MPFR

The Multiple Precision Floating-Point Reliable Library
Edition 2.3.1
January 2008

The MPFR team
mpfr@loria.fr

This manual documents how to install and use the Multiple Precision Floating-Point Reliable Library, version 2.3.1.

Table of Contents

MPFR Copying Conditions

1 Introduction to MPFR

MPFR is a portable library written in C for arbitrary precision arithmetic on floating-point numbers. It is based on the GNU MP library. It aims to extend the class of floating-point numbers provided by the GNU MP library by a precise semantics. The main differences with the `mpf` class from GNU MP are:

- the `mpfr` code is portable, i.e. the result of any operation does not depend (or should not) on the machine word size `mp_bits_per_limb` (32 or 64 on most machines);
- the precision in bits can be set exactly to any valid value for each variable (including very small precision);
- `mpfr` provides the four rounding modes from the IEEE 754-1985 standard.

In particular, with a precision of 53 bits, `mpfr` should be able to exactly reproduce all computations with double-precision machine floating-point numbers (`double` type in C), except the default exponent range is much wider and subnormal numbers are not implemented but can be emulated.

This version of MPFR is released under the GNU Lesser General Public License. It is permitted to link MPFR to non-free programs, as long as when distributing them the MPFR source code and a means to re-link with a modified MPFR library is provided.

1.1 How to Use This Manual

Everyone should read Chapter 4 [MPFR Basics], page 6. If you need to install the library yourself, you need to read Chapter 2 [Installing MPFR], page 3, too.

The rest of the manual can be used for later reference, although it is probably a good idea to glance through it.

2 Installing MPFR

2.1 How to Install

Here are the steps needed to install the library on Unix systems (more details are provided in the 'INSTALL' file):

1. To build MPFR, you first have to install GNU MP (version 4.1 or higher) on your computer. You need a C compiler, preferably GCC, but any reasonable compiler should work. And you need a standard Unix 'make' program, plus some other standard Unix utility programs.
2. In the MPFR build directory, type './configure'

   This will prepare the build and setup the options according to your system. If you get error messages, you might check that you use the same compiler and compile options as for GNU MP (see the 'INSTALL' file).
3. 'make'

   This will compile MPFR, and create a library archive file 'libmpfr.a'. A dynamic library may be produced too (see configure).
4. 'make check'

   This will make sure MPFR was built correctly. If you get error messages, please report this to 'mpfr@loria.fr'. (See Chapter 3 [Reporting Bugs], page 5, for information on what to include in useful bug reports.)
5. 'make install'

   This will copy the files 'mpfr.h' and 'mpf2mpfr.h' to the directory '/usr/local/include', the file 'libmpfr.a' to the directory '/usr/local/lib', and the file 'mpfr.info' to the directory '/usr/local/share/info' (or if you passed the '--prefix' option to 'configure', using the prefix directory given as argument to '--prefix' instead of '/usr/local').

2.2 Other 'make' Targets

There are some other useful make targets:

- 'mpfr.info' or 'info'

  Create an info version of the manual, in 'mpfr.info'.
- 'mpfr.dvi' or 'dvi'

  Create a DVI version of the manual, in 'mpfr.dvi'.
- 'mpfr.ps'

  Create a Postscript version of the manual, in 'mpfr.ps'.
- 'clean'

  Delete all object files and archive files, but not the configuration files.
- 'distclean'

  Delete all files not included in the distribution.
- 'uninstall'

  Delete all files copied by 'make install'.

2.3 Known Build Problems

MPFR suffers from all bugs from the GNU MP library, plus many more.

Please report other problems to 'mpfr@loria.fr'. See Chapter 3 [Reporting Bugs], page 5. Some bug fixes are available on the MPFR web page http://www.mpfr.org/.

2.4 Getting the Latest Version of MPFR

The latest version of MPFR is available from http://www.mpfr.org/.

3 Reporting Bugs

If you think you have found a bug in the MPFR library, first have a look on the MPFR web page http://www.mpfr.org/: perhaps this bug is already known, in which case you may find there a workaround for it. Otherwise, please investigate and report it. We have made this library available to you, and it is not to ask too much from you, to ask you to report the bugs that you find.

There are a few things you should think about when you put your bug report together.

You have to send us a test case that makes it possible for us to reproduce the bug. Include instructions on how to run the test case.

You also have to explain what is wrong; if you get a crash, or if the results printed are incorrect and in that case, in what way.

Please include compiler version information in your bug report. This can be extracted using '`cc -V`' on some machines, or, if you're using gcc, '`gcc -v`'. Also, include the output from '`uname -a`' and the MPFR version (the GMP version may be useful too).

If your bug report is good, we will do our best to help you to get a corrected version of the library; if the bug report is poor, we won't do anything about it (aside of chiding you to send better bug reports).

Send your bug report to: '`mpfr@loria.fr`'.

If you think something in this manual is unclear, or downright incorrect, or if the language needs to be improved, please send a note to the same address.

4 MPFR Basics

All declarations needed to use MPFR are collected in the include file '`mpfr.h`'. It is designed to work with both C and C++ compilers. You should include that file in any program using the MPFR library:

```
#include <mpfr.h>
```

4.1 Nomenclature and Types

A *floating-point number* or *float* for short, is an arbitrary precision significand (also called mantissa) with a limited precision exponent. The C data type for such objects is `mpfr_t` (internally defined as a one-element array of a structure, and `mpfr_ptr` is the C data type representing a pointer to this structure). A floating-point number can have three special values: Not-a-Number (NaN) or plus or minus Infinity. NaN represents an uninitialized object, the result of an invalid operation (like 0 divided by 0), or a value that cannot be determined (like +Infinity minus +Infinity). Moreover, like in the IEEE 754-1985 standard, zero is signed, i.e. there are both +0 and -0; the behavior is the same as in the IEEE 754-1985 standard and it is generalized to the other functions supported by MPFR.

The *precision* is the number of bits used to represent the significand of a floating-point number; the corresponding C data type is `mp_prec_t`. The precision can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`. In the current implementation, `MPFR_PREC_MIN` is equal to 2.

Warning! MPFR needs to increase the precision internally, in order to provide accurate results (and in particular, correct rounding). Do not attempt to set the precision to any value near `MPFR_PREC_MAX`, otherwise MPFR will abort due to an assertion failure. Moreover, you may reach some memory limit on your platform, in which case the program may abort, crash or have undefined behavior (depending on your C implementation).

The *rounding mode* specifies the way to round the result of a floating-point operation, in case the exact result can not be represented exactly in the destination significand; the corresponding C data type is `mp_rnd_t`.

A *limb* means the part of a multi-precision number that fits in a single word. (We chose this word because a limb of the human body is analogous to a digit, only larger, and containing several digits.) Normally a limb contains 32 or 64 bits. The C data type for a limb is `mp_limb_t`.

4.2 Function Classes

There is only one class of functions in the MPFR library:

1. Functions for floating-point arithmetic, with names beginning with `mpfr_`. The associated type is `mpfr_t`.

4.3 MPFR Variable Conventions

As a general rule, all MPFR functions expect output arguments before input arguments. This notation is based on an analogy with the assignment operator.

MPFR allows you to use the same variable for both input and output in the same expression. For example, the main function for floating-point multiplication, `mpfr_mul`, can be used like this: `mpfr_mul (x, x, x, rnd_mode)`. This computes the square of x with rounding mode `rnd_mode` and puts the result back in x.

Before you can assign to an MPFR variable, you need to initialize it by calling one of the special initialization functions. When you're done with a variable, you need to clear it out, using one of the functions for that purpose.

A variable should only be initialized once, or at least cleared out between each initialization. After a variable has been initialized, it may be assigned to any number of times.

For efficiency reasons, avoid to initialize and clear out a variable in loops. Instead, initialize it before entering the loop, and clear it out after the loop has exited.

You don't need to be concerned about allocating additional space for MPFR variables, since any variable has a significand of fixed size. Hence unless you change its precision, or clear and reinitialize it, a floating-point variable will have the same allocated space during all its life.

4.4 Rounding Modes

The following four rounding modes are supported:

- `GMP_RNDN`: round to nearest
- `GMP_RNDZ`: round toward zero
- `GMP_RNDU`: round toward plus infinity
- `GMP_RNDD`: round toward minus infinity

The '`round to nearest`' mode works as in the IEEE 754-1985 standard: in case the number to be rounded lies exactly in the middle of two representable numbers, it is rounded to the one with the least significant bit set to zero. For example, the number 5/2, which is represented by (10.1) in binary, is rounded to (10.0)=2 with a precision of two bits, and not to (11.0)=3. This rule avoids the *drift* phenomenon mentioned by Knuth in volume 2 of The Art of Computer Programming (Section 4.2.2).

Most MPFR functions take as first argument the destination variable, as second and following arguments the input variables, as last argument a rounding mode, and have a return value of type `int`, called the *ternary value*. The value stored in the destination variable is correctly rounded, i.e. MPFR behaves as if it computed the result with an infinite precision, then rounded it to the precision of this variable. The input variables are regarded as exact (in particular, their precision does not affect the result).

As a consequence, in case of a non-zero real rounded result, the error on the result is less or equal to 1/2 ulp (unit in the last place) of the target in the rounding to nearest mode, and less than 1 ulp of the target in the directed rounding modes (a ulp is the weight of the least significant represented bit of the target after rounding).

Unless documented otherwise, functions returning an `int` return a ternary value. If the ternary value is zero, it means that the value stored in the destination variable is the exact result of the corresponding mathematical function. If the ternary value is positive (resp. negative), it means the value stored in the destination variable is greater (resp. lower) than the exact result. For example with the `GMP_RNDU` rounding mode, the ternary value is usually positive, except when the result is exact, in which case it is zero. In the case of an infinite result, it is considered as inexact when it was obtained by overflow, and exact otherwise. A NaN result (Not-a-Number) always corresponds to an exact return value. The opposite of a returned ternary value is guaranteed to be representable in an `int`.

Unless documented otherwise, functions returning a `1` (or any other value specified in this manual) for special cases (like `acos(0)`) should return an overflow or an underflow if `1` is not representable in the current exponent range.

4.5 Floating-Point Values on Special Numbers

This section specifies the floating-point values (of type `mpfr_t`) returned by MPFR functions. For functions returning several values (like `mpfr_sin_cos`), the rules apply to each result separately.

Functions can have one or several input arguments. An input point is a mapping from these input arguments to the set of the MPFR numbers. When none of its components are NaN, an input point can also be seen as a tuple in the extended real numbers (the set of the real numbers with both infinities).

When the input point is in the domain of the mathematical function, the result is rounded as described in Section "Rounding Modes" (but see below for the specification of the sign of an exact zero). Otherwise the general rules from this section apply unless stated otherwise in the description of the MPFR function (Chapter 5 [MPFR Interface], page 10).

When the input point is not in the domain of the mathematical function but is in its closure in the extended real numbers and the function can be extended by continuity, the result is the obtained limit. Examples: `mpfr_hypot` on (+Inf,0) gives +Inf. But `mpfr_pow` cannot be defined on (1,+Inf) using this rule, as one can find sequences (x_n,y_n) such that x_n goes to 1, y_n goes to +Inf and $(x_n)^{y_n}$ goes to any positive value when n goes to the infinity.

When the input point is in the closure of the domain of the mathematical function and an input argument is +0 (resp. -0), one considers the limit when the corresponding argument approaches 0 from above (resp. below). If the limit is not defined (e.g., `mpfr_log` on -0), the behavior must be specified in the description of the MPFR function.

When the result is equal to 0, its sign is determined by considering the limit as if the input point were not in the domain: If one approaches 0 from above (resp. below), the result is +0 (resp. -0). In the other cases, the sign must be specified in the description of the MPFR function. Example: `mpfr_sin` on +0 gives +0.

When the input point is not in the closure of the domain of the function, the result is NaN. Example: `mpfr_sqrt` on -17 gives NaN.

When an input argument is NaN, the result is NaN, possibly except when a partial function is constant on the finite floating-point numbers; such a case is always explicitly specified in Chapter 5 [MPFR Interface], page 10. Example: `mpfr_hypot` on (NaN,0) gives NaN, but `mpfr_hypot` on (NaN,+Inf) gives +Inf (as specified in Section 5.7 [Special Functions], page 19), since for any finite input x, `mpfr_hypot` on (x,+Inf) gives +Inf.

4.6 Exceptions

MPFR supports 5 exception types:

- Underflow: An underflow occurs when the exact result of a function is a non-zero real number and the result obtained after the rounding, assuming an unbounded exponent range (for the rounding), has an exponent smaller than the minimum exponent of the current range. In the round-to-nearest mode, the halfway case is rounded toward zero.

  Note: This is not the single definition of the underflow. MPFR chooses to consider the underflow after rounding. The underflow before rounding can also be defined. For instance, consider a function that has the exact result $7 \times 2^{e-4}$, where e is the smallest exponent (for a significand between 1/2 and 1) in the current range, with a 2-bit target precision and rounding toward plus infinity. The exact result has the exponent $e-1$. With the underflow before rounding, such a function call would yield an underflow, as $e-1$ is outside the current exponent range. However, MPFR first considers the rounded result assuming an unbounded exponent range. The exact result cannot be represented exactly in precision 2, and here, it is rounded to 0.5×2^e, which is representable in the current exponent range. As a consequence, this will not yield an underflow in MPFR.

- Overflow: An overflow occurs when the exact result of a function is a non-zero real number and the result obtained after the rounding, assuming an unbounded exponent range (for the rounding), has an exponent larger than the maximum exponent of the current range. In the round-to-nearest mode, the result is infinite.
- NaN: A NaN exception occurs when the result of a function is a NaN.
- Inexact: An inexact exception occurs when the result of a function cannot be represented exactly and must be rounded.
- Range error: A range exception occurs when a function that does not return a MPFR number (such as comparisons and conversions to an integer) has an invalid result (e.g. an argument is NaN in `mpfr_cmp` or in a conversion to an integer).

MPFR has a global flag for each exception, which can be cleared, set or tested by functions described in Section 5.12 [Exception Related Functions], page 26.

Differences with the ISO C99 standard:

- In C, only quiet NaNs are specified, and a NaN propagation does not raise an invalid exception. Unless explicitly stated otherwise, MPFR sets the NaN flag whenever a NaN is generated, even when a NaN is propagated (e.g. in NaN + NaN), as if all NaNs were signaling.
- An invalid exception in C corresponds to either a NaN exception or a range error in MPFR.

5 MPFR Interface

The floating-point functions expect arguments of type `mpfr_t`.

The MPFR floating-point functions have an interface that is similar to the GNU MP integer functions. The function prefix for floating-point operations is `mpfr_`.

There is one significant characteristic of floating-point numbers that has motivated a difference between this function class and other GNU MP function classes: the inherent inexactness of floating-point arithmetic. The user has to specify the precision for each variable. A computation that assigns a variable will take place with the precision of the assigned variable; the cost of that computation should not depend from the precision of variables used as input (on average).

The semantics of a calculation in MPFR is specified as follows: Compute the requested operation exactly (with "infinite accuracy"), and round the result to the precision of the destination variable, with the given rounding mode. The MPFR floating-point functions are intended to be a smooth extension of the IEEE 754-1985 arithmetic. The results obtained on one computer should not differ from the results obtained on a computer with a different word size.

MPFR does not keep track of the accuracy of a computation. This is left to the user or to a higher layer. As a consequence, if two variables are used to store only a few significant bits, and their product is stored in a variable with large precision, then MPFR will still compute the result with full precision.

The value of the standard C macro `errno` may be set to non-zero by any MPFR function or macro, whether or not there is an error.

5.1 Initialization Functions

An `mpfr_t` object must be initialized before storing the first value in it. The functions `mpfr_init` and `mpfr_init2` are used for that purpose.

`void mpfr_init2` (*mpfr_t* `x`, *mp_prec_t* `prec`) [Function]

Initialize *x*, set its precision to be **exactly** *prec* bits and its value to NaN. (Warning: the corresponding `mpf` functions initialize to zero instead.)

Normally, a variable should be initialized once only or at least be cleared, using `mpfr_clear`, between initializations. To change the precision of a variable which has already been initialized, use `mpfr_set_prec`. The precision *prec* must be an integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX` (otherwise the behavior is undefined).

`void mpfr_clear` (*mpfr_t* `x`) [Function]

Free the space occupied by *x*. Make sure to call this function for all `mpfr_t` variables when you are done with them.

`void mpfr_init` (*mpfr_t* `x`) [Function]

Initialize *x* and set its value to NaN.

Normally, a variable should be initialized once only or at least be cleared, using `mpfr_clear`, between initializations. The precision of *x* is the default precision, which can be changed by a call to `mpfr_set_default_prec`.

`void mpfr_set_default_prec` (*mp_prec_t* `prec`) [Function]

Set the default precision to be **exactly** *prec* bits. The precision of a variable means the number of bits used to store its significand. All subsequent calls to `mpfr_init` will use this precision, but previously initialized variables are unaffected. This default precision is set to 53 bits initially. The precision can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`.

`mp_prec_t mpfr_get_default_prec` (*void*) [Function]
Return the default MPFR precision in bits.

Here is an example on how to initialize floating-point variables:

```
{
  mpfr_t x, y;
  mpfr_init (x);                /* use default precision */
  mpfr_init2 (y, 256);          /* precision exactly 256 bits */
  ...
  /* When the program is about to exit, do ... */
  mpfr_clear (x);
  mpfr_clear (y);
}
```

The following functions are useful for changing the precision during a calculation. A typical use would be for adjusting the precision gradually in iterative algorithms like Newton-Raphson, making the computation precision closely match the actual accurate part of the numbers.

`void mpfr_set_prec` (*mpfr_t* **x**, *mp_prec_t* **prec**) [Function]
Reset the precision of *x* to be **exactly** *prec* bits, and set its value to NaN. The previous value stored in *x* is lost. It is equivalent to a call to `mpfr_clear(x)` followed by a call to `mpfr_init2(x, prec)`, but more efficient as no allocation is done in case the current allocated space for the significand of *x* is enough. The precision *prec* can be any integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX`.

In case you want to keep the previous value stored in *x*, use `mpfr_prec_round` instead.

`mp_prec_t mpfr_get_prec` (*mpfr_t* **x**) [Function]
Return the precision actually used for assignments of *x*, i.e. the number of bits used to store its significand.

5.2 Assignment Functions

These functions assign new values to already initialized floats (see Section 5.1 [Initialization Functions], page 10). When using any functions using `intmax_t`, you must include `<stdint.h>` or `<inttypes.h>` before '`mpfr.h`', to allow '`mpfr.h`' to define prototypes for these functions.

`int mpfr_set` (*mpfr_t* **rop**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
`int mpfr_set_ui` (*mpfr_t* **rop**, *unsigned long int* **op**, *mp_rnd_t* **rnd**) [Function]
`int mpfr_set_si` (*mpfr_t* **rop**, *long int* **op**, *mp_rnd_t* **rnd**) [Function]
`int mpfr_set_uj` (*mpfr_t* **rop**, *uintmax_t* **op**, *mp_rnd_t* **rnd**) [Function]
`int mpfr_set_sj` (*mpfr_t* **rop**, *intmax_t* **op**, *mp_rnd_t* **rnd**) [Function]
`int mpfr_set_d` (*mpfr_t* **rop**, *double* **op**, *mp_rnd_t* **rnd**) [Function]
`int mpfr_set_ld` (*mpfr_t* **rop**, *long double* **op**, *mp_rnd_t* **rnd**) [Function]
`int mpfr_set_decimal64` (*mpfr_t* **rop**, *_Decimal64* **op**, *mp_rnd_t* **rnd**) [Function]
`int mpfr_set_z` (*mpfr_t* **rop**, *mpz_t* **op**, *mp_rnd_t* **rnd**) [Function]
`int mpfr_set_q` (*mpfr_t* **rop**, *mpq_t* **op**, *mp_rnd_t* **rnd**) [Function]
`int mpfr_set_f` (*mpfr_t* **rop**, *mpf_t* **op**, *mp_rnd_t* **rnd**) [Function]
Set the value of *rop* from *op*, rounded toward the given direction *rnd*. Note that the input 0 is converted to +0 by `mpfr_set_ui`, `mpfr_set_si`, `mpfr_set_sj`, `mpfr_set_uj`, `mpfr_set_`

z, mpfr_set_q and mpfr_set_f, regardless of the rounding mode. If the system doesn't support the IEEE-754 standard, mpfr_set_d, mpfr_set_ld and mpfr_set_decimal64 might not preserve the signed zeros. The mpfr_set_decimal64 function is built only with the configure option '--enable-decimal-float', which also requires '--with-gmp-build', and when the compiler or system provides the '_Decimal64' data type (GCC version 4.2.0 is known to support this data type, but only when configured with '--enable-decimal-float' too). mpfr_set_q might not be able to work if the numerator (or the denominator) can not be representable as a mpfr_t.

Note: If you want to store a floating-point constant to a mpfr_t, you should use mpfr_set_str (or one of the MPFR constant functions, such as mpfr_const_pi for π) instead of mpfr_set_d, mpfr_set_ld or mpfr_set_decimal64. Otherwise the floating-point constant will be first converted into a reduced-precision (e.g., 53-bit) binary number before MPFR can work with it.

int mpfr_set_ui_2exp (*mpfr_t* rop, *unsigned long int* op, *mp_exp_t* e, *mp_rnd_t* rnd) [Function]
int mpfr_set_si_2exp (*mpfr_t* rop, *long int* op, *mp_exp_t* e, *mp_rnd_t* rnd) [Function]
int mpfr_set_uj_2exp (*mpfr_t* rop, *uintmax_t* op, *intmax_t* e, *mp_rnd_t* rnd) [Function]
int mpfr_set_sj_2exp (*mpfr_t* rop, *intmax_t* op, *intmax_t* e, *mp_rnd_t* rnd) [Function]

Set the value of *rop* from $op \times 2^e$, rounded toward the given direction *rnd*. Note that the input 0 is converted to +0.

int mpfr_set_str (*mpfr_t* rop, *const char* *s, *int* base, *mp_rnd_t* rnd) [Function]

Set *rop* to the value of the whole string *s* in base *base*, rounded in the direction *rnd*. See the documentation of mpfr_strtofr for a detailed description of the valid string formats. This function returns 0 if the entire string up to the final null character is a valid number in base *base*; otherwise it returns -1, and *rop* may have changed.

int mpfr_strtofr (*mpfr_t* rop, *const char* *nptr, *char* **endptr, *int* base, *mp_rnd_t* rnd) [Function]

Read a floating-point number from a string *nptr* in base *base*, rounded in the direction *rnd*; *base* must be either 0 (to detect the base, as described below) or a number from 2 to 36 (otherwise the behavior is undefined). If *nptr* starts with valid data, the result is stored in *rop* and *endptr points to the character just after the valid data (if *endptr* is not a null pointer); otherwise *rop* is set to zero and the value of *nptr* is stored in the location referenced by *endptr* (if *endptr* is not a null pointer). The usual ternary value is returned.

Parsing follows the standard C strtod function with some extensions. Case is ignored. After optional leading whitespace, one has a subject sequence consisting of an optional sign (+ or -), and either numeric data or special data. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-whitespace character, that is of the expected form.

The form of numeric data is a non-empty sequence of significand digits with an optional decimal point, and an optional exponent consisting of an exponent prefix followed by an optional sign and a non-empty sequence of decimal digits. A significand digit is either a decimal digit or a Latin letter (62 possible characters), with a = 10, b = 11, ..., z = 36; its value must be strictly less than the base. The decimal point can be either the one defined by the current locale or the period (the first one is accepted for consistency with the C standard and the practice, the second one is accepted to allow the programmer to provide MPFR numbers from strings in a way that does not depend on the current locale). The exponent prefix can be `e` or `E` for bases up to 10, or `@` in any base; it indicates a multiplication by a power of the base. In bases 2 and 16, the exponent prefix can also be `p` or `P`, in which case it introduces a binary exponent: it indicates a multiplication by a power of 2 (there is a difference only for base 16). The value of an exponent is always written in base 10. In base 2, the significand can start with `0b` or `0B`, and in base 16, it can start with `0x` or `0X`.

If the argument *base* is 0, then the base is automatically detected as follows. If the significand starts with `0b` or `0B`, base 2 is assumed. If the significand starts with `0x` or `0X`, base 16 is assumed. Otherwise base 10 is assumed.

Note: The exponent must contain at least a digit. Otherwise the possible exponent prefix and sign are not part of the number (which ends with the significand). Similarly, if `0b`, `0B`, `0x` or `0X` is not followed by a binary/hexadecimal digit, then the subject sequence stops at the character `0`.

Special data (for infinities and NaN) can be `@inf@` or `@nan@(n-char-sequence)`, and if *base* ≤ 16, it can also be `infinity`, `inf`, `nan` or `nan(n-char-sequence)`, all case insensitive. A `n-char-sequence` is a non-empty string containing only digits, Latin letters and the underscore (0, 1, 2, . . . , 9, a, b, . . . , z, A, B, . . . , Z, _). Note: one has an optional sign for all data, even NaN.

`void mpfr_set_inf` (*mpfr_t* `x`, *int* `sign`) [Function]
`void mpfr_set_nan` (*mpfr_t* `x`) [Function]
Set the variable *x* to infinity or NaN (Not-a-Number) respectively. In `mpfr_set_inf`, *x* is set to plus infinity iff *sign* is nonnegative.

`void mpfr_swap` (*mpfr_t* `x`, *mpfr_t* `y`) [Function]
Swap the values *x* and *y* efficiently. Warning: the precisions are exchanged too; in case the precisions are different, `mpfr_swap` is thus not equivalent to three `mpfr_set` calls using a third auxiliary variable.

5.3 Combined Initialization and Assignment Functions

`int mpfr_init_set` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Macro]
`int mpfr_init_set_ui` (*mpfr_t* `rop`, *unsigned long int* `op`, *mp_rnd_t* `rnd`) [Macro]
`int mpfr_init_set_si` (*mpfr_t* `rop`, *signed long int* `op`, *mp_rnd_t* `rnd`) [Macro]
`int mpfr_init_set_d` (*mpfr_t* `rop`, *double* `op`, *mp_rnd_t* `rnd`) [Macro]
`int mpfr_init_set_ld` (*mpfr_t* `rop`, *long double* `op`, *mp_rnd_t* `rnd`) [Macro]
`int mpfr_init_set_z` (*mpfr_t* `rop`, *mpz_t* `op`, *mp_rnd_t* `rnd`) [Macro]
`int mpfr_init_set_q` (*mpfr_t* `rop`, *mpq_t* `op`, *mp_rnd_t* `rnd`) [Macro]
`int mpfr_init_set_f` (*mpfr_t* `rop`, *mpf_t* `op`, *mp_rnd_t* `rnd`) [Macro]
Initialize *rop* and set its value from *op*, rounded in the direction *rnd*. The precision of *rop* will be taken from the active default precision, as set by `mpfr_set_default_prec`.

`int mpfr_init_set_str` (*mpfr_t* `x`, *const char* `*s`, *int* `base`, *mp_rnd_t* `rnd`) [Function]
Initialize *x* and set its value from the string *s* in base *base*, rounded in the direction *rnd*. See `mpfr_set_str`.

5.4 Conversion Functions

`double mpfr_get_d` (*mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`long double mpfr_get_ld` (*mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

`_Decimal64 mpfr_get_decimal64` (*mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]

Convert *op* to a `double` (respectively `_Decimal64` or `long double`), using the rounding mode *rnd*. If *op* is NaN, some fixed NaN (either quiet or signaling) or the result of 0.0/0.0 is returned. If *op* is ±Inf, an infinity of the same sign or the result of ±1.0/0.0 is returned. If *op* is zero, these functions return a zero, trying to preserve its sign, if possible. The `mpfr_get_decimal64` function is built only under some conditions: see the documentation of `mpfr_set_decimal64`.

`double mpfr_get_d_2exp` (*long* ***exp**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
`long double mpfr_get_ld_2exp` (*long* ***exp**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]

Return d and set *exp* such that $0.5 \le |d| < 1$ and $d \times 2^{exp}$ equals *op* rounded to double (resp. long double) precision, using the given rounding mode. If *op* is zero, then a zero of the same sign (or an unsigned zero, if the implementation does not have signed zeros) is returned, and *exp* is set to 0. If *op* is NaN or an infinity, then the corresponding double precision (resp. long-double precision) value is returned, and *exp* is undefined.

`long mpfr_get_si` (*mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
`unsigned long mpfr_get_ui` (*mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
`intmax_t mpfr_get_sj` (*mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
`uintmax_t mpfr_get_uj` (*mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]

Convert *op* to a `long`, an `unsigned long`, an `intmax_t` or an `uintmax_t` (respectively) after rounding it with respect to *rnd*. If *op* is NaN, the result is undefined. If *op* is too big for the return type, it returns the maximum or the minimum of the corresponding C type, depending on the direction of the overflow. The flag erange is set too. See also `mpfr_fits_slong_p`, `mpfr_fits_ulong_p`, `mpfr_fits_intmax_p` and `mpfr_fits_uintmax_p`.

`mp_exp_t mpfr_get_z_exp` (*mpz_t* **rop**, *mpfr_t* **op**) [Function]

Put the scaled significand of *op* (regarded as an integer, with the precision of *op*) into *rop*, and return the exponent *exp* (which may be outside the current exponent range) such that *op* exactly equals $rop \times 2^{\mathrm{exp}}$. If the exponent is not representable in the `mp_exp_t` type, the behavior is undefined.

`void mpfr_get_z` (*mpz_t* **rop**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]

Convert *op* to a `mpz_t`, after rounding it with respect to *rnd*. If *op* is NaN or Inf, the result is undefined.

`int mpfr_get_f` (*mpf_t* **rop**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]

Convert *op* to a `mpf_t`, after rounding it with respect to *rnd*. Return zero iff no error occurred, in particular a non-zero value is returned if *op* is NaN or Inf, which do not exist in `mpf`.

`char * mpfr_get_str` (*char* ***str**, *mp_exp_t* ***expptr**, *int* **b**, *size_t* **n**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]

Convert *op* to a string of digits in base *b*, with rounding in the direction *rnd*, where *n* is either zero (see below) or the number of significant digits; in the latter case, *n* must be greater or equal to 2. The base may vary from 2 to 36.

The generated string is a fraction, with an implicit radix point immediately to the left of the first digit. For example, the number −3.1416 would be returned as `"−31416"` in the string and 1 written at *expptr*. If *rnd* is to nearest, and *op* is exactly in the middle of two possible outputs, the one with an even last digit is chosen (for an odd base, this may not correspond to an even significand).

If *n* is zero, the number of digits of the significand is chosen large enough so that re-reading the printed value with the same precision, assuming both output and input use rounding to nearest, will recover the original value of *op*. More precisely, in most cases, the chosen precision of *str* is the minimal precision depending on *n* and *b* only that satisfies the above property, i.e., $m = 1 + \lceil n \frac{\log 2}{\log b} \rceil$, but in some very rare cases, it might be $m + 1$.

If *str* is a null pointer, space for the significand is allocated using the current allocation function, and a pointer to the string is returned. To free the returned string, you must use `mpfr_free_str`.

If *str* is not a null pointer, it should point to a block of storage large enough for the significand, i.e., at least `max(n + 2, 7)`. The extra two bytes are for a possible minus sign, and for the terminating null character.

If the input number is an ordinary number, the exponent is written through the pointer *expptr* (the current minimal exponent for 0).

A pointer to the string is returned, unless there is an error, in which case a null pointer is returned.

`void mpfr_free_str` (*char* `*str`) [Function]

Free a string allocated by `mpfr_get_str` using the current unallocation function (preliminary interface). The block is assumed to be `strlen(str)+1` bytes. For more information about how it is done: see section "Custom Allocation" in *GNU MP*.

`int mpfr_fits_ulong_p` (*mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_fits_slong_p` (*mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_fits_uint_p` (*mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_fits_sint_p` (*mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_fits_ushort_p` (*mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_fits_sshort_p` (*mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_fits_intmax_p` (*mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_fits_uintmax_p` (*mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Return non-zero if *op* would fit in the respective C data type, when rounded to an integer in the direction *rnd*.

5.5 Basic Arithmetic Functions

`int mpfr_add` (*mpfr_t* `rop`, *mpfr_t* `op1`, *mpfr_t* `op2`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_add_ui` (*mpfr_t* `rop`, *mpfr_t* `op1`, *unsigned long int* `op2`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_add_si` (*mpfr_t* `rop`, *mpfr_t* `op1`, *long int* `op2`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_add_z` (*mpfr_t* `rop`, *mpfr_t* `op1`, *mpz_t* `op2`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_add_q` (*mpfr_t* `rop`, *mpfr_t* `op1`, *mpq_t* `op2`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to $op1 + op2$ rounded in the direction *rnd*. For types having no signed zero, it is considered unsigned (i.e. $(+0) + 0 = (+0)$ and $(-0) + 0 = (-0)$).

`int mpfr_sub` (*mpfr_t* `rop`, *mpfr_t* `op1`, *mpfr_t* `op2`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_ui_sub` (*mpfr_t* `rop`, *unsigned long int* `op1`, *mpfr_t* `op2`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_sub_ui` (*mpfr_t* `rop`, *mpfr_t* `op1`, *unsigned long int* `op2`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_si_sub` (*mpfr_t* `rop`, *long int* `op1`, *mpfr_t* `op2`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_sub_si` (*mpfr_t* `rop`, *mpfr_t* `op1`, *long int* `op2`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_sub_z` (*mpfr_t* `rop`, *mpfr_t* `op1`, *mpz_t* `op2`, *mp_rnd_t* `rnd`) [Function]

`int mpfr_sub_q (mpfr_t rop, mpfr_t op1, mpq_t op2, mp_rnd_t rnd)` [Function]

Set *rop* to *op1* − *op2* rounded in the direction *rnd*. For types having no signed zero, it is considered unsigned (i.e. (+0) − 0 = (+0), (−0) − 0 = (−0), 0 − (+0) = (−0) and 0 − (−0) = (+0)).

`int mpfr_mul (mpfr_t rop, mpfr_t op1, mpfr_t op2, mp_rnd_t rnd)` [Function]
`int mpfr_mul_ui (mpfr_t rop, mpfr_t op1, unsigned long int op2, mp_rnd_t rnd)` [Function]
`int mpfr_mul_si (mpfr_t rop, mpfr_t op1, long int op2, mp_rnd_t rnd)` [Function]
`int mpfr_mul_z (mpfr_t rop, mpfr_t op1, mpz_t op2, mp_rnd_t rnd)` [Function]
`int mpfr_mul_q (mpfr_t rop, mpfr_t op1, mpq_t op2, mp_rnd_t rnd)` [Function]

Set *rop* to $op1 \times op2$ rounded in the direction *rnd*. When a result is zero, its sign is the product of the signs of the operands (for types having no signed zero, it is considered positive).

`int mpfr_sqr (mpfr_t rop, mpfr_t op, mp_rnd_t rnd)` [Function]

Set *rop* to op^2 rounded in the direction *rnd*.

`int mpfr_div (mpfr_t rop, mpfr_t op1, mpfr_t op2, mp_rnd_t rnd)` [Function]
`int mpfr_ui_div (mpfr_t rop, unsigned long int op1, mpfr_t op2, mp_rnd_t rnd)` [Function]
`int mpfr_div_ui (mpfr_t rop, mpfr_t op1, unsigned long int op2, mp_rnd_t rnd)` [Function]
`int mpfr_si_div (mpfr_t rop, long int op1, mpfr_t op2, mp_rnd_t rnd)` [Function]
`int mpfr_div_si (mpfr_t rop, mpfr_t op1, long int op2, mp_rnd_t rnd)` [Function]
`int mpfr_div_z (mpfr_t rop, mpfr_t op1, mpz_t op2, mp_rnd_t rnd)` [Function]
`int mpfr_div_q (mpfr_t rop, mpfr_t op1, mpq_t op2, mp_rnd_t rnd)` [Function]

Set *rop* to $op1/op2$ rounded in the direction *rnd*. When a result is zero, its sign is the product of the signs of the operands (for types having no signed zero, it is considered positive).

`int mpfr_sqrt (mpfr_t rop, mpfr_t op, mp_rnd_t rnd)` [Function]
`int mpfr_sqrt_ui (mpfr_t rop, unsigned long int op, mp_rnd_t rnd)` [Function]

Set *rop* to $\sqrt{op}$ rounded in the direction *rnd*. Return −0 if *op* is −0 (to be consistent with the IEEE 754-1985 standard). Set *rop* to NaN if *op* is negative.

`int mpfr_cbrt (mpfr_t rop, mpfr_t op, mp_rnd_t rnd)` [Function]
`int mpfr_root (mpfr_t rop, mpfr_t op, unsigned long int k, mp_rnd_t rnd)` [Function]

Set *rop* to the cubic root (resp. the kth root) of *op* rounded in the direction *rnd*. An odd (resp. even) root of a negative number (including −Inf) returns a negative number (resp. NaN). The kth root of −0 is defined to be −0, whatever the parity of k.

`int mpfr_pow (mpfr_t rop, mpfr_t op1, mpfr_t op2, mp_rnd_t rnd)` [Function]
`int mpfr_pow_ui (mpfr_t rop, mpfr_t op1, unsigned long int op2, mp_rnd_t rnd)` [Function]
`int mpfr_pow_si (mpfr_t rop, mpfr_t op1, long int op2, mp_rnd_t rnd)` [Function]
`int mpfr_pow_z (mpfr_t rop, mpfr_t op1, mpz_t op2, mp_rnd_t rnd)` [Function]
`int mpfr_ui_pow_ui (mpfr_t rop, unsigned long int op1, unsigned long int op2, mp_rnd_t rnd)` [Function]
`int mpfr_ui_pow (mpfr_t rop, unsigned long int op1, mpfr_t op2, mp_rnd_t rnd)` [Function]

Set *rop* to $op1^{op2}$, rounded in the direction *rnd*. Special values are currently handled as described in the ISO C99 standard for the `pow` function (note this may change in future versions):

- `pow(±0, y)` returns plus or minus infinity for y a negative odd integer.
- `pow(±0, y)` returns plus infinity for y negative and not an odd integer.
- `pow(±0, y)` returns plus or minus zero for y a positive odd integer.
- `pow(±0, y)` returns plus zero for y positive and not an odd integer.
- `pow(-1, ±Inf)` returns 1.
- `pow(+1, y)` returns 1 for any y, even a NaN.
- `pow(x, y)` returns NaN for finite negative x and finite non-integer y.
- `pow(x, -Inf)` returns plus infinity for $0 < |x| < 1$, and plus zero for $|x| > 1$.
- `pow(x, +Inf)` returns plus zero for $0 < |x| < 1$, and plus infinity for $|x| > 1$.
- `pow(-Inf, y)` returns minus zero for y a negative odd integer.
- `pow(-Inf, y)` returns plus zero for y negative and not an odd integer.
- `pow(-Inf, y)` returns minus infinity for y a positive odd integer.
- `pow(-Inf, y)` returns plus infinity for y positive and not an odd integer.
- `pow(+Inf, y)` returns plus zero for y negative, and plus infinity for y positive.

`int mpfr_neg` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to $-op$ rounded in the direction *rnd*. Just changes the sign if *rop* and *op* are the same variable.

`int mpfr_abs` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the absolute value of *op*, rounded in the direction *rnd*. Just changes the sign if *rop* and *op* are the same variable.

`int mpfr_dim` (*mpfr_t* `rop`, *mpfr_t* `op1`, *mpfr_t* `op2`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the positive difference of *op1* and *op2*, i.e., $op1 - op2$ rounded in the direction *rnd* if $op1 > op2$, and +0 otherwise. Returns NaN when *op1* or *op2* is NaN.

`int mpfr_mul_2ui` (*mpfr_t* `rop`, *mpfr_t* `op1`, *unsigned long int* `op2`, *mp_rnd_t* `rnd`) [Function]

`int mpfr_mul_2si` (*mpfr_t* `rop`, *mpfr_t* `op1`, *long int* `op2`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to $op1 \times 2^{op2}$ rounded in the direction *rnd*. Just increases the exponent by *op2* when *rop* and *op1* are identical.

`int mpfr_div_2ui` (*mpfr_t* `rop`, *mpfr_t* `op1`, *unsigned long int* `op2`, *mp_rnd_t* `rnd`) [Function]

`int mpfr_div_2si` (*mpfr_t* `rop`, *mpfr_t* `op1`, *long int* `op2`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to $op1/2^{op2}$ rounded in the direction *rnd*. Just decreases the exponent by *op2* when *rop* and *op1* are identical.

5.6 Comparison Functions

`int mpfr_cmp` (*mpfr_t* `op1`, *mpfr_t* `op2`) [Function]

`int mpfr_cmp_ui` (*mpfr_t* `op1`, *unsigned long int* `op2`) [Function]

`int mpfr_cmp_si` (*mpfr_t* `op1`, *signed long int* `op2`) [Function]

`int mpfr_cmp_d` (*mpfr_t* `op1`, *double* `op2`) [Function]

`int mpfr_cmp_ld` (*mpfr_t* `op1`, *long double* `op2`) [Function]

`int mpfr_cmp_z` (*mpfr_t* `op1`, *mpz_t* `op2`) [Function]

`int mpfr_cmp_q` (*mpfr_t* `op1`, *mpq_t* `op2`) [Function] `int mpfr_cmp_f (mpfr_t op1, mpf_t op2)` [Function]

Compare *op1* and *op2*. Return a positive value if $op1 > op2$, zero if $op1 = op2$, and a negative value if $op1 < op2$. Both *op1* and *op2* are considered to their full own precision, which may differ. If one of the operands is NaN, set the erange flag and return zero.

Note: These functions may be useful to distinguish the three possible cases. If you need to distinguish two cases only, it is recommended to use the predicate functions (e.g., `mpfr_equal_p` for the equality) described below; they behave like the IEEE-754 comparisons, in particular when one or both arguments are NaN. But only floating-point numbers can be compared (you may need to do a conversion first).

`int mpfr_cmp_ui_2exp (mpfr_t op1, unsigned long int op2, mp_exp_t e)` [Function]
`int mpfr_cmp_si_2exp (mpfr_t op1, long int op2, mp_exp_t e)` [Function]

Compare *op1* and $op2 \times 2^e$. Similar as above.

`int mpfr_cmpabs (mpfr_t op1, mpfr_t op2)` [Function]

Compare $|op1|$ and $|op2|$. Return a positive value if $|op1| > |op2|$, zero if $|op1| = |op2|$, and a negative value if $|op1| < |op2|$. If one of the operands is NaN, set the erange flag and return zero.

`int mpfr_nan_p (mpfr_t op)` [Function]
`int mpfr_inf_p (mpfr_t op)` [Function]
`int mpfr_number_p (mpfr_t op)` [Function]
`int mpfr_zero_p (mpfr_t op)` [Function]

Return non-zero if *op* is respectively NaN, an infinity, an ordinary number (i.e. neither NaN nor an infinity) or zero. Return zero otherwise.

`int mpfr_sgn (mpfr_t op)` [Macro]

Return a positive value if $op > 0$, zero if $op = 0$, and a negative value if $op < 0$. If the operand is NaN, set the erange flag and return zero.

`int mpfr_greater_p (mpfr_t op1, mpfr_t op2)` [Function]

Return non-zero if $op1 > op2$, zero otherwise.

`int mpfr_greaterequal_p (mpfr_t op1, mpfr_t op2)` [Function]

Return non-zero if $op1 \geq op2$, zero otherwise.

`int mpfr_less_p (mpfr_t op1, mpfr_t op2)` [Function]

Return non-zero if $op1 < op2$, zero otherwise.

`int mpfr_lessequal_p (mpfr_t op1, mpfr_t op2)` [Function]

Return non-zero if $op1 \leq op2$, zero otherwise.

`int mpfr_lessgreater_p (mpfr_t op1, mpfr_t op2)` [Function]

Return non-zero if $op1 < op2$ or $op1 > op2$ (i.e. neither *op1*, nor *op2* is NaN, and $op1 \neq op2$), zero otherwise (i.e. *op1* and/or *op2* are NaN, or $op1 = op2$).

`int mpfr_equal_p (mpfr_t op1, mpfr_t op2)` [Function]

Return non-zero if $op1 = op2$, zero otherwise (i.e. *op1* and/or *op2* are NaN, or $op1 \neq op2$).

`int mpfr_unordered_p (mpfr_t op1, mpfr_t op2)` [Function]

Return non-zero if *op1* or *op2* is a NaN (i.e. they cannot be compared), zero otherwise.

5.7 Special Functions

All those functions, except explicitly stated, return zero for an exact return value, a positive value for a return value larger than the exact result, and a negative value otherwise.

Important note: in some domains, computing special functions (either with correct or incorrect rounding) is expensive, even for small precision, for example the trigonometric and Bessel functions for large argument.

`int mpfr_log` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_log2` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_log10` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the natural logarithm of *op*, $\log_2 op$ or $\log_{10} op$, respectively, rounded in the direction *rnd*. Return $-$Inf if *op* is -0 (i.e. the sign of the zero has no influence on the result).

`int mpfr_exp` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_exp2` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_exp10` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the exponential of *op*, to 2^{op} or to 10^{op}, respectively, rounded in the direction *rnd*.

`int mpfr_cos` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_sin` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_tan` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the cosine of *op*, sine of *op*, tangent of *op*, rounded in the direction *rnd*.

`int mpfr_sec` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_csc` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_cot` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the secant of *op*, cosecant of *op*, cotangent of *op*, rounded in the direction *rnd*.

`int mpfr_sin_cos` (*mpfr_t* `sop`, *mpfr_t* `cop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set simultaneously *sop* to the sine of *op* and *cop* to the cosine of *op*, rounded in the direction *rnd* with the corresponding precisions of *sop* and *cop*, which must be different variables. Return 0 iff both results are exact.

`int mpfr_acos` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_asin` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_atan` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the arc-cosine, arc-sine or arc-tangent of *op*, rounded in the direction *rnd*. Note that since `acos(-1)` returns the floating-point number closest to π according to the given rounding mode, this number might not be in the output range $0 \le rop < \pi$ of the arc-cosine function; still, the result lies in the image of the output range by the rounding function. The same holds for `asin(-1)`, `asin(1)`, `atan(-Inf)`, `atan(+Inf)`.

`int mpfr_atan2` (*mpfr_t* `rop`, *mpfr_t* `y`, *mpfr_t* `x`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the arc-tangent2 of *y* and *x*, rounded in the direction *rnd*: if `x > 0`, `atan2(y, x) = atan (y/x)`; if `x < 0`, `atan2(y, x) = sign(y)*(Pi - atan (|y/x|))`. As for `atan`, in case the exact mathematical result is $+\pi$ or $-\pi$, its rounded result might be outside the function output range.

`atan2(y, 0)` does not raise any floating-point exception. Special values are currently handled as described in the ISO C99 standard for the `atan2` function (note this may change in future versions):

- `atan2(+0, -0)` returns $+\pi$.
- `atan2(-0, -0)` returns $-\pi$.
- `atan2(+0, +0)` returns +0.
- `atan2(-0, +0)` returns -0.
- `atan2(+0, x)` returns $+\pi$ for $x < 0$.
- `atan2(-0, x)` returns $-\pi$ for $x < 0$.
- `atan2(+0, x)` returns +0 for $x > 0$.
- `atan2(-0, x)` returns -0 for $x > 0$.
- `atan2(y, 0)` returns $-\pi/2$ for $y < 0$.
- `atan2(y, 0)` returns $+\pi/2$ for $y > 0$.
- `atan2(+Inf, -Inf)` returns $+3 * \pi/4$.
- `atan2(-Inf, -Inf)` returns $-3 * \pi/4$.
- `atan2(+Inf, +Inf)` returns $+\pi/4$.
- `atan2(-Inf, +Inf)` returns $-\pi/4$.
- `atan2(+Inf, x)` returns $+\pi/2$ for finite x.
- `atan2(-Inf, x)` returns $-\pi/2$ for finite x.
- `atan2(y, -Inf)` returns $+\pi$ for finite $y > 0$.
- `atan2(y, -Inf)` returns $-\pi$ for finite $y < 0$.
- `atan2(y, +Inf)` returns +0 for finite $y > 0$.
- `atan2(y, +Inf)` returns -0 for finite $y < 0$.

`int mpfr_cosh` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_sinh` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_tanh` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the hyperbolic cosine, sine or tangent of *op*, rounded in the direction *rnd*.

`int mpfr_sech` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_csch` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_coth` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the hyperbolic secant of *op*, cosecant of *op*, cotangent of *op*, rounded in the direction *rnd*.

`int mpfr_acosh` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_asinh` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_atanh` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the inverse hyperbolic cosine, sine or tangent of *op*, rounded in the direction *rnd*.

`int mpfr_fac_ui` (*mpfr_t* `rop`, *unsigned long int* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the factorial of the `unsigned long int` *op*, rounded in the direction *rnd*.

`int mpfr_log1p` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the logarithm of one plus *op*, rounded in the direction *rnd*.

`int mpfr_expm1` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the exponential of *op* minus one, rounded in the direction *rnd*.

`int mpfr_eint` (*mpfr_t* `y`, *mpfr_t* `x`, *mp_rnd_t* `rnd`) [Function]

Set *y* to the exponential integral of *x*, rounded in the direction *rnd*. For positive *x*, the exponential integral is the sum of Euler's constant, of the logarithm of *x*, and of the sum for k from 1 to infinity of $x^k/k/k!$. For negative *x*, the returned value is NaN.

int mpfr_gamma (*mpfr_t* **rop**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
Set *rop* to the value of the Gamma function on *op*, rounded in the direction *rnd*. When *op* is a negative integer, NaN is returned.

int mpfr_lngamma (*mpfr_t* **rop**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
Set *rop* to the value of the logarithm of the Gamma function on *op*, rounded in the direction *rnd*. When -2k-1 $\leq x \leq$ -2k, k being a non-negative integer, NaN is returned. See also mpfr_lgamma.

int mpfr_lgamma (*mpfr_t* **rop**, *int* ***signp**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
Set *rop* to the value of the logarithm of the absolute value of the Gamma function on *op*, rounded in the direction *rnd*. The sign (1 or −1) of Gamma(*op*) is returned in the object pointed to by *signp*. When *op* is an infinity or a non-positive integer, +Inf is returned. When *op* is NaN, −Inf or a negative integer, **signp* is undefined, and when *op* is ±0, **signp* is the sign of the zero.

int mpfr_zeta (*mpfr_t* **rop**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
int mpfr_zeta_ui (*mpfr_t* **rop**, *unsigned long* **op**, *mp_rnd_t* **rnd**) [Function]
Set *rop* to the value of the Riemann Zeta function on *op*, rounded in the direction *rnd*.

int mpfr_erf (*mpfr_t* **rop**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
Set *rop* to the value of the error function on *op*, rounded in the direction *rnd*.

int mpfr_erfc (*mpfr_t* **rop**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
Set *rop* to the value of the complementary error function on *op*, rounded in the direction *rnd*.

int mpfr_j0 (*mpfr_t* **rop**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
int mpfr_j1 (*mpfr_t* **rop**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
int mpfr_jn (*mpfr_t* **rop**, *long* **n**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
Set *rop* to the value of the first order Bessel function of order 0, 1 and *n* on *op*, rounded in the direction *rnd*. When *op* is NaN, *rop* is always set to NaN. When *op* is plus or minus Infinity, *rop* is set to +0. When *op* is zero, and *n* is not zero, *rop* is +0 or −0 depending on the parity and sign of *n*, and the sign of *op*.

int mpfr_y0 (*mpfr_t* **rop**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
int mpfr_y1 (*mpfr_t* **rop**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
int mpfr_yn (*mpfr_t* **rop**, *long* **n**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]
Set *rop* to the value of the second order Bessel function of order 0, 1 and *n* on *op*, rounded in the direction *rnd*. When *op* is NaN or negative, *rop* is always set to NaN. When *op* is +Inf, *rop* is +0. When *op* is zero, *rop* is +Inf or −Inf depending on the parity and sign of *n*.

int mpfr_fma (*mpfr_t* **rop**, *mpfr_t* **op1**, *mpfr_t* **op2**, *mpfr_t* **op3**, *mp_rnd_t* **rnd**) [Function]
Set *rop* to $op1 \times op2 + op3$, rounded in the direction *rnd*.

int mpfr_fms (*mpfr_t* **rop**, *mpfr_t* **op1**, *mpfr_t* **op2**, *mpfr_t* **op3**, *mp_rnd_t* **rnd**) [Function]
Set *rop* to $op1 \times op2 - op3$, rounded in the direction *rnd*.

int mpfr_agm (*mpfr_t* **rop**, *mpfr_t* **op1**, *mpfr_t* **op2**, *mp_rnd_t* **rnd**) [Function]
Set *rop* to the arithmetic-geometric mean of *op1* and *op2*, rounded in the direction *rnd*. The arithmetic-geometric mean is the common limit of the sequences u[n] and v[n], where

u[0]=*op1*, v[0]=*op2*, u[n+1] is the arithmetic mean of u[n] and v[n], and v[n+1] is the geometric mean of u[n] and v[n]. If any operand is negative, the return value is NaN.

`int mpfr_hypot` (*mpfr_t* **rop**, *mpfr_t* **x**, *mpfr_t* **y**, *mp_rnd_t* **rnd**) [Function]

Set *rop* to the Euclidean norm of *x* and *y*, i.e. $\sqrt{x^2+y^2}$, rounded in the direction *rnd*. Special values are currently handled as described in Section F.9.4.3 of the ISO C99 standard, for the `hypot` function (note this may change in future versions): If *x* or *y* is an infinity, then plus infinity is returned in *rop*, even if the other number is NaN.

`int mpfr_const_log2` (*mpfr_t* **rop**, *mp_rnd_t* **rnd**) [Function]
`int mpfr_const_pi` (*mpfr_t* **rop**, *mp_rnd_t* **rnd**) [Function]
`int mpfr_const_euler` (*mpfr_t* **rop**, *mp_rnd_t* **rnd**) [Function]
`int mpfr_const_catalan` (*mpfr_t* **rop**, *mp_rnd_t* **rnd**) [Function]

Set *rop* to the logarithm of 2, the value of π, of Euler's constant 0.577..., of Catalan's constant 0.915..., respectively, rounded in the direction *rnd*. These functions cache the computed values to avoid other calculations if a lower or equal precision is requested. To free these caches, use `mpfr_free_cache`.

`void mpfr_free_cache` (*void*) [Function]

Free various caches used by MPFR internally, in particular the caches used by the functions computing constants (currently `mpfr_const_log2`, `mpfr_const_pi`, `mpfr_const_euler` and `mpfr_const_catalan`). You should call this function when terminating a thread.

`int mpfr_sum` (*mpfr_t* **rop**, *mpfr_ptr const* **tab**[], *unsigned long* **n**, *mp_rnd_t* **rnd**) [Function]

Set *ret* to the sum of all elements of *tab* whose size is *n*, rounded in the direction *rnd*. Warning, *tab* is a table of pointers to mpfr_t, not a table of mpfr_t (preliminary interface). The returned `int` value is zero when the computed value is the exact value, and non-zero when this cannot be guaranteed, without giving the direction of the error as the other functions do.

5.8 Input and Output Functions

This section describes functions that perform input from an input/output stream, and functions that output to an input/output stream. Passing a null pointer for a *stream* argument to any of these functions will make them read from `stdin` and write to `stdout`, respectively.

When using any of these functions, you must include the `<stdio.h>` standard header before '`mpfr.h`', to allow '`mpfr.h`' to define prototypes for these functions.

`size_t mpfr_out_str` (*FILE* ***stream**, *int* **base**, *size_t* **n**, *mpfr_t* **op**, *mp_rnd_t* **rnd**) [Function]

Output *op* on stream *stream*, as a string of digits in base *base*, rounded in the direction *rnd*. The base may vary from 2 to 36. Print *n* significant digits exactly, or if *n* is 0, enough digits so that *op* can be read back exactly (see `mpfr_get_str`).

In addition to the significant digits, a decimal point (defined by the current locale) at the right of the first digit and a trailing exponent in base 10, in the form '`eNNN`', are printed. If *base* is greater than 10, '`@`' will be used instead of '`e`' as exponent delimiter.

Return the number of bytes written, or if an error occurred, return 0.

`size_t mpfr_inp_str` (*mpfr_t* `rop`, *FILE* `*stream`, *int* `base`, *mp_rnd_t* `rnd`) [Function]

Input a string in base *base* from stream *stream*, rounded in the direction *rnd*, and put the read float in *rop*.

This function reads a word (defined as a sequence of characters between whitespace) and parses it using `mpfr_set_str` (it may change). See the documentation of `mpfr_strtofr` for a detailed description of the valid string formats.

Return the number of bytes read, or if an error occurred, return 0.

5.9 Integer Related Functions

`int mpfr_rint` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_ceil` (*mpfr_t* `rop`, *mpfr_t* `op`) [Function]
`int mpfr_floor` (*mpfr_t* `rop`, *mpfr_t* `op`) [Function]
`int mpfr_round` (*mpfr_t* `rop`, *mpfr_t* `op`) [Function]
`int mpfr_trunc` (*mpfr_t* `rop`, *mpfr_t* `op`) [Function]

Set *rop* to *op* rounded to an integer. `mpfr_rint` rounds to the nearest representable integer in the given rounding mode, `mpfr_ceil` rounds to the next higher or equal representable integer, `mpfr_floor` to the next lower or equal representable integer, `mpfr_round` to the nearest representable integer, rounding halfway cases away from zero, and `mpfr_trunc` to the next representable integer toward zero.

The returned value is zero when the result is exact, positive when it is greater than the original value of *op*, and negative when it is smaller. More precisely, the returned value is 0 when *op* is an integer representable in *rop*, 1 or -1 when *op* is an integer that is not representable in *rop*, 2 or -2 when *op* is not an integer.

Note that `mpfr_round` is different from `mpfr_rint` called with the rounding to nearest mode (where halfway cases are rounded to an even integer or significand). Note also that no double rounding is performed; for instance, 4.5 (100.1 in binary) is rounded by `mpfr_round` to 4 (100 in binary) in 2-bit precision, though `round(4.5)` is equal to 5 and 5 (101 in binary) is rounded to 6 (110 in binary) in 2-bit precision.

`int mpfr_rint_ceil` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_rint_floor` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_rint_round` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]
`int mpfr_rint_trunc` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to *op* rounded to an integer. `mpfr_rint_ceil` rounds to the next higher or equal integer, `mpfr_rint_floor` to the next lower or equal integer, `mpfr_rint_round` to the nearest integer, rounding halfway cases away from zero, and `mpfr_rint_trunc` to the next integer toward zero. If the result is not representable, it is rounded in the direction *rnd*. The returned value is the ternary value associated with the considered round-to-integer function (regarded in the same way as any other mathematical function).

`int mpfr_frac` (*mpfr_t* `rop`, *mpfr_t* `op`, *mp_rnd_t* `rnd`) [Function]

Set *rop* to the fractional part of *op*, having the same sign as *op*, rounded in the direction *rnd* (unlike in `mpfr_rint`, *rnd* affects only how the exact fractional part is rounded, not how the fractional part is generated).

`int mpfr_remainder` (*mpfr_t* `r`, *mpfr_t* `x`, *mpfr_t* `y`, *mp_rnd_t* `rnd`) [Function] `int mpfr_remquo (mpfr_t r, long* q, mpfr_t x, mpfr_t y, mp_rnd_t rnd)` [Function]

Set *r* to the remainder of the division of *x* by *y*, with quotient rounded to the nearest integer (ties rounded to even), and *r* rounded according to the direction *rnd*. If *r* is zero, it has the sign of *x*. The return value is the inexact flag corresponding to *r*. Additionally, `mpfr_remquo` stores the low significant bits from the quotient in **q* (more precisely the number of bits in a `long` minus one), with the sign of *x* divided by *y* (except if those low bits are all zero, in which case zero is returned). Note that *x* may be so large in magnitude relative to *y* that an exact representation of the quotient is not practical. These functions are useful for additive argument reduction.

`int mpfr_integer_p (mpfr_t op)` [Function]

Return non-zero iff *op* is an integer.

5.10 Miscellaneous Functions

`void mpfr_nexttoward (mpfr_t x, mpfr_t y)` [Function]

If *x* or *y* is NaN, set *x* to NaN. Otherwise, if *x* is different from *y*, replace *x* by the next floating-point number (with the precision of *x* and the current exponent range) in the direction of *y*, if there is one (the infinite values are seen as the smallest and largest floating-point numbers). If the result is zero, it keeps the same sign. No underflow or overflow is generated.

`void mpfr_nextabove (mpfr_t x)` [Function]

Equivalent to `mpfr_nexttoward` where *y* is plus infinity.

`void mpfr_nextbelow (mpfr_t x)` [Function]

Equivalent to `mpfr_nexttoward` where *y* is minus infinity.

`int mpfr_min (mpfr_t rop, mpfr_t op1, mpfr_t op2, mp_rnd_t rnd)` [Function]

Set *rop* to the minimum of *op1* and *op2*. If *op1* and *op2* are both NaN, then *rop* is set to NaN. If *op1* or *op2* is NaN, then *rop* is set to the numeric value. If *op1* and *op2* are zeros of different signs, then *rop* is set to -0.

`int mpfr_max (mpfr_t rop, mpfr_t op1, mpfr_t op2, mp_rnd_t rnd)` [Function]

Set *rop* to the maximum of *op1* and *op2*. If *op1* and *op2* are both NaN, then *rop* is set to NaN. If *op1* or *op2* is NaN, then *rop* is set to the numeric value. If *op1* and *op2* are zeros of different signs, then *rop* is set to +0.

`int mpfr_urandomb (mpfr_t rop, gmp_randstate_t state)` [Function]

Generate a uniformly distributed random float in the interval $0 \leq rop < 1$. Return 0, unless the exponent is not in the current exponent range, in which case *rop* is set to NaN and a non-zero value is returned. The second argument is a `gmp_randstate_t` structure which should be created using the GMP `gmp_randinit` function, see the GMP manual.

`void mpfr_random (mpfr_t rop)` [Function]

Generate a uniformly distributed random float in the interval $0 \leq rop < 1$. This function is deprecated; `mpfr_urandomb` should be used instead.

`void mpfr_random2 (mpfr_t rop, mp_size_t size, mp_exp_t exp)` [Function]

Generate a random float of at most *size* limbs, with long strings of zeros and ones in the binary representation. The exponent of the number is in the interval $-exp$ to *exp*. This function is useful for testing functions and algorithms, since this kind of random numbers have proven to be more likely to trigger corner-case bugs. Negative random numbers are generated when *size* is negative. Put +0 in *rop* when size if zero. The internal state of the default pseudorandom number generator is modified by a call to this function (the same one as GMP if MPFR was built using '`--with-gmp-build`').

`mp_exp_t mpfr_get_exp` (*mpfr_t* `x`) [Function]

Get the exponent of *x*, assuming that *x* is a non-zero ordinary number and the significand is chosen in [1/2,1). The behavior for NaN, infinity or zero is undefined.

`int mpfr_set_exp` (*mpfr_t* `x`, *mp_exp_t* `e`) [Function]

Set the exponent of *x* if *e* is in the current exponent range, and return 0 (even if *x* is not a non-zero ordinary number); otherwise, return a non-zero value. The significand is assumed to be in [1/2,1).

`int mpfr_signbit` (*mpfr_t* `op`) [Function]

Return a non-zero value iff *op* has its sign bit set (i.e. if it is negative, −0, or a NaN whose representation has its sign bit set).

`int mpfr_setsign` (*mpfr_t* `rop`, *mpfr_t* `op`, *int* `s`, *mp_rnd_t* `rnd`) [Function]

Set the value of *rop* from *op*, rounded toward the given direction *rnd*, then set (resp. clear) its sign bit if *s* is non-zero (resp. zero), even when *op* is a NaN.

`int mpfr_copysign` (*mpfr_t* `rop`, *mpfr_t* `op1`, *mpfr_t* `op2`, *mp_rnd_t* `rnd`) [Function]

Set the value of *rop* from *op1*, rounded toward the given direction *rnd*, then set its sign bit to that of *op2* (even when *op1* or *op2* is a NaN). This function is equivalent to `mpfr_setsign (`*rop*`,` *op1*`, mpfr_signbit (`*op2*`),` *rnd*`)`.

`const char * mpfr_get_version` (*void*) [Function]

Return the MPFR version, as a null-terminated string.

`MPFR_VERSION` [Macro]
`MPFR_VERSION_MAJOR` [Macro]
`MPFR_VERSION_MINOR` [Macro]
`MPFR_VERSION_PATCHLEVEL` [Macro]
`MPFR_VERSION_STRING` [Macro]

`MPFR_VERSION` is the version of MPFR as a preprocessing constant. `MPFR_VERSION_MAJOR`, `MPFR_VERSION_MINOR` and `MPFR_VERSION_PATCHLEVEL` are respectively the major, minor and patch level of MPFR version, as preprocessing constants. `MPFR_VERSION_STRING` is the version (with an optional suffix, used in development and pre-release versions) as a string constant, which can be compared to the result of `mpfr_get_version` to check at run time the header file and library used match:

```
if (strcmp (mpfr_get_version (), MPFR_VERSION_STRING))
  fprintf (stderr, "Warning: header and library do not match\n");
```

Note: Obtaining different strings is not necessarily an error, as in general, a program compiled with some old MPFR version can be dynamically linked with a newer MPFR library version (if allowed by the library versioning system).

`long MPFR_VERSION_NUM` (`major`, `minor`, `patchlevel`) [Macro]

Create an integer in the same format as used by `MPFR_VERSION` from the given *major*, *minor* and *patchlevel*. Here is an example of how to check the MPFR version at compile time:

```
#if (!defined(MPFR_VERSION) || (MPFR_VERSION<MPFR_VERSION_NUM(2,1,0)))
```

```
# error "Wrong MPFR version."
#endif
```

`const char * mpfr_get_patches` (*void*) [Function]
Return a null-terminated string containing the ids of the patches applied to the MPFR library (contents of the 'PATCHES' file), separated by spaces. Note: If the program has been compiled with an older MPFR version and is dynamically linked with a new MPFR library version, the ids of the patches applied to the old (compile-time) MPFR version are not available (however this information should not have much interest in general).

5.11 Rounding Mode Related Functions

`void mpfr_set_default_rounding_mode` (*mp_rnd_t* `rnd`) [Function]
Set the default rounding mode to *rnd*. The default rounding mode is to nearest initially.

`mp_rnd_t mpfr_get_default_rounding_mode` (*void*) [Function]
Get the default rounding mode.

`int mpfr_prec_round` (*mpfr_t* `x`, *mp_prec_t* `prec`, *mp_rnd_t* `rnd`) [Function]
Round *x* according to *rnd* with precision *prec*, which must be an integer between `MPFR_PREC_MIN` and `MPFR_PREC_MAX` (otherwise the behavior is undefined). If *prec* is greater or equal to the precision of *x*, then new space is allocated for the significand, and it is filled with zeros. Otherwise, the significand is rounded to precision *prec* with the given direction. In both cases, the precision of *x* is changed to *prec*.

`int mpfr_round_prec` (*mpfr_t* `x`, *mp_rnd_t* `rnd`, *mp_prec_t* `prec`) [Function]
[This function is obsolete. Please use `mpfr_prec_round` instead.]

`const char * mpfr_print_rnd_mode` (*mp_rnd_t* `rnd`) [Function]
Return the input string (GMP_RNDD, GMP_RNDU, GMP_RNDN, GMP_RNDZ) corresponding to the rounding mode *rnd* or a null pointer if *rnd* is an invalid rounding mode.

5.12 Exception Related Functions

`mp_exp_t mpfr_get_emin` (*void*) [Function]
`mp_exp_t mpfr_get_emax` (*void*) [Function]
Return the (current) smallest and largest exponents allowed for a floating-point variable. The smallest positive value of a floating-point variable is $1/2 \times 2^{\mathrm{emin}}$ and the largest value has the form $(1 - \varepsilon) \times 2^{\mathrm{emax}}$.

`int mpfr_set_emin` (*mp_exp_t* `exp`) [Function]
`int mpfr_set_emax` (*mp_exp_t* `exp`) [Function]
Set the smallest and largest exponents allowed for a floating-point variable. Return a non-zero value when *exp* is not in the range accepted by the implementation (in that case the smallest or largest exponent is not changed), and zero otherwise. If the user changes the exponent range, it is her/his responsibility to check that all current floating-point variables are in the new allowed range (for example using `mpfr_check_range`), otherwise the subsequent behavior will be undefined, in the sense of the ISO C standard.

`mp_exp_t mpfr_get_emin_min` (*void*) [Function]
`mp_exp_t mpfr_get_emin_max` (*void*) [Function]
`mp_exp_t mpfr_get_emax_min` (*void*) [Function]

`mp_exp_t mpfr_get_emax_max` (*void*) [Function]

Return the minimum and maximum of the smallest and largest exponents allowed for `mpfr_set_emin` and `mpfr_set_emax`. These values are implementation dependent; it is possible to create a non portable program by writing `mpfr_set_emax(mpfr_get_emax_max())` and `mpfr_set_emin(mpfr_get_emin_min())` since the values of the smallest and largest exponents become implementation dependent.

`int mpfr_check_range` (*mpfr_t* `x`, *int* `t`, *mp_rnd_t* `rnd`) [Function]

This function forces *x* to be in the current range of acceptable values, *t* being the current ternary value: negative if *x* is smaller than the exact value, positive if *x* is larger than the exact value and zero if *x* is exact (before the call). It generates an underflow or an overflow if the exponent of *x* is outside the current allowed range; the value of *t* may be used to avoid a double rounding. This function returns zero if the rounded result is equal to the exact one, a positive value if the rounded result is larger than the exact one, a negative value if the rounded result is smaller than the exact one. Note that unlike most functions, the result is compared to the exact one, not the input value *x*, i.e. the ternary value is propagated.

`int mpfr_subnormalize` (*mpfr_t* `x`, *int* `t`, *mp_rnd_t* `rnd`) [Function]

This function rounds *x* emulating subnormal number arithmetic: if *x* is outside the subnormal exponent range, it just propagates the ternary value *t*; otherwise, it rounds *x* to precision `EXP(x)-emin+1` according to rounding mode *rnd* and previous ternary value *t*, avoiding double rounding problems. More precisely in the subnormal domain, denoting by *e* the value of `emin`, *x* is rounded in fixed-point arithmetic to an integer multiple of 2^{e-1}; as a consequence, $1.5 \times 2^{e-1}$ when *t* is zero is rounded to 2^e with rounding to nearest.

`PREC(x)` is not modified by this function. *rnd* and *t* must be the used rounding mode for computing *x* and the returned ternary value when computing *x*. The subnormal exponent range is from `emin` to `emin+PREC(x)-1`. If the result cannot be represented in the current exponent range (due to a too small `emax`), the behavior is undefined. Note that unlike most functions, the result is compared to the exact one, not the input value *x*, i.e. the ternary value is propagated. This is a preliminary interface.

This is an example of how to emulate double IEEE-754 arithmetic using MPFR:

```
{
  mpfr_t xa, xb;
  int i;
  volatile double a, b;

  mpfr_set_default_prec (53);
  mpfr_set_emin (-1073);
  mpfr_set_emax (1024);

  mpfr_init (xa); mpfr_init (xb);

  b = 34.3; mpfr_set_d (xb, b, GMP_RNDN);
  a = 0x1.1235P-1021; mpfr_set_d (xa, a, GMP_RNDN);

  a /= b;
  i = mpfr_div (xa, xa, xb, GMP_RNDN);
  i = mpfr_subnormalize (xa, i, GMP_RNDN);

  mpfr_clear (xa); mpfr_clear (xb);
```

```
    }
```

Warning: this emulates a double IEEE-754 arithmetic with correct rounding in the subnormal range, which may not be the case for your hardware.

`void mpfr_clear_underflow` (*void*) [Function]
`void mpfr_clear_overflow` (*void*) [Function]
`void mpfr_clear_nanflag` (*void*) [Function]
`void mpfr_clear_inexflag` (*void*) [Function]
`void mpfr_clear_erangeflag` (*void*) [Function]

Clear the underflow, overflow, invalid, inexact and erange flags.

`void mpfr_set_underflow` (*void*) [Function]
`void mpfr_set_overflow` (*void*) [Function]
`void mpfr_set_nanflag` (*void*) [Function]
`void mpfr_set_inexflag` (*void*) [Function]
`void mpfr_set_erangeflag` (*void*) [Function]

Set the underflow, overflow, invalid, inexact and erange flags.

`void mpfr_clear_flags` (*void*) [Function]

Clear all global flags (underflow, overflow, inexact, invalid, erange).

`int mpfr_underflow_p` (*void*) [Function]
`int mpfr_overflow_p` (*void*) [Function]
`int mpfr_nanflag_p` (*void*) [Function]
`int mpfr_inexflag_p` (*void*) [Function]
`int mpfr_erangeflag_p` (*void*) [Function]

Return the corresponding (underflow, overflow, invalid, inexact, erange) flag, which is non-zero iff the flag is set.

5.13 Advanced Functions

All the given interfaces are preliminary. They might change incompatibly in future revisions.

`MPFR_DECL_INIT` (*name*, *prec*) [Macro]

This macro declares *name* as an automatic variable of type `mpfr_t`, initializes it and sets its precision to be **exactly** *prec* bits and its value to NaN. *name* must be a valid identifier. You must use this macro in the declaration section. This macro is much faster than using `mpfr_init2` but has some drawbacks:

- You **must not** call `mpfr_clear` with variables created with this macro (The storage is allocated at the point of declaration and deallocated when the brace-level is exited.).
- You **can not** change their precision.
- You **should not** create variables with huge precision with this macro.
- Your compiler must support '`Non-Constant Initializers`' (standard in C++ and ISO C99) and '`Token Pasting`' (standard in ISO C89). If *prec* is not a compiler constant, your compiler must support '`Variable-length automatic arrays`' (standard in ISO C99). '`GCC 2.95.3`' supports all these features. If you compile your program with gcc in c89 mode and with '`-pedantic`', you may want to define the `MPFR_USE_EXTENSION` macro to avoid warnings due to the `MPFR_DECL_INIT` implementation.

`void mpfr_inits` (*mpfr_t* `x`, *...*) [Function]

Initialize all the `mpfr_t` variables of the given `va_list`, set their precision to be the default precision and their value to NaN. See `mpfr_init` for more details. The `va_list` is assumed to be composed only of type `mpfr_t`. It begins from *x*. It ends when it encounters a null pointer.

`void mpfr_inits2` (*mp_prec_t* `prec`, *mpfr_t* `x`, *...*) [Function]

Initialize all the `mpfr_t` variables of the given `va_list`, set their precision to be **exactly** *prec* bits and their value to NaN. See `mpfr_init2` for more details. The `va_list` is assumed to be composed only of type `mpfr_t`. It begins from *x*. It ends when it encounters a null pointer.

`void mpfr_clears` (*mpfr_t* `x`, *...*) [Function]

Free the space occupied by all the `mpfr_t` variables of the given `va_list`. See `mpfr_clear` for more details. The `va_list` is assumed to be composed only of type `mpfr_t`. It begins from *x*. It ends when it encounters a null pointer.

Here is an example of how to use multiple initialization functions:

```
{
  mpfr_t x, y, z, t;
  mpfr_inits2 (256, x, y, z, t, (void *) 0);
  ...
  mpfr_clears (x, y, z, t, (void *) 0);
}
```

5.14 Compatibility With MPF

A header file '`mpf2mpfr.h`' is included in the distribution of MPFR for compatibility with the GNU MP class MPF. After inserting the following two lines after the `#include <gmp.h>` line,

```
#include <mpfr.h>
#include <mpf2mpfr.h>
```

any program written for MPF can be compiled directly with MPFR without any changes. All operations are then performed with the default MPFR rounding mode, which can be reset with `mpfr_set_default_rounding_mode`.

Warning: the `mpf_init` and `mpf_init2` functions initialize to zero, whereas the corresponding `mpfr` functions initialize to NaN: this is useful to detect uninitialized values, but is slightly incompatible with `mpf`.

`void mpfr_set_prec_raw` (*mpfr_t* `x`, *mp_prec_t* `prec`) [Function]

Reset the precision of *x* to be **exactly** *prec* bits. The only difference with `mpfr_set_prec` is that *prec* is assumed to be small enough so that the significand fits into the current allocated memory space for *x*. Otherwise the behavior is undefined.

`int mpfr_eq` (*mpfr_t* `op1`, *mpfr_t* `op2`, *unsigned long int* `op3`) [Function]

Return non-zero if *op1* and *op2* are both non-zero ordinary numbers with the same exponent and the same first *op3* bits, both zero, or both infinities of the same sign. Return zero otherwise. This function is defined for compatibility with `mpf`. Do not use it if you want to know whether two numbers are close to each other; for instance, 1.011111 and 1.100000 are regarded as different for any value of *op3* larger than 1.

`void mpfr_reldiff` (*mpfr_t* `rop`, *mpfr_t* `op1`, *mpfr_t* `op2`, *mp_rnd_t* `rnd`) [Function]

Compute the relative difference between *op1* and *op2* and store the result in *rop*. This function does not guarantee the correct rounding on the relative difference; it just computes $|op1 - op2|/op1$, using the rounding mode *rnd* for all operations and the precision of *rop*.

`int mpfr_mul_2exp` (*mpfr_t* `rop`, *mpfr_t* `op1`, *unsigned long int* `op2`, *mp_rnd_t* `rnd`) [Function]

`int mpfr_div_2exp` (*mpfr_t* `rop`, *mpfr_t* `op1`, *unsigned long int* `op2`, *mp_rnd_t* `rnd`) [Function]

See `mpfr_mul_2ui` and `mpfr_div_2ui`. These functions are only kept for compatibility with MPF.

5.15 Custom Interface

Some applications use a stack to handle the memory and their objects. However, the MPFR memory design is not well suited for such a thing. So that such applications are able to use MPFR, an auxiliary memory interface has been created: the Custom Interface.

The following interface allows them to use MPFR in two ways:

- Either they directly store the MPFR FP number as a `mpfr_t` on the stack.
- Either they store their own representation of a FP number on the stack and construct a new temporary `mpfr_t` each time it is needed.

Nothing has to be done to destroy the FP numbers except garbaging the used memory: all the memory stuff (allocating, destroying, garbaging) is kept to the application.

Each function in this interface is also implemented as a macro for efficiency reasons: for example `mpfr_custom_init (s, p)` uses the macro, while `(mpfr_custom_init) (s, p)` uses the function.

Note 1: MPFR functions may still initialize temporary FP numbers using standard mpfr_init. See Custom Allocation (GNU MP).

Note 2: MPFR functions may use the cached functions (mpfr_const_pi for example), even if they are not explicitly called. You have to call `mpfr_free_cache` each time you garbage the memory iff mpfr_init, through GMP Custom Allocation, allocates its memory on the application stack.

Note 3: This interface is preliminary.

`size_t mpfr_custom_get_size` (*mp_prec_t* `prec`) [Function]

Return the needed size in bytes to store the significand of a FP number of precision *prec*.

`void mpfr_custom_init` (*void* `*significand`, *mp_prec_t* `prec`) [Function]

Initialize a significand of precision *prec*. *significand* must be an area of `mpfr_custom_get_size (prec)` bytes at least and be suitably aligned for an array of `mp_limb_t`.

`void mpfr_custom_init_set` (*mpfr_t* `x`, *int* `kind`, *mp_exp_t* `exp`, *mp_prec_t* `prec`, *void* `*significand`) [Function]

Perform a dummy initialization of a `mpfr_t` and set it to:

- if `ABS(kind) == MPFR_NAN_KIND`, *x* is set to NaN;
- if `ABS(kind) == MPFR_INF_KIND`, *x* is set to the infinity of sign `sign(kind)`;
- if `ABS(kind) == MPFR_ZERO_KIND`, *x* is set to the zero of sign `sign(kind)`;
- if `ABS(kind) == MPFR_REGULAR_KIND`, *x* is set to a regular number: `x = sign(kind)*significand*2^exp`

In all cases, it uses *significand* directly for further computing involving *x*. It will not allocate anything. A FP number initialized with this function cannot be resized using `mpfr_set_prec`, or cleared using `mpfr_clear`! *significand* must have been initialized with `mpfr_custom_init` using the same precision *prec*.

int **mpfr_custom_get_kind** (*mpfr_t x*) [Function]

Return the current kind of a `mpfr_t` as used by `mpfr_custom_init_set`. The behavior of this function for any `mpfr_t` not initialized with `mpfr_custom_init_set` is undefined.

void * **mpfr_custom_get_mantissa** (*mpfr_t x*) [Function]

Return a pointer to the significand used by a `mpfr_t` initialized with `mpfr_custom_init_set`. The behavior of this function for any `mpfr_t` not initialized with `mpfr_custom_init_set` is undefined.

mp_exp_t **mpfr_custom_get_exp** (*mpfr_t x*) [Function]

Return the exponent of *x*, assuming that *x* is a non-zero ordinary number. The return value for NaN, Infinity or Zero is unspecified but doesn't produce any trap. The behavior of this function for any `mpfr_t` not initialized with `mpfr_custom_init_set` is undefined.

void **mpfr_custom_move** (*mpfr_t x, void *new_position*) [Function]

Inform MPFR that the significand has moved due to a garbage collect and update its new position to `new_position`. However the application has to move the significand and the `mpfr_t` itself. The behavior of this function for any `mpfr_t` not initialized with `mpfr_custom_init_set` is undefined.

See the test suite for examples.

5.16 Internals

The following types and functions were mainly designed for the implementation of `mpfr`, but may be useful for users too. However no upward compatibility is guaranteed. You may need to include '`mpfr-impl.h`' to use them.

The `mpfr_t` type consists of four fields.

- The `_mpfr_prec` field is used to store the precision of the variable (in bits); this is not less than `MPFR_PREC_MIN`.
- The `_mpfr_sign` field is used to store the sign of the variable.
- The `_mpfr_exp` field stores the exponent. An exponent of 0 means a radix point just above the most significant limb. Non-zero values n are a multiplier 2^n relative to that point. A NaN, an infinity and a zero are indicated by a special value of the exponent.
- Finally, the `_mpfr_d` is a pointer to the limbs, least significant limbs stored first. The number of limbs in use is controlled by `_mpfr_prec`, namely ceil(`_mpfr_prec`/`mp_bits_per_limb`). Non-singular values always have the most significant bit of the most significant limb set to 1. When the precision does not correspond to a whole number of limbs, the excess bits at the low end of the data are zero.

int **mpfr_can_round** (*mpfr_t b, mp_exp_t err, mp_rnd_t rnd1, mp_rnd_t rnd2, mp_prec_t prec*) [Function]

Assuming *b* is an approximation of an unknown number *x* in the direction *rnd1* with error at most two to the power E(b)-*err* where E(b) is the exponent of *b*, return a non-zero value if one is able to round correctly *x* to precision *prec* with the direction *rnd2*, and 0 otherwise (including for NaN and Inf). This function **does not modify** its arguments.

Note: if one wants to also determine the correct ternary value when rounding *b* to precision *prec*, a useful trick is the following:

```
if (mpfr_can_round (b, err, rnd1, GMP_RNDZ, prec + (rnd2 == GMP_RNDN)))
   ...
```

Indeed, if *rnd2* is `GMP_RNDN`, this will check if one can round to *prec*+1 bits with a directed rounding: if so, one can surely round to nearest to *prec* bits, and in addition one can determine the correct ternary value, which would not be the case when *b* is near from a value exactly representable on *prec* bits.

`double mpfr_get_d1` (*mpfr_t* `op`) [Function]

Convert *op* to a `double`, using the default MPFR rounding mode (see function `mpfr_set_default_rounding_mode`). This function is obsolete.

Contributors

The main developers consist of Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier and Paul Zimmermann.

We would like to thank Jean-Michel Muller and Joris van der Hoeven for very fruitful discussions at the beginning of that project, Torbjörn Granlund and Kevin Ryde for their help about design issues, and Nathalie Revol for her careful reading of a previous version of this documentation. Kevin Ryde did a tremendous job for the portability of MPFR, and integrating it into GMP 4.x; alas the GMP developers decided in January 2004 not to include MPFR any more.

Sylvie Boldo from ENS-Lyon, France, contributed the functions `mpfr_agm` and `mpfr_log`. Emmanuel Jeandel, from ENS-Lyon too, contributed the generic hypergeometric code in `generic.c`, as well as the `mpfr_exp3`, a first implementation of the sine and cosine, and improved versions of `mpfr_const_log2` and `mpfr_const_pi`. Mathieu Dutour contributed the functions `mpfr_atan` and `mpfr_asin`, and a previous version of `mpfr_gamma`; David Daney contributed the hyperbolic and inverse hyperbolic functions, the base-2 exponential, and the factorial function. Fabrice Rouillier contributed the original version of '`mul_ui.c`', the '`gmp_op.c`' file, and helped to the Windows porting. Jean-Luc Rémy contributed the `mpfr_zeta` code. Ludovic Meunier helped in the design of the `mpfr_erf` code. Damien Stehlé contributed the `mpfr_get_ld_2exp` function.

The development of the MPFR library would not have been possible without the continuous support of INRIA, and of the LORIA and LIP laboratories. In particular the main authors were or are members of the PolKA, Spaces, Cacao project-teams at LORIA (Nancy, France) and of the Arenaire project-team at LIP (Lyon, France). The development of MPFR was also supported by a grant (202F0659 00 MPN 121) from the Conseil Régional de Lorraine in 2002.

References

- Torbjörn Granlund, "GNU MP: The GNU Multiple Precision Arithmetic Library", version 4.1.2, 2002.
- IEEE standard for binary floating-point arithmetic, Technical Report ANSI-IEEE Standard 754-1985, New York, 1985. Approved March 21, 1985: IEEE Standards Board; approved July 26, 1985: American National Standards Institute, 18 pages.
- Donald E. Knuth, "The Art of Computer Programming", vol 2, "Seminumerical Algorithms", 2nd edition, Addison-Wesley, 1981.
- Jean-Michel Muller, "Elementary Functions, Algorithms and Implementation", Birkhauser, Boston, 1997.

Appendix A GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

   The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the

Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

Function and Type Index