MPFR

The Multiple Precision Floating-Point Reliable Library
Edition 2.0.2
November 2003

The MPFR team
mpfr@loria.fr

This manual documents how to install and use the Multiple Precision Floating-Point Reliable Library, version 2.0.2.

Table of Contents

MPFR Copying Conditions

This library is *free*; this means that everyone is free to use it and free to redistribute it on a free basis. The library is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of this library that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the library, that you receive source code or else can get it if you want it, that you can change this library or use pieces of it in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the MPFR library, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the MPFR library. If it is modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the license for the MPFR library are found in the Lesser General Public License that accompanies the source code. See the file COPYING.LIB.

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.

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 5. 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

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. MPFR needs some internal GMP header files that are not installed. So, keep the GMP build directory as is, at least until you have built MPFR.
2. In the MPFR build directory, type './configure --with-gmp-include=GMPBUILD --with-gmp-lib=GMPINSTALL/lib' where 'GMPBUILD' is the GMP build directory and 'GMPINSTALL' the directory where you have installed GMP. Because of the internal header files required by MPFR, the option '--with-gmp=GMPINSTALL' is not sufficient and should not be used. 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' in the working directory. No dynamic library is provided yet.
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 4, 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/info' (or if you passed the '--prefix' option to 'configure', using the prefix directory given as argument to '--prefix' instead of '/usr/local').

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'.

2.1 Known Build Problems

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

Please report other problems to 'mpfr@loria.fr'. See Chapter 3 [Reporting Bugs], page 4. Some bug fixes are available on the MPFR web page http://www.loria.fr/projets/mpfr/ or 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/` or `http://www.loria.fr/projets/mpfr/`: 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`'.

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 mantissa with a limited precision exponent. The C data type for such objects is `mpfr_t`. 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 mantissa 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 and `MPFR_PREC_MAX` is equal to `ULONG_MAX`/2.

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 mantissa; 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 mantissa 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 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`.

mp_rnd_t **__gmpfr_default_rounding_mode** — Global Variable

The default rounding mode (to nearest initially).

4.5 Getting the Latest Version of MPFR

The latest version of MPFR is available from http://www.mpfr.org/ or http://www.loria.fr/projets/mpfr/.

5 Floating-point Functions

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.

5.1 Rounding Modes

The following four rounding modes are supported:

- `GMP_RNDN`: round to nearest
- `GMP_RNDZ`: round towards zero
- `GMP_RNDU`: round towards plus infinity
- `GMP_RNDD`: round towards 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 exactly 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).

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.

`void` **mpfr_set_default_rounding_mode** (`mp_rnd_t` *rnd*) Function

Sets the default rounding mode to *rnd*. The default rounding mode is to nearest initially.

`int` **mpfr_prec_round** (`mpfr_t` *x*, `mp_prec_t` *prec*, `mp_rnd_t` *rnd*) Function

Rounds *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 mantissa, and it is filled with zeros. Otherwise, the mantissa 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.]

`char *` **mpfr_print_rnd_mode** (`mp_rnd_t` *rnd*) Function

Returns 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.

5.2 Exceptions

Note: Overflow handling is still experimental and currently implemented very partially. If an overflow occurs internally at the wrong place, anything can happen (crash, wrong results, etc).

`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.

`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.

`void` **mpfr_clear_underflow** (`void`) Function
`void` **mpfr_clear_overflow** (`void`) Function
`void` **mpfr_clear_nanflag** (`void`) Function
`void` **mpfr_clear_inexflag** (`void`) Function

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

`void` **mpfr_clear_flags** (`void`) Function

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

`int` **mpfr_underflow_p** (`void`) Function
`int` **mpfr_overflow_p** (`void`) Function
`int` **mpfr_nanflag_p** (`void`) Function
`int` **mpfr_inexflag_p** (`void`) Function

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

5.3 Initialization Functions

`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 mantissa. 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

Returns the default MPFR precision in bits.

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_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_init2** (`mpfr_t` *x*, `mp_prec_t` *prec*) Function

Initialize *x*, set its precision to be **exactly** *prec* bits and its value to NaN.

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.

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 mantissa 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 mantissa.

`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 mantissa fits into the current allocated memory space for *x*. Otherwise the behavior is undefined.

5.4 Assignment Functions

These functions assign new values to already initialized floats (see Section 5.3 [Initialization Functions], page 9).

`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_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_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 towards the given direction *rnd*.

`int` **mpfr_set_str** (`mpfr_t` *x*, `const char` **s*, `int` *base*, `mp_rnd_t` *rnd*) Function

Set *x* to the value of the whole string *s* in base *base* (between 2 and 36), rounded in direction *rnd*. See the documentation of `mpfr_inp_str` for a detailed description of the valid string formats. This function returns 0 if the entire string up to the final `\0` is a valid number in base *base*; otherwise it returns −1.

`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.

5.5 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`.

5.6 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

Convert *op* to a `double` (respectively `long double`), using the rounding mode *rnd*.

`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.

`double` **mpfr_get_d_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 precision, using the given rounding mode.

`long` **mpfr_get_si** (`mpfr_t` *op*, `mp_rnd_t` *rnd*) Function
`unsigned long` **mpfr_get_ui** (`mpfr_t` *op*, `mp_rnd_t` *op*) Function

Convert *op* to a `long` or `unsigned long`, after rounding it with respect to *rnd*. If *op* is NaN or Inf, or too big for the return type, the result is undefined.

See also `mpfr_fits_slong_p` and `mpfr_fits_ulong_p`.

`mp_exp_t` **mpfr_get_z_exp** (`mpz_t` *z*, `mpfr_t` *op*) Function

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

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

Convert *op* to a string of digits in base *base*, with rounding in direction *rnd*. 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 *n* is zero, the number of digits of the mantissa is determined automatically from the precision of *op* and the value of *base*. Warning: this functionality may disappear or change in future versions. Otherwise generate exactly *n* significant digits, which must be at least 2.

If *str* is a null pointer, space for the mantissa is allocated using the current allocation function, and a pointer to the string is returned. The block will be `strlen(s)+1` bytes. For more information on how this block is allocated and how to free it: see section “Custom Allocation” in *GNU MP*.

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

If *n* is 0, note that the space requirements for *str* in this case will be impossible for the user to predetermine. Therefore, one needs to pass a null pointer for the string argument whenever *n* is 0.

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.

`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

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

5.7 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_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*.

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_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*.

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_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*.

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_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*.

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 *rop* 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

Set *rop* to the cubic root (defined over the real numbers) of *op* rounded in the direction *rnd*.

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_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:

- `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 *x*, 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*.

`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
`int` **mpfr_mul_2exp** (`mpfr_t` *rop*, `mpfr_t` *op1*, `unsigned 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. [`mpfr_mul_2exp` is kept for upward compatibility.]

`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
`int` **mpfr_div_2exp** (`mpfr_t` *rop*, `mpfr_t` *op1*, `unsigned 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. [`mpfr_div_2exp` is kept for upward compatibility.]

5.8 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

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 (Not-a-Number), the behavior is undefined.

`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 (Not-a-Number), 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`, but does not make much sense.

`int` **mpfr_nan_p** (`mpfr_t` *op*) Function
`int` **mpfr_inf_p** (`mpfr_t` *op*) Function
`int` **mpfr_number_p** (`mpfr_t` *op*) Function

Return non-zero if *op* is respectively Not-a-Number (NaN), an infinity, an ordinary number (i.e. neither Not-a-Number nor an infinity). Return zero otherwise.

`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 exact 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_sgn** (`mpfr_t` *op*) Macro

Return a positive value if $op > 0$, zero if $op = 0$, and a negative value if $op < 0$. Its result is undefined when *op* is NaN (Not-a-Number).

This function is actually implemented as a macro. It may evaluate its argument multiple times.

`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.

5.9 Special Functions

All those functions, except explicitely 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.

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*.

int **mpfr_exp** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function

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

int **mpfr_exp2** (mpfr_t *rop*, mpfr_t *op*, mp_rnd_t *rnd*) Function

Set *rop* to 2^{op}, 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_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*. 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*.

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_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_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*.

int **mpfr_zeta** (mpfr_t *rop*, mpfr_t *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_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_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].

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

Set *rop* to the logarithm of 2, the value of π, the value of Euler's constant 0.577..., respectively, rounded in the direction *rnd*. These functions cache the computed values to avoid other calculations if a lower or equal precision is requested. There is currently no way to free the cache.

5.10 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 direction *rnd*. The base may vary from 2 to 36. Print *n* significant digits exactly, or if *n* is 0, the maximum number of digits accurately representable by *op* (this feature may disappear).

In addition to the significant digits, a decimal point 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 direction *rnd*, and put the read float in *rop*. The string is of the form 'M@N' or, if the base is 10 or less, alternatively 'MeN' or 'MEN', or, if the base is 16, alternatively 'MpB' or 'MPB'. 'M' is the mantissa in the specified base, 'N' is the exponent written in decimal for the specified base, and in base 16, 'B' is the binary exponent written in decimal (i.e. it indicates the power of 2 by which the mantissa is to be scaled). The argument *base* may be in the range 2 to 36.

Special values can be read as follows (the case does not matter): `@NaN@`, `@Inf@`, `+@Inf@` and `-@Inf@`, possibly followed by other characters; if the base is smaller or equal to 16, the following strings are accepted too: `NaN`, `Inf`, `+Inf` and `-Inf`.

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

5.11 Miscellaneous 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 towards 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 the nearest mode (where halfway cases are rounded to an even integer or mantissa). 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_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_integer_p** (mpfr_t *op*) Function

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

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_urandomb** (mpfr_t *rop*, gmp_randstate_t *state*) Function

Generate a uniformly distributed random float in the interval $0 \le rop < 1$. 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.

void **mpfr_random** (mpfr_t *rop*) Function

Generate a uniformly distributed random float in the interval $0 \le rop < 1$. 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.

5.12 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_size field is used to store the number of allocated limbs, with the high bits reserved to store the sign (bit 31), the NaN flag (bit 30), and the Infinity flag (bit 29); thus bits 0 to 28 remain for the number of allocated limbs, with a maximal value of 536870911. A NaN is indicated by the NaN flag set, and the other fields are undefined. An Infinity is indicated by the NaN flag clear and the Infinity flag set; the sign bit of an Infinity indicates the sign, the limb data and the exponent are undefined.

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.

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/BITS_PER_MP_LIMB). Zeros are represented by the most significant limb being zero, other limb data and the exponent are undefined (this implies that the corresponding objects may contain invalid values, thus should not be evaluated even if they are not taken into account). Non-zero 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. When the precision has been lowered by mpfr_set_prec, the space allocated at _mpfr_d remains as given by _mpfr_size, but _mpfr_prec indicates how much of that space is actually used.

int **mpfr_add_one_ulp** (`mpfr_t` *x*, `mp_rnd_t` *rnd*) Function

Add one unit in last place (ulp) to *x* if *x* is finite and positive, subtract one ulp if *x* is finite and negative; otherwise, *x* is not changed. The return value is zero unless an overflow occurs, in which case the `mpfr_add_one_ulp` function behaves like a conventional addition.

int **mpfr_sub_one_ulp** (`mpfr_t` *x*, `mp_rnd_t` *rnd*) Function

Subtract one ulp to *x* if *x* is finite and positive, add one ulp if *x* is finite and negative; otherwise, *x* is not changed. The return value is zero unless an underflow occurs, in which case the `mpfr_sub_one_ulp` function behaves like a conventional subtraction.

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

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

Get the exponent of *x*, assuming that *x* is a non-zero ordinary number.

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 1.

void **mpfr_set_str_binary** (`mpfr_t` *x*, `const char` **s*) Function

Set *x* to the value of the binary number in string *s*, which has to be of the form +/-xxxx.xxxxxxEyy. The exponent is read in decimal, but is interpreted as the power of two to be multiplied by the mantissa. The mantissa length of *s* has to be less or equal to the precision of *x*, otherwise an error occurs. If *s* starts with `N`, it is interpreted as NaN (Not-a-Number); if it starts with `I` after the sign, it is interpreted as infinity, with the corresponding sign.

void **mpfr_print_binary** (`mpfr_t` *float*) Function

Output *float* on stdout in raw binary format (the exponent is written in decimal, yet).

Contributors

The main developers consist of Guillaume Hanrot, Vincent Lefèvre, Kevin Ryde 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 their suggestions for an easy integration into GNU MP, and Nathalie Revol for her careful reading of a previous version of this documentation.

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`, 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.

The development of the MPFR library would not have been possible without the continuous support of LORIA, INRIA and INRIA Lorraine. 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

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other written 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 that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

   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. (For example, 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.

   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 "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and 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 has been designed to thwart or discourage subsequent modification by readers is not Transparent. 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 designed for human modification. Opaque formats include PostScript, PDF, 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 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.

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 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 publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. 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 less than five).

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", and 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. In any section entitled "Acknowledgments" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments 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 as "Endorsements" or to conflict in title with any Invariant Section.

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.

   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 "Acknowledgments", 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, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they 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 quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around 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 provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

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 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

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