32_t st_flags;   /* user defined flags for file */
    uint32_t st_gen;     /* file generation number */
#if defined(_NETBSD_SOURCE)
    struct timespec st_birthtimespec;   /* time of inode creation */
#else
    time_t    st_birthtime;             /* time of inode creation */
    long      st_birthtimensec;         /* nsec of inode creation */
#endif
};
```

Table 11-7 lists the time-related fields of the `struct stat`.

**Table 11-7 stat time-related fields**

| Field | Description |
|---|---|
| `st_atime` | Time when file data was last accessed.<br>Changed by the `mknod`, `utimes`, and `read` system calls. |
| `st_mtime` | Time when file data was last modified.<br>Changed by the `mknod`, `utimes`, and `write` system calls. |
| `st_ctime` | Time when file status was last changed (file metadata modification).<br>Changed by the `chflags`, `chmod`, `chown`, `link`, `mknod`, `rename`, `unlink`, `utimes`, and `write` system calls. |
| `st_birthtime` | Time when the `inode` was created. |

If `_NETBSD_SOURCE` is defined, the time-related fields are defined as:

```
#if defined(_NETBSD_SOURCE)
#define st_atime                st_atimespec.tv_sec
#define st_atimensec            st_atimespec.tv_nsec
#define st_mtime                st_mtimespec.tv_sec
#define st_mtimensec            st_mtimespec.tv_nsec
#define st_ctime                st_ctimespec.tv_sec
#define st_ctimensec            st_ctimespec.tv_nsec
#define st_birthtime            st_birthtimespec.tv_sec
#define st_birthtimensec        st_birthtimespec.tv_nsec
#endif
```

Table 11-8 lists the size-related fields of the `struct stat`.

**Table 11-8 stat size-related fields**

| Field | Description |
|---|---|
| `st_size` | The size of the file in bytes.<br>A directory will be a multiple of the size of the dirent structure. Some file systems (notably ZFS) return the number of entries in the directory instead of the size in bytes. |
| `st_blksize` | The optimal input/output block size for the file. |
| `st_blocks` | The actual number of blocks allocated for the file in 512-byte units.<br>Because short symbolic links are stored in the `inode`, this number might be zero. |

The status information word `st_mode` has the following bits:

```
#define S_IFMT 0170000          /* type of file */
#define        S_IFIFO  0010000  /* named pipe (fifo) */
#define        S_IFCHR  0020000  /* character special */
#define        S_IFDIR  0040000  /* directory */
#define        S_IFBLK  0060000  /* block special */
#define        S_IFREG  0100000  /* regular */
#define        S_IFLNK  0120000  /* symbolic link */
#define        S_IFSOCK 0140000  /* socket */
#define        S_IFWHT  0160000  /* whiteout */
#define S_ISUID 0004000  /* set user id on execution */
```

```
#define S_ISGID 0002000  /* set group id on execution */
#define S_ISVTX 0001000  /* save swapped text even after use */
#define S_IRUSR 0000400  /* read permission, owner */
#define S_IWUSR 0000200  /* write permission, owner */
#define S_IXUSR 0000100  /* execute/search permission, owner */
#define S_IRGRP 0000040  /* read permission, group */
#define S_IWGRP 0000020  /* write permission, group */
#define S_IXGRP 0000010  /* execute/search permission, group */
#define S_IROTH 0000004  /* read permission, other */
#define S_IWOTH 0000002  /* write permission, other */
#define S_IXOTH 0000001  /* execute/search permission, other */
```

The status information word `st_flags` has the following bits:

```
#define UF_NODUMP    0x00000001 /* do not dump file */
#define UF_IMMUTABLE 0x00000002 /* file may not be changed */
#define UF_APPEND    0x00000004 /* writes file may only append */
#define UF_OPAQUE    0x00000008 /* directory is opaque wrt union */
#define SF_ARCHIVED  0x00010000 /* file is archived */
#define SF_IMMUTABLE 0x00020000 /* file may not be changed */
#define SF_APPEND    0x00040000 /* writes file may only append */
```

Upon successful completion a value of `0` is returned. Otherwise, a value of `-1` is returned and `errno` is set to indicate the error.

# 11.26 <sys/time.h>

Include the system header <sys/time.h> to define a timeval structure and the gettimeofday function.

```
struct timeval {
        time_t       tv_sec;      /* seconds since Jan. 1, 1970 */
        suseconds_t  tv_usec;     /* and microseconds */
};

struct timezone {
        int     tz_minuteswest; /* of Greenwich */
        int     tz_dsttime;     /* type of dst correction to apply
};

int gettimeofday(struct timeval * restrict tp, void * restrict tzp);
[POSIX]
gettimeofday
int gettimeofday(struct timeval * restrict tp, void * restrict tzp);
[POSIX]
```

Time zone information is no longer provided by this interface. For information on how to retrieve it, see localtime.

The system's notion of the current UTC time is obtained with the gettimeofday call. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware dependent, and the time might be updated continuously or in *ticks*. If tp is NULL, the time will not be returned or set. Despite being declared void *, the objects pointed to by tzp shall be of type struct timezone.

The timezone structure is provided only for source compatibility. gettimeofday will always return zeroes.

# 11.27 <sys/times.h>

Include the system header <sys/times.h> to define the tms structure and the times function prototype.

```
struct tms {
    clock_t tms_utime;
    clock_t tms_stime;
    clock_t tms_cutime;
    clock_t tms_cstime;
};

clock_t times(struct tms *buffer); [POSIX]
times
clock_t times(struct tms *buffer); [POSIX]
```

This interface is obsoleted by gettimeofday.

The times function returns the value of time in clock ticks since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time (UTC).

The number of clock ticks per second can be determined by calling sysconf with the _SC_CLK_TCK request. It is typically (but not always) between 60 and 1024.

At the common rate of 100 ticks per second on many NetBSD ports and with a 32-bit unsigned clock_t, this value first wrapped in 1971.

The times call also fills in the structure pointed to by buffer with time accounting information.

Table 11-9 10-9 lists the elements of the tms structure.

**Table 11-9 tms structure elements**

| Element | Description |
|---|---|
| tms_utime | The CPU time charged for the execution of user instructions. |
| tms_stime | The CPU time charged for execution by the system on behalf of the process. |
| tms_cutime | The sum of the tms_utime s and tms_cutime s of the child processes. |
| tms_cstime | The sum of the tms_stimes and tms_cstimes of the child processes. |

All times are measured in clock ticks, as defined above. At 100 ticks per second, and with a 32-bit unsigned clock_t, the values wrap after 497 days.

The times of a terminated child process are included in the tms_cutime and tms_cstime elements of the parent when one of the wait functions returns the process ID of the terminated child to the parent. If an error occurs, times returns the value ((clock_t)-1), and sets errno to indicate the error.

# 11.28 <tgmath.h>

[Added with C99]

Include the standard header `<tgmath>` to define several families of generic functions. A generic function has the same name for two or more distinct parameter lists, as in:

```
float _Complex fc;
long double ld;

cos(1)       // same as cos((double)1)
cos(ld)      // same as cosl(ld)
pow(fc, ld)  // same as cpowl((long double _Complex)fc,
                    (long double _Complex)ld)
```

The actual (non-generic) function called depends on the type of the actual argument on a function call:

- For a function argument that can have a floating point type, an integer argument is converted (by a type `cast`) to double before determining the function to call.
- For a function with two or more arguments, the arguments are each converted as above, and then converted (by a type `cast`) to the type of the sum of all the arguments before determining the function to call.

The actual function called is the one whose parameters exactly match the type of the converted argument.

**NOTE:** C++ adds sufficient overloads (not shown here) to provide the same conversion rules. In this implementation, these overloads are declared in `<math.h>` and `<complex.h>`, and hence are present whether or not you include `<tgmath.h>`.

```
#include <complex.h>
#include <math.h>

/* FUNCTIONS */
double acos(double x);
float acos(float x);
long double acos(long double x);
double _Complex acos(double _Complex x);
float _Complex acos(float _Complex x);
long double _Complex acos(long double _Complex x);

double acosh(double x);
float acosh(float x);
long double acosh(long double x);
double _Complex acosh(double _Complex x);
float _Complex acosh(float _Complex x);
long double _Complex acosh(long double _Complex x);

double asin(double x);
float asin(float x);
long double asin(long double x);
double _Complex asin(double _Complex x);
float _Complex asin(float _Complex x);
long double _Complex asin(long double _Complex x);
```

```
double asinh(double x);
float asinh(float x);
long double asinh(long double x);
double _Complex asinh(double _Complex x);
float _Complex asinh(float _Complex x);
long double _Complex asinh(long double _Complex x);

double atan(double x);
float atan(float x);
long double atan(long double x);
double _Complex atan(double _Complex x);
float _Complex atan(float _Complex x);
long double _Complex atan(long double _Complex x);

double atan2(double y, double x);
float atan2(float y, float x);
long double atan2(long double y, long double x);

double atanh(double x);
float atanh(float x);
long double atanh(long double x);
double _Complex atanh(double _Complex x);
float _Complex atanh(float _Complex x);
long double _Complex atanh(long double _Complex x);

double carg(double _Complex x);
float carg(float _Complex x);
long double carg(long double _Complex x);

double ceil(double x);
float ceil(float x);
long double ceil(long double x);

double cbrt(double x);
float cbrt(float x);
long double cbrt(long double x);

double cimag(double _Complex x);
float cimag(float _Complex x);
long double cimag(long double _Complex x);

double _Complex conj(double _Complex x);
float _Complex conj(float _Complex x);
long double _Complex conj(long double _Complex x);

double cos(double x);
float cos(float x);
long double cos(long double x);
double _Complex cos(double _Complex x);
float _Complex cos(float _Complex x);
long double _Complex cos(long double _Complex x);

double copysign(double x, double y);
float copysign(float x, float y);
long double copysign(long double x, long double y);
```

```
double _Complex cproj(double _Complex x);
float _Complex cproj(float _Complex x);
long double _Complex cproj(long double _Complex x);

double cosh(double x);
float cosh(float x);
long double cosh(long double x);
double _Complex cosh(double _Complex x);
float _Complex cosh(float _Complex x);
long double _Complex cosh(long double _Complex x);

double creal(double _Complex x);
float creal(float _Complex x);
long double creal(long double _Complex x);

double erf(double x);
float erf(float x);
long double erf(long double x);

double erfc(double x);
float erfc(float x);
long double erfc(long double x);

double exp(double x);
float exp(float x);
long double exp(long double x);
double _Complex exp(double _Complex x);
float _Complex exp(float _Complex x);
long double _Complex exp(long double _Complex x);

double exp2(double x);
float exp2(float x);
long double exp2(long double x);

double expm1(double x);
float expm1(float x);
long double expm1(long double x);

double fabs(double x);
float fabs(float x);
long double fabs(long double x);
double fabs(double _Complex x);
float fabs(float _Complex x);
long double fabs(long double _Complex x);

double fdim(double x, double y);
float fdim(float x, float y);
long double fdim(long double x, long double y);

double floor(double x);
float floor(float x);
long double floor(long double x);

double fma(double x, double y, double z);
float fma(float x, float y, float z);
```

```
long double fma(long double x, long double y, long double z);

double fmax(double x, double y);
float fmax(float x, float y);
long double fmax(long double x, long double y);

double fmin(double x, double y);
float fmin(float x, float y);
long double fmin(long double x, long double y);

double fmod(double x, double y);
float fmod(float x, float y);
long double fmod(long double x, long double y);

double frexp(double x, int *pexp);
float frexp(float x, int *pexp);
long double frexp(long double x, int *pexp);

double hypot(double x, double y);
float hypot(float x, float y);
long double hypot(long double x, long double y);

int ilogb(double x);
int ilogb(float x);
int ilogb(long double x);

double ldexp(double x, int ex);
float ldexp(float x, int ex);
long double ldexp(long double x, int ex);

double lgamma(double x);
float lgamma(float x);
long double lgamma(long double x);

long long llrint(double x);
long long llrint(float x);
long long llrint(long double x);

long long llround(double x);
long long llround(float x);
long long llround(long double x);

double log(double x);
float log(float x);
long double log(long double x);
double _Complex log(double _Complex x);
float _Complex log(float _Complex x);
long double _Complex log(long double _Complex x);

double log10(double x);
float log10(float x);
long double log10(long double x);

double log1p(double x);
float log1p(float x);
long double log1p(long double x);
```

```
double log2(double x);
float log2(float x);
long double log2(long double x);

double logb(double x);
float logb(float x);
long double logb(long double x);

long lrint(double x);
long lrint(float x);
long lrint(long double x);

long lround(double x);
long lround(float x);
long lround(long double x);

double modf(double x, double *pint);
float modf(float x, float *pint);
long double modf(long double x, long double *pint);

double nearbyint(double x);
float nearbyint(float x);
long double nearbyint(long double x);

double nextafter(double x, double y);
float nextafter(float x, float y);
long double nextafter(long double x, long double y);

double nexttoward(double x, long double y);
float nexttoward(float x, long double y);
long double nexttoward(long double x, long double y);

double pow(double x, double y);
float pow(float x, float y);
long double pow(long double x, long double y);
double _Complex pow(double _Complex x);
float _Complex pow(float _Complex x);
long double _Complex pow(long double _Complex x);

double remainder(double x, double y);
float remainder(float x, float y);
long double remainder(long double x, long double y);

double remquo(double x, double y, int *pquo);
float remquo(float x, float y, int *pquo);
long double remquo(long double x, long double y, int *pquo);

double rint(double x);
float rint(float x);
long double rint(long double x);

double round(double x);
float round(float x);
long double round(long double x);
```

```
double scalbln(double x, long ex);
float scalbln(float x, long ex);
long double scalbln(long double x, long ex);
double scalbn(double x, int ex);
float scalbn(float x, int ex);
long double scalbn(long double x, int ex);

double sin(double x);
float sin(float x);
long double sin(long double x);
double _Complex sin(double _Complex x);
float _Complex sin(float _Complex x);
long double _Complex sin(long double _Complex x);

double sinh(double x);
float sinh(float x);
long double sinh(long double x);
double _Complex sinh(double _Complex x);
float _Complex sinh(float _Complex x);
long double _Complex sinh(long double _Complex x);

double sqrt(double x);
float sqrt(float x);
long double sqrt(long double x);
double _Complex sqrt(double _Complex x);
float _Complex sqrt(float _Complex x);
long double _Complex sqrt(long double _Complex x);

double tan(double x);
float tan(float x);
long double tan(long double x);
double _Complex tan(double _Complex x);
float _Complex tan(float _Complex x);
long double _Complex tan(long double _Complex x);

double tanh(double x);
float tanh(float x);
long double tanh(long double x);
double _Complex tanh(double _Complex x);
float _Complex tanh(float _Complex x);
long double _Complex tanh(long double _Complex x);

double tgamma(double x);
float tgamma(float x);
long double tgamma(long double x);

double trunc(double x);
float trunc(float x);
long double trunc(long double x);
```

## 11.28.1 acos

```
double acos(double x);
float acos(float x);
long double acos(long double x);
double _Complex acos(double _Complex x);
float _Complex acos(float _Complex x);
long double _Complex acos(long double _Complex x);
```

The function returns the arccosine of `x`.

## 11.28.2 acosh

```
double acosh(double x);
float acosh(float x);
long double acosh(long double x);
double _Complex acosh(double _Complex x);
float _Complex acosh(float _Complex x);
long double _Complex acosh(long double _Complex x);
```

The function returns the hyperbolic arccosine of `x`.

## 11.28.3 carg

```
double carg(double _Complex x);
float carg(float _Complex x);
long double carg(long double _Complex x);
```

The function returns the phase angle of `x`.

## 11.28.4 asin

```
double asin(double x);
float asin(float x);
long double asin(long double x);
double _Complex asin(double _Complex x);
float _Complex asin(float _Complex x);
long double _Complex asin(long double _Complex x);
```

The function returns the arcsine of `x`.

## 11.28.5 asinh

```
double asinh(double x);
float asinh(float x);
long double asinh(long double x);
double _Complex asinh(double _Complex x);
float _Complex asinh(float _Complex x);
long double _Complex asinh(long double _Complex x);
```

The function returns the hyperbolic arcsine of `x`.

## 11.28.6 atan

```
double atan(double x);
float atan(float x);
long double atan(long double x);
double _Complex atan(double _Complex x);
float _Complex atan(float _Complex x);
long double _Complex atan(long double _Complex x);
```

The function returns the arctangent of `x`.

## 11.28.7 atan2

```
double atan2(double y, double x);
float atan2(float y, float x);
long double atan2(long double y, long double x);
```

The function returns the angle whose tangent is `y/x`, in the full angular range `[-pi, +pi]` radians. A domain error might occur if both `x` and `y` are zero.

## 11.28.8 atanh

```
double atanh(double x);
float atanh(float x);
long double atanh(long double x);
double _Complex atanh(double _Complex x);
float _Complex atanh(float _Complex x);
long double _Complex atanh(long double _Complex x);
```

The function returns the hyperbolic arctangent of `x`.

## 11.28.9 cbrt

```
double cbrt(double x);
float cbrt(float x);
long double cbrt(long double x);
```

The function returns the real cube root of `x,x^(1/3)`.

## 11.28.10 ceil

```
double ceil(double x);
float ceil(float x);
long double ceil(long double x);
```

The function returns the smallest integer value not less than `x`.

## 11.28.11 cimag

```
double cimag(double _Complex x);
float cimag(float _Complex x);
long double cimag(long double _Complex x);
```

The function returns the imaginary part of `x`.

## 11.28.12 conj

```
double _Complex conj(double _Complex x);
float _Complex conj(float _Complex x);
long double _Complex conj(long double _Complex x);
```

The function returns the conjugate of `x`.

## 11.28.13 copysign

```
double copysign(double x, double y);
float copysign(float x, float y);
long double copysign(long double x, long double y);
```

The function returns `x`, with its sign bit replaced from `y`.

## 11.28.14 cos

```
double cos(double x);
float cos(float x);
long double cos(long double x);
double _Complex cos(double _Complex x);
float _Complex cos(float _Complex x);
long double _Complex cos(long double _Complex x);
```

The function returns the cosine of `x`.

## 11.28.15 cosh

```
double cosh(double x);
float cosh(float x);
long double cosh(long double x);
double _Complex cosh(double _Complex x);
float _Complex cosh(float _Complex x);
long double _Complex cosh(long double _Complex x);
```

The function returns the hyperbolic cosine of `x`.

## 11.28.16cproj

```
double _Complex cproj(double _Complex x);
float _Complex cproj(float _Complex x);
long double _Complex cproj(long double _Complex x);
```

The function returns a projection of x onto the Riemann sphere. Specifically, if either component of x is an infinity of either sign, the function returns a value whose real part is positive infinity and whose imaginary part is zero with the same sign as the imaginary part of x. Otherwise, the function returns x.

## 11.28.17creal

```
double creal(double _Complex x);
float creal(float _Complex x);
long double creal(long double _Complex x);
```

The function returns the real part of x.

## 11.28.18erf

```
double erf(double x);
float erf(float x);
long double erf(long double x);
```

The function returns the error function of x.

## 11.28.19erfc

```
double erfc(double x);
float erfc(float x);
long double erfc(long double x);
```

The function returns the complementary error function of x.

## 11.28.20exp

```
double exp(double x);
float exp(float x);
long double exp(long double x);
double _Complex exp(double _Complex x);
float _Complex exp(float _Complex x);
long double _Complex exp(long double _Complex x);
```

The function returns the exponential of x, e^x.

## 11.28.21exp2

```
double exp2(double x);
float exp2(float x);
long double exp2(long double x);
```

The function returns two raised to the power `x, 2^x`.

## 11.28.22expm1

```
double expm1(double x);
float expm1(float x);
long double expm1(long double x);
```

The function returns one less than the exponential function of `x, e^x - 1`.

## 11.28.23fabs

```
double fabs(double x);
float fabs(float x);
long double fabs(long double x);
double fabs(double _Complex x);
float fabs(float _Complex x);
long double fabs(long double _Complex x);
```

The function returns the magnitude of `x, |x|`.

## 11.28.24fdim

```
double fdim(double x, double y);
float fdim(float x, float y);
long double fdim(long double x, long double y);
```

The function returns the larger of `x - y` and zero.

## 11.28.25floor

```
double floor(double x);
float floor(float x);
long double floor(long double x);
```

The function returns the largest integer value not greater than `x`.

## 11.28.26fma

```
double fma(double x, double y, double z);
float fma(float x, float y, float z);
long double fma(long double x, long double y, long double z);
```

The function returns `x * y + z`, to arbitrary intermediate precision.

## 11.28.27fmax

```
double fmax(double x, double y);
float fmax(float x, float y);
long double fmax(long double x, long double y);
```

The function returns the larger (more positive) of `x` and `y`

## 11.28.28fmin

```
double fmin(double x, double y);
float fmin(float x, float y);
long double fmin(long double x, long double y);
```

The function returns the smaller (more negative) of `x` and `y`.

## 11.28.29fmod

```
double fmod(double x, double y);
float fmod(float x, float y);
long double fmod(long double x, long double y);
```

The function returns the remainder of `x/y`, which is defined as follows:

- If `y` is zero, the function either reports a domain error or simply returns zero.
- Otherwise, if `0 <= x`, the value is `x - i*y` for some integer `i` such that:

  ```
  0 <= i*|y| <= x < (i + 1)*|y|
  ```

- Otherwise, `x < 0` and the value is `x - i*y` for some integer `i` such that:

  ```
  i*|y| <= x < (i + 1)*|y| <= 0
  ```

## 11.28.30frexp

```
double frexp(double x, int *pexp);
float frexp(float x, int *pexp);
long double frexp(long double x, int *pexp);
```

The function determines a fraction `frac` and an exponent integer `ex` that represent the value of `x`. It returns the value `frac` and stores the integer `ex` in `*pexp`, such that:

```
|frac| is in the interval [1/2, 1) or is zero
x == frac * 2^ex
```

If `x` is zero, `*pexp` is also zero.

## 11.28.31hypot

```
double hypot(double x, double y);
float hypot(float x, float y);
long double hypot(long double x, long double y);
```

The function returns the square root of `x^2 + y^2`.

## 11.28.32ilogb

```
int ilogb(double x);
int ilogb(float x);
int ilogb(long double x);
```

The function returns:

- For `x` not-a-number (NaN), the value of the macro `FP_ILOGBNAN`
- For x equal to zero, the value of the macro `FP_ILOGB0`
- For x equal to positive or negative infinity, the value of the macro `INT_MAX`

Otherwise, it returns `(int)logb(x)`.

## 11.28.33ldexp

```
double ldexp(double x, int ex);
float ldexp(float x, int ex);
long double ldexp(long double x, int ex);
```

The function returns `x` * 2^ex.

## 11.28.34lgamma

```
double lgamma(double x);
float lgamma(float x);
long double lgamma(long double x);
```

The function returns the natural logarithm of the absolute value of the gamma function of `x`.

## 11.28.35llrint

```
long long llrint(double x);
long long llrint(float x);
long long llrint(long double x);
```

The function returns the nearest `long long` integer to `x`, consistent with the current rounding mode. It raises an invalid floating point exception if the magnitude of the rounded value is too large to represent. And it raises an inexact floating point exception if the return value does not equal `x`.

## 11.28.36llround

```
long long llround(double x);
long long llround(float x);
long long llround(long double x);
```

The function returns the nearest `long long` integer to `x`, rounding halfway values away from zero, regardless of the current rounding mode.

## 11.28.37log

```
double log(double x);
float log(float x);
long double log(long double x);
double _Complex log(double _Complex x);
float _Complex log(float _Complex x);
long double _Complex log(long double _Complex x);
```

The function returns the natural logarithm of `x`.

## 11.28.38log10

```
double log10(double x);
float log10(float x);
long double log10(long double x);
```

The function returns the base-10 logarithm of `x`. A domain error occurs if `x < 0`.

## 11.28.39log1p

```
double log1p(double x);
float log1p(float x);
long double log1p(long double x);
```

The function returns the natural logarithm of `1 + x`. A domain error occurs if `x < -1`.

## 11.28.40log2

```
double log2(double x);
float log2(float x);
long double log2(long double x);
```

The function returns the base-2 logarithm of `x`. A domain error occurs if `x < 0`.

## 11.28.41logb

```
double logb(double x);
float logb(float x);
long double logb(long double x);
```

The function determines an integer exponent `ex` and a fraction `frac` that represent the value of a finite `x`. It returns the value ex such that:

```
x == frac * ex^FLT_RADIX
|frac| is in the interval [1, FLT_RADIX)
```

A domain error might occur if `x` is zero.

## 11.28.42lrint

```
long lrint(double x);
long lrint(float x);
long lrint(long double x);
```

The function returns the nearest `long` integer to `x`, consistent with the current rounding mode. It raises an invalid floating point exception if the magnitude of the rounded value is too large to represent. And it raises an inexact floating point exception if the return value does not equal `x`.

## 11.28.43lround

```
long lround(double x);
long lround(float x);
long lround(long double x);
```

The function returns the nearest `long` integer to `x`, rounding halfway values away from zero, regardless of the current rounding mode.

## 11.28.44modf

```
double modf(double x, double *pint);
float modf(float x, float *pint);
long double modf(long double x, long double *pint);
```

The function determines an integer `i` plus a fraction `frac` that represent the value of `x`. It returns the value `frac` and stores the integer `i` in `*pint`, such that:

- x == frac + i
- `|frac|` is in the interval `[0, 1)`
- Both `frac` and `i` have the same sign as x

## 11.28.45nearbyint

```
double nearbyint(double x);
float nearbyint(float x);
long double nearbyint(long double x);
```

The function returns `x` rounded to the nearest integer, using the current rounding mode but without raising an inexact floating point exception.

## 11.28.46nextafter

```
double nextafter(double x, double y);
float nextafter(float x, float y);
long double nextafter(long double x, long double y);
```

The function returns:

- If `x < y`, the next representable value after `x`
- If `x == y`, `y`
- If `x > y`, the next representable value before `x`

## 11.28.47nexttoward

```
double nexttoward(double x, long double y);
float nexttoward(float x, long double y);
long double nexttoward(long double x, long double y);
```

The function returns:

- if `x < y`, the next representable value after `x`
- If `x == y`, `y`
- If `x > y`, the next representable value before `x`

## 11.28.48pow

```
double pow(double x, double y);
float pow(float x, float y);
long double pow(long double x, long double y);
double _Complex pow(double _Complex x, double _Complex y);
float _Complex pow(float _Complex x, float _Complex y);
long double _Complex pow(long double _Complex x, long double _Complex
y);
```

The function returns `x` raised to the power `y, x^y`.

## 11.28.49remainder

```
double remainder(double x, double y);
float remainder(float x, float y);
long double remainder(long double x, long double y);
```

The function effectively returns `remquo(x, y, &temp)`, where `temp` is a temporary object of type `int` local to the function.

## 11.28.50remquo

```
double remquo(double x, double y, int *pquo);
float remquo(float x, float y, int *pquo);
long double remquo(long double x, long double y, int *pquo);
```

The function computes the remainder `rem == x - n*y`, where `n == x/y` rounded to the nearest integer, or to the nearest even integer if `|n - x/y| == 1/2`. If `rem` is zero, it has the same sign as `x`. A domain error occurs if `y` is zero.

The function stores in `*pquo` at least three of the low order bits of `|x/y|`, negated if `x/y < 0`. It returns rem.

## 11.28.51rint

```
double rint(double x);
float rint(float x);
long double rint(long double x);
```

The function returns `x` rounded to the nearest integer, using the current rounding mode. It might raise an inexact floating point exception if the return value does not equal `x`.

## 11.28.52round

```
double round(double x);
float round(float x);
long double round(long double x);
```

The function returns `x` rounded to the nearest integer `n`, or to the value with larger magnitude if `|n - x| == 1/2`.

## 11.28.53scalbln

```
double scalbln(double x, long ex);
float scalbln(float x, long ex);
long double scalbln(long double x, long ex);
```

The function returns `x * FLT_RADIX^ex`.

## 11.28.54scalbn

```
double scalbn(double x, int ex);
float scalbn(float x, int ex);
long double scalbn(long double x, int ex);
```

The function returns `x * FLT_RADIX^ex`.

## 11.28.55sin

```
double sin(double x);
float sin(float x);
long double sin(long double x);
double _Complex sin(double _Complex x);
float _Complex sin(float _Complex x);
long double _Complex sin(long double _Complex x);
```

The function returns the sine of `x`.

## 11.28.56sinh

```
double sinh(double x);
float sinh(float x);
long double sinh(long double x);
double _Complex sinh(double _Complex x);
float _Complex sinh(float _Complex x);
long double _Complex sinh(long double _Complex x);
```

The function returns the hyperbolic sine of `x`.

## 11.28.57sqrt

```
double sqrt(double x);
float sqrt(float x);
long double sqrt(long double x);
double _Complex sqrt(double _Complex x);
float _Complex sqrt(float _Complex x);
long double _Complex sqrt(long double _Complex x);
```

The function returns the real square root of `x,x^(1/2)`.

## 11.28.58tan

```
double tan(double x);
float tan(float x);
long double tan(long double x);
double _Complex tan(double _Complex x);
float _Complex tan(float _Complex x);
long double _Complex tan(long double _Complex x);
```

The function returns the tangent of `x`.

## 11.28.59tanh

```
double tanh(double x);
float tanh(float x);
long double tanh(long double x);
double _Complex tanh(double _Complex x);
float _Complex tanh(float _Complex x);
long double _Complex tanh(long double _Complex x);
```

The function returns the hyperbolic tangent of `x`.

## 11.28.60tgamma

```
double tgamma(double x);
float tgamma(float x);
long double tgamma(long double x);
```

The function computes the gamma function of `x`. A domain error occurs if x is a negative integer.

## 11.28.61trunc

```
double trunc(double x);
float trunc(float x);
long double trunc(long double x);
```

The function returns `x` rounded to the nearest integer n not larger in magnitude than `x` (toward zero).

# 11.29 <time.h>

Include the standard header `<time.h>` to declare several functions that help you manipulate times. Figure 11-6 summarizes the functions and the object types that they convert between.

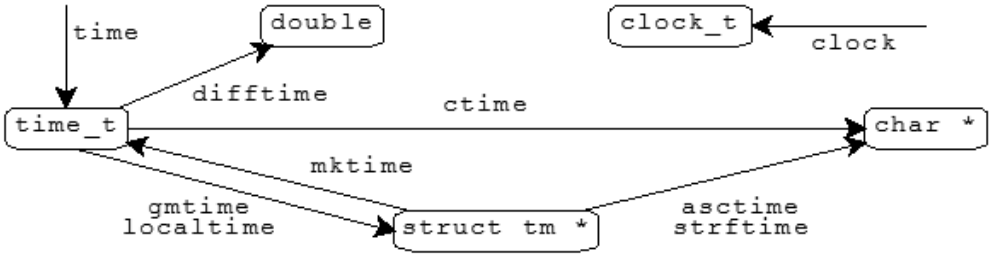


**Figure 11-6 Time conversion functions**

The functions share two `static-duration` objects that hold values computed by the functions:

- A time string of type `array of char`
- A time structure of type `struct tm`

A call to one of these functions can alter the value that was stored earlier in a `static-duration` object by another of these functions.

```
/* MACROS */
#define CLOCKS_PER_SEC <integer constant expression > 0>
#define NULL <either 0, 0L, or (void *)0> [0 in C++]

/* TYPES */
typedef a-type clock_t;
typedef ui-type size_t;
typedef a-type time_t;
struct tm;

/* FUNCTIONS */
char *asctime(const struct tm *tptr);
char *asctime_r(const struct tm restrict *tm, char * restrict buf);
[POSIX]
clock_t clock(void);
char *ctime(const time_t *tod);
char *ctime_r(const time_t *clock, char *buf); [POSIX]
double difftime(time_t t1, time_t t0);

struct tm *gmtime(const time_t *tod);
struct tm *gmtime_r(const time_t * restrict clock, struct tm *
restrict result); [POSIX]
struct tm *localtime(const time_t *tod);
struct tm *localtime_r(const time_t * restrict clock, struct tm *
restrict result); [POSIX]
time_t mktime(struct tm *tptr);
size_t strftime(char *restrict s, size_t n, const char *restrict
format, const struct tm *restrict tptr);
```

```
char * strptime(const char * restrict buf, const char * restrict
format, struct tm * restrict tm); [POSIX]
time_t time(time_t *tod);
```

## 11.29.1 asctime

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

The function stores in the `static-duration` time string a 26-character English-language representation of the time encoded in `*tptr`. It returns the address of the `static-duration` time string. The text representation takes the form:

```
Sun Dec2 06:55:15 1979\n\0
```

## 11.29.2 asctime_r

```
char *asctime_r(const struct tm restrict *tm, char * restrict buf);
[POSIX]
```

The function provides the same functionality as `asctime`, differing in that the caller must supply a buffer area `buf` with a size of at least 26 bytes, in which the result is stored.

## 11.29.3 clock

```
clock_t clock(void);
```

The function returns the number of clock ticks of elapsed processor time, counting from a time related to program startup, or it returns `-1` if the target environment cannot measure elapsed processor time.

## 11.29.4 CLOCKS_PER_SEC

```
#define CLOCKS_PER_SEC <integer constant expression > 0>
```

The macro yields the number of clock ticks, returned by clock, in one second.

## 11.29.5 clock_t

```
typedef a-type clock_t;
```

The type is the arithmetic type a-type of an object that you declare to hold the value returned by clock, representing elapsed processor time.

### 11.29.6 ctime

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

The function converts the calendar time in `*tod` to a text representation of the local time in the `static-duration` time string. It returns the address of that string. It is equivalent to `asctime(localtime(tod))`.

### 11.29.7 ctime_r

```
char *ctime_r(const time_t *clock, char *buf); [POSIX]
```

The function provides the same functionality as `ctime` differing in that the caller must supply a buffer area `buf` with a size of at least 26 bytes, in which the result is stored.

### 11.29.8 difftime

```
double difftime(time_t t1, time_t t0);
```

The function returns the difference `t1 - t0`, in seconds, between the calendar time `t0` and the calendar time `t1`.

### 11.29.9 gmtime

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

The function stores in the `static-duration` time structure an encoding of the calendar time in `*tod`, expressed as Coordinated Universal Time, or UTC [sic]. (UTC was formerly Greenwich Mean Time, or GMT).

It returns the address of that structure, or [added with C99] a NULL pointer if it cannot generate the encoding.

### 11.29.10 gmtime_r

```
struct tm *gmtime_r(const time_t * restrict clock, struct tm *
restrict result); [POSIX]
```

The function provide the same functionality as `gmtime`, differing in that the caller must supply a buffer area result in which the result is stored.

### 11.29.11 localtime

```
struct tm *localtime(const time_t *tod);
```

The function stores in the `static-duration` time structure an encoding of the calendar time in `*tod`, expressed as local time. It returns the address of that structure, or [added with C99] a NULL pointer if it cannot generate the encoding.

## 11.29.12localtime_r

```
struct tm *localtime_r(const time_t * restrict clock, struct tm *
restrict result); [POSIX]
```

The function provides the same functionality as `localtime`, differing in that the caller must supply a buffer area result in which the result is stored. Also, `localtime_r` does not imply initialization of the local time conversion information; the application may need to do so by calling `tzset`.

## 11.29.13mktime

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

The function alters the values stored in `*tptr` to represent an equivalent encoded local time, but with the values of all members within their normal ranges. It then determines the values `tptr->wday` and `tptr->yday` from the values of the other members. It returns the calendar time equivalent to the encoded time, or it returns a value of `-1` if the calendar time cannot be represented.

## 11.29.14NULL

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
```

The macro yields a NULL pointer constant that is usable as an address constant expression.

## 11.29.15size_t

```
typedef ui-type size_t;
```

The type is the unsigned integer type `ui-type` of an object that you declare to store the result of the `sizeof` operator.

## 11.29.16strftime

```
size_t strftime(char *restrict s, size_t n, const char *restrict
format, const struct tm *restrict tptr);
```

The function generates formatted text, under the control of the format `format` and the values stored in the time structure `*tptr`. It stores each generated character in successive locations of the array object of size `n` whose first element has the address `s`. The function then stores a NULL character in the next location of the array.

It returns `x`, the number of characters generated, if `x < n`; otherwise, it returns zero and the values stored in the array are indeterminate.

For each multibyte character other than `%` in the format, the function stores that multibyte character in the array object. Each occurrence of `%` followed by an optional qualifier and another character in the format is a conversion specifier. The optional qualifiers [added with C99] are:

- `E` – To represent times in terms of a locale-specific era (such as 1 BC instead of 0000).
- `O` – To represent numeric values with a set of locale-specific alternate digits (such as first instead of 1).

For each conversion specifier, the function stores a replacement character sequence.

The following table lists all conversion specifiers defined for `strftime`. The fields used in `*tptr` follow. Example replacement character sequences in parentheses follow each description. All examples are for the "C" locale, which ignores any optional qualifier, using the date and time Sunday, 2 December 1979 at 06:55:15 AM EST.

For a Sunday week of the year, week 1 begins with the first Sunday on or after 1 January. For a Monday week of the year, week 1 begins with the first Monday on or after 1 January. An ISO 8601 week of the year is the same as a Monday week of the year, with the following exceptions:

- If 1 January is a Tuesday, Wednesday, or Thursday, the week number is one greater. Moreover, days back to and including the immediately preceding Monday in the preceding year are included in week 1 of the current year.
- If 1 January is a Friday, Saturday, or Sunday, days up to but not including the immediately following Monday in the current year are included in the last week (52 or 53) of the preceding year.

Table 11-10 the conversion specifications marked with a + are all added with C99.

**Table 11-10 strftime conversion specifications**

| Specifier | Fields | Description (example) |
|---|---|---|
| `%a` | `tm_wday` | Abbreviated weekday name (Sun) |
| `%A` | `tm_wday` | Full weekday name (Sunday) |
| `%b` | `tm_mon` | Abbreviated month name (Dec) |
| `%B` | `tm_mon` | Full month name (December) |
| `%c` | `[all]` | Date and time (Sun Dec2 06:55:15 1979) |

**Table 11-10 strftime conversion specifications**

| Specifier | Fields | Description (example) |
|---|---|---|
| %Ec | [all] | + Era-specific date and time |
| %C | tm_year | + Year/100 (19) |
| %EC | tm_mday | |
| tm_mon | | |
| tm_year | + era specific era name | |
| %d | tm_mday | Day of the month (02) |
| %D | tm_mday | |
| tm_mon | | |
| tm_year | + month/day/year from 01/01/00 (12/02/79) | |
| %e | tm_mday | + Day of the month, leading space for zero (2) |
| %F | tm_mday | |
| tm_mon | | |
| tm_year | + year-month-day (1979-12-02) | |
| %g | tm_wday | |
| tm_yday | | |
| tm_year | + year for ISO 8601 week, from 00 (79) | |
| %G | tm_wday | |
| tm_yday | | |
| tm_year | + year for ISO 8601 week, from 0000 (1979) | |
| %h | tm_mon | + Same as %b (Dec) |
| %H | tm_hour | Hour of the 24-hour day, from 00 (06) |
| %I | tm_hour | Hour of the 12-hour day, from 01 (06) |
| %j | tm_yday | Day of the year, from 001 (336) |
| %m | tm_mon | Month of the year, from 01 (12) |
| %M | tm_min | Minutes after the hour (55) |
| %n | + newline character \n | |
| %p | tm_hour | AM/PM indicator (AM) |
| %r | tm_sec | |
| tm_min | | |
| tm_hour | + 12-hour time, from 01:00:00 AM (06:55:15 AM) | |
| %Er | tm_sec | |
| tm_min | | |
| tm_hour | | |

**Table 11-10 strftime conversion specifications**

| Specifier | Fields | Description (example) |
|---|---|---|
| tm_mday | | |
| tm_mon | | |
| tm_year | + era-specific date and 12-hour time | |
| %R | tm_min | |
| tm_hour | + hour:minute, from 01:00 (06:55) | |
| %S | tm_sec | Seconds after the minute (15) |
| %t | + horizontal tab character \t | |
| %T | tm_sec | |
| tm_min | | |
| tm_hour | + 24-hour time, from 00:00:00 (06:55:15) | |
| %u | tm_wday | + ISO 8601 day of the week, to 7 for Sunday (7) |
| %U | tm_wday | |
| tm_yday | Sunday week of the year, from 00 (48) | |
| %V | tm_wday | |
| tm_yday | | |
| tm_year | + ISO 8601 week of the year, from 01 (48) | |
| %w | tm_wday | Day of the week, from 0 for Sunday (0) |
| %W | tm_wday | |
| tm_yday | Monday week of the year, from 00 (48) | |
| %x | [all] | Date (02/12/79) |
| %Ex | [all] | + Era-specific date |
| %X | [all] | Time, from 00:00:00 (06:55:15) |
| %EX | [all] | + Era-specific time |
| %y | tm_year | Year of the century, from 00 (79) |
| %Ey | tm_mday | |
| tm_mon | | |
| tm_year | + year of the era | |
| %Y | tm_year | Year (1979) |
| %EY | tm_mday | |
| tm_mon | | |
| tm_year | + era-specific era name and year of the era | |

**Table 11-10 strftime conversion specifications**

| Specifier | Fields | Description (example) |
|---|---|---|
| %z | tm_isdst + time zone (hours east*100 + minutes), if any (-0500) | |
| %Z | tm_isdst time zone name, if any (EST) | |
| %% | percent character % | |

The current locale category LC_TIME can affect these replacement character sequences.

## 11.29.17 strptime

```
char * strptime(const char * restrict buf, const char * restrict
format, struct tm * restrict tm); [POSIX]
```

The function converts the character string pointed to by buf to values that are stored in the tm structure pointed to by tm, using the format specified by format.

The format string consists of zero or more conversion specifications, whitespace characters as defined by isspace, and ordinary characters. All ordinary characters in format are compared directly against the corresponding characters in buf; comparisons that fail will cause strptime to fail. Whitespace characters in format match any number of whitespace characters in buf, including none.

A conversion specification consists of a percent sign (%) followed by one or two conversion characters which specify the replacement required. There must be white-space or other non-alphanumeric characters between any two conversion specifications.

Conversion of alphanumeric strings (such as month and weekday names) is done without regard to case. Conversion specifications which cannot be matched will cause strptime to fail.

The LC_TIME category defines the locale values for the conversion specifications. Table 11-11 lists the supported conversion specifications.

**Table 11-11 strptime conversion specifications**

| Spec | Description |
|---|---|
| %a | The day of week, using the locale's weekday names; either the abbreviated or full name can be specified. |
| %A | The same as %a. |
| %b | The month, using the locale's month names; either the abbreviated or full name can be specified. |
| %B | The same as %b. |
| %c | The date and time, using the locale's date and time format. |
| %C | The century number [0,99]; leading zeros are permitted but not required. This conversion should be used in conjunction with the %y conversion. |
| %d | The day of month [1,31]; leading zeros are permitted but not required. |

**Table 11-11 strptime conversion specifications**

| Spec | Description |
|---|---|
| %D | The date as %m/%d/%y. |
| %e | The same as %d. |
| %F | The date as %Y-%m-%d (the ISO 8601 date format). |
| %g | The year corresponding to the ISO week number, without the century. (A NetBSD extension.) |
| %G | The year corresponding to the ISO week number, with the century. (A NetBSD extension.) |
| %h | The same as %b. |
| %H | The hour (24-hour clock) [0,23]; leading zeros are permitted but not required. |
| %I | The hour (12-hour clock) [1,12]; leading zeros are permitted but not required. |
| %j | The day number of the year [1,366]; leading zeros are permitted but not required. |
| %k | The same as %H. |
| %l | The same as %I. |
| %m | The month number [1,12]; leading zeros are permitted but not required. |
| %M | The minute [0,59]; leading zeros are permitted but not required. |
| %n | Any white-space, including none. |
| %p | The locale's equivalent of a.m. or p.m. |
| %r | The time (12-hour clock) with %p, using the locale's time format. |
| %R | The time as %H:%M. |
| %S | The seconds [0,61]; leading zeros are permitted but not required. |
| %s | The number of seconds since the Epoch, UTC (see mktime). (A NetBSD extension.) |
| %t | Any white-space, including none. |
| %T | The time as %H:%M:%S. |
| %u | The day of the week as a decimal number, where Monday = 1. (A NetBSD extension.) |
| %U | The week number of the year (Sunday as the first day of the week) as a decimal number [0,53]; leading zeros are permitted but not required. All days in a year preceding the first Sunday are considered to be in week 0. |
| %V | The ISO 8601:1988 week number as a decimal number. If the week (starting on Monday) that contains January 1 has more than three days in the new year, it is considered the first week of the year. If it has fewer than four days in the new year, it is considered the last week of the previous year. Weeks are numbered from 1 to 53. (A NetBSD extension.) |
| %w | The weekday as a decimal number [0,6], with 0 representing Sunday; leading zeros are permitted but not required. |
| %W | The week number of the year (Monday as the first day of the week) as a decimal number [0,53]; leading zeros are permitted but not required. All days in a year preceding the first Monday are considered to be in week 0. |
| %x | The date, using the locale's date format. |
| %X | The time, using the locale's time format. |
| %y | The year within the 20th century [69,99] or the 21st century [0,68]; leading zeros are permitted but not required. If specified in conjunction with %C, specifies the year [0,99] within that century. |

**Table 11-11 strptime conversion specifications**

| Spec | Description |
|---|---|
| %Y | The year, including the century (i.e., 1996). |
| %z | An ISO 8601 or RFC-2822 timezone specification. This is one of the following:<br>■ The offset from Coordinated Universal Time (UTC) specified as: [+-]hhmm, [+-]hh:mm', or [+-]hh<br>■ UTC specified as: GMT' (Greenwich Mean Time), UT (Universal Time), or Z (Zulu Time)<br>■ A three-character US timezone specified as: EDT, EST, CDT, CST, MDT, MST, PDT, or PST, where<br>❒ The first letter stands for Eastern (E), Central (C), Mountain (M) or Pacific (P)<br>❒ The second letter stands for Daylight (D or summer) time or Standard (S) time<br>■ A single letter military timezone specified as: A through I and K through Y. (A NetBSD extension.) |
| %Z | Time zone name or no characters when time zone information is unavailable. (A NetBSD extension.)<br>**NOTE:** The %Z format specifier only accepts timezone abbreviations of the local timezone, or the value GMT. This limitation is caused by the ambiguity of overloaded timezone abbreviations, for example EST is both Eastern Standard Time and Eastern Australia Summer Time. |
| %% | Matches a literal %. No argument is converted. |

## Modified conversion specifications

For compatibility, certain conversion specifications can be modified by the `E` and `O` modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified conversion specification. Because there are currently neither alternative formats nor specifications supported by the system, the behavior will be as if the unmodified conversion specification were used.

Case is ignored when matching string items in `buf`, such as month and weekday names.

If successful, the `strptime` function returns a pointer to the character following the last character parsed. Otherwise, a NULL pointer is returned.

## 11.29.18time

```
time_t time(time_t *tod);
```

If `tod` is not a NULL pointer, the function stores the current calendar time in `*tod`. The function returns the current calendar time, if the target environment can determine it; otherwise, it returns `-1`.

## 11.29.19time_t

```
typedef a-type time_t;
```

The type is the arithmetic type `a-type` of an object that you declare to hold the value returned by time. The value represents calendar time.

## 11.29.20tm

```
struct tm {
    int tm_sec;     seconds after the minute (from 0)
    int tm_min;     minutes after the hour (from 0)
    int tm_hour;    hour of the day (from 0)
    int tm_mday;    day of the month (from 1)
    int tm_mon;     month of the year (from 0)
    int tm_year;    years since 1900 (from 0)
    int tm_wday;    days since Sunday (from 0)
    int tm_yday;    day of the year (from 0)
    int tm_isdst;   Daylight Saving Time flag
    };
```

The structure contains members that describe various properties of the calendar time. The members shown above can occur in any order, interspersed with additional members. The comment following each member briefly describes its meaning.

The member `tm_isdst` contains:

- A positive value if Daylight Saving Time is in effect
- Zero if Daylight Saving Time is not in effect
- A negative value if the status of Daylight Saving Time is not known (so the target environment should attempt to determine its status)

# 11.30 <uchar.h>

[Added with TR19769]

Include the added header <uchar.h> so that you can work with either 16-bit or 32-bit character encodings regardless of the size of wchar_t.

Use of this header does not require the additions to the C language mandated by TR19769, which include additional literals of the form u'x', u"abc", U'x', and U"abc".

```
/* MACROS */
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
#define __STDC_UTF_16__ unspecified
#define __STDC_UTF_32__ unspecified

/* TYPES */
typedef i-type char16_t;
typedef i-type char32_t;
typedef o-type mbstate_t;
typedef ui-type size_t;

/* FUNCTIONS */
size_t c16rtomb(char *restrict s, char16_t wc,
    mbstate_t *restrict ps);
size_t c32rtomb(char *restrict s, char16_t wc,
    mbstate_t *restrict ps);
size_t mbrtoc16(char16_t *restrict pwc, const char *restrict s,
    size_t n, mbstate_t *restrict ps);
size_t mbrtoc32(char16_t *restrict pwc, const char *restrict s,
    size_t n, mbstate_t *restrict ps);
c16rtomb
size_t c16rtomb(char *restrict s, wchar_t wc,
     mbstate_t *restrict ps);
```

The function determines the number of bytes needed to represent the wide character wc as a multibyte character, if possible. (Not all values representable as type wchar_t are necessarily valid wide-character codes.) The conversion state for the multibyte string is assumed to be *ps.

If s is not a NULL pointer and wc is a valid wide-character code, the function determines x, the number of bytes needed to represent wc as a multibyte character, and stores the converted bytes in the array of char beginning at s. (x cannot be greater than MB_CUR_MAX, defined in <stdlib.h>.) If wc is a NULL wide character, the function stores any shift sequence needed to restore the initial shift state. followed by a NULL byte. The resulting conversion state is the initial conversion state.

If s is a NULL pointer, the function effectively returns c16rtomb(buf, L'\0', ps), where buf is a buffer internal to the function. (The function thus returns the number of bytes needed to restore the initial conversion state and to terminate the multibyte string pending from a previous call to c16rtomb or wcsrtombs for the same string and conversion state.)

The function returns:

- `(size_t)-1` – If `wc` is an invalid wide-character code, in which case the function stores the value `EILSEQ` in `errno` (both macros defined in `<errno.h>`) and leaves the resulting conversion state undefined
- `x` – The number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that `x` bytes have been generated

## 11.30.1 c32rtomb

```
size_t c32rtomb(char *restrict s, wchar_t wc,
    mbstate_t *restrict ps);
```

The function determines the number of bytes needed to represent the wide character `wc` as a multibyte character, if possible. (Not all values representable as type `wchar_t` are necessarily valid wide-character codes.) The conversion state for the multibyte string is assumed to be `*ps`.

If `s` is not a NULL pointer and `wc` is a valid wide-character code, the function determines `x`, the number of bytes needed to represent `wc` as a multibyte character, and stores the converted bytes in the array of `char` beginning at `s`. (`x` cannot be greater than `MB_CUR_MAX`, defined in `<stdlib.h>`.) If `wc` is a NULL wide character, the function stores any shift sequence needed to restore the initial shift state followed by a NULL byte. The resulting conversion state is the initial conversion state.

If `s` is a NULL pointer, the function effectively returns `c32rtomb(buf, L'\0', ps)`, where `buf` is a buffer internal to the function. (The function thus returns the number of bytes needed to restore the initial conversion state and to terminate the multibyte string pending from a previous call to `c32rtomb` or `wcsrtombs` for the same string and conversion state.)

The function returns:

- `(size_t)-1` – If `wc` is an invalid wide-character code, in which case the function stores the value `EILSEQ` in `errno` (both macros defined in `<errno.h>`) and leaves the resulting conversion state undefined
- `x` – The number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that `x` bytes have been generated

## 11.30.2 char16_t

```
typedef i-type char16_t;
```

The type is the integer type `i-type` of a 16-bit character constant, such as `u'X'`. Declare an object of type `char16_t` to hold a 16-bit wide character.

### 11.30.3 char32_t

```
typedef i-type char32_t;
```

The type is the integer type `i-type` of a 32-bit character constant, such as `u'X'`. Declare an object of type `char32_t` to hold a 32-bit wide character.

### 11.30.4 mbrtoc16

```
size_t mbrtoc16(char16_t *restrict pwc, const char *restrict s,
    size_t n, mbstate_t *restrict ps);
```

The function determines the number of bytes in a multibyte string that completes the next multibyte character, if possible. The conversion state for the multibyte string is assumed to be `*ps`.

If `s` is not a NULL pointer, the function determines `x`, the number of bytes in the multibyte string `s` that complete or contribute to the next multibyte character. (`x` cannot be greater than `n`.) Otherwise, the function effectively returns `mbrtoc16(0, "", 1, ps)`, ignoring `pwc` and `n`. (The function thus returns zero only if the conversion state indicates that no incomplete multibyte character is pending from a previous call to `mbrlen`, `mbrtoc16`, or `mbsrtowcs` for the same string and conversion state.)

If `pwc` is not a NULL pointer, the function converts a completed multibyte character to its corresponding wide-character value and stores that value in `*pwc`.

The function returns:

- `(size_t)-3` – If no additional bytes are needed to complete the next multibyte character, in which case the resulting conversion state indicates that no additional bytes have been converted and the next multibyte character has been produced
- `(size_t)-2` – If, after converting all `n` characters, the resulting conversion state indicates an incomplete multibyte character
- `(size_t)-1` – If the function detects an encoding error before completing the next multibyte character, in which case the function stores the value `EILSEQ` in `errno` (both macros defined in `<errno.h>`) and leaves the resulting conversion state undefined
- Zero – If the next completed character is a NULL character, in which case the resulting conversion state is the initial conversion state
- `x` – The number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that `x` bytes have been converted and the next multibyte character has been produced

## 11.30.5 mbrtoc32

```
size_t mbrtoc32(char32_t *restrict pwc, const char *restrict s,
    size_t n, mbstate_t *restrict ps);
```

The function determines the number of bytes in a multibyte string that completes the next multibyte character, if possible. The conversion state for the multibyte string is assumed to be `*ps`.

If `s` is not a NULL pointer, the function determines `x`, the number of bytes in the multibyte string `s` that complete or contribute to the next multibyte character. (`x` cannot be greater than `n`.) Otherwise, the function effectively returns `mbrtoc32(0, "", 1, ps)`, ignoring `pwc` and `n`. (The function thus returns zero only if the conversion state indicates that no incomplete multibyte character is pending from a previous call to `mbrlen`, `mbrtoc32`, or `mbsrtowcs` for the same string and conversion state.)

If `pwc` is not a NULL pointer, the function converts a completed multibyte character to its corresponding wide-character value and stores that value in `*pwc`.

The function returns:

- `(size_t)-3` – If no additional bytes are needed to complete the next multibyte character, in which case the resulting conversion state indicates that no additional bytes have been converted and the next multibyte character has been produced
- `(size_t)-2` – If, after converting all `n` characters, the resulting conversion state indicates an incomplete multibyte character
- `(size_t)-1` – If the function detects an encoding error before completing the next multibyte character, in which case the function stores the value `EILSEQ` in `errno` (both macros defined in `<errno.h>`) and leaves the resulting conversion state undefined
- Zero – If the next completed character is a NULL character, in which case the resulting conversion state is the initial conversion state
- `x` – The number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that `x` bytes have been converted and the next multibyte character has been produced

## 11.30.6 mbstate_t

```
typedef o-type mbstate_t;
```

The type is an object type `o-type` that can represent a conversion state for any of the functions `c16rtomb`, `c32rtomb`, `mbrtoc16`, or `mbrtoc32`. A definition of the following form ensures that `mbst` represents the initial conversion state:

```
mbstate_t mbst = {0};
```

However, other values stored in an object of type `mbstate_t` can also represent this state. To test safely for this state, use the function `mbsinit`, declared in `<wchar.h>`.

### 11.30.7 NULL

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
```

The macro yields a NULL pointer constant that is usable as an address constant expression.

### 11.30.8 size_t

```
typedef ui-type size_t;
```

The type is the unsigned integer type `ui-type` of an object that you declare to store the result of the `sizeof` operator.

### 11.30.9 __STDC_UTF_16__

```
#define __STDC_UTF_16__ unspecified
```

The header defines the macro only if the functions `c16rtomb` and `mbrtoc16` treat elements of type `char16_t` as characters with the UTF-16 encoding.

### 11.30.10 __STDC_UTF_32__

```
#define __STDC_UTF_32__ unspecified
```

The header defines the macro only if the functions `c32rtomb` and `mbrtoc32` treat elements of type `char32_t` as characters with the UTF-32 encoding.

# 11.31 <unistd.h>

The <unistd.h> header defines the symbolic constants and structures which are not already defined or declared in some other header.

```
int access(const char *path, int mode); [POSIX]
unsigned int alarm(unsigned int seconds); [POSIX]
char *getcwd(char *buf, size_t size); [POSIX]
pid_t getpid(void); [POSIX]
char *getwd(char *path_name); [POSIX]
int isatty(int fd); [POSIX]
void swab(const void *src, void *dest, size_t len); [POSIX]
long int sysconf(int name); [POSIX]
int unlink(const char *path); [POSIX]

extern char *optarg;
extern int optind;
extern int optopt;
extern int opterr;
extern int optreset;
int getopt(int argc, char * const argv[], const char *optstring);
[POSIX]
access
int access(const char *path, int mode); [POSIX]
```

The access function checks the accessibility of the file named by path for the access permissions indicated by mode. The value of mode is the bitwise inclusive OR of the access permissions to be checked (R_OK for read permission, W_OK for write permission and X_OK for execute/search permission) or the existence test, F_OK. All components of the pathname path are checked for access permissions (including F_OK).

When Hexagon standalone programs execute on a simulator that is running on the Windows Operating system, the check for X_OK is not supported due to restrictions in the Windows platform.

The real user ID is used in place of the effective user ID and the real group access list (including the real group ID) are used in place of the effective ID for verifying permission.

If a process has super-user privileges and indicates success for R_OK or W_OK, the file may not actually have read or write permission bits set. If a process has super-user privileges and indicates success for X_OK, at least one of the user, group, or other execute bits is set. (However, the file may still not be executable.)

If path cannot be found or if an access modes would not be granted, a -1 value is returned; otherwise a 0 value is returned.

### 11.31.1 alarm

```
unsigned int alarm(unsigned int seconds); [POSIX]
```

The alarm function sets a timer to deliver the signal `SIGALRM` to the calling process seconds after the call to alarm. If an alarm has already been set with alarm but has not been delivered, another call to alarm will supersede the prior call. The request `alarm(0)` voids the current alarm and the signal `SIGALRM` will not be delivered. The maximum number of seconds allowed is 2,147,483,647.

The return value of `alarm` is the amount of time left on the timer from a previous call to alarm. If no alarm is currently set, the return value is `0`. If there is an error setting the timer, `alarm` returns `((unsigned int) -1)`.

### 11.31.2 getcwd

```
char *getcwd(char *buf, size_t size); [POSIX]
```

The function copies the absolute pathname of the current working directory into the memory referenced by `buf` and returns a pointer to `buf`. The size argument is the size, in bytes, of the array referenced by `buf`.

If `buf` is NULL, space is allocated as necessary to store the pathname. This space can later be freed with `free`.

Upon successful completion, a pointer to the pathname is returned. Otherwise a NULL pointer is returned and the global variable `errno` is set to indicate the error. In addition, getwd copies the error message associated with `errno` into the memory referenced by `buf`.

### 11.31.3 getopt

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

int getopt(int argc, char * const argv[], const char *optstring);
[POSIX]
```

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

The option string `optstring` may contain the following elements: individual characters, and characters followed by a colon to indicate an option argument is to follow. For example, an option string `"x"` recognizes an option `"-x"`, and an option string `"x:"` recognizes an option and argument `"-x argument"`. It does not matter to `getopt` if a following argument has leading whitespace.

On return from getopt, optarg points to an option argument, if it is anticipated, and the variable optind contains the index to the next argv argument for a subsequent call to getopt. The variable optopt saves the last known option character returned

The variables opterr and optind are both initialized to 1. The optind variable can be set to another value before a set of calls to getopt in order to skip over more or less argv entries.

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

The getopt function returns -1 when the argument list is exhausted. The interpretation of options in the argument list can be canceled by the option -- (double dash) which causes getopt to signal the end of argument processing and return -1. When all options have been processed (up to the first non-option argument), getopt returns -1.

The getopt function returns the next known option character in optstring. If getopt encounters a character not found in optstring or if it detects a missing option argument, it returns "?" (question mark). If optstring has a leading ":". a missing option argument causes ":" to be returned instead of "?". In either case, the variable optopt is set to the character that caused the error. The getopt function returns -1 when the argument list is exhausted.

For example:

```
extern char *optarg;
extern int optind;
int bflag, ch, fd;

bflag = 0;
while ((ch = getopt(argc, argv, "bf:")) != -1) {
   switch (ch) {
   case 'b':
 bflag = 1;
 break;
   case 'f':
 if ((fd = open(optarg, O_RDONLY, 0)) < 0) {
    (void)fprintf(stderr,
   "myname: %s: %s\n", optarg, strerror(errno));
    exit(1);
 }
 break;
   case '?':
   default:
 usage();
   }
}
argc -= optind;
argv += optind;
```

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

Option arguments are allowed to begin with "-"; this is reasonable but reduces the amount of error checking possible.

### Bugs

The getopt function was once specified to return EOF instead of -1. This was changed by IEEE Std. 1003.2-1992 ("POSIX.2") to decouple getopt from <stdio.h>.

A single dash ("-") can be specified as a character in optstring, however it should never have an argument associated with it. This allows getopt to be used with programs that expect "-" as an option flag.

This practice is wrong, and should not be used in any current development. It is provided for backwards compatibility only. Care should be taken not to use "-" as the first character in optstring to avoid a semantic conflict with GNU getopt, which assigns different meaning to an optstring that begins with a "-". By default, a single dash causes getopt to return -1.

It is also possible to handle digits as option letters. This allows getopt to be used with programs that expect a number ("-3") as an option. This practice is wrong, and should not be used in any current development. It is provided for backward compatibility only. The following code fragment works in most cases.

```
int ch;
long length;
char *p;

while ((ch = getopt(argc, argv, "0123456789")) != -1) {
        switch (ch) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
     p = argv[optind - 1];
     if (p[0] == '-' && p[1] == ch && !p[2])
  length = ch - '0';
     else
  length = strtol(argv[optind] + 1, NULL, 10);
     break;
        }
}
```

## 11.31.4 getpid

```
pid_t getpid(void); [POSIX]
```

The function returns the process ID of the calling process. The ID is guaranteed to be unique and is useful for constructing temporary file names.

When programs are compiled in standalone mode there is no job control. When the getpid routine is called the return value will be the thread number the program is executing on.

### 11.31.5 getwd

```
char *getwd(char *path_name); [POSIX]
```

The function is a compatibility routine that calls `getcwd` with its `buf` argument and a size of `MAXPATHLEN` (as defined in the include file `<sys/param.h>`). Obviously, `buf` should be at least `MAXPATHLEN` bytes in length.

These routines have traditionally been used by programs to save the name of a working directory for the purpose of returning to it. A much faster and less error-prone method of accomplishing this is to open the current directory (`.`) and use the `fchdir` function to return.

Upon successful completion, a pointer to the pathname is returned. Otherwise a NULL pointer is returned and the global variable `errno` is set to indicate the error. In addition, `getwd` copies the error message associated with `errno` into the memory referenced by `buf`.

Because `getwd` does not know the length of the supplied buffer, it is possible for a long (but valid) path to overflow the buffer and provide a means for an attacker to exploit the caller. `getcwd` should be used in place of `getwd` (the latter is only provided for compatibility purposes).

### 11.31.6 isatty

```
int isatty(int fd); [POSIX]
```

The function determines if the file descriptor `fd` refers to a valid terminal type device.

The function returns `1` if `fd` is associated with a terminal device; otherwise it returns `0` and `errno` is set to indicate the error.

### 11.31.7 swab

```
void swab(const void *src, void *dest, size_t len); [POSIX]
```

The function copies `len` bytes from the location referenced by `src` to the location referenced by `dest`, swapping adjacent bytes. The argument `len` must be even number.

### 11.31.8 sysconf

```
long int sysconf(int name); [POSIX]
```

The function provides a method for applications to determine the current value of a configurable system limit or option variable. The name argument specifies the system variable to be queried. Symbolic constants for each name value are found in the include file `<unistd.h>`.

Table 11-12 lists the `sysconf` values.

**Table 11-12 sysconf values**

| Value | Description |
|---|---|
| _SC_ARG_MAX | The maximum bytes of argument to execve(2). |
| _SC_ATEXIT_MAX | The maximum number of functions that can be registered with atexit(3). |
| _SC_BARRIERS | The version of IEEE Std. 1003.1 (“POSIX.1”) and its Barriers option to which the system attempts to conform, otherwise -1. |
| _SC_CLOCK_SELECTION | Return the POSIX version the implementation of the Clock Selection Option on this system conforms to, or -1 if unavailable. |
| _SC_CHILD_MAX | The maximum number of simultaneous processes per user ID. |
| _SC_CLK_TCK | The number of clock ticks per second. |
| _SC_FSYNC | Return 1 if the File Synchronization Option is available on this system, otherwise -1. |
| _SC_HOST_NAME_MAX | The maximum size of a hostname, including NULL. |
| _SC_IOV_MAX | The maximum number of iovec structures that a process has available for use with preadv(2), pwritev(2), readv(2), recvmsg(2), sendmsg(2) or writev(2). |
| _SC_JOB_CONTROL | Return 1 if job control is available on this system, otherwise -1. |
| _SC_LOGIN_NAME_MAX | Returns the size of the storage required for a login name, in bytes, including the terminating NUL. |
| _SC_MAPPED_FILES | Return 1 if the Memory Mapped Files Option is available on this system, otherwise -1. |
| _SC_MEMLOCK | Return 1 if the Process Memory Locking Option is available on this system, otherwise -1. |
| _SC_MEMLOCK_RANGE | Return 1 if the Range Memory Locking Option is available on this system, otherwise -1. |
| _SC_MEMORY_PROTECTION | Return 1 if the Memory Protection Option is available on this system, otherwise -1. |
| _SC_MONOTONIC_CLOCK | Return the POSIX version the implementation of the Monotonic Clock Option on this system conforms to, or -1 if unavailable. |
| _SC_NGROUPS_MAX | The maximum number of supplemental groups. |
| _SC_OPEN_MAX | The maximum number of open files per process. |
| _SC_PAGESIZE | The size of a system page in bytes. |
| _SC_PASS_MAX | The maximum length of the password, not counting NULL. |
| _SC_READER_WRITER_LOCKS | The version of IEEE Std. 1003.1 (“POSIX.1”) and its Read-Write Locks option to which the system attempts to conform, otherwise -1. |
| _SC_REGEXP | Return 1 if POSIX regular expressions are available on this system, otherwise -1. |
| _SC_SEMAPHORES | The version of IEEE Std. 1003.1 (“POSIX.1”) and its Semaphores option to which the system attempts to conform, otherwise -1. Availability of the Semaphores option depends on the P1003_1B_SEMAPHORE kernel option. |
| _SC_SHELL | Return 1 if POSIX shell is available on this system, otherwise -1. |

**Table 11-12 sysconf values**

| Value | Description |
|---|---|
| _SC_SPIN_LOCKS | The version of IEEE Std. 1003.1 (“POSIX.1”) and its Spin Locks option to which the system attempts to conform, otherwise -1. |
| _SC_STREAM_MAX | The minimum maximum number of streams that a process may have open at any one time. |
| _SC_SYMLOOP_MAX | The maximum number of symbolic links that can be expanded in a path name. |
| _SC_SYNCHRONIZED_IO | Return 1 if the Synchronized I/O Option is available on this system, otherwise -1. |
| _SC_THREADS | The version of IEEE Std. 1003.1 (“POSIX.1”) and its Threads option to which the system attempts to conform, otherwise -1. |
| _SC_TIMERS | The version of IEEE Std. 1003.1 (“POSIX.1”) and its Timers option to which the system attempts to conform, otherwise -1. |
| _SC_TZNAME_MAX | The minimum maximum number of types supported for the name of a timezone. |
| _SC_SAVED_IDS | Returns 1 if saved set-group and saved set-user ID is available, otherwise -1. |
| _SC_VERSION | The version of ISO/IEC 9945 (POSIX 1003.1) with which the system attempts to comply. |
| _SC_XOPEN_SHM | Return 1 if the X/Open Portability Guide Issue 4, Version 2 (“XPG4.2”) Shared Memory option is available on this system, otherwise -1. Availability of the Shared Memory option depends on the SYSVSHM kernel option. |
| _SC_BC_BASE_MAX | The maximum ibase/obase values in the bc(1) utility. |
| _SC_BC_DIM_MAX | The maximum array size in the bc(1) utility. |
| _SC_BC_SCALE_MAX | The maximum scale value in the bc(1) utility. |
| _SC_BC_STRING_MAX | The maximum string length in the bc(1) utility. |
| _SC_COLL_WEIGHTS_MAX | The maximum number of weights that can be assigned to any entry of the LC_COLLATE order keyword in the locale definition file. |
| _SC_EXPR_NEST_MAX | The maximum number of expressions that can be nested within parenthesis by the expr(1) utility. |
| _SC_LINE_MAX | The maximum length in bytes of a text-processing utility's input line. |
| _SC_RE_DUP_MAX | The maximum number of repeated occurrences of a regular expression permitted when using interval notation. |
| _SC_2_VERSION | The version of POSIX 1003.2 with which the system attempts to comply. |
| _SC_2_C_BIND | Return 1 if the system's C-language development facilities sup- port the C-Language Bindings Option, otherwise -1. |
| _SC_2_C_DEV | Return 1 if the system supports the C-Language Development Utilities Option, otherwise -1. |
| _SC_2_CHAR_TERM | Return 1 if the system supports at least one terminal type capable of all operations described in POSIX 1003.2, otherwise -1. |
| _SC_2_FORT_DEV | Return 1 if the system supports the FORTRAN Development Utilities Option, otherwise -1. |

**Table 11-12 sysconf values**

| Value | Description |
|---|---|
| _SC_2_FORT_RUN | Return 1 if the system supports the FORTRAN Runtime Utilities Option, otherwise -1. |
| _SC_2_LOCALEDEF | Return 1 if the system supports the creation of locales, other- wise -1. |
| _SC_2_SW_DEV | Return 1 if the system supports the Software Development Utilities Option, otherwise -1. |
| _SC_2_UPE | Return 1 if the system supports the User Portability Utilities Option, otherwise -1. |
| _SC_GETGR_R_SIZE_MAX | The minimum size of the buffer passed to getgrgid_r(3) and getgrnam_r(3). |
| _SC_GETPW_R_SIZE_MAX | The minimum size of the buffer passed to getpwnam_r(3) and getpwuid_r(3). |
| _SC_NPROCESSORS_CONF | The number of processors configured. |
| _SC_NPROCESSORS_ONLN | The number of processors online (capable of running processes). |

If the call to sysconf is not successful, -1 is returned and errno is set appropriately. Otherwise, if the variable is associated with functionality that is not supported, -1 is returned and errno is not modified. Otherwise, the current variable value is returned.

## 11.31.9 unlink

```
int unlink(const char *path); [POSIX]
```

The function removes a link to a file. If path names a symbolic link, unlink removes the symbolic link and does not affect any file or directory named by the contents of the symbolic link. Otherwise, unlink removes the link named by path and decrements the link count of the file referenced by the link.

When the file's link count becomes 0 and no process has the file open, the space occupied by the file will be freed and the file is no longer accessible. If one or more processes have the file open when the last link is removed, the link is removed before unlink returns, but the removal of the file contents is postponed until all references to the file are closed.

The path argument must not name a directory unless the process has appropriate privileges and the implementation supports using unlink on directories.

Upon successful completion, unlink marks for update the st_ctime and st_mtime fields of the parent directory. If the file's link count is not 0, the st_ctime field of the file is marked for update. Upon successful completion, 0 is returned. Otherwise, -1 is returned, errno is set to indicate the error, and the file is not unlinked.

# 11.32 <wchar.h>

[Added with Amendment 1]

Include the standard header <wchar.h> so that you can perform input and output operations on wide streams or manipulate wide strings.

```
/* MACROS */
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
#define WCHAR_MAX <#if expression >= 127>
#define WCHAR_MIN <#if expression <= 0>
#define WEOF <wint_t constant expression>

/* TYPES */
typedef o-type mbstate_t;
typedef ui-type size_t;
typedef i-type wchar_t; [Keyword in C++]
typedef i_type wint_t; struct tm;

/* FUNCTIONS */
wint_t fgetwc(FILE *stream);
wchar_t *fgetws(wchar_t *restrict s, int n, FILE *restrict stream);
wint_t fputwc(wchar_t c, FILE *stream);
int fputws(const wchar_t *restrict s, FILE *restrict stream);
int fwide(FILE *stream, int mode);
wint_t getwc(FILE *stream);
wint_t getwchar(void);
wint_t putwc(wchar_t c, FILE *stream);
wint_t putwchar(wchar_t c);
wint_t ungetwc(wint_t c, FILE *stream);

int fwscanf(FILE *restrict stream, const wchar_t *restrict format,
...);
int swscanf(const wchar_t *restrict s, const wchar_t *restrict format,
...);
int wscanf(const wchar_t *restrict format, ...);
int fwprintf(FILE *restrict stream, const wchar_t *restrict format,
...);
int swprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict
format, ...);
int wprintf(const wchar_t *restrict format, ...);
int vfwscanf(FILE *restrict stream, const wchar_t *restrict format,
va_list arg); [Added with C99]
int vswscanf(const wchar_t *restrict s, const wchar_t *restrict
format, va_list arg); [Added with C99]
int vwscanf(const wchar_t *restrict format, va_list arg); [Added with
C99]
int vfwprintf(FILE *restrict stream, const wchar_t *restrict format,
va_list arg);
int vswprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict
format, va_list arg);
int vwprintf(const wchar_t *restrict format, va_list arg);

size_t wcsftime(wchar_t *restrict s, size_t maxsize, const wchar_t
*restrict format, const struct tm *restrict timeptr);
```

```
wint_t btowc(int c);
size_t mbrlen(const char *restrict s, size_t n, mbstate_t *restrict
ps);
size_t mbrtowc(wchar_t *restrict pwc, const char *restrict s, size_t
n, mbstate_t *restrict ps);
int mbsinit(const mbstate_t *ps);
size_t mbsrtowcs(wchar_t *restrict dst, const char **restrict src,
size_t len, mbstate_t *restrict ps);
size_t wcrtomb(char *restrict s, wchar_t wc, mbstate_t *restrict ps);
size_t wcsrtombs(char *restrict dst, const wchar_t **restrict src,
size_t len, mbstate_t *restrict ps);
double wcstod(const wchar_t *restrict nptr, wchar_t **restrict
endptr);
float wcstof(const wchar_t *restrict nptr, wchar_t **restrict endptr);
[Added with C99]
long double wcstold(const wchar_t *restrict nptr, wchar_t **restrict
endptr); [Added with C99]
long long wcstoll(const wchar_t *restrict nptr, wchar_t **restrict
endptr, int base); [Added with C99]
unsigned long long wcstoull(const wchar_t *restrict nptr, wchar_t
**restrict endptr, int base); [Added with C99]
long wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr,
int base);
unsigned long wcstoul(const wchar_t *restrict nptr, wchar_t **restrict
endptr, int base);
int wctob(wint_t c);

wchar_t *wcscat(wchar_t *restrict s1, const wchar_t *restrict s2);
int wcscmp(const wchar_t *s1, const wchar_t *s2);
int wcscoll(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcscpy(wchar_t *restrict s1, const wchar_t *restrict s2);
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
size_t wcslen(const wchar_t *s);
wchar_t *wcsncat(wchar_t *restrict s1, const wchar_t *restrict s2,
size_t n);
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wcsncpy(wchar_t *restrict s1, const wchar_t *restrict s2,
size_t n);
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcstok(wchar_t *restrict s1, const wchar_t *restrict s2,
wchar_t **restrict ptr);
size_t wcsxfrm(wchar_t *restrict s1, const wchar_t *restrict s2,
size_t n);
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemcpy(wchar_t *restrict s1, const wchar_t *restrict s2,
size_t n);
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);

wchar_t *wcschr(const wchar_t *s, wchar_t c); [Not in C++]
const wchar_t *wcschr(const wchar_t *s, wchar_t c); [C++ only]
wchar_t *wcschr(wchar_t *s, wchar_t c); [C++ only]

wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2); [Not in C++]
const wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2); [C++ only]
```

```
wchar_t *wcspbrk(wchar_t *s1, const wchar_t *s2); [C++ only]

wchar_t *wcsrchr(const wchar_t *s, wchar_t c); [Not in C++]
const wchar_t *wcsrchr(const wchar_t *s, wchar_t c); [C++ only]
wchar_t *wcsrchr(wchar_t *s, wchar_t c); [C++ only]

wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2); [Not in C++]
const wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2); [C++ only]
wchar_t *wcsstr(wchar_t *s1, const wchar_t *s2); [C++ only]

wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n); [Not in C++]
const wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n); [C++
only]
wchar_t *wmemchr(wchar_t *s, wchar_t c, size_t n); [C++ only]
```

### 11.32.1 btowc

```
wint_t btowc(int c);
```

The function returns `WEOF` if `c` equals `EOF`. Otherwise, it converts `(unsigned char)c` as a one-byte multibyte character beginning in the initial conversion state, as if by calling `mbrtowc`. If the conversion succeeds, the function returns the wide-character conversion. Otherwise, it returns `WEOF`.

### 11.32.2 fgetwc

```
wint_t fgetwc(FILE *stream);
```

The function reads the next wide character `c` (if present) from the input stream `stream`, advances the file-position indicator (if defined), and returns `(wint_t)c`. If the function sets either the end-of-file indicator or the error indicator, it returns `WEOF`.

### 11.32.3 fgetws

```
wchar_t *fgetws(wchar_t *restrict s, int n, FILE *restrict stream);
```

The function reads wide characters from the input stream `stream` and stores them in successive elements of the array beginning at `s` and continuing until it stores `n-1` wide characters, stores an NL wide character, or sets the end-of-file or error indicators.

If `fgetws` stores any wide characters, it concludes by storing a NULL wide character in the next element of the array. It returns `s` if it stores any wide characters and has not set the error indicator for the stream; otherwise, it returns a NULL pointer. If it sets the error indicator, the array contents are indeterminate.

## 11.32.4 fputwc

```
wint_t fputwc(wchar_t c, FILE *stream);
```

The function writes the wide character `c` to the output stream `stream`, advances the file-position indicator (if defined), and returns `(wint_t)c`. If the function sets the error indicator for the stream, it returns `WEOF`.

## 11.32.5 fputws

```
int fputws(const wchar_t *restrict s, FILE *restrict stream);
```

The function accesses wide characters from string `s` and writes them to the output stream `stream`. The function does not write the terminating NULL wide character. It returns a non-negative value if it has not set the error indicator; otherwise, it returns `WEOF`.

## 11.32.6 fwide

```
int fwide(FILE *stream, int mode);
```

The function determines the orientation of the stream `stream`. If `mode` is greater than zero, it first attempts to make the stream wide oriented. If `mode` is less than zero, it first attempts to make the stream byte oriented. In any event, the function returns:

- A value greater than zero if the stream is left-wide oriented
- Zero if the stream is left unbound
- A value less than zero if the stream is left-byte oriented

In no event will the function alter the orientation of a stream once it has been oriented.

## 11.32.7 fwprintf

```
int fwprintf(FILE *restrict stream, const wchar_t *restrict format,
...);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated wide character to the stream `stream`. It returns the number of wide characters generated, or it returns a negative value if the function sets the error indicator for the stream.

## 11.32.8 fwscanf

```
int fwscanf(FILE *restrict stream, const wchar_t *restrict format,
...);
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned character from the stream `stream`. It returns the number of input items matched and assigned, or it returns `EOF` if the function does not store values before it sets the end-of-file or error indicator for the stream.

## 11.32.9 getwc

```
wint_t getwc(FILE *stream);
```

The function has the same effect as `fgetwc(stream)` except that a macro version of `getwc` can evaluate stream more than once.

## 11.32.10 getwchar

```
wint_t getwchar(void);
```

The function has the same effect as `fgetwc(stdin)`.

## 11.32.11 mbrlen

```
size_t mbrlen(const char *restrict s, size_t n, mbstate_t *restrict
ps);
```

The function is equivalent to the call:

```
mbrtowc(0, s, n, ps != 0 ? ps : &internal)
```

Where `internal` is an object of type `mbstate_t` internal to the `mbrlen` function. At program startup, `internal` is initialized to the initial conversion state. No other library function alters the value stored in `internal`.

The function returns:

- `(size_t)-3` – If no additional bytes are needed to complete the next multibyte character, in which case the resulting conversion state indicates that no additional bytes have been converted and the next multibyte character has been produced
- `(size_t)-2` – If, after converting all `n` characters, the resulting conversion state indicates an incomplete multibyte character
- `(size_t)-1` – If the function detects an encoding error before completing the next multibyte character, in which case the function stores the value `EILSEQ` in `errno` and leaves the resulting conversion state undefined
- Zero – If the next completed character is a NULL character, in which case the resulting conversion state is the initial conversion state
- `x` – The number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that `x` bytes have been converted and the next multibyte character has been produced

Thus, `mbrlen` effectively returns the number of bytes that would be consumed in successfully converting a multibyte character to a wide character (without storing the converted wide character), or an error code if the conversion cannot succeed.

## 11.32.12mbrtowc

```
size_t mbrtowc(wchar_t *restrict pwc, const char *restrict s, size_t
n, mbstate_t *restrict ps);
```

The function determines the number of bytes in a multibyte string that completes the next multibyte character, if possible.

If `ps` is not a NULL pointer, the conversion state for the multibyte string is assumed to be `*ps`. Otherwise, it is assumed to be `&internal`, where internal is an object of type `mbstate_t` internal to the `mbrtowc` function. At program startup, `internal` is initialized to the initial conversion state. No other library function alters the value stored in `internal`.

If `s` is not a NULL pointer, the function determines `x`, the number of bytes in the multibyte string `s` that complete or contribute to the next multibyte character. (`x` cannot be greater than `n`.) Otherwise, the function effectively returns `mbrtowc(0, "", 1, ps)`, ignoring `pwc` and `n`. (The function thus returns zero only if the conversion state indicates that no incomplete multibyte character is pending from a previous call to `mbrlen`, `mbrtowc`, or `mbsrtowcs` for the same string and conversion state.)

If `pwc` is not a NULL pointer, the function converts a completed multibyte character to its corresponding wide-character value and stores that value in `*pwc`.

The function returns:

- `(size_t)-3` – If no additional bytes are needed to complete the next multibyte character, in which case the resulting conversion state indicates that no additional bytes have been converted and the next multibyte character has been produced
- `(size_t)-2` – If, after converting all `n` characters, the resulting conversion state indicates an incomplete multibyte character
- `(size_t)-1` – If the function detects an encoding error before completing the next multibyte character, in which case the function stores the value `EILSEQ` in `errno` and leaves the resulting conversion state undefined
- Zero – If the next completed character is a NULL character, in which case the resulting conversion state is the initial conversion state
- `x` – The number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that `x` bytes have been converted and the next multibyte character has been produced

## 11.32.13mbsinit

```
int mbsinit(const mbstate_t *ps);
```

The function returns a nonzero value if `ps` is a NULL pointer or if `*ps` designates an initial conversion state. Otherwise, it returns zero.

## 11.32.14mbsrtowcs

```
size_t mbsrtowcs(wchar_t *restrict dst, const char **restrict src,
size_t len, mbstate_t *restrict ps);
```

The function converts the multibyte string beginning at `*src` to a sequence of wide characters as if by repeated calls of the form:

```
x = mbrtowc(dst, *src, n, ps != 0 ? ps : &internal)
```

Where `n` is a value `> 0`, and internal is an object of type `mbstate_t` internal to the `mbsrtowcs` function. At program startup, `internal` is initialized to the initial conversion state. No other library function alters the value stored in `internal`.

If `dst` is not a NULL pointer, the `mbsrtowcs` function stores at most `len` wide characters by calls to `mbrtowc`. The function effectively increments `dst` by one and `*src` by `x` after each call to `mbrtowc` that stores a converted wide character. After a call that returns zero, `mbsrtowcs` stores a NULL wide character at `dst` and stores a NULL pointer at `*src`.

If `dst` is a NULL pointer, `len` is effectively assigned a large value.

The function returns:

- `(size_t)-1` if a call to `mbrtowc` returns `(size_t)-1`, indicating that it has detected an encoding error before completing the next multibyte character
- The number of multibyte characters successfully converted, not including the terminating NULL character

## 11.32.15mbstate_t

```
typedef o-type mbstate_t;
```

The type is an object type `o-type` that can represent a conversion state for any of the functions `mbrlen`, `mbrtowc`, `mbsrtowcs`, `wcrtomb`, or `wcsrtombs`. A definition of the following form ensures that `mbst` represents the initial conversion state:

```
mbstate_t mbst = {0};
```

However, other values stored in an object of type `mbstate_t` can also represent this state. To test safely for this state, use the function `mbsinit`.

## 11.32.16NULL

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
```

The macro yields a NULL pointer constant that is usable as an address constant expression.

## 11.32.17 putwc

```
wint_t putwc(wchar_t c, FILE *stream);
```

The function has the same effect as `fputwc(c, stream)` except that a macro version of `putwc` can evaluate stream more than once.

## 11.32.18 putwchar

```
wint_t putwchar(wchar_t c);
```

The function has the same effect as `fputwc(c, stdout)`.

## 11.32.19 size_t

```
typedef ui-type size_t;
```

The type is the unsigned integer type `ui-type` of an object that you declare to store the result of the `sizeof` operator.

## 11.32.20 swprintf

```
int swprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict
format, ...);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and stores each generated character in successive locations of the array object whose first element has the address `s`. The function concludes by storing a NULL wide character in the next location of the array. If it cannot generate and store all characters in an array of size `n`, the function returns a negative number. Otherwise, it returns the number of wide characters generated—not including the NULL wide character.

## 11.32.21 swscanf

```
int swscanf(const wchar_t *restrict s, const wchar_t *restrict format,
...);
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It accesses each scanned character from successive locations of the array object whose first element has the address `s`. It returns the number of items matched and assigned, or it returns `EOF` if the function does not store values before it accesses a NULL wide character from the array.

## 11.32.22tm

```
struct tm;
```

The structure contains members that describe various properties of the calendar time. The declaration in this header leaves `struct tm` an incomplete type. Include the header `<time.h>` to complete the type.

## 11.32.23ungetwc

```
wint_t ungetwc(wint_t c, FILE *stream);
```

If `c` is not equal to `WEOF`, the function stores `(wchar_t)c` in the object whose address is `stream` and clears the end-of-file indicator. If `c` equals `WEOF` or the store cannot occur, the function returns `WEOF`; otherwise, it returns `(wchar_t)c`. A subsequent library function call that reads a wide character from the stream `stream` obtains this stored value, which is then forgotten.

Thus, you can effectively push back a wide character to a stream after reading a wide character.

## 11.32.24vfwprintf

```
int vfwprintf(FILE *restrict stream, const wchar_t *restrict format,
va_list arg);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated wide character to the stream `stream`. It returns the number of wide characters generated, or it returns a negative value if the function sets the error indicator for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

## 11.32.25vfwscanf

```
int vfwscanf(FILE *restrict stream, const wchar_t *restrict format,
va_list ap); [Added with C99]
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned wide character from the stream `stream`. It returns the number of input items matched and assigned, or it returns `WEOF` if the function does not store values before it sets the end-of-file or error indicator for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

## 11.32.26vswprintf

```
int vswprintf(wchar_t *restrict s, size_t n, const wchar_t *restrict
format, va_list arg);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and stores each generated wide character in successive locations of the array object whose first element has the address `s`. The function concludes by storing a NULL wide character in the next location of the array.

If it cannot generate and store all characters in an array of size `n`, the function returns a negative number. Otherwise, it returns the number of wide characters generated—not including the NULL wide character.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

## 11.32.27vswscanf

```
int vswscanf(const wchar_t *restrict s, const wchar_t *restrict
format, va_list ap); [Added with C99]
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It accesses each scanned wide character from successive locations of the array object whose first element has the address `s`. It returns the number of items matched and assigned, or it returns `WEOF` if the function does not store values before it accesses a NULL character from the array.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

## 11.32.28vwprintf

```
int vwprintf(const wchar_t *restrict format, va_list arg);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated wide character to the stream `stdout`. It returns the number of characters generated, or a negative value if the function sets the error indicator for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

## 11.32.29vwscanf

`int vwscanf(const wchar_t *restrict format, va_list ap);` [Added with`C99`]

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned wide character from the stream `stdin`. It returns the number of input items matched and assigned, or it returns `WEOF` if the function does not store values before it sets the end-of-file or error indicators for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

## 11.32.30WCHAR_MAX

`#define WCHAR_MAX <#if expression >= 127>`

The macro yields the maximum value for type `wchar_t`.

## 11.32.31WCHAR_MIN

`#define WCHAR_MIN <#if expression <= 0>`

The macro yields the minimum value for type `wchar_t`.

## 11.32.32wchar_t

`typedef i-type wchar_t;` [Keyword in C++]

The type is the integer type `i-type` of a wide-character constant, such as `L'X'`. Declare an object of type `wchar_tto` hold a wide character.

## 11.32.33wcrtomb

`size_t wcrtomb(char *restrict s, wchar_t wc, mbstate_t *restrict ps);`

The function determines the number of bytes needed to represent the wide character `wc` as a multibyte character, if possible. (Not all values representable as type `wchar_t` are necessarily valid wide-character codes.)

If `ps` is not a NULL pointer, the conversion state for the multibyte string is assumed to be `*ps`. Otherwise, it is assumed to be `&internal`, where `internal` is an object of type `mbstate_t` internal to the `wcrtomb` function. At program startup, `internal` is initialized to the initial conversion state. No other library function alters the value stored in `internal`.

If s is not a NULL pointer and wc is a valid wide-character code, the function determines x, the number of bytes needed to represent wc as a multibyte character, and stores the converted bytes in the array of char beginning at s. (x cannot be greater than MB_CUR_MAX.) If wc is a NULL wide character, the function stores any shift sequence needed to restore the initial shift state, followed by a NULL byte. The resulting conversion state is the initial conversion state.

If s is a NULL pointer, the function effectively returns wcrtomb(buf, L'\0', ps), where buf is a buffer internal to the function. (The function thus returns the number of bytes needed to restore the initial conversion state and to terminate the multibyte string pending from a previous call to wcrtomb or wcsrtombs for the same string and conversion state.)

The function returns:

- (size_t)-1 – If wc is an invalid wide-character code, in which case the function stores the value EILSEQ in errno and leaves the resulting conversion state undefined
- x – The number of bytes needed to complete the next multibyte character, in which case the resulting conversion state indicates that x bytes have been generated

## 11.32.34wcscat

```
wchar_t *wcscat(wchar_t *restrict s1, const wchar_t *restrict s2);
```

The function copies the wide string s2, including its terminating NULL wide character, to successive elements of the array that stores the wide string s1, beginning with the element that stores the terminating NULL wide character of s1. It returns s1.

## 11.32.35wcschr

```
wchar_t *wcschr(const wchar_t *s, wchar_t c); [Not in C++]
const wchar_t *wcschr(const wchar_t *s, wchar_t c); [C++ only]
wchar_t *wcschr(wchar_t *s, wchar_t c); [C++ only]
```

The function searches for the first element of the wide strings that equals c. It considers the terminating NULL wide character as part of the wide string. If successful, the function returns the address of the matching element; otherwise, it returns a NULL pointer.

## 11.32.36wcscmp

```
int wcscmp(const wchar_t *s1, const wchar_t *s2);
```

The function compares successive elements from two wide strings, s1 and s2, until it finds elements that are not equal.

- If all elements are equal, the function returns zero.
- If the differing element from s1 is greater than the element from s2, the function returns a positive number.
- Otherwise, the function returns a negative number.

## 11.32.37wcscoll

```
int wcscoll(const wchar_t *s1, const wchar_t *s2);
```

The function compares two wide strings, `s1` and `s2`, using a comparison rule that depends on the current locale. If `s1` compares greater than `s2` by this rule, the function returns a positive number. If the two wide strings compare equal, it returns zero. Otherwise, it returns a negative number.

## 11.32.38wcscpy

```
wchar_t *wcscpy(wchar_t *restrict s1, const wchar_t *restrict s2);
```

The function copies the wide string `s2`, including its terminating NULL wide character, to successive elements of the array whose first element has the address `s1`. It returns `s1`.

## 11.32.39wcscspn

```
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

The function searches for the first element s1[i] in the wide string `s1` that equals any one of the elements of the wide string `s2` and returns i. Each terminating NULL wide character is considered part of its wide string.

## 11.32.40wcsftime

```
size_t wcsftime(wchar_t *restrict s, size_t maxsize, const wchar_t
*restrict format, const struct tm *restrict timeptr);
```

The function generates formatted text, under the control of the format `format` and the values stored in the time structure `*tptr`. It stores each generated wide character in successive locations of the array object of size `n` whose first element has the address `s`. The function then stores a NULL wide character in the next location of the array. It returns `x`, the number of wide characters generated, if `x < n`; otherwise, it returns zero, and the values stored in the array are indeterminate.

For each wide character other than `%` in the format, the function stores that wide character in the array object. Each occurrence of `%` followed by another character in the format is a conversion specifier. For each conversion specifier, the function stores a replacement wide character sequence. Conversion specifiers are the same as for the function `strftime`. The current locale category `LC_TIME` can affect these replacement character sequences.

## 11.32.41wcslen

```
size_t wcslen(const wchar_t *s);
```

The function returns the number of wide characters in the wide strings, not including its terminating NULL wide character.

## 11.32.42wcsncat

```
wchar_t *wcsncat(wchar_t *restrict s1, const wchar_t *restrict s2,
size_t n);
```

The function copies the wide string `s2`, not including its terminating NULL wide character, to successive elements of the array that stores the wide string `s1`, beginning with the element that stores the terminating NULL wide character of `s1`. The function copies no more than n wide characters from `s2`. It then stores a NULL wide character, in the next element to be altered in `s1`, and returns `s1`.

## 11.32.43wcsncmp

```
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

The function compares successive elements from two wide strings, `s1` and `s2`, until it finds elements that are not equal or until it has compared the first n elements of the two wide strings.

- If all elements are equal, the function returns zero.
- If the differing element from `s1` is greater than the element from `s2`, the function returns a positive number.
- Otherwise, it returns a negative number.

## 11.32.44wcsncpy

```
wchar_t *wcsncpy(wchar_t *restrict s1, const wchar_t *restrict s2,
size_t n);
```

The function copies the wide string `s2`, not including its terminating NULL wide character, to successive elements of the array whose first element has the address `s1`. It copies no more than n wide characters from `s2`. The function then stores zero or more NULL wide characters in the next elements to be altered `ins1` until it stores a total of n wide characters. It returns `s1`.

## 11.32.45wcspbrk

```
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2); [Not in C++]
const wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2); [C++ only]
wchar_t *wcspbrk(wchar_t *s1, const wchar_t *s2); [C++ only]
```

The function searches for the first element `s1[i]` in the wide string `s1` that equals any one of the elements of the wide string `s2`. It considers each terminating NULL wide character as part of its wide string. If `s1[i]` is not the terminating NULL wide character, the function returns `&s1[i]`; otherwise, it returns a NULL pointer.

## 11.32.46wcsrchr

```
wchar_t *wcsrchr(const wchar_t *s, wchar_t c); [Not in C++]
const wchar_t *wcsrchr(const wchar_t *s, wchar_t c); [C++ only]
wchar_t *wcsrchr(wchar_t *s, wchar_t c); [C++ only]
```

The function searches for the last element of the wide string `s` that equals `c`. It considers the terminating NULL wide character as part of the wide string. If successful, the function returns the address of the matching element; otherwise, it returns a NULL pointer.

## 11.32.47wcsrtombs

```
size_t wcsrtombs(char *restrict dst, const wchar_t **restrict src,
size_t len, mbstate_t *restrict ps);
```

The function converts the wide-character string beginning at `*src` to a sequence of multibyte characters as if by repeated calls of the form:

```
x = wcrtomb(dst ? dst : buf, *src, ps != 0 ? ps : &internal)
```

Where `buf` is an array of type char and internal is an object of type `mbstate_t`, both internal to the `wcsrtombs` function. At program startup, `internal` is initialized to the initial conversion state. No other library function alters the value stored in `internal`.

If `dst` is not a NULL pointer, the `wcsrtombs` function stores at most `len` bytes by calls to `wcrtomb`. The function effectively increments `dst` by `x` and `*src` by one after each call to `wcrtomb` that stores a complete converted multibyte character in the remaining space available. After a call that stores a complete NULL multibyte character at `dst` (including any shift sequence needed to restore the initial shift state), the function stores a NULL pointer at `*src`.

If `dst` is a NULL pointer, `len` is effectively assigned a large value.

The function returns:

- `(size_t)-1` if a call to `wcrtomb` returns `(size_t)-1`, indicating that it has detected an invalid wide-character code
- The number of bytes successfully converted, not including the terminating NULL byte

## 11.32.48wcsspn

```
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
```

The function searches for the first element `s1[i]` in the wide string `s1` that equals none of the elements of the wide string `s2` and returns `i`. It considers the terminating NULL wide character as part of the wide string `s1` only.

## 11.32.49wcsstr

```
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2); [Not in C++]
const wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2); [C++ only]
wchar_t *wcsstr(wchar_t *s1, const wchar_t *s2); [C++ only]
```

The function searches for the first sequence of elements in the wide string `s1` that matches the sequence of elements in the wide string `s2`, not including its terminating NULL wide character. If successful, the function returns the address of the matching first element; otherwise, it returns a NULL pointer.

## 11.32.50wcstod

```
double wcstod(const wchar_t *restrict nptr, wchar_t **restrict
endptr);
```

The function converts the initial wide characters of the wide strings to an equivalent value `x` of type `double`. If `endptr` is not a NULL pointer, the function stores a pointer to the unconverted remainder of the wide string in `*endptr`. The function then returns `x`.

The initial wide characters of the wide string `s` must match the same pattern as recognized by the function `strtod`, where each wide character `wc` is converted as if by calling `wctob(wc))`.

If the wide string `s` matches this pattern, its equivalent value is the value returned by `strtod` for the converted sequence. If the wide string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If a range error occurs, `wcstod` behaves exactly as the functions declared in `<math.h>`.

## 11.32.51wcstof

```
float wcstof(const wchar_t *restrict nptr,wchar_t **restrict endptr);
```

The function converts the initial wide characters of the wide strings to an equivalent value `x` of type `float`. If `endptr` is not a NULL pointer, the function stores a pointer to the unconverted remainder of the wide string in `*endptr`. The function then returns `x`.

The initial wide characters of the wide string `s` must match the same pattern as recognized by the function `strtod`, where each wide character `wc` is converted as if by calling `wctob(wc))`.

If the wide string `s` matches this pattern, its equivalent value is the value returned by `strtof` for the converted sequence. If the wide string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If a range error occurs, `wcstod` behaves exactly as the functions declared in `<math.h>`.

## 11.32.52wcstok

```
wchar_t *wcstok(wchar_t *restrict s1, const wchar_t *restrict s2,
wchar_t **restrict ptr);
```

If `s1` is not a NULL pointer, the function begins a search of the wide string `s1`. Otherwise, it begins a search of the wide string whose address was last stored in `*ptr` on an earlier call to the function, as described below. The search proceeds as follows:

- The function searches the wide string for begin, the address of the first element that equals none of the elements of the wide string `s2` (a set of token separators). It considers the terminating NULL character as part of the search wide string only.
- If the search does not find an element, the function stores the address of the terminating NULL wide character in `*ptr` (so that a subsequent search beginning with that address will fail) and returns a NULL pointer. Otherwise, the function searches from begin for end, the address of the first element that equals any one of the elements of the wide string `s2`. It again considers the terminating NULL wide character as part of the search string only.
- If the search does not find an element, the function stores the address of the terminating NULL wide character in `*ptr`. Otherwise, it stores a NULL wide character in the element whose address is `end`. Then it stores the address of the next element after end in `*ptr` (so that a subsequent search beginning with that address will continue with the remaining elements of the string) and returns `begin`.

## 11.32.53wcstol

```
long wcstol(const wchar_t *restrict nptr, wchar_t **restrict endptr,
int base);
```

The function converts the initial wide characters of the wide strings to an equivalent value `x` of type `long`. If `endptr` is not a NULL pointer, the function stores a pointer to the unconverted remainder of the wide string in `*endptr`. The function then returns `x`.

The initial wide characters of the wide string `s` must match the same pattern as recognized by the function `strtol`, with the same base argument, where each wide character `wc` is converted as if by calling `wctob(wc))`.

If the wide string `s` matches this pattern, its equivalent value is the value returned by `strtol`, with the same base argument, for the converted sequence. If the wide string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If the equivalent value is too large in magnitude to represent as type `long`, `wcstol` stores the value of `ERANGE` in `errno` and returns either `LONG_MAX` if `x` is positive or `LONG_MIN` if `x` is negative.

## 11.32.54wcstold

```
long double wcstof(const wchar_t *restrict nptr,wchar_t **restrict
endptr);
```

The function converts the initial wide characters of the wide strings to an equivalent value `x` of type `long double`. If `endptr` is not a NULL pointer, the function stores a pointer to the unconverted remainder of the wide string in `*endptr`. The function then returns `x`.

The initial wide characters of the wide string `s` must match the same pattern as recognized by the function `strtod`, where each wide character `wc` is converted as if by calling `wctob(wc))`.

If the wide string `s` matches this pattern, its equivalent value is the value returned by `strtold` for the converted sequence. If the wide string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If a range error occurs, `wcstod` behaves exactly as the functions declared in `<math.h>`.

## 11.32.55wcstoll

```
long long wcstoll(const wchar_t *restrict nptr, wchar_t **restrict
endptr, int base);
```

The function converts the initial wide characters of the wide strings to an equivalent value `x` of type `long long`. If `endptr` is not a NULL pointer, the function stores a pointer to the unconverted remainder of the wide string in `*endptr`. The function then returns `x`.

The initial wide characters of the wide string `s` must match the same pattern as recognized by the function `strtol`, with the same base argument, where each wide character `wc` is converted as if by calling `wctob(wc))`.

If the wide string `s` matches this pattern, its equivalent value is the value returned by `strtoll`, with the same base argument, for the converted sequence. If the wide strings does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If the equivalent value is too large in magnitude to represent as type `long long`, `wcstoll` stores the value of `ERANGE` in `errno` and returns either `LLONG_MAX` if `x` is positive or `LLONG_MIN` if `x` is negative.

## 11.32.56wcstoul

```
unsigned long wcstoul(const wchar_t *restrict nptr,wchar_t **restrict
endptr, int base);
```

The function converts the initial wide characters of the wide strings to an equivalent value `x` of type unsigned `long`. If `endptr` is not a NULL pointer, it stores a pointer to the unconverted remainder of the wide string in `*endptr`. The function then returns `x`.

wcstoul converts strings exactly as does `wcstol`, but checks only if the equivalent value is too large to represent as type unsigned `long`. In this case, `wcstoul` stores the value of `ERANGE` in `errno` and returns `ULONG_MAX`.

## 11.32.57wcstoull

```
unsigned long long wcstoull(const wchar_t *restrict nptr,wchar_t
**restrict endptr, int base);
```

The function converts the initial wide characters of the wide strings to an equivalent value `x` of type `unsigned long long`. If `endptr` is not a NULL pointer, it stores a pointer to the unconverted remainder of the wide string in `*endptr`. The function then returns `x`.

`wcstoull` converts strings exactly as does `wcstoll`, but checks only if the equivalent value is too large to represent as type unsigned `long long`. In this case, `wcstoull` stores the value of `ERANGE` in `errno` and returns `ULLONG_MAX`.

## 11.32.58wcsxfrm

```
size_t wcsxfrm(wchar_t *restrict s1, const wchar_t *restrict s2,
size_t n);
```

The function stores a wide string in the array whose first element has the address `s1`. It stores no more than `n` wide characters, including the terminating NULL wide character, and returns the number of wide characters needed to represent the entire wide string, not including the terminating NULL wide character. If the value returned is `n` or greater, the values stored in the array are indeterminate. (If `n` is zero, `s1` can be a NULL pointer.)

`wcsxfrm` generates the wide string it stores from the wide string `s2` by using a transformation rule that depends on the current locale. For example, if `x` is a transformation of `s1` and `y` is a transformation of `s2`, `wcscmp(x, y)` returns the same value as `wcscoll(s1, s2)`.

## 11.32.59wctob

```
int wctob(wint_t c);
```

The function determines whether `c` can be represented as a one-byte multibyte character `x`, beginning in the initial shift state. (It effectively calls `wcrtomb` to make the conversion.) If so, the function returns `x`. Otherwise, it returns `EOF`.

## 11.32.60WEOF

```
#define WEOF <wint_t constant expression>
```

The macro yields the return value, of type `wint_t`, used to signal the end of a wide stream or to report an error condition.

## 11.32.61wint_t

```
typedef i_type wint_t;
```

The type is the integer type `i_type` that can represent all values of type wchar_t as well as the value of the macro `WEOF`, and that does not change when promoted.

## 11.32.62wmemchr

```
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n); [Not in C++]
const wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n); [C++
only]
wchar_t *wmemchr(wchar_t *s, wchar_t c, size_t n); [C++ only]
```

The function searches for the first element of an array beginning at the address s with size `n`, that equals `c`. If successful, it returns the address of the matching element; otherwise, it returns a NULL pointer.

## 11.32.63wmemcmp

```
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

The function compares successive elements from two arrays beginning at the addresses `s1` and `2` (both of size `n`), until it finds elements that are not equal:

- If all elements are equal, the function returns zero.
- If the differing element from `s1` is greater than the element from `s2`, the function returns a positive number.
- Otherwise, the function returns a negative number.

## 11.32.64wmemcpy

```
wchar_t *wmemcpy(wchar_t *restrict s1, const wchar_t *restrict s2,
size_t n);
```

The function copies the array beginning at the address `s2` to the array beginning at the address `s1` (both of size `n`). It returns `s1`. The elements of the arrays can be accessed and stored in any order.

## 11.32.65wmemmove

```
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

The function copies the array beginning at `s2` to the array beginning at `s1` (both of size `n`). It returns `s1`.

If the arrays overlap, the function accesses each of the element values from `s2` before it stores a new value in that element, so the copy is not corrupted.

## 11.32.66wmemset

```
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

The function stores `c` in each of the elements of the array beginning at `s`, with size `n`. It returns `s`.

## 11.32.67wprintf

```
int wprintf(const wchar_t *restrict format, ...);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated wide character to the stream `stdout`. It returns the number of wide characters generated, or it returns a negative value if the function sets the error indicator for the stream.

## 11.32.68wscanf

```
int wscanf(const wchar_t *restrict format, ...);
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned wide character from the stream `stdin`. It returns the number of input items matched and assigned, or it returns `EOF` if the function does not store values before it sets the end-of-file or error indicators for the stream.

# 11.33 <wctype.h>

[Added with Amendment 1]

Include the standard header `<wctype.h>` to declare several functions that are useful for classifying and mapping codes from the target wide-character set.

Every function that has a parameter of type wint_t can accept the value of the macro `WEOF` or any valid wide-character code (of type `wchar_t`). Thus, the argument can be the value returned by any of the functions: `btowc`, `fgetwc`, `fputwc`, `getwc`, `getwchar`, `putwc`, `putwchar`, `towctrans`, `towlower`, `towupper`, or `ungetwc`. You must not call these functions with other wide-character argument values.

The wide-character classification functions are strongly related to the (byte) character classification functions. Each function `isXXX` has a corresponding wide-character classification function `iswXXX`. Moreover, the wide-character classification functions are interrelated much the same way as their corresponding byte functions, with two added provisos:

- The function `iswprint`, unlike `isprint`, can return a nonzero value for additional space characters besides the wide-character equivalent of space (`L' '`). Any such additional characters return a nonzero value for `iswspace` and return zero for `iswgraph` or `iswpunct`.
- The characters in each wide-character class are a superset of the characters in the corresponding byte class. If the call `isXXX(c)` returns a nonzero value, the corresponding call `iswXXX(btowc(c))` also returns a nonzero value.

An implementation can define additional characters that return nonzero for some of these functions. Any character set can contain additional characters that return nonzero for:

- `iswcntrl` (provided the characters cause `iswprint` to return zero)
- `iswpunct` (provided the characters cause `iswalnum` to return zero)

Moreover, a locale other than the "C" locale can define additional characters for:

- `iswalpha`, `iswupper`, and `iswlower` (provided the characters cause `iswcntrl`, `iswdigit`, `iswpunct`, and `iswspace` to return zero)
- `iswblank` (provided the characters cause `iswalnum` to return zero)
- `iswspace` (provided the characters cause `iswpunct` to return zero)

The last rule differs slightly from the corresponding rule for the function `isspace`, as indicated above. Note also that an implementation can define a locale other than the "C" locale in which a character can cause `iswalpha` (and hence `iswalnum`) to return nonzero, yet still `causeiswupper` and `iswlower` to return zero.

### 11.33.1 WEOF

```
#define WEOF <wint_t constant expression>
```

The macro yields the return value, of type `wint_t`, used to signal the end of a wide stream or to report an error condition.

```
/* TYPES */
typedef s_type wctrans_t;
typedef s_type wctype_t;
typedef i_type wint_t;

/* FUNCTIONS */ int iswalnum(wint_t c);
int iswalpha(wint_t c);
int iswblank(wint_t c); [Added with C99]
int iswcntrl(wint_t c);
int iswctype(wint_t c, wctype_t category);
int iswdigit(wint_t c);
int iswgraph(wint_t c);
int iswlower(wint_t c);
int iswprint(wint_t c);
int iswpunct(wint_t c);
int iswspace(wint_t c);
int iswupper(wint_t c);
int iswxdigit(wint_t c);

wint_t towctrans(wint_t c, wctrans_t category);
wint_t towlower(wint_t c);
wint_t towupper(wint_t c);

wctrans_t wctrans(const char *property);
wctype_t wctype(const char *property);
```

### 11.33.2 iswalnum

```
int iswalnum(wint_t c);
```

The function returns nonzero if `c` is any of:

```
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 z
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 Z
0 1 2 3 4 5 6 7 8 9
```

Or any other locale-specific alphabetic character.

### 11.33.3 iswalpha

```
int iswalpha(wint_t c);
```

The function returns nonzero if `c` is any of:

```
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 z
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 Z
```

Or any other locale-specific alphabetic character.

### 11.33.4 iswblank

```
int iswblank(wint_t c); [Added with C99]
```

The function returns nonzero if `c` is any of:

```
HT space
```

Or any other locale-specific blank character.

### 11.33.5 iswcntrl

```
int iswcntrl(wint_t c);
```

The function returns nonzero if `c` is any of:

```
BEL BS CR FF HT NL VT
```

Or any other implementation-defined control character.

### 11.33.6 iswctype

```
int iswctype(wint_t c, wctype_t category);
```

The function returns nonzero if `c` is any character in the category `category`. The value of `category` must have been returned by an earlier successful call to `wctype`.

### 11.33.7 iswdigit

```
int iswdigit(wint_t c);
```

The function returns nonzero if `c` is any of:

```
0 1 2 3 4 5 6 7 8 9
```

### 11.33.8 iswgraph

```
int iswgraph(wint_t c);
```

The function returns nonzero if `c` is any character for which either `iswalnum` or `iswpunct` returns nonzero.

### 11.33.9 iswlower

```
int iswlower(wint_t c);
```

The function returns nonzero if `c` is any of:

```
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 z
```

Or any other locale-specific lowercase character.

## 11.33.10 iswprint

```
int iswprint(wint_t c);
```

The function returns nonzero if `c` is space, a character for which `iswgraph` returns nonzero, or an implementation-defined subset of the characters for which `iswspace` returns nonzero.

## 11.33.11 iswpunct

```
int iswpunct(wint_t c);
```

The function returns nonzero if `c` is any of:

```
! " # % & ' ( ) ; < = > ? [ \ ] * + , -. / : ^ _ { | } ~
```

Or any other implementation-defined punctuation character.

## 11.33.12 iswspace

```
int iswspace(wint_t c);
```

The function returns nonzero if `c` is any of:

```
CR FF HT NL VT space
```

Or any other locale-specific space character.

## 11.33.13 iswupper

```
int iswupper(wint_t c);
```

The function returns nonzero if `c` is any of:

```
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 Z
```

Or any other locale-specific uppercase character.

## 11.33.14 iswxdigit

```
int iswxdigit(wint_t c);
```

The function returns nonzero if `c` is any of

```
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

## 11.33.15towctrans

```
wint_t towctrans(wint_t c, wctrans_t category);
```

The function returns the transformation of the character `c`, using the transform in the category `category`. The value of category must have been returned by an earlier successful call `towctrans`.

## 11.33.16towlower

```
wint_t towlower(wint_t c);
```

The function returns the corresponding lowercase letter if one exists and `ifiswupper(c)`; otherwise, it returns `c`.

## 11.33.17towupper

```
wint_t towupper(wint_t c);
```

The function returns both the corresponding uppercase letter, if one exists, and `ifiswlower(c)`; otherwise, it returns `c`.

## 11.33.18wctrans

```
wctrans_t wctrans(const char *property);
```

The function determines a mapping from one set of wide-character codes to another. If the `LC_CTYPE` category of the current locale does not define a mapping whose name matches the property string property, the function returns zero. Otherwise, it returns a nonzero value suitable for use as the second argument to a subsequent call to `towctrans`.

The following pairs of calls have the same behavior in all locales (but an implementation can define additional mappings even in the "C" locale):

```
towlower(c) same as towctrans(c, wctrans("tolower"))towupper(c) same
as towctrans(c, wctrans("toupper"))
```

## 11.33.19wctrans_t

```
typedef s_type wctrans_t;
```

The type is the scalar type `s-type` that can represent locale-specific character mappings, as specified by the return value of `wctrans`.

## 11.33.20wctype

```
wctype_t wctype(const char *property);wctrans_t wctrans(const char
*property);
```

The function determines a classification rule for wide-character codes. If the `LC_CTYPE` category of the current locale does not define a classification rule whose name matches the property string property, the function returns zero. Otherwise, it returns a nonzero value suitable for use as the second argument to a subsequent call `totowctrans`.

The following pairs of calls have the same behavior in all locales (but an implementation can define additional classification rules even in the "C" locale):

```
iswalnum(c)same as iswctype(c, wctype("alnum"))
iswalpha(c)same as iswctype(c, wctype("alpha"))
iswblank(c)same as iswctype(c, wctype("blank"))
iswcntrl(c)same as iswctype(c, wctype("cntrl"))
iswdigit(c)same as iswctype(c, wctype("digit"))
iswgraph(c)same as iswctype(c, wctype("graph"))
iswlower(c)same as iswctype(c, wctype("lower"))
iswprint(c)same as iswctype(c, wctype("print"))
iswpunct(c)same as iswctype(c, wctype("punct"))
iswspace(c)same as iswctype(c, wctype("space"))
iswupper(c)same as iswctype(c, wctype("upper"))
iswxdigit(c) same as iswctype(c, wctype("xdigit"))
```

## 11.33.21wctype_t

```
typedef s_type wctype_t;
```

The type is the scalar type `s-type` that can represent locale-specific character classifications, as specified by the return value `ofwctype`.

## 11.33.22wint_t

```
typedef i_type wint_t;
```

The type is the integer type `i_type` that can represent all values of type `wchar_t` as well as the value of `WEOF`, and that does not change when promoted.

# 12 Copyright and license notices

## 12.1 Dinkumware notice

Dinkumware, Ltd.
398 Main Street
Concord MA 01742

**Dinkum® C++ Library** developed by P.J. Plauger
**Dinkum C++ Library Reference** developed by P.J. Plauger
**Dinkum C Library Reference** developed by P.J. Plauger and Jim Brodie
Additional libraries and documentation developed by Dinkumware, Ltd.

The Dinkum C++ Library and additional libraries, in machine-readable or printed form (Dinkum Library), and the Dinkum C++ Library Reference and additional documentation, in machine-readable or printed form (Dinkum Reference), hereafter in whole or in part the Product, are all copyrighted by P.J. Plauger and/or Dinkumware, Ltd. ALL RIGHTS RESERVED. The Product is derived in part from books copyright © 1992-2001 by P.J. Plauger.

Dinkumware, Ltd. and P.J. Plauger (Licensor) retain exclusive ownership of this Product. It is licensed to you (Licensee) in accordance with the terms specifically stated in this Notice. If you have obtained this Product from a third party or under a special license from Dinkumware, Ltd., additional restrictions may also apply. You must otherwise treat the Product the same as other copyrighted material, such as a book or recording. You may also exercise certain rights particular to computer software under copyright law. In particular:

- You may use the Library portion of the Product (if present) to compile and link with C/C++ code to produce executable files.
- You may freely distribute such executable files for no additional license fee to Licensor.
- You may make one or more backup copies of the Product for archival purposes.
- You may permanently transfer ownership of the Product to another party only if the other party agrees to the terms stated in this Notice and you transfer or destroy all copies of the Product that are in your possession.
- You must preserve this Notice and all copyright notices with any copy you make of the Product.
- You may not loan, rent, or sublicense the Product.
- You may not copy or distribute, in any form, any part of this Product for any purpose not specifically permitted by this Notice.

This copy of the Product is licensed for use by a limited number of developers, which is specified as part of the packaging for this Product. A license for up to ten users, for example, limits to ten the number of developers reasonably able to use the Product at any instant of time. Thus, ten is the maximum number of possible concurrent users, not the number of actual concurrent users. A single-user license is for use by just one developer.

Anyone who accesses this software has a moral responsibility not to aid or abet illegal copying by others. Licensor recognizes that the machine-readable format of the Product makes it particularly conducive to sharing within multi-user systems and across networks. Such use is permitted only so long as Licensee does not exceed the maximum number of possible concurrent users and takes reasonable precautions to protect the Product against unauthorized copying and against public access. In particular, please note that the ability to access this copy does not imply permission to use it or to copy it. Please note also that Licensor has expended considerable professional effort in the production of this Product, and continues to do so to keep it current.

Licensor warrants that the Product as shipped performs substantially in accordance with its documented purpose, and that the medium on which the Product is provided is free from defects in material and workmanship. To the extent permitted by law, any implied warranties on the Product are limited to 90 days.

Licensor's entire liability under this warranty shall be, at Licensor's option, either to refund the license fee paid by Licensee or to replace the medium on which the Product is provided. This is also Licensee's exclusive remedy. To qualify for this remedy, Licensee must demonstrate satisfactory proof of purchase to Licensor and return the Product in reasonably good condition to Licensor.

LICENSOR OTHERWISE MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THIS PRODUCT, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. LICENSOR SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING THIS PRODUCT, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. TO THE EXTENT PERMITTED BY LAW, LICENSOR SHALL NOT BE LIABLE FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES.

By using this Product, you agree to abide by the intellectual property laws and all other applicable laws of the USA, and the terms described above. You may be held legally responsible for any infringement that is caused or encouraged by your failure to abide by the terms of this Notice.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the government is subject to the restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause as DFARS 52.227-7013 and FAR 52.227-19. Unpublished rights are reserved under the Copyright Laws of the USA. Contractor/ Manufacturer is DINKUMWARE, LTD., 398 Main Street, Concord MA 01742.

The terms of this notice shall be governed by the laws of the Commonwealth of Massachusetts. THE RIGHTS AND OBLIGATIONS OF THE PARTIES SHALL NOT BE GOVERNED BY THE PROVISIONS OF THE U.N. CONVENTION FOR THE INTERNATIONAL SALE OF GOODS, 1980.

This Copyright and License Notice is the entire agreement of the parties with respect to the matters set forth herein, and supersedes any other oral or written agreements or communications relating thereto, and shall alone be binding. No provision appearing on any purchase order, quotation form, or other form originated by either party shall be applicable.

Dinkumware and Dinkum are registered trademarks of Dinkumware, Ltd.

End of Copyright and License Notice

### References


- ANSI Standard X3.159-1989 (New York NY: American National Standards Institute, 1989). The original C Standard, developed by the ANSI-authorized committee X3J11. The Rationale that accompanies the C Standard explains many of the decisions that went into it, if you can get your hands on a copy.
- ISO/IEC Standard 9899:1990 (Geneva: International Standards Organization, 1990). Until 1999, the official C Standard around the world. Aside from formatting details and section numbering, the ISO C Standard is identical to the ANSI C Standard.
- ISO/IEC Amendment 1 to Standard 9899:1990 (Geneva: International Standards Organization, 1995). The first (and only) amendment to the C Standard. It provides substantial support for manipulating large character sets.
- ISO/IEC Standard 9899:1999 (Geneva: International Standards Organization, 1999) as corrected through 2003. The official C Standard around the world, replacing ISO/IEC Standard 9899:1990.
- ISO/IEC Standard 14882:1998 (Geneva: International Standards Organization, 1998) as corrected through 2003. The official C++ Standard around the world. The ISO C++ Standard is identical to the ANSI C++ Standard.
- P.J. Plauger, The Standard C Library (Englewood Cliffs NJ: Prentice Hall, 1992). Contains a complete implementation of the Standard C library, as of 1992 at least, as well as text from the library portion of the C Standard and guidance in using the Standard C library.
- P.J. Plauger, The Draft Standard C++ Library (Englewood Cliffs NJ: Prentice Hall, 1995). Contains a complete implementation of the draft Standard C++ library as of early 1994.
- P.J. Plauger, Alexander Stepanov, Meng Lee, and David R. Musser, The Standard Template Library (Englewood Cliffs NJ: Prentice Hall, 2001). Contains a complete implementation of the Standard Template Library as incorporated into the C++ Standard.

### Bug reports

The author welcomes reports of any errors or omissions. Please report any bugs or difficulties to:

Dinkumware Support
Dinkumware, Ltd.
398 Main Street
Concord MA, 01742-2321
USA

+1-978-371-2773 (UTC -4 hours, -5 November through March)
+1-978-371-9014 (FAX)

support@dinkumware.com

## 12.2 NetBSD notices

Portions of this document are licensed under the following licenses.

**a64l.3:**



This code is derived from software contributed to The NetBSD Foundation by Klaus Klein.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

   This product includes software developed by the NetBSD Foundation, Inc. and its contributors.
4. Neither the name of The NetBSD Foundation nor the names of its contributors can be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE NETBSD FOUNDATION, INC. AND CONTRIBUTORS “AS IS”" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### access.2:

### alarm.3:

### bcmp.3:



This code is derived from software contributed to Berkeley by Chris Torek.

### bcopy.3:



This code is derived from software contributed to Berkeley by Chris Torek.

### bzero.3:



This code is derived from software contributed to Berkeley by Chris Torek.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors can be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### fcntl.2:

### ffs.3:



This code is derived from software contributed to Berkeley by Chris Torek.

### getcwd.3:

### getsubopt.3:

### gettimeofday.2:

### hcreate.3:

### index.3:



This code is derived from software contributed to Berkeley by Chris Torek.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors can be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### isascii.3:

### isinf.3:

### isnan.3:



Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors can be used to endorse or promote products derived from this software without specific prior written permission.

### j0.3:

**memcpy.3:**

### rindex.3:

### stat.2:

### strptime.3:

### swab.3:

### sysconf.3:

**times.3:**

### toascii.3:



Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors can be used to endorse or promote products derived from this software without specific prior written permission.

**tolower.3:**



This code is derived from software contributed to Berkeley by the American National Standards Committee X3, on Information Processing Systems.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. 3. Neither the name of the University nor the names of its contributors can be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### tsearch.3:

### unlink.2: