Go to the first, previous, next, last section, table of contents.

Arithmetic Functions

This chapter contains information about functions for doing basic arithmetic operations, such as splitting a float into its integer and fractional parts or retrieving the imaginary part of a complex value. These functions are declared in the header files `math.h' and `complex.h'.

Integers

The C language defines several integer data types: integer, short integer, long integer, and character, all in both signed and unsigned varieties. The GNU C compiler extends the language to contain long long integers as well.

The C integer types were intended to allow code to be portable among machines with different inherent data sizes (word sizes), so each type may have different ranges on different machines. The problem with this is that a program often needs to be written for a particular range of integers, and sometimes must be written for a particular size of storage, regardless of what machine the program runs on.

To address this problem, the GNU C library contains C type definitions you can use to declare integers that meet your exact needs. Because the GNU C library header files are customized to a specific machine, your program source code doesn't have to be.

These `typedef`s are in `stdint.h'.

If you require that an integer be represented in exactly N bits, use one of the following types, with the obvious mapping to bit size and signedness:

- int8_t
- int16_t
- int32_t
- int64_t
- uint8_t
- uint16_t
- uint32_t
- uint64_t

If your C compiler and target machine do not allow integers of a certain size, the corresponding above type does not exist.

If you don't need a specific storage size, but want the smallest data structure with *at least* N bits, use one of these:

- int_least8_t
- int_least16_t
- int_least32_t
- int_least64_t
- uint_least8_t

- uint_least16_t
- uint_least32_t
- uint_least64_t

If you don't need a specific storage size, but want the data structure that allows the fastest access while having at least N bits (and among data structures with the same access speed, the smallest one), use one of these:

- int_fast8_t
- int_fast16_t
- int_fast32_t
- int_fast64_t
- uint_fast8_t
- uint_fast16_t
- uint_fast32_t
- uint_fast64_t

If you want an integer with the widest range possible on the platform on which it is being used, use one of the following. If you use these, you should write code that takes into account the variable size and range of the integer.

- intmax_t
- uintmax_t

The GNU C library also provides macros that tell you the maximum and minimum possible values for each integer data type. The macro names follow these examples: `INT32_MAX`, `UINT8_MAX`, `INT_FAST32_MIN`, `INT_LEAST64_MIN`, `UINTMAX_MAX`, `INTMAX_MAX`, `INTMAX_MIN`. Note that there are no macros for unsigned integer minima. These are always zero.

There are similar macros for use with C's built in integer types which should come with your C compiler. These are described in section Data Type Measurements.

Don't forget you can use the C `sizeof` function with any of these data types to get the number of bytes of storage each uses.

Integer Division

This section describes functions for performing integer division. These functions are redundant when GNU CC is used, because in GNU C the `` `/' `` operator always rounds towards zero. But in other C implementations, `` `/' `` may round differently with negative arguments. `div` and `ldiv` are useful because they specify how to round the quotient: towards zero. The remainder has the same sign as the numerator.

These functions are specified to return a result *r* such that the value `r.quot`**denominator* + `r.rem` equals *numerator*.

To use these facilities, you should include the header file `` `stdlib.h' `` in your program.

Data Type: **div_t**

This is a structure type used to hold the result returned by the `div` function. It has the following members:

```
int quot
```

The quotient from the division.

`int rem`
The remainder from the division.

Function: div_t **div** *(int numerator, int denominator)*
This function `div` computes the quotient and remainder from the division of *numerator* by *denominator*, returning the result in a structure of type `div_t`.

If the result cannot be represented (as in a division by zero), the behavior is undefined.

Here is an example, albeit not a very useful one.

```
div_t result;
result = div (20, -6);
```

Now `result.quot` is `-3` and `result.rem` is `2`.

Data Type: **ldiv_t**
This is a structure type used to hold the result returned by the `ldiv` function. It has the following members:

`long int quot`
The quotient from the division.
`long int rem`
The remainder from the division.

(This is identical to `div_t` except that the components are of type `long int` rather than `int`.)

Function: ldiv_t **ldiv** *(long int numerator, long int denominator)*
The `ldiv` function is similar to `div`, except that the arguments are of type `long int` and the result is returned as a structure of type `ldiv_t`.

Data Type: **lldiv_t**
This is a structure type used to hold the result returned by the `lldiv` function. It has the following members:

`long long int quot`
The quotient from the division.
`long long int rem`
The remainder from the division.

(This is identical to `div_t` except that the components are of type `long long int` rather than `int`.)

Function: lldiv_t **lldiv** *(long long int numerator, long long int denominator)*
The `lldiv` function is like the `div` function, but the arguments are of type `long long int` and the result is returned as a structure of type `lldiv_t`.

The `lldiv` function was added in ISO C99.

Data Type: **imaxdiv_t**
This is a structure type used to hold the result returned by the `imaxdiv` function. It has the following members:

`intmax_t quot`
: The quotient from the division.

`intmax_t rem`
: The remainder from the division.

(This is identical to `div_t` except that the components are of type `intmax_t` rather than `int`.)

See section Integers for a description of the `intmax_t` type.

Function: imaxdiv_t **imaxdiv** *(intmax_t numerator, intmax_t denominator)*
: The `imaxdiv` function is like the `div` function, but the arguments are of type `intmax_t` and the result is returned as a structure of type `imaxdiv_t`.

See section Integers for a description of the `intmax_t` type.

The `imaxdiv` function was added in ISO C99.

Floating Point Numbers

Most computer hardware has support for two different kinds of numbers: integers (@math{...-3, -2, -1, 0, 1, 2, 3...}) and floating-point numbers. Floating-point numbers have three parts: the **mantissa**, the **exponent**, and the **sign bit**. The real number represented by a floating-point value is given by @ifnottex @math{(s ? -1 : 1) @mul{} 2^e @mul{} M} where @math{s} is the sign bit, @math{e} the exponent, and @math{M} the mantissa. See section Floating Point Representation Concepts, for details. (It is possible to have a different **base** for the exponent, but all modern hardware uses @math{2}.)

Floating-point numbers can represent a finite subset of the real numbers. While this subset is large enough for most purposes, it is important to remember that the only reals that can be represented exactly are rational numbers that have a terminating binary expansion shorter than the width of the mantissa. Even simple fractions such as @math{1/5} can only be approximated by floating point.

Mathematical operations and functions frequently need to produce values that are not representable. Often these values can be approximated closely enough for practical purposes, but sometimes they can't. Historically there was no way to tell when the results of a calculation were inaccurate. Modern computers implement the IEEE 754 standard for numerical computations, which defines a framework for indicating to the program when the results of calculation are not trustworthy. This framework consists of a set of **exceptions** that indicate why a result could not be represented, and the special values **infinity** and **not a number** (NaN).

Floating-Point Number Classification Functions

ISO C99 defines macros that let you determine what sort of floating-point number a variable holds.

Macro: int **fpclassify** *(float-type x)*
: This is a generic macro which works on all floating-point types and which returns a value of type `int`. The possible values are:

`FP_NAN`
: The floating-point number *x* is "Not a Number" (see section Infinity and NaN)

`FP_INFINITE`
: The value of *x* is either plus or minus infinity (see section Infinity and NaN)

`FP_ZERO`
: The value of *x* is zero. In floating-point formats like IEEE 754, where zero can be signed, this value is also returned if *x* is negative zero.

`FP_SUBNORMAL`
: Numbers whose absolute value is too small to be represented in the normal format are represented in an alternate, **denormalized** format (see section Floating Point Representation Concepts). This format is less precise but can represent values closer to zero. `fpclassify` returns this value for values of *x* in this alternate format.

`FP_NORMAL`
: This value is returned for all other values of *x*. It indicates that there is nothing special about the number.

`fpclassify` is most useful if more than one property of a number must be tested. There are more specific macros which only test one property at a time. Generally these macros execute faster than `fpclassify`, since there is special hardware support for them. You should therefore use the specific macros whenever possible.

Macro: int **isfinite** *(float-type x)*
: This macro returns a nonzero value if *x* is finite: not plus or minus infinity, and not NaN. It is equivalent to

```
(fpclassify (x) != FP_NAN && fpclassify (x) != FP_INFINITE)
```

`isfinite` is implemented as a macro which accepts any floating-point type.

Macro: int **isnormal** *(float-type x)*
: This macro returns a nonzero value if *x* is finite and normalized. It is equivalent to

```
(fpclassify (x) == FP_NORMAL)
```

Macro: int **isnan** *(float-type x)*
: This macro returns a nonzero value if *x* is NaN. It is equivalent to

```
(fpclassify (x) == FP_NAN)
```

Another set of floating-point classification functions was provided by BSD. The GNU C library also supports these functions; however, we recommend that you use the ISO C99 macros in new code. Those are standard and will be available more widely. Also, since they are macros, you do not have to worry about the type of their argument.

Function: int **isinf** *(double x)*
Function: int **isinff** *(float x)*
Function: int **isinfl** *(long double x)*
: This function returns `-1` if *x* represents negative infinity, `1` if *x* represents positive infinity, and `0` otherwise.

Function: int **isnan** *(double x)*
Function: int **isnanf** *(float x)*
Function: int **isnanl** *(long double x)*
This function returns a nonzero value if *x* is a "not a number" value, and zero otherwise.

Note: The `isnan` macro defined by ISO C99 overrides the BSD function. This is normally not a problem, because the two routines behave identically. However, if you really need to get the BSD function for some reason, you can write

```
(isnan) (x)
```

Function: int **finite** *(double x)*
Function: int **finitef** *(float x)*
Function: int **finitel** *(long double x)*
This function returns a nonzero value if *x* is finite or a "not a number" value, and zero otherwise.

Function: double **infnan** *(int error)*
This function is provided for compatibility with BSD. Its argument is an error code, `EDOM` or `ERANGE`; `infnan` returns the value that a math function would return if it set `errno` to that value. See section Error Reporting by Mathematical Functions. `-ERANGE` is also acceptable as an argument, and corresponds to `-HUGE_VAL` as a value.

In the BSD library, on certain machines, `infnan` raises a fatal signal in all cases. The GNU library does not do likewise, because that does not fit the ISO C specification.

Portability Note: The functions listed in this section are BSD extensions.

Errors in Floating-Point Calculations

FP Exceptions

The IEEE 754 standard defines five **exceptions** that can occur during a calculation. Each corresponds to a particular sort of error, such as overflow.

When exceptions occur (when exceptions are **raised**, in the language of the standard), one of two things can happen. By default the exception is simply noted in the floating-point **status word**, and the program continues as if nothing had happened. The operation produces a default value, which depends on the exception (see the table below). Your program can check the status word to find out which exceptions happened.

Alternatively, you can enable **traps** for exceptions. In that case, when an exception is raised, your program will receive the `SIGFPE` signal. The default action for this signal is to terminate the program. See section Signal Handling, for how you can change the effect of the signal.

In the System V math library, the user-defined function `matherr` is called when certain exceptions occur inside math library functions. However, the Unix98 standard deprecates this interface. We support it for historical compatibility, but recommend that you do not use it in new programs.

The exceptions defined in IEEE 754 are:

`Invalid Operation'
: This exception is raised if the given operands are invalid for the operation to be performed. Examples are (see IEEE 754, section 7):
  1. Addition or subtraction: @math{@infinity{} - @infinity{}}. (But @math{@infinity{} + @infinity{} = @infinity{}}).
  2. Multiplication: @math{0 @mul{} @infinity{}}.
  3. Division: @math{0/0} or @math{@infinity{}/@infinity{}}.
  4. Remainder: @math{x} REM @math{y}, where @math{y} is zero or @math{x} is infinite.
  5. Square root if the operand is less then zero. More generally, any mathematical function evaluated outside its domain produces this exception.
  6. Conversion of a floating-point number to an integer or decimal string, when the number cannot be represented in the target format (due to overflow, infinity, or NaN).
  7. Conversion of an unrecognizable input string.
  8. Comparison via predicates involving @math{<} or @math{>}, when one or other of the operands is NaN. You can prevent this exception by using the unordered comparison functions instead; see section Floating-Point Comparison Functions.

  If the exception does not trap, the result of the operation is NaN.

`Division by Zero'
: This exception is raised when a finite nonzero number is divided by zero. If no trap occurs the result is either @math{+@infinity{}} or @math{-@infinity{}}, depending on the signs of the operands.

`Overflow'
: This exception is raised whenever the result cannot be represented as a finite value in the precision format of the destination. If no trap occurs the result depends on the sign of the intermediate result and the current rounding mode (IEEE 754, section 7.3):
  1. Round to nearest carries all overflows to @math{@infinity{}} with the sign of the intermediate result.
  2. Round toward @math{0} carries all overflows to the largest representable finite number with the sign of the intermediate result.
  3. Round toward @math{-@infinity{}} carries positive overflows to the largest representable finite number and negative overflows to @math{-@infinity{}}.
  4. Round toward @math{@infinity{}} carries negative overflows to the most negative representable finite number and positive overflows to @math{@infinity{}}.

  Whenever the overflow exception is raised, the inexact exception is also raised.

`Underflow'
: The underflow exception is raised when an intermediate result is too small to be calculated accurately, or if the operation's result rounded to the destination precision is too small to be normalized. When no trap is installed for the underflow exception, underflow is signaled (via the underflow flag) only when both tininess and loss of accuracy have been detected. If no trap handler is installed the operation continues with an imprecise small value, or zero if the destination precision cannot hold the small exact result.

`Inexact'

This exception is signalled if a rounded result is not exact (such as when calculating the square root of two) or a result overflows without an overflow trap.

Infinity and NaN

IEEE 754 floating point numbers can represent positive or negative infinity, and **NaN** (not a number). These three values arise from calculations whose result is undefined or cannot be represented accurately. You can also deliberately set a floating-point variable to any of them, which is sometimes useful. Some examples of calculations that produce infinity or NaN:

@ifnottex

```
@math{1/0 = @infinity{}}
@math{log (0) = -@infinity{}}
@math{sqrt (-1) = NaN}
```

When a calculation produces any of these values, an exception also occurs; see section FP Exceptions.

The basic operations and math functions all accept infinity and NaN and produce sensible output. Infinities propagate through calculations as one would expect: for example, @math{2 + @infinity{} = @infinity{}}, @math{4/@infinity{} = 0}, atan @math{(@infinity{}) = @pi{}/2}. NaN, on the other hand, infects any calculation that involves it. Unless the calculation would produce the same result no matter what real value replaced NaN, the result is NaN.

In comparison operations, positive infinity is larger than all values except itself and NaN, and negative infinity is smaller than all values except itself and NaN. NaN is **unordered**: it is not equal to, greater than, or less than anything, *including itself*. `x == x` is false if the value of `x` is NaN. You can use this to test whether a value is NaN or not, but the recommended way to test for NaN is with the `isnan` function (see section Floating-Point Number Classification Functions). In addition, `<`, `>`, `<=`, and `>=` will raise an exception when applied to NaNs.

`` `math.h' `` defines macros that allow you to explicitly set a variable to infinity or NaN.

Macro: float **INFINITY**
: An expression representing positive infinity. It is equal to the value produced by mathematical operations like `1.0 / 0.0`. `-INFINITY` represents negative infinity.

  You can test whether a floating-point value is infinite by comparing it to this macro. However, this is not recommended; you should use the `isfinite` macro instead. See section Floating-Point Number Classification Functions.

  This macro was introduced in the ISO C99 standard.

Macro: float **NAN**
: An expression representing a value which is "not a number". This macro is a GNU extension, available only on machines that support the "not a number" value--that is to say, on all machines that support IEEE floating point.

  You can use `` `#ifdef NAN' `` to test whether the machine supports NaN. (Of course, you must arrange for GNU extensions to be visible, such as by defining

_GNU_SOURCE, and then you must include `math.h'.)

IEEE 754 also allows for another unusual value: negative zero. This value is produced when you divide a positive number by negative infinity, or when a negative result is smaller than the limits of representation. Negative zero behaves identically to zero in all calculations, unless you explicitly test the sign bit with signbit or copysign.

Examining the FPU status word

ISO C99 defines functions to query and manipulate the floating-point status word. You can use these functions to check for untrapped exceptions when it's convenient, rather than worrying about them in the middle of a calculation.

These constants represent the various IEEE 754 exceptions. Not all FPUs report all the different exceptions. Each constant is defined if and only if the FPU you are compiling for supports that exception, so you can test for FPU support with `#ifdef'. They are defined in `fenv.h'.

FE_INEXACT
: The inexact exception.

FE_DIVBYZERO
: The divide by zero exception.

FE_UNDERFLOW
: The underflow exception.

FE_OVERFLOW
: The overflow exception.

FE_INVALID
: The invalid exception.

The macro FE_ALL_EXCEPT is the bitwise OR of all exception macros which are supported by the FP implementation.

These functions allow you to clear exception flags, test for exceptions, and save and restore the set of exceptions flagged.

Function: int **feclearexcept** *(int excepts)*
: This function clears all of the supported exception flags indicated by *excepts*.

  The function returns zero in case the operation was successful, a non-zero value otherwise.

Function: int **feraiseexcept** *(int excepts)*
: This function raises the supported exceptions indicated by *excepts*. If more than one exception bit in *excepts* is set the order in which the exceptions are raised is undefined except that overflow (FE_OVERFLOW) or underflow (FE_UNDERFLOW) are raised before inexact (FE_INEXACT). Whether for overflow or underflow the inexact exception is also raised is also implementation dependent.

  The function returns zero in case the operation was successful, a non-zero value otherwise.

Function: int **fetestexcept** *(int excepts)*
: Test whether the exception flags indicated by the parameter *except* are

currently set. If any of them are, a nonzero value is returned which specifies which exceptions are set. Otherwise the result is zero.

To understand these functions, imagine that the status word is an integer variable named *status*. `feclearexcept` is then equivalent to `` `status &= ~excepts' `` and `fetestexcept` is equivalent to `` `(status & excepts)' ``. The actual implementation may be very different, of course.

Exception flags are only cleared when the program explicitly requests it, by calling `feclearexcept`. If you want to check for exceptions from a set of calculations, you should clear all the flags first. Here is a simple example of the way to use `fetestexcept`:

```
{
  double f;
  int raised;
  feclearexcept (FE_ALL_EXCEPT);
  f = compute ();
  raised = fetestexcept (FE_OVERFLOW | FE_INVALID);
  if (raised & FE_OVERFLOW) { /* ... */ }
  if (raised & FE_INVALID) { /* ... */ }
  /* ... */
}
```

You cannot explicitly set bits in the status word. You can, however, save the entire status word and restore it later. This is done with the following functions:

Function: int **fegetexceptflag** *(fexcept_t *flagp, int excepts)*
This function stores in the variable pointed to by *flagp* an implementation-defined value representing the current setting of the exception flags indicated by *excepts*.

The function returns zero in case the operation was successful, a non-zero value otherwise.

Function: int **fesetexceptflag** *(const fexcept_t *flagp, int excepts*) This function restores the flags for the exceptions indicated by *excepts* to the values stored in the variable pointed to by *flagp*.

The function returns zero in case the operation was successful, a non-zero value otherwise.

Note that the value stored in `fexcept_t` bears no resemblance to the bit mask returned by `fetestexcept`. The type may not even be an integer. Do not attempt to modify an `fexcept_t` variable.

Error Reporting by Mathematical Functions

Many of the math functions are defined only over a subset of the real or complex numbers. Even if they are mathematically defined, their result may be larger or smaller than the range representable by their return type. These are known as **domain errors**, **overflows**, and **underflows**, respectively. Math functions do several things when one of these errors occurs. In this manual we will refer to the complete response as **signalling** a domain error, overflow, or underflow.

When a math function suffers a domain error, it raises the invalid exception and returns NaN. It also sets *errno* to `EDOM`; this is for compatibility with old systems that

do not support IEEE 754 exception handling. Likewise, when overflow occurs, math functions raise the overflow exception and return @math{@infinity{}} or @math{-@infinity{}} as appropriate. They also set *errno* to `ERANGE`. When underflow occurs, the underflow exception is raised, and zero (appropriately signed) is returned. *errno* may be set to `ERANGE`, but this is not guaranteed.

Some of the math functions are defined mathematically to result in a complex value over parts of their domains. The most familiar example of this is taking the square root of a negative number. The complex math functions, such as `csqrt`, will return the appropriate complex value in this case. The real-valued functions, such as `sqrt`, will signal a domain error.

Some older hardware does not support infinities. On that hardware, overflows instead return a particular very large number (usually the largest representable number). `` `math.h' `` defines macros you can use to test for overflow on both old and new hardware.

Macro: double **HUGE_VAL**
Macro: float **HUGE_VALF**
Macro: long double **HUGE_VALL**
: An expression representing a particular very large number. On machines that use IEEE 754 floating point format, `HUGE_VAL` is infinity. On other machines, it's typically the largest positive number that can be represented.

  Mathematical functions return the appropriately typed version of `HUGE_VAL` or `-HUGE_VAL` when the result is too large to be represented.

Rounding Modes

Floating-point calculations are carried out internally with extra precision, and then rounded to fit into the destination type. This ensures that results are as precise as the input data. IEEE 754 defines four possible rounding modes:

Round to nearest.
: This is the default mode. It should be used unless there is a specific need for one of the others. In this mode results are rounded to the nearest representable value. If the result is midway between two representable values, the even representable is chosen. **Even** here means the lowest-order bit is zero. This rounding mode prevents statistical bias and guarantees numeric stability: round-off errors in a lengthy calculation will remain smaller than half of `FLT_EPSILON`.

Round toward plus Infinity.
: All results are rounded to the smallest representable value which is greater than the result.

Round toward minus Infinity.
: All results are rounded to the largest representable value which is less than the result.

Round toward zero.
: All results are rounded to the largest representable value whose magnitude is less than that of the result. In other words, if the result is negative it is rounded up; if it is positive, it is rounded down.

`` `fenv.h' `` defines constants which you can use to refer to the various rounding

modes. Each one will be defined if and only if the FPU supports the corresponding rounding mode.

`FE_TONEAREST`
: Round to nearest.

`FE_UPWARD`
: Round toward @math{+@infinity{}}.

`FE_DOWNWARD`
: Round toward @math{-@infinity{}}.

`FE_TOWARDZERO`
: Round toward zero.

Underflow is an unusual case. Normally, IEEE 754 floating point numbers are always normalized (see section Floating Point Representation Concepts). Numbers smaller than @math{2^r} (where @math{r} is the minimum exponent, `FLT_MIN_RADIX-1` for *float*) cannot be represented as normalized numbers. Rounding all such numbers to zero or @math{2^r} would cause some algorithms to fail at 0. Therefore, they are left in denormalized form. That produces loss of precision, since some bits of the mantissa are stolen to indicate the decimal point.

If a result is too small to be represented as a denormalized number, it is rounded to zero. However, the sign of the result is preserved; if the calculation was negative, the result is **negative zero**. Negative zero can also result from some operations on infinity, such as @math{4/-@infinity{}}. Negative zero behaves identically to zero except when the `copysign` or `signbit` functions are used to check the sign bit directly.

At any time one of the above four rounding modes is selected. You can find out which one with this function:

Function: int **fegetround** *(void)*
: Returns the currently selected rounding mode, represented by one of the values of the defined rounding mode macros.

To change the rounding mode, use this function:

Function: int **fesetround** *(int round)*
: Changes the currently selected rounding mode to *round*. If *round* does not correspond to one of the supported rounding modes nothing is changed. `fesetround` returns zero if it changed the rounding mode, a nonzero value if the mode is not supported.

You should avoid changing the rounding mode if possible. It can be an expensive operation; also, some hardware requires you to compile your program differently for it to work. The resulting code may run slower. See your compiler documentation for details.

Floating-Point Control Functions

IEEE 754 floating-point implementations allow the programmer to decide whether traps will occur for each of the exceptions, by setting bits in the **control word**. In C, traps result in the program receiving the `SIGFPE` signal; see section Signal Handling.

Note: IEEE 754 says that trap handlers are given details of the exceptional

situation, and can set the result value. C signals do not provide any mechanism to pass this information back and forth. Trapping exceptions in C is therefore not very useful.

It is sometimes necessary to save the state of the floating-point unit while you perform some calculation. The library provides functions which save and restore the exception flags, the set of exceptions that generate traps, and the rounding mode. This information is known as the **floating-point environment**.

The functions to save and restore the floating-point environment all use a variable of type `fenv_t` to store information. This type is defined in `` `fenv.h' ``. Its size and contents are implementation-defined. You should not attempt to manipulate a variable of this type directly.

To save the state of the FPU, use one of these functions:

Function: int **fegetenv** *(fenv_t *envp)*
: Store the floating-point environment in the variable pointed to by *envp*.

  The function returns zero in case the operation was successful, a non-zero value otherwise.

Function: int **feholdexcept** *(fenv_t *envp)*
: Store the current floating-point environment in the object pointed to by *envp*. Then clear all exception flags, and set the FPU to trap no exceptions. Not all FPUs support trapping no exceptions; if `feholdexcept` cannot set this mode, it returns nonzero value. If it succeeds, it returns zero.

The functions which restore the floating-point environment can take these kinds of arguments:

- Pointers to `fenv_t` objects, which were initialized previously by a call to `fegetenv` or `feholdexcept`.
- The special macro `FE_DFL_ENV` which represents the floating-point environment as it was available at program start.
- Implementation defined macros with names starting with `FE_` and having type `fenv_t *`. If possible, the GNU C Library defines a macro `FE_NOMASK_ENV` which represents an environment where every exception raised causes a trap to occur. You can test for this macro using `#ifdef`. It is only defined if `_GNU_SOURCE` is defined. Some platforms might define other predefined environments.

To set the floating-point environment, you can use either of these functions:

Function: int **fesetenv** *(const fenv_t *envp)*
: Set the floating-point environment to that described by *envp*.

  The function returns zero in case the operation was successful, a non-zero value otherwise.

Function: int **feupdateenv** *(const fenv_t *envp)*
: Like `fesetenv`, this function sets the floating-point environment to that described by *envp*. However, if any exceptions were flagged in the status word before `feupdateenv` was called, they remain flagged after the call. In other words, after `feupdateenv` is called, the status word is the bitwise OR of the previous status word and the one saved in *envp*.

The function returns zero in case the operation was successful, a non-zero value otherwise.

To control for individual exceptions if raising them causes a trap to occur, you can use the following two functions.

Portability Note: These functions are all GNU extensions.

Function: int **feenableexcept** *(int excepts)*
This functions enables traps for each of the exceptions as indicated by the parameter *except*. The individual excepetions are described in section Examining the FPU status word. Only the specified exceptions are enabled, the status of the other exceptions is not changed.

The function returns the previous enabled exceptions in case the operation was successful, `-1` otherwise.

Function: int **fedisableexcept** *(int excepts)*
This functions disables traps for each of the exceptions as indicated by the parameter *except*. The individual excepetions are described in section Examining the FPU status word. Only the specified exceptions are disabled, the status of the other exceptions is not changed.

The function returns the previous enabled exceptions in case the operation was successful, `-1` otherwise.

Function: int **fegetexcept** *(int excepts)*
The function returns a bitmask of all currently enabled exceptions. It returns `-1` in case of failure.

Arithmetic Functions

The C library provides functions to do basic operations on floating-point numbers. These include absolute value, maximum and minimum, normalization, bit twiddling, rounding, and a few others.

Absolute Value

These functions are provided for obtaining the **absolute value** (or **magnitude**) of a number. The absolute value of a real number *x* is *x* if *x* is positive, -*x* if *x* is negative. For a complex number *z*, whose real part is *x* and whose imaginary part is *y*, the absolute value is `sqrt (x*x + y*y)`.

Prototypes for `abs`, `labs` and `llabs` are in `stdlib.h'; `imaxabs` is declared in `inttypes.h'; `fabs`, `fabsf` and `fabsl` are declared in `math.h'. `cabs`, `cabsf` and `cabsl` are declared in `complex.h'.

Function: int **abs** *(int number)*
Function: long int **labs** *(long int number)*
Function: long long int **llabs** *(long long int number)*
Function: intmax_t **imaxabs** *(intmax_t number)*
These functions return the absolute value of *number*.

Most computers use a two's complement integer representation, in which the

absolute value of `INT_MIN` (the smallest possible `int`) cannot be represented; thus, `abs (INT_MIN)` is not defined.

`llabs` and `imaxdiv` are new to ISO C99.

See section Integers for a description of the `intmax_t` type.

Function: double **fabs** *(double number)*
Function: float **fabsf** *(float number)*
Function: long double **fabsl** *(long double number)*
This function returns the absolute value of the floating-point number *number*.

Function: double **cabs** *(complex double z)*
Function: float **cabsf** *(complex float z)*
Function: long double **cabsl** *(complex long double z)*
These functions return the absolute value of the complex number *z* (see section Complex Numbers). The absolute value of a complex number is:

```
sqrt (creal (z) * creal (z) + cimag (z) * cimag (z))
```

This function should always be used instead of the direct formula because it takes special care to avoid losing precision. It may also take advantage of hardware support for this operation. See `hypot` in section Exponentiation and Logarithms.

Normalization Functions

The functions described in this section are primarily provided as a way to efficiently perform certain low-level manipulations on floating point numbers that are represented internally using a binary radix; see section Floating Point Representation Concepts. These functions are required to have equivalent behavior even if the representation does not use a radix of 2, but of course they are unlikely to be particularly efficient in those cases.

All these functions are declared in `` `math.h' ``.

Function: double **frexp** *(double value, int *exponent)*
Function: float **frexpf** *(float value, int *exponent)*
Function: long double **frexpl** *(long double value, int *exponent)*
These functions are used to split the number *value* into a normalized fraction and an exponent.

If the argument *value* is not zero, the return value is *value* times a power of two, and is always in the range 1/2 (inclusive) to 1 (exclusive). The corresponding exponent is stored in `*exponent`; the return value multiplied by 2 raised to this exponent equals the original number *value*.

For example, `frexp (12.8, &exponent)` returns `0.8` and stores `4` in `exponent`.

If *value* is zero, then the return value is zero and zero is stored in `*exponent`.

Function: double **ldexp** *(double value, int exponent)*
Function: float **ldexpf** *(float value, int exponent)*
Function: long double **ldexpl** *(long double value, int exponent)*
These functions return the result of multiplying the floating-point number

value by 2 raised to the power *exponent*. (It can be used to reassemble floating-point numbers that were taken apart by `frexp`.)

For example, `ldexp (0.8, 4)` returns `12.8`.

The following functions, which come from BSD, provide facilities equivalent to those of `ldexp` and `frexp`.

Function: double **logb** *(double x)*
Function: float **logbf** *(float x)*
Function: long double **logbl** *(long double x)*
These functions return the integer part of the base-2 logarithm of *x*, an integer value represented in type `double`. This is the highest integer power of `2` contained in *x*. The sign of *x* is ignored. For example, `logb (3.5)` is `1.0` and `logb (4.0)` is `2.0`.

When `2` raised to this power is divided into *x*, it gives a quotient between `1` (inclusive) and `2` (exclusive).

If *x* is zero, the return value is minus infinity if the machine supports infinities, and a very small number if it does not. If *x* is infinity, the return value is infinity.

For finite *x*, the value returned by `logb` is one less than the value that `frexp` would store into `*`*`exponent`*.

Function: double **scalb** *(double value, int exponent)*
Function: float **scalbf** *(float value, int exponent)*
Function: long double **scalbl** *(long double value, int exponent)*
The `scalb` function is the BSD name for `ldexp`.

Function: long long int **scalbn** *(double x, int n)*
Function: long long int **scalbnf** *(float x, int n)*
Function: long long int **scalbnl** *(long double x, int n)*
`scalbn` is identical to `scalb`, except that the exponent *n* is an `int` instead of a floating-point number.

Function: long long int **scalbln** *(double x, long int n)*
Function: long long int **scalblnf** *(float x, long int n)*
Function: long long int **scalblnl** *(long double x, long int n)*
`scalbln` is identical to `scalb`, except that the exponent *n* is a `long int` instead of a floating-point number.

Function: long long int **significand** *(double x)*
Function: long long int **significandf** *(float x)*
Function: long long int **significandl** *(long double x)*
`significand` returns the mantissa of *x* scaled to the range @math{[1, 2)}. It is equivalent to `scalb (`*`x`*`, (double) -ilogb (`*`x`*`))`.

This function exists mainly for use in certain standardized tests of IEEE 754 conformance.

Rounding Functions

The functions listed here perform operations such as rounding and truncation of floating-point values. Some of these functions convert floating point numbers to integer values. They are all declared in `math.h'.

You can also convert floating-point numbers to integers simply by casting them to `int`. This discards the fractional part, effectively rounding towards zero. However, this only works if the result can actually be represented as an `int`---for very large numbers, this is impossible. The functions listed here return the result as a `double` instead to get around this problem.

Function: double **ceil** *(double x)*
Function: float **ceilf** *(float x)*
Function: long double **ceill** *(long double x)*
These functions round *x* upwards to the nearest integer, returning that value as a `double`. Thus, `ceil (1.5)` is `2.0`.

Function: double **floor** *(double x)*
Function: float **floorf** *(float x)*
Function: long double **floorl** *(long double x)*
These functions round *x* downwards to the nearest integer, returning that value as a `double`. Thus, `floor (1.5)` is `1.0` and `floor (-1.5)` is `-2.0`.

Function: double **trunc** *(double x)*
Function: float **truncf** *(float x)*
Function: long double **truncl** *(long double x)*
The `trunc` functions round *x* towards zero to the nearest integer (returned in floating-point format). Thus, `trunc (1.5)` is `1.0` and `trunc (-1.5)` is `-1.0`.

Function: double **rint** *(double x)*
Function: float **rintf** *(float x)*
Function: long double **rintl** *(long double x)*
These functions round *x* to an integer value according to the current rounding mode. See section Floating Point Parameters, for information about the various rounding modes. The default rounding mode is to round to the nearest integer; some machines support other modes, but round-to-nearest is always used unless you explicitly select another.

If *x* was not initially an integer, these functions raise the inexact exception.

Function: double **nearbyint** *(double x)*
Function: float **nearbyintf** *(float x)*
Function: long double **nearbyintl** *(long double x)*
These functions return the same value as the `rint` functions, but do not raise the inexact exception if *x* is not an integer.

Function: double **round** *(double x)*
Function: float **roundf** *(float x)*
Function: long double **roundl** *(long double x)*
These functions are similar to `rint`, but they round halfway cases away from zero instead of to the nearest even integer.

Function: long int **lrint** *(double x)*
Function: long int **lrintf** *(float x)*
Function: long int **lrintl** *(long double x)*

These functions are just like `rint`, but they return a `long int` instead of a floating-point number.

Function: long long int **llrint** *(double x)*
Function: long long int **llrintf** *(float x)*
Function: long long int **llrintl** *(long double x)*
These functions are just like `rint`, but they return a `long long int` instead of a floating-point number.

Function: long int **lround** *(double x)*
Function: long int **lroundf** *(float x)*
Function: long int **lroundl** *(long double x)*
These functions are just like `round`, but they return a `long int` instead of a floating-point number.

Function: long long int **llround** *(double x)*
Function: long long int **llroundf** *(float x)*
Function: long long int **llroundl** *(long double x)*
These functions are just like `round`, but they return a `long long int` instead of a floating-point number.

Function: double **modf** *(double value, double *integer-part)*
Function: float **modff** *(float value, float *integer-part)*
Function: long double **modfl** *(long double value, long double *integer-part)*
These functions break the argument *value* into an integer part and a fractional part (between `-1` and `1`, exclusive). Their sum equals *value*. Each of the parts has the same sign as *value*, and the integer part is always rounded toward zero.

`modf` stores the integer part in `*integer-part`, and returns the fractional part. For example, `modf (2.5, &intpart)` returns `0.5` and stores `2.0` into `intpart`.

Remainder Functions

The functions in this section compute the remainder on division of two floating-point numbers. Each is a little different; pick the one that suits your problem.

Function: double **fmod** *(double numerator, double denominator)*
Function: float **fmodf** *(float numerator, float denominator)*
Function: long double **fmodl** *(long double numerator, long double denominator)*
These functions compute the remainder from the division of *numerator* by *denominator*. Specifically, the return value is `numerator - n * denominator`, where *n* is the quotient of *numerator* divided by *denominator*, rounded towards zero to an integer. Thus, `fmod (6.5, 2.3)` returns `1.9`, which is `6.5` minus `4.6`.

The result has the same sign as the *numerator* and has magnitude less than the magnitude of the *denominator*.

If *denominator* is zero, `fmod` signals a domain error.

Function: double **drem** *(double numerator, double denominator)*
Function: float **dremf** *(float numerator, float denominator)*
Function: long double **dreml** *(long double numerator, long double denominator)*
These functions are like `fmod` except that they rounds the internal quotient *n* to

the nearest integer instead of towards zero to an integer. For example, `drem (6.5, 2.3)` returns `-0.4`, which is `6.5` minus `6.9`.

The absolute value of the result is less than or equal to half the absolute value of the *denominator*. The difference between `fmod (numerator, denominator)` and `drem (numerator, denominator)` is always either *denominator*, minus *denominator*, or zero.

If *denominator* is zero, `drem` signals a domain error.

Function: double **remainder** *(double numerator, double denominator)*
Function: float **remainderf** *(float numerator, float denominator)*
Function: long double **remainderl** *(long double numerator, long double denominator)*
This function is another name for `drem`.

Setting and modifying single bits of FP values

There are some operations that are too complicated or expensive to perform by hand on floating-point numbers. ISO C99 defines functions to do these operations, which mostly involve changing single bits.

Function: double **copysign** *(double x, double y)*
Function: float **copysignf** *(float x, float y)*
Function: long double **copysignl** *(long double x, long double y)*
These functions return *x* but with the sign of *y*. They work even if *x* or *y* are NaN or zero. Both of these can carry a sign (although not all implementations support it) and this is one of the few operations that can tell the difference.

`copysign` never raises an exception.

This function is defined in IEC 559 (and the appendix with recommended functions in IEEE 754/IEEE 854).

Function: int **signbit** *(float-type x)*
`signbit` is a generic macro which can work on all floating-point types. It returns a nonzero value if the value of *x* has its sign bit set.

This is not the same as `x < 0.0`, because IEEE 754 floating point allows zero to be signed. The comparison `-0.0 < 0.0` is false, but `signbit (-0.0)` will return a nonzero value.

Function: double **nextafter** *(double x, double y)*
Function: float **nextafterf** *(float x, float y)*
Function: long double **nextafterl** *(long double x, long double y)*
The `nextafter` function returns the next representable neighbor of *x* in the direction towards *y*. The size of the step between *x* and the result depends on the type of the result. If @math{*x* = *y*} the function simply returns *y*. If either value is `NaN`, `NaN` is returned. Otherwise a value corresponding to the value of the least significant bit in the mantissa is added or subtracted, depending on the direction. `nextafter` will signal overflow or underflow if the result goes outside of the range of normalized numbers.

This function is defined in IEC 559 (and the appendix with recommended

functions in IEEE 754/IEEE 854).

Function: double **nexttoward** *(double x, long double y)*
Function: float **nexttowardf** *(float x, long double y)*
Function: long double **nexttowardl** *(long double x, long double y)*
These functions are identical to the corresponding versions of `nextafter` except that their second argument is a `long double`.

Function: double **nan** *(const char *tagp)*
Function: float **nanf** *(const char *tagp)*
Function: long double **nanl** *(const char *tagp)*
The `nan` function returns a representation of NaN, provided that NaN is supported by the target platform. `nan ("`*n-char-sequence*`")` is equivalent to `strtod ("NAN(`*n-char-sequence*`)")`.

The argument *tagp* is used in an unspecified manner. On IEEE 754 systems, there are many representations of NaN, and *tagp* selects one. On other systems it may do nothing.

Floating-Point Comparison Functions

The standard C comparison operators provoke exceptions when one or other of the operands is NaN. For example,

```
int v = a < 1.0;
```

will raise an exception if *a* is NaN. (This does *not* happen with `==` and `!=`; those merely return false and true, respectively, when NaN is examined.) Frequently this exception is undesirable. ISO C99 therefore defines comparison functions that do not raise exceptions when NaN is examined. All of the functions are implemented as macros which allow their arguments to be of any floating-point type. The macros are guaranteed to evaluate their arguments only once.

Macro: int **isgreater** *(real-floating x, real-floating y)*
This macro determines whether the argument *x* is greater than *y*. It is equivalent to `(`*x*`) > (`*y*`)`, but no exception is raised if *x* or *y* are NaN.

Macro: int **isgreaterequal** *(real-floating x, real-floating y)*
This macro determines whether the argument *x* is greater than or equal to *y*. It is equivalent to `(`*x*`) >= (`*y*`)`, but no exception is raised if *x* or *y* are NaN.

Macro: int **isless** *(real-floating x, real-floating y)*
This macro determines whether the argument *x* is less than *y*. It is equivalent to `(`*x*`) < (`*y*`)`, but no exception is raised if *x* or *y* are NaN.

Macro: int **islessequal** *(real-floating x, real-floating y)*
This macro determines whether the argument *x* is less than or equal to *y*. It is equivalent to `(`*x*`) <= (`*y*`)`, but no exception is raised if *x* or *y* are NaN.

Macro: int **islessgreater** *(real-floating x, real-floating y)*
This macro determines whether the argument *x* is less or greater than *y*. It is equivalent to `(`*x*`) < (`*y*`) || (`*x*`) > (`*y*`)` (although it only evaluates *x* and *y* once), but no exception is raised if *x* or *y* are NaN.

This macro is not equivalent to *x* `!=` *y*, because that expression is true if *x* or *y*

are NaN.

Macro: int **isunordered** *(real-floating x, real-floating y)*
This macro determines whether its arguments are unordered. In other words, it is true if *x* or *y* are NaN, and false otherwise.

Not all machines provide hardware support for these operations. On machines that don't, the macros can be very slow. Therefore, you should not use these functions when NaN is not a concern.

Note: There are no macros `isequal` or `isunequal`. They are unnecessary, because the `==` and `!=` operators do *not* throw an exception if one or both of the operands are NaN.

Miscellaneous FP arithmetic functions

The functions in this section perform miscellaneous but common operations that are awkward to express with C operators. On some processors these functions can use special machine instructions to perform these operations faster than the equivalent C code.

Function: double **fmin** *(double x, double y)*
Function: float **fminf** *(float x, float y)*
Function: long double **fminl** *(long double x, long double y)*
The `fmin` function returns the lesser of the two values *x* and *y*. It is similar to the expression

```
((x) < (y) ? (x) : (y))
```

except that *x* and *y* are only evaluated once.

If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

Function: double **fmax** *(double x, double y)*
Function: float **fmaxf** *(float x, float y)*
Function: long double **fmaxl** *(long double x, long double y)*
The `fmax` function returns the greater of the two values *x* and *y*.

If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

Function: double **fdim** *(double x, double y)*
Function: float **fdimf** *(float x, float y)*
Function: long double **fdiml** *(long double x, long double y)*
The `fdim` function returns the positive difference between *x* and *y*. The positive difference is @math{*x* - *y*} if *x* is greater than *y*, and @math{0} otherwise.

If *x*, *y*, or both are NaN, NaN is returned.

Function: double **fma** *(double x, double y, double z)*
Function: float **fmaf** *(float x, float y, float z)*
Function: long double **fmal** *(long double x, long double y, long double z)*
The `fma` function performs floating-point multiply-add. This is the operation @math{(*x* @mul{} *y*) + *z*}, but the intermediate result is not rounded to the

destination type. This can sometimes improve the precision of a calculation.

This function was introduced because some processors have a special instruction to perform multiply-add. The C compiler cannot use it directly, because the expression `x*y + z' is defined to round the intermediate result. `fma` lets you choose when you want to round only once.

On processors which do not implement multiply-add in hardware, `fma` can be very slow since it must avoid intermediate rounding. `math.h' defines the symbols `FP_FAST_FMA`, `FP_FAST_FMAF`, and `FP_FAST_FMAL` when the corresponding version of `fma` is no slower than the expression `x*y + z'. In the GNU C library, this always means the operation is implemented in hardware.

Complex Numbers

ISO C99 introduces support for complex numbers in C. This is done with a new type qualifier, `complex`. It is a keyword if and only if `complex.h' has been included. There are three complex types, corresponding to the three real types: `float complex`, `double complex`, and `long double complex`.

To construct complex numbers you need a way to indicate the imaginary part of a number. There is no standard notation for an imaginary floating point constant. Instead, `complex.h' defines two macros that can be used to create complex numbers.

Macro: const float complex **_Complex_I**
This macro is a representation of the complex number "@math{0+1i}". Multiplying a real floating-point value by `_Complex_I` gives a complex number whose value is purely imaginary. You can use this to construct complex constants:

```
@math{3.0 + 4.0i} = 3.0 + 4.0 * _Complex_I
```

Note that `_Complex_I * _Complex_I` has the value `-1`, but the type of that value is `complex`.

`_Complex_I` is a bit of a mouthful. `complex.h' also defines a shorter name for the same constant.

Macro: const float complex **I**
This macro has exactly the same value as `_Complex_I`. Most of the time it is preferable. However, it causes problems if you want to use the identifier `I` for something else. You can safely write

```
#include <complex.h>
#undef I
```

if you need `I` for your own purposes. (In that case we recommend you also define some other short name for `_Complex_I`, such as `J`.)

Projections, Conjugates, and Decomposing of Complex Numbers

ISO C99 also defines functions that perform basic operations on complex numbers, such as decomposition and conjugation. The prototypes for all these functions are in `complex.h'. All functions are available in three variants, one for each of the three complex types.

Function: double **creal** *(complex double z)*
Function: float **crealf** *(complex float z)*
Function: long double **creall** *(complex long double z)*
These functions return the real part of the complex number *z*.

Function: double **cimag** *(complex double z)*
Function: float **cimagf** *(complex float z)*
Function: long double **cimagl** *(complex long double z)*
These functions return the imaginary part of the complex number *z*.

Function: complex double **conj** *(complex double z)*
Function: complex float **conjf** *(complex float z)*
Function: complex long double **conjl** *(complex long double z)*
These functions return the conjugate value of the complex number *z*. The conjugate of a complex number has the same real part and a negated imaginary part. In other words, `conj(a + bi) = a + -bi'.

Function: double **carg** *(complex double z)*
Function: float **cargf** *(complex float z)*
Function: long double **cargl** *(complex long double z)*
These functions return the argument of the complex number *z*. The argument of a complex number is the angle in the complex plane between the positive real axis and a line passing through zero and the number. This angle is measured in the usual fashion and ranges from @math{0} to @math{2@pi{}}.

`carg` has a branch cut along the positive real axis.

Function: complex double **cproj** *(complex double z)*
Function: complex float **cprojf** *(complex float z)*
Function: complex long double **cprojl** *(complex long double z)*
These functions return the projection of the complex value *z* onto the Riemann sphere. Values with a infinite imaginary part are projected to positive infinity on the real axis, even if the real part is NaN. If the real part is infinite, the result is equivalent to

```
INFINITY + I * copysign (0.0, cimag (z))
```

Parsing of Numbers

This section describes functions for "reading" integer and floating-point numbers from a string. It may be more convenient in some cases to use `sscanf` or one of the related functions; see section Formatted Input. But often you can make a program more robust by finding the tokens in the string by hand, then converting the numbers one by one.

Parsing of Integers

The `str' functions are declared in `stdlib.h' and those beginning with `wcs' are declared in `wchar.h'. One might wonder about the use of `restrict' in the prototypes of the functions in this section. It is seemingly useless but the ISO C standard uses it (for the functions defined there) so we have to do it as well.

Function: long int **strtol** *(const char *restrict string, char **restrict tailptr, int base)*

The `strtol' ("string-to-long") function converts the initial part of *string* to a signed integer, which is returned as a value of type `long int'.

This function attempts to decompose *string* as follows:

- A (possibly empty) sequence of whitespace characters. Which characters are whitespace is determined by the `isspace' function (see section Classification of Characters). These are discarded.
- An optional plus or minus sign (`+' or `-').
- A nonempty sequence of digits in the radix specified by *base*. If *base* is zero, decimal radix is assumed unless the series of digits begins with `0' (specifying octal radix), or `0x' or `0X' (specifying hexadecimal radix); in other words, the same syntax used for integer constants in C. Otherwise *base* must have a value between `2' and `36'. If *base* is `16', the digits may optionally be preceded by `0x' or `0X'. If base has no legal value the value returned is `0l' and the global variable `errno' is set to `EINVAL'.
- Any remaining characters in the string. If *tailptr* is not a null pointer, `strtol' stores a pointer to this tail in `*tailptr'.

If the string is empty, contains only whitespace, or does not contain an initial substring that has the expected syntax for an integer in the specified *base*, no conversion is performed. In this case, `strtol' returns a value of zero and the value stored in `*tailptr' is the value of *string*.

In a locale other than the standard `"C"' locale, this function may recognize additional implementation-dependent syntax.

If the string has valid syntax for an integer but the value is not representable because of overflow, `strtol' returns either `LONG_MAX' or `LONG_MIN' (see section Range of an Integer Type), as appropriate for the sign of the value. It also sets `errno' to `ERANGE' to indicate there was overflow.

You should not check for errors by examining the return value of `strtol', because the string might be a valid representation of `0l', `LONG_MAX', or `LONG_MIN'. Instead, check whether *tailptr* points to what you expect after the number (e.g. `'\0'' if the string should end after the number). You also need to clear *errno* before the call and check it afterward, in case there was overflow.

There is an example at the end of this section.

Function: long int **wcstol** *(const wchar_t *restrict string, wchar_t **restrict tailptr, int base)*

The `wcstol' function is equivalent to the `strtol' function in nearly all aspects but handles wide character strings.

The `wcstol' function was introduced in Amendment 1 of ISO C90.

Function: unsigned long int **strtoul** *(const char *retrict string, char **restrict*

tailptr, int base)

The `strtoul` ("string-to-unsigned-long") function is like `strtol` except it converts to an `unsigned long int` value. The syntax is the same as described above for `strtol`. The value returned on overflow is `ULONG_MAX` (see section Range of an Integer Type).

If *string* depicts a negative number, `strtoul` acts the same as *strtol* but casts the result to an unsigned integer. That means for example that `strtoul` on `"-1"` returns `ULONG_MAX` and an input more negative than `LONG_MIN` returns (`ULONG_MAX` + 1) / 2.

`strtoul` sets *errno* to `EINVAL` if *base* is out of range, or `ERANGE` on overflow.

Function: unsigned long int **wcstoul** *(const wchar_t *restrict string, wchar_t **restrict tailptr, int base)*

The `wcstoul` function is equivalent to the `strtoul` function in nearly all aspects but handles wide character strings.

The `wcstoul` function was introduced in Amendment 1 of ISO C90.

Function: long long int **strtoll** *(const char *restrict string, char **restrict tailptr, int base)*

The `strtoll` function is like `strtol` except that it returns a `long long int` value, and accepts numbers with a correspondingly larger range.

If the string has valid syntax for an integer but the value is not representable because of overflow, `strtoll` returns either `LONG_LONG_MAX` or `LONG_LONG_MIN` (see section Range of an Integer Type), as appropriate for the sign of the value. It also sets `errno` to `ERANGE` to indicate there was overflow.

The `strtoll` function was introduced in ISO C99.

Function: long long int **wcstoll** *(const wchar_t *restrict string, wchar_t **restrict tailptr, int base)*

The `wcstoll` function is equivalent to the `strtoll` function in nearly all aspects but handles wide character strings.

The `wcstoll` function was introduced in Amendment 1 of ISO C90.

Function: long long int **strtoq** *(const char *restrict string, char **restrict tailptr, int base)*

`strtoq` ("string-to-quad-word") is the BSD name for `strtoll`.

Function: long long int **wcstoq** *(const wchar_t *restrict string, wchar_t **restrict tailptr, int base)*

The `wcstoq` function is equivalent to the `strtoq` function in nearly all aspects but handles wide character strings.

The `wcstoq` function is a GNU extension.

Function: unsigned long long int **strtoull** *(const char *restrict string, char **restrict tailptr, int base)*

The `strtoull` function is related to `strtoll` the same way `strtoul` is related to `strtol`.

The `strtoull` function was introduced in ISO C99.

Function: unsigned long long int **wcstoull** *(const wchar_t *restrict string, wchar_t **restrict tailptr, int base)*

The `wcstoull` function is equivalent to the `strtoull` function in nearly all aspects but handles wide character strings.

The `wcstoull` function was introduced in Amendment 1 of ISO C90.

Function: unsigned long long int **strtouq** *(const char *restrict string, char **restrict tailptr, int base)*

`strtouq` is the BSD name for `strtoull`.

Function: unsigned long long int **wcstouq** *(const wchar_t *restrict string, wchar_t **restrict tailptr, int base)*

The `wcstouq` function is equivalent to the `strtouq` function in nearly all aspects but handles wide character strings.

The `wcstoq` function is a GNU extension.

Function: intmax_t **strtoimax** *(const char *restrict string, char **restrict tailptr, int base)*

The `strtoimax` function is like `strtol` except that it returns a `intmax_t` value, and accepts numbers of a corresponding range.

If the string has valid syntax for an integer but the value is not representable because of overflow, `strtoimax` returns either `INTMAX_MAX` or `INTMAX_MIN` (see section Integers), as appropriate for the sign of the value. It also sets `errno` to `ERANGE` to indicate there was overflow.

See section Integers for a description of the `intmax_t` type. The `strtoimax` function was introduced in ISO C99.

Function: intmax_t **wcstoimax** *(const wchar_t *restrict string, wchar_t **restrict tailptr, int base)*

The `wcstoimax` function is equivalent to the `strtoimax` function in nearly all aspects but handles wide character strings.

The `wcstoimax` function was introduced in ISO C99.

Function: uintmax_t **strtoumax** *(const char *restrict string, char **restrict tailptr, int base)*

The `strtoumax` function is related to `strtoimax` the same way that `strtoul` is related to `strtol`.

See section Integers for a description of the `intmax_t` type. The `strtoumax` function was introduced in ISO C99.

Function: uintmax_t **wcstoumax** *(const wchar_t *restrict string, wchar_t **restrict tailptr, int base)*

The `wcstoumax` function is equivalent to the `strtoumax` function in nearly all aspects but handles wide character strings.

The `wcstoumax` function was introduced in ISO C99.

Function: long int **atol** *(const char *string)*
This function is similar to the `strtol` function with a *base* argument of `10`, except that it need not detect overflow errors. The `atol` function is provided mostly for compatibility with existing code; using `strtol` is more robust.

Function: int **atoi** *(const char *string)*
This function is like `atol`, except that it returns an `int`. The `atoi` function is also considered obsolete; use `strtol` instead.

Function: long long int **atoll** *(const char *string)*
This function is similar to `atol`, except it returns a `long long int`.

The `atoll` function was introduced in ISO C99. It too is obsolete (despite having just been added); use `strtoll` instead.

All the functions mentioned in this section so far do not handle alternative representations of characters as described in the locale data. Some locales specify thousands separator and the way they have to be used which can help to make large numbers more readable. To read such numbers one has to use the `scanf` functions with the `` `'' `` flag.

Here is a function which parses a string as a sequence of integers and returns the sum of them:

```
int
sum_ints_from_string (char *string)
{
  int sum = 0;

  while (1) {
    char *tail;
    int next;

    /* Skip whitespace by hand, to detect the end.  */
    while (isspace (*string)) string++;
    if (*string == 0)
      break;

    /* There is more nonwhitespace,  */
    /* so it ought to be another number.  */
    errno = 0;
    /* Parse it.  */
    next = strtol (string, &tail, 0);
    /* Add it in, if not overflow.  */
    if (errno)
      printf ("Overflow\n");
    else
      sum += next;
    /* Advance past it.  */
    string = tail;
  }

  return sum;
}
```

Parsing of Floats

The `` `str' `` functions are declared in `` `stdlib.h' `` and those beginning with `` `wcs' `` are declared in `` `wchar.h' ``. One might wonder about the use of `restrict` in the prototypes of the functions in this section. It is seemingly useless but the ISO C standard uses it (for the functions defined there) so we have to do it as well.

Function: double **strtod** *(const char *restrict string, char **restrict tailptr)*

The `strtod` ("string-to-double") function converts the initial part of *string* to a floating-point number, which is returned as a value of type `double`.

This function attempts to decompose *string* as follows:

- A (possibly empty) sequence of whitespace characters. Which characters are whitespace is determined by the `isspace` function (see section Classification of Characters). These are discarded.
- An optional plus or minus sign (`` `+' `` or `` `-' ``).
- A floating point number in decimal or hexadecimal format. The decimal format is:
  - A nonempty sequence of digits optionally containing a decimal-point character--normally `` `.' ``, but it depends on the locale (see section Generic Numeric Formatting Parameters).
  - An optional exponent part, consisting of a character `` `e' `` or `` `E' ``, an optional sign, and a sequence of digits.

  The hexadecimal format is as follows:
  - A 0x or 0X followed by a nonempty sequence of hexadecimal digits optionally containing a decimal-point character--normally `` `.' ``, but it depends on the locale (see section Generic Numeric Formatting Parameters).
  - An optional binary-exponent part, consisting of a character `` `p' `` or `` `P' ``, an optional sign, and a sequence of digits.
- Any remaining characters in the string. If *tailptr* is not a null pointer, a pointer to this tail of the string is stored in `*tailptr`.

If the string is empty, contains only whitespace, or does not contain an initial substring that has the expected syntax for a floating-point number, no conversion is performed. In this case, `strtod` returns a value of zero and the value returned in `*tailptr` is the value of *string*.

In a locale other than the standard `"C"` or `"POSIX"` locales, this function may recognize additional locale-dependent syntax.

If the string has valid syntax for a floating-point number but the value is outside the range of a `double`, `strtod` will signal overflow or underflow as described in section Error Reporting by Mathematical Functions.

`strtod` recognizes four special input strings. The strings `"inf"` and `"infinity"` are converted to @math{@infinity{}}, or to the largest representable value if the floating-point format doesn't support infinities. You can prepend a `"+"` or `"-"` to specify the sign. Case is ignored when scanning these strings.

The strings `"nan"` and `"nan(chars...)"` are converted to NaN. Again, case is ignored. If *chars...* are provided, they are used in some unspecified fashion to select a particular representation of NaN (there can be several).

Since zero is a valid result as well as the value returned on error, you should check for errors in the same way as for `strtol`, by examining *errno* and *tailptr*.

Function: float **strtof** *(const char *string, char **tailptr)*

Function: long double **strtold** *(const char *string, char **tailptr)*

These functions are analogous to `strtod`, but return `float` and `long double` values

respectively. They report errors in the same way as `strtod`. `strtof` can be substantially faster than `strtod`, but has less precision; conversely, `strtold` can be much slower but has more precision (on systems where `long double` is a separate type).

These functions have been GNU extensions and are new to ISO C99.

Function: double **wcstod** *(const wchar_t *restrict string, wchar_t **restrict tailptr)*
Function: float **wcstof** *(const wchar_t *string, wchar_t **tailptr)*
Function: long double **wcstold** *(const wchar_t *string, wchar_t **tailptr)*
The `wcstod`, `wcstof`, and `wcstol` functions are equivalent in nearly all aspect to the `strtod`, `strtof`, and `strtold` functions but it handles wide character string.

The `wcstod` function was introduced in Amendment 1 of ISO C90. The `wcstof` and `wcstold` functions were introduced in ISO C99.

Function: double **atof** *(const char *string)*
This function is similar to the `strtod` function, except that it need not detect overflow and underflow errors. The `atof` function is provided mostly for compatibility with existing code; using `strtod` is more robust.

The GNU C library also provides `` `_l' `` versions of these functions, which take an additional argument, the locale to use in conversion. See section Parsing of Integers.

Old-fashioned System V number-to-string functions

The old System V C library provided three functions to convert numbers to strings, with unusual and hard-to-use semantics. The GNU C library also provides these functions and some natural extensions.

These functions are only available in glibc and on systems descended from AT&T Unix. Therefore, unless these functions do precisely what you need, it is better to use `sprintf`, which is standard.

All these functions are defined in `` `stdlib.h' ``.

Function: char * **ecvt** *(double value, int ndigit, int *decpt, int *neg)*
The function `ecvt` converts the floating-point number *value* to a string with at most *ndigit* decimal digits. The returned string contains no decimal point or sign. The first digit of the string is non-zero (unless *value* is actually zero) and the last digit is rounded to nearest. `*`*`decpt`* is set to the index in the string of the first digit after the decimal point. `*`*`neg`* is set to a nonzero value if *value* is negative, zero otherwise.

If *ndigit* decimal digits would exceed the precision of a `double` it is reduced to a system-specific value.

The returned string is statically allocated and overwritten by each call to `ecvt`.

If *value* is zero, it is implementation defined whether `*`*`decpt`* is `0` or `1`.

For example: `ecvt (12.3, 5, &d, &n)` returns `"12300"` and sets *d* to `2` and *n* to `0`.

Function: char * **fcvt** *(double value, int ndigit, int *decpt, int *neg)*
The function `fcvt` is like `ecvt`, but *ndigit* specifies the number of digits after the decimal point. If *ndigit* is less than zero, *value* is rounded to the @math{*ndigit*+1}'th place to the left of the decimal point. For example, if *ndigit* is `-1`, *value* will be rounded to the nearest 10. If *ndigit* is negative and larger than the number of digits to the left of the decimal point in *value*, *value* will be rounded to one significant digit.

If *ndigit* decimal digits would exceed the precision of a `double` it is reduced to a system-specific value.

The returned string is statically allocated and overwritten by each call to `fcvt`.

Function: char * **gcvt** *(double value, int ndigit, char *buf)*
`gcvt` is functionally equivalent to `` `sprintf(buf, "%*g", ndigit, value' ``. It is provided only for compatibility's sake. It returns *buf*.

If *ndigit* decimal digits would exceed the precision of a `double` it is reduced to a system-specific value.

As extensions, the GNU C library provides versions of these three functions that take `long double` arguments.

Function: char * **qecvt** *(long double value, int ndigit, int *decpt, int *neg)*
This function is equivalent to `ecvt` except that it takes a `long double` for the first parameter and that *ndigit* is restricted by the precision of a `long double`.

Function: char * **qfcvt** *(long double value, int ndigit, int *decpt, int *neg)*
This function is equivalent to `fcvt` except that it takes a `long double` for the first parameter and that *ndigit* is restricted by the precision of a `long double`.

Function: char * **qgcvt** *(long double value, int ndigit, char *buf)*
This function is equivalent to `gcvt` except that it takes a `long double` for the first parameter and that *ndigit* is restricted by the precision of a `long double`.

The `ecvt` and `fcvt` functions, and their `long double` equivalents, all return a string located in a static buffer which is overwritten by the next call to the function. The GNU C library provides another set of extended functions which write the converted string into a user-supplied buffer. These have the conventional `_r` suffix.

`gcvt_r` is not necessary, because `gcvt` already uses a user-supplied buffer.

Function: char * **ecvt_r** *(double value, int ndigit, int *decpt, int *neg, char *buf, size_t len)*
The `ecvt_r` function is the same as `ecvt`, except that it places its result into the user-specified buffer pointed to by *buf*, with length *len*.

This function is a GNU extension.

Function: char * **fcvt_r** *(double value, int ndigit, int *decpt, int *neg, char *buf, size_t len)*
The `fcvt_r` function is the same as `fcvt`, except that it places its result into the user-specified buffer pointed to by *buf*, with length *len*.

This function is a GNU extension.

Function: char * **qecvt_r** *(long double value, int ndigit, int *decpt, int *neg, char *buf, size_t len)*

The `qecvt_r` function is the same as `qecvt`, except that it places its result into the user-specified buffer pointed to by *buf*, with length *len*.

This function is a GNU extension.

Function: char * **qfcvt_r** *(long double value, int ndigit, int *decpt, int *neg, char *buf, size_t len)*

The `qfcvt_r` function is the same as `qfcvt`, except that it places its result into the user-specified buffer pointed to by *buf*, with length *len*.

This function is a GNU extension.

Go to the first, previous, next, last section, table of contents.