This manual documents how to install and use the Multiple Precision Floating-Point Reliable Library, version 4.0.0.


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, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in Appendix A [GNU Free Documentation License], page 56.
## Table of Contents

**MPFR Copying Conditions** .................................................. 1

1 **Introduction to MPFR** .................................................. 2  
   1.1 How to Use This Manual ............................................. 2

2 **Installing MPFR** .......................................................... 3  
   2.1 How to Install ...................................................... 3  
   2.2 Other ‘make’ Targets ............................................... 4  
   2.3 Build Problems .................................................... 4  
   2.4 Getting the Latest Version of MPFR ............................. 4

3 **Reporting Bugs** .......................................................... 5

4 **MPFR Basics** ............................................................. 6  
   4.1 Headers and Libraries ............................................... 6  
   4.2 Nomenclature and Types .......................................... 7  
   4.3 MPFR Variable Conventions ..................................... 7  
   4.4 Rounding Modes .................................................. 8  
   4.5 Floating-Point Values on Special Numbers .................. 9  
   4.6 Exceptions .......................................................... 9  
   4.7 Memory Handling ................................................. 10  
   4.8 Getting the Best Efficiency Out of MPFR .................... 11

5 **MPFR Interface** .......................................................... 12  
   5.1 Initialization Functions ......................................... 12  
   5.2 Assignment Functions ............................................ 14  
   5.3 Combined Initialization and Assignment Functions ........ 17  
   5.4 Conversion Functions ............................................ 17  
   5.5 Basic Arithmetic Functions ..................................... 20  
   5.6 Comparison Functions ............................................ 23  
   5.7 Special Functions ................................................ 24  
   5.8 Input and Output Functions ..................................... 29  
   5.9 Formatted Output Functions ..................................... 30  
      5.9.1 Requirements ................................................ 30  
      5.9.2 Format String ............................................... 30  
      5.9.3 Functions .................................................... 33  
   5.10 Integer and Remainder Related Functions ................... 33  
   5.11 Rounding-Related Functions ................................... 35  
   5.12 Miscellaneous Functions ....................................... 37  
   5.13 Exception Related Functions .................................. 40  
   5.14 Compatibility With MPF ....................................... 44  
   5.15 Custom Interface ................................................ 45  
   5.16 Internals ........................................................... 46
MPFR Copying Conditions

The GNU MPFR library (or MPFR for short) 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 GNU 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 GNU 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 GNU MPFR library are found in the Lesser General Public License that accompanies the source code. See the file COPYING.LESSER.
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 provide a class of floating-point numbers with precise semantics. The main characteristics of MPFR, which make it differ from most arbitrary precision floating-point software tools, are:

- the MPFR code is portable, i.e., the result of any operation does not depend on the machine word size \texttt{mp\_bits\_per\_limb} (64 on most current processors), possibly except in faithful rounding. It does not depend either on the machine rounding mode or rounding precision;
- the precision in bits can be set \textit{exactly} to any valid value for each variable (including very small precision);
- MPFR provides the four rounding modes from the IEEE 754-1985 standard, plus away-from-zero, as well as for basic operations as for other mathematical functions. Faithful rounding (partially supported) is provided too, but the results may no longer be reproducible.

In particular, with a precision of 53 bits and in any of the four standard rounding modes, MPFR is able to exactly reproduce all computations with double-precision machine floating-point numbers (e.g., \texttt{double} type in C, with a C implementation that rigorously follows Annex F of the ISO C99 standard and \texttt{FP\_CONTRACT} pragma set to \texttt{OFF}) on the four arithmetic operations and the square root, 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, version 3 or any later version. It is permitted to link MPFR to most 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. To use the library you will need to refer to Chapter 5 [MPFR Interface], page 12.

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

The MPFR library is already installed on some GNU/Linux distributions, but the development files necessary to the compilation such as mpfr.h are not always present. To check that MPFR is fully installed on your computer, you can check the presence of the file mpfr.h in /usr/include, or try to compile a small program having #include <mpfr.h> (since mpfr.h may be installed somewhere else). For instance, you can try to compile:

```c
#include <stdio.h>
#include <mpfr.h>
int main (void)
{
    printf ("MPFR library: %-12s
MPFR header: %s (based on %d.%d.%d)\n",
            mpfr_get_version (), MPFR_VERSION_STRING,
            MPFR_VERSION_MAJOR, MPFR_VERSION_MINOR, MPFR_VERSION_PATCHLEVEL);
    return 0;
}
```

with

```
cc -o version version.c -lmpfr -lgmp
```

and if you get errors whose first line looks like

```
version.c:2:19: error: mpfr.h: No such file or directory
```

then MPFR is probably not installed. Running this program will give you the MPFR version.

If MPFR is not installed on your computer, or if you want to install a different version, please follow the steps below.

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 the standard Unix `make' command, plus some other standard Unix utility commands. Then, in the MPFR build directory, type the following commands.

   ```
   ./configure
   ```

   This will prepare the build and setup the options according to your system. You can give options to specify the install directories (instead of the default /usr/local), threading support, and so on. See the INSTALL file and/or the output of `./configure --help' for more information, in particular if you get error messages.

2. `./configure'

   This will prepare the build and setup the options according to your system. You can give options to specify the install directories (instead of the default /usr/local), threading support, and so on. See the INSTALL file and/or the output of `./configure --help' for more information, in particular if you get error messages.

3. `make'

   This will compile MPFR, and create a library archive file libmpfr.a. On most platforms, a dynamic library will be produced too.

4. `make check'

   This will make sure that MPFR was built correctly. If any test fails, information about this failure can be found in the tests/test-suite.log file. If you want the contents of this file to be automatically output in case of failure, you can set the `VERBOSE' environment variable to 1 before running `make check', for instance by typing:

   `VERBOSE=1 make check'
In case of failure, you may want to check whether the problem is already known. If not, please report this failure to the MPFR mailing-list ‘mpfr@inria.fr’. For details, See Chapter 3 [Reporting Bugs], page 5.

5. ‘make install’
This will copy the files mpfr.h and mpf2mpfr.h to the directory /usr/local/include, the library files (libmpfr.a and possibly others) to the directory /usr/local/lib, the file mpfr.info to the directory /usr/local/share/info, and some other documentation files to the directory /usr/local/share/doc/mpfr (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 or update an info version of the manual, in mpfr.info.
  This file is already provided in the MPFR archives.
- ‘mpfr.pdf’ or ‘pdf’
  Create a PDF version of the manual, in mpfr.pdf.
- ‘mpfr.dvi’ or ‘dvi’
  Create a DVI version of the manual, in mpfr.dvi.
- ‘mpfr.ps’ or ‘ps’
  Create a Postscript version of the manual, in mpfr.ps.
- ‘mpfr.html’ or ‘html’
  Create a HTML version of the manual, in several pages in the directory doc/mpfr.html; if you want only one output HTML file, then type ‘makeinfo --html --no-split mpfr.texi’ from the ‘doc’ directory instead.
- ‘clean’
  Delete all object files and archive files, but not the configuration files.
- ‘distclean’
  Delete all generated files not included in the distribution.
- ‘uninstall’
  Delete all files copied by ‘make install’.

2.3 Build Problems
In case of problem, please read the INSTALL file carefully before reporting a bug, in particular section “In case of problem”. Some problems are due to bad configuration on the user side (not specific to MPFR). Problems are also mentioned in the FAQ http://www.mpfr.org/faq.html.

Please report problems to the MPFR mailing-list ‘mpfr@inria.fr’. See Chapter 3 [Reporting Bugs], page 5. Some bug fixes are available on the MPFR 4.0.0 web page http://www.mpfr.org/mpfr-4.0.0/.

2.4 Getting the Latest Version of MPFR
3 Reporting Bugs

If you think you have found a bug in the MPFR library, first have a look on the MPFR 4.0.0 web page http://www.mpfr.org/mpfr-4.0.0/ and the FAQ http://www.mpfr.org/faq.html: perhaps this bug is already known, in which case you may find there a workaround for it. You might also look in the archives of the MPFR mailing-list: https://sympa.inria.fr/sympa/arc/mpfr. 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, i.e., a small self-content program, using no other library than MPFR. 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 you get 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 are using GCC, ‘gcc -v’. Also, include the output from ‘uname -a’ and the MPFR version (the GMP version may be useful too). If you get a failure while running ‘make’ or ‘make check’, please include the config.log file in your bug report, and in case of test failure, the tests/test-suite.log file 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 will not do anything about it (aside of chiding you to send better bug reports).

Send your bug report to the MPFR mailing-list ‘mpfr@inria.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

4.1 Headers and Libraries

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:

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

Note however that prototypes for MPFR functions with `FILE *` parameters are provided only if `<stdio.h>` is included too (before `mpfr.h`):

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

Likewise `stdarg.h` (or `<varargs.h>`) is required for prototypes with `va_list` parameters, such as `mpfr_vprintf`.

And for 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. Moreover, under some platforms (in particular with C++ compilers), users may need to define `MPFR_USE_INTMAX_T` (and should do it for portability) before `mpfr.h` has been included; of course, it is possible to do that on the command line, e.g., with `-DMPFR_USE_INTMAX_T`.

Note: If `mpfr.h` and/or `gmp.h` (used by `mpfr.h`) are included several times (possibly from another header file), `stdio.h` and/or `stdarg.h` (or `<varargs.h>`) should be included before the first inclusion of `mpfr.h` or `gmp.h`. Alternatively, you can define `MPFR_USE_FILE` (for MPFR I/O functions) and/or `MPFR_USE_VA_LIST` (for MPFR functions with `va_list` parameters) anywhere before the last inclusion of `mpfr.h`. As a consequence, if your file is a public header that includes `mpfr.h`, you need to use the latter method.

When calling a MPFR macro, it is not allowed to have previously defined a macro with the same name as some keywords (currently `do`, `while` and `sizeof`).

You can avoid the use of MPFR macros encapsulating functions by defining the `MPFR_USE_NO_MACRO` macro before `mpfr.h` is included. In general this should not be necessary, but this can be useful when debugging user code: with some macros, the compiler may emit spurious warnings with some warning options, and macros can prevent some prototype checking.

All programs using MPFR must link against both `libmpfr` and `libgmp` libraries. On a typical Unix-like system this can be done with `'-lmmpfr -lgmp'` (in that order), for example:

```
gcc myprogram.c -lmpfr -lgmp
```

MPFR is built using Libtool and an application can use that to link if desired, see GNU Libtool.

If MPFR has been installed to a non-standard location, then it may be necessary to set up environment variables such as `C_INCLUDE_PATH` and `LIBRARY_PATH`, or use `-I` and `-L` compiler options, in order to point to the right directories. For a shared library, it may also be necessary to set up some sort of run-time library path (e.g., `LD_LIBRARY_PATH`) on some systems. Please read the `INSTALL` file for additional information.

Alternatively, it is possible to use `pkg-config` (a file `mpfr.pc` is provided as of MPFR 4.0):

```
cc myprogram.c $(pkg-config --cflags --libs mpfr)
```
Chapter 4: MPFR Basics

Note that the ‘MPFR_’ and ‘mpfr_’ prefixes are reserved for MPFR. As a general rule, in order to avoid clashes, software using MPFR (directly or indirectly) and system headers/libraries should not define macros and symbols using these prefixes.

4.2 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 standard, zero is signed, i.e., there are both +0 and −0; the behavior is the same as in the IEEE 754 standard and it is generalized to the other functions supported by MPFR. Unless documented otherwise, the sign bit of a NaN is unspecified.

The precision is the number of bits used to represent the significand of a floating-point number; the corresponding C data type is mpfr_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 1.

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 mpfr_rnd_t.

MPFR has a global (or per-thread) flag for each supported exception and provides operations on flags (Section 4.6 [Exceptions], page 9). This C data type is used to represent a group of flags (or a mask).

4.3 MPFR Variable Conventions

Before you can assign to a MPFR variable, you need to initialize it by calling one of the special initialization functions. When you are 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 do not 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.

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). This computes the square of x with rounding mode rnd and puts the result back in x.
4.4 Rounding Modes

The following rounding modes are supported:

- **MPFR_RNDN**: round to nearest (roundTiesToEven in IEEE 754-2008),
- **MPFR_RNDZ**: round toward zero (roundTowardZero in IEEE 754-2008),
- **MPFR_RNDU**: round toward plus infinity (roundTowardPositive in IEEE 754-2008),
- **MPFR_RNDD**: round toward minus infinity (roundTowardNegative in IEEE 754-2008),
- **MPFR_RNDA**: round away from zero.

- **MPFR_RNDF**: faithful rounding. This feature is currently experimental. Specific support for this rounding mode has been added to some functions, such as the basic operations (addition, subtraction, multiplication, square, division, square root) or when explicitly documented. It might also work with other functions, as it is possible that they do not need modification in their code; even though a correct behavior is not guaranteed yet (corrections were done when failures occurred in the test suite, but almost nothing has been checked manually), failures should be regarded as bugs and reported, so that they can be fixed.

The ‘round to nearest’ mode works as in the IEEE 754 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 2.5, 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).

The **MPFR_RNDF** mode works as follows: the computed value is either that corresponding to **MPFR_RNDD** or that corresponding to **MPFR_RNDU**. In particular when those values are identical, i.e., when the result of the corresponding operation is exactly representable, that exact result is returned. Thus, the computed result can take at most two possible values, and in absence of underflow/overflow, the corresponding error is strictly less than one ulp (unit in the last place) of that result and of the exact result. For **MPFR_RNDF**, the ternary value (defined below) and the inexact flag (defined later, as with the other flags) are unspecified, the divide-by-zero flag is as with other roundings, and the underflow and overflow flags match what would be obtained in the case the computed value is the same as with **MPFR_RNDD** or **MPFR_RNDU**. The results may not be reproducible.

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 that result in the rounding to nearest mode, and less than 1 ulp of that result in the directed rounding modes (a ulp is the weight of the least significant represented bit of the result 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 **MPFR_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 as result the value 1 (or any other value specified in this manual) for special cases (like `acos(0)`) yield an overflow or an underflow if that value 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 (where by “returned” we mean here the modified value of the destination object, which should not be mixed with the ternary return value of type `int` of those 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 12).

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 possible. If the limit is not defined (e.g., `mpfr_sqrt` and `mpfr_log` on `-0`), the behavior is specified in the description of the MPFR function, but must be consistent with the rule from the above paragraph (e.g., `mpfr_log` on `±0` gives `-Inf`).

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`); for example, `mpfr_sin` on `-0` gives `-0` and `mpfr_acos` on 1 gives `+0` (in all rounding modes). In the other cases, the sign is specified in the description of the MPFR function; for example `mpfr_max` on `-0` and `+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 12. 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 24), since for any finite or infinite input `x`, `mpfr_hypot` on `(x,+Inf)` gives `+Inf`.

### 4.6 Exceptions

MPFR defines a global (or per-thread) flag for each supported exception. A macro evaluating to a power of two is associated with each flag and exception, in order to be able to specify a group of flags (or a mask) by OR’ing such macros.
Flags can be cleared (lowered), set (raised), and tested by functions described in Section 5.13 [Exception Related Functions], page 40.

The supported exceptions are listed below. The macro associated with each exception is in parentheses.

- **Underflow (MPFR_FLAGS_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 value of the current exponent range. (In the round-to-nearest mode, the halfway case is rounded toward zero.)

  Note: This is not the single possible 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 $1$), 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 $0.5 \times 2^e$, which is representable in the current exponent range. As a consequence, this will not yield an underflow in MPFR.

- **Overflow (MPFR_FLAGS_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 value of the current exponent range. In the round-to-nearest mode, the result is infinite. Note: unlike the underflow case, there is only one possible definition of overflow here.

- **Divide-by-zero (MPFR_FLAGS_DIVBY0):** An exact infinite result is obtained from finite inputs.

- **NaN (MPFR_FLAGS_NAN):** A NaN exception occurs when the result of a function is NaN.

- **Inexact (MPFR_FLAGS_INEXACT):** An inexact exception occurs when the result of a function cannot be represented exactly and must be rounded.

- **Range error (MPFR_FLAGS_ERANGE):** 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 a conversion to an integer cannot be represented in the target type).

Moreover, the group consisting of all the flags is represented by the MPFR_FLAGS_ALL macro (if new flags are added in future MPFR versions, they will be added to this macro too).

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.

### 4.7 Memory Handling

MPFR functions may create caches, e.g., when computing constants such as $\pi$, either because the user has called a function like mpfr_const_pi directly or because such a function was called internally by the MPFR library itself to compute some other function. When more precision is needed, the value is automatically recomputed; a minimum of 10% increase of the precision is guaranteed to avoid too many recomputations.
MPFR functions may also create thread-local pools for internal use to avoid the cost of memory allocation. The pools can be freed with `mpfr_free_pool` (but with a default MPFR build, they should not take much memory, as the allocation size is limited).

At any time, the user can free various caches and pools with `mpfr_free_cache` and `mpfr_free_cache2`. It is strongly advised to free thread-local caches before terminating a thread, and all caches before exiting when using tools like `valgrind` (to avoid memory leaks being reported).

MPFR allocates its memory either on the stack (for temporary memory only) or with the same allocator as the one configured for GMP: see Section “Custom Allocation” in GNU MP. This means that the application must make sure that data allocated with the current allocator will not be reallocated or freed with a new allocator. So, in practice, if an application needs to change the allocator with `mp_set_memory_functions`, it should first free all data allocated with the current allocator: for its own data, with `mpfr_clear`, etc.; for the caches and pools, with `mpfr_mp_memory_cleanup` in all threads where MPFR is potentially used. This function is currently equivalent to `mpfr_free_cache`, but `mpfr_mp_memory_cleanup` is the recommended way in case the allocation method changes in the future (for instance, one may choose to allocate the caches for floating-point constants with `malloc` to avoid freeing them if the allocator changes). Developers should also be aware that MPFR may also be used indirectly by libraries, so that libraries based on MPFR should provide a clean-up function calling `mpfr_mp_memory_cleanup` and/or warn their users about this issue.

Note: For multithreaded applications, the allocator must be valid in all threads where MPFR may be used; data allocated in one thread may be reallocated and/or freed in some other thread.

MPFR internal data such as flags, the exponent range, the default precision and rounding mode, and caches (i.e., data that are not accessed via parameters) are either global (if MPFR has not been compiled as thread safe) or per-thread (thread local storage, TLS). The initial values of TLS data after a thread is created entirely depend on the compiler and thread implementation (MPFR simply does a conventional variable initialization, the variables being declared with an implementation-defined TLS specifier).

Writers of libraries using MPFR should be aware that the application and/or another library used by the application may also use MPFR, so that changing the exponent range, the default precision, or the default rounding mode may have an effect on this other use of MPFR since these data are not duplicated (unless they are in a different thread). Therefore any such value changed in a library function should be restored before the function returns (unless the purpose of the function is to do such a change). Writers of software using MPFR should also be careful when changing such a value if they use a library using MPFR (directly or indirectly), in order to make sure that such a change is compatible with the library.

4.8 Getting the Best Efficiency Out of MPFR

Here are a few hints to get the best efficiency out of MPFR:

- you should avoid allocating and clearing variables. Reuse variables whenever possible, allocate or clear outside of loops, pass temporary variables to subroutines instead of allocating them inside the subroutines;
- use `mpfr_swap` instead of `mpfr_set` whenever possible. This will avoid copying the significands;
- avoid using MPFR from C++, or make sure your C++ interface does not perform unnecessary allocations or copies;
- MPFR functions work in-place: to compute \( a = a + b \) you don’t need an auxiliary variable, you can directly write `mpfr_add (a, a, b, ...)`. 


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 functions. The function prefix for floating-point operations is mpfr_.

The user has to specify the precision of 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 on 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 arithmetic. The results obtained on a given computer are identical to those obtained on a computer with a different word size, or with a different compiler or operating system.

MPFR does not keep track of the accuracy of a computation. This is left to the user or to a higher layer (for example the MPFI library for interval arithmetic). 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 after calling any MPFR function or macro, whether or not there is an error. Except when documented, MPFR will not set errno, but functions called by the MPFR code (libc functions, memory allocator, etc.) may do so.

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.

```c
void mpfr_init2 (mpfr_t x, mpfr_prec_t prec) [Function]
Initialize x, set its precision to be exactly prec bits and its value to NaN. (Warning: the corresponding MPF function initializes 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).

```c
void mpfr_inits2 (mpfr_prec_t prec, mpfr_t x, ...) [Function]
Initialize all the mpfr_t variables of the given variable argument 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 (or equivalently mpfr_ptr). It begins from x, and ends when it encounters a null pointer (whose type must also be mpfr_ptr).
```

```c
void mpfr_clear (mpfr_t x) [Function]
Free the space occupied by the significand of x. Make sure to call this function for all mpfr_t variables when you are done with them.
```

```c
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 (or equivalently
mpfr_ptr). It begins from \( x \), and ends when it encounters a null pointer (whose type must also be mpfr_ptr).

Here is an example of how to use multiple initialization functions (since NULL is not necessarily defined in this context, we use (mpfr_ptr) 0 instead, but (mpfr_ptr) NULL is also correct).

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

**void mpfr_init (mpfr_t x)**

Initialize \( x \), set its precision to the default precision, and set its value to NaN. The default precision can be changed by a call to mpfr_set_default_prec.

Warning! In a given program, some other libraries might change the default precision and not restore it. Thus it is safer to use mpfr_init2.

**void mpfr_inits (mpfr_t x, ...)**

Initialize all the mpfr_t variables of the given va_list, set their precision to 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 (or equivalently mpfr_ptr). It begins from \( x \), and ends when it encounters a null pointer (whose type must also be mpfr_ptr).

Warning! In a given program, some other libraries might change the default precision and not restore it. Thus it is safer to use mpfr_inits2.

**MPFR_DECL_INIT (name, prec)**

This macro declares \( \text{name} \) as an automatic variable of type mpfr_t, initializes it and sets its precision to be exactly \( \text{prec} \) bits and its value to NaN. \( \text{name} \) 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 **cannot** 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 \( \text{prec} \) is not a constant expression, your compiler must support ‘variable-length automatic arrays’ (standard in ISO C99). GCC 2.95.3 and above 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_set_default_prec (mpfr_prec_t prec)**

Set the default precision to be exactly \( \text{prec} \) bits, where \( \text{prec} \) can be any integer between MPFR_PREC_MIN and MPFR_PREC_MAX. The precision of a variable means the number of bits used to store its significand. All subsequent calls to mpfr_init or mpfr_inits will use this precision, but previously initialized variables are unaffected. The default precision is set to 53 bits initially.
Note: when MPFR is built with the ‘--enable-thread-safe’ configure option, the default precision is local to each thread. See Section 4.7 [Memory Handling], page 10, for more information.

\texttt{mpfr\_prec\_t mpfr\_get\_default\_prec (void)} \quad [Function]

Return the current default MPFR precision in bits. See the documentation of \texttt{mpfr\_set\_default\_prec}.

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

\begin{verbatim}
{
    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);
    mpfr\_free\_cache (); /* free the cache for constants like pi */
}
\end{verbatim}

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.

\texttt{void mpfr\_set\_prec (mpfr\_t x, mpfr\_prec\_t prec)} \quad [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 \texttt{mpfr\_clear(x)} followed by a call to \texttt{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 \texttt{MPFR\_PREC\_MIN} and \texttt{MPFR\_PREC\_MAX}. In case you want to keep the previous value stored in \(x\), use \texttt{mpfr\_prec\_round} instead.

Warning! You must not use this function if \(x\) was initialized with \texttt{MPFR\_DECL\_INIT} or with \texttt{mpfr\_custom\_init\_set} (see Section 5.15 [Custom Interface], page 45).

\texttt{mpfr\_prec\_t mpfr\_get\_prec (mpfr\_t x)} \quad [Function]

Return the precision 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 12).

\texttt{int mpfr\_set (mpfr\_t rop, mpfr\_t op, mpfr\_rnd\_t rnd)} \quad [Function]
\texttt{int mpfr\_set\_ui (mpfr\_t rop, unsigned long int op, mpfr\_rnd\_t rnd)} \quad [Function]
\texttt{int mpfr\_set\_si (mpfr\_t rop, long int op, mpfr\_rnd\_t rnd)} \quad [Function]
\texttt{int mpfr\_set\_uj (mpfr\_t rop, uintmax\_t op, mpfr\_rnd\_t rnd)} \quad [Function]
\texttt{int mpfr\_set\_sj (mpfr\_t rop, intmax\_t op, mpfr\_rnd\_t rnd)} \quad [Function]
\texttt{int mpfr\_set\_flt (mpfr\_t rop, float op, mpfr\_rnd\_t rnd)} \quad [Function]
\texttt{int mpfr\_set\_d (mpfr\_t rop, double op, mpfr\_rnd\_t rnd)} \quad [Function]
\texttt{int mpfr\_set\_ld (mpfr\_t rop, long double op, mpfr\_rnd\_t rnd)} \quad [Function]
\texttt{int mpfr\_set\_float128 (mpfr\_t rop, __float128 op, mpfr\_rnd\_t rnd)} \quad [Function]
\texttt{int mpfr\_set\_decimal64 (mpfr\_t rop, _Decimal64 op, mpfr\_rnd\_t rnd)} \quad [Function]
Chapter 5: MPFR Interface

```
int mpfr_set_z (mpfr_t rop, mpz_t op, mpfr_rnd_t rnd) [Function]
int mpfr_set_q (mpfr_t rop, mpq_t op, mpfr_rnd_t rnd) [Function]
int mpfr_set_f (mpfr_t rop, mpfr_t op, mpfr_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_uj`, `mpfr_set_sj`. The `mpfr_set_float128` function is built only with the configure option `--enable-float128`, which requires the compiler or system provides the `__float128` data type (GCC 4.3 or later supports this data type); to use `mpfr_set_float128`, one should define the macro `MPFR_WANT_FLOAT128` before including `mpfr.h`. `mpfr_set_z`, `mpfr_set_q` and `mpfr_set_f`, regardless of the rounding mode. If the system does not support the IEEE 754 standard, `mpfr_set_flt`, `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`, and when the compiler or system provides the `_Decimal64` data type (recent versions of GCC support this data type); to use `mpfr_set_decimal64`, one should define the macro `MPFR_WANT_DECIMAL_FLOATS` before including `mpfr.h`. `mpfr_set_q` might fail if the numerator (or the denominator) can not be represented as a `mpfr_t`. For `mpfr_set`, the sign of a NaN is propagated in order to mimic the IEEE 754 copy operation. But contrary to IEEE 754, the NaN flag is set as usual.

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_flt`, `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 (or decimal, for `mpfr_set_decimal64`) number before MPFR can work with it.

```
int mpfr_set_ui_2exp (mpfr_t rop, unsigned long int op, mpfr_exp_t e, mpfr_rnd_t rnd) [Function]
int mpfr_set_si_2exp (mpfr_t rop, long int op, mpfr_exp_t e, mpfr_rnd_t rnd) [Function]
int mpfr_set_uj_2exp (mpfr_t rop, uintmax_t op, intmax_t e, mpfr_rnd_t rnd) [Function]
int mpfr_set_sj_2exp (mpfr_t rop, intmax_t op, intmax_t e, mpfr_rnd_t rnd) [Function]
int mpfr_set_z_2exp (mpfr_t rop, mpz_t op, mpfr_exp_t e, mpfr_rnd_t rnd) [Function]
```

Set the value of `rop` from `op × 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, mpfr_rnd_t rnd) [Function]
```

Set rop to the value of the 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. Contrary to `mpfr_strtofr`, `mpfr_set_str` requires the whole string to represent a valid floating-point number.

The meaning of the return value differs from other MPFR functions: it is 0 if the entire string up to the final null character is a valid number in base `base`; otherwise it is -1, and rop may have changed (users interested in the [ternary value], page 8, should use `mpfr_strtofr` instead).

Note: it is preferable to use `mpfr_strtofr` if one wants to distinguish between an infinite `rop` value coming from an infinite `s` or from an overflow.
int mpfr_strtofr (mpfr_t rop, const char *nptr, char **endptr, int base, mpfr_rnd_t rnd)
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 62 (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 (for consistency with strtod) 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. 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 = 35; case is ignored in bases less or equal to 36, in bases larger than 36, a = 36, b = 37, . . ., z = 61. The value of a significand digit 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 the exponent, called binary exponent, indicates a multiplication by a power of 2 instead of the base (there is a difference only for base 16); in base 16 for example 1p2 represents 4 whereas 1@2 represents 256. The value of an exponent is always written in base 10.

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 (if present) 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', thus 0 is read.

Special data (for infinities and NaN) can be @inf@ or @nan@n-char-sequence-opt, and if base \leq 16, it can also be infinity, inf, nan or nan(n-char-sequence-opt), all case insensitive. A n-char-sequence-opt is a possibly 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. For example, -@nan@This_Is_Not_17 is a valid representation for NaN in base 17.

void mpfr_set_nan (mpfr_t x) [Function]
void mpfr_set_inf (mpfr_t x, int sign) [Function]
void mpfr_set_zero (mpfr_t x, int sign) [Function]
Set the variable x to NaN (Not-a-Number), infinity or zero respectively. In mpfr_set_inf or mpfr_set_zero, x is set to plus infinity or plus zero iff sign is nonnegative; in mpfr_set_nan, the sign bit of the result is unspecified.
### 5.3 Combined Initialization and Assignment Functions

```c
int mpfr_init_set (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Macro]
int mpfr_init_set_ui (mpfr_t rop, unsigned long int op, mpfr_rnd_t rnd) [Macro]
int mpfr_init_set_si (mpfr_t rop, long int op, mpfr_rnd_t rnd) [Macro]
int mpfr_init_set_d (mpfr_t rop, double op, mpfr_rnd_t rnd) [Macro]
int mpfr_init_set_ld (mpfr_t rop, long double op, mpfr_rnd_t rnd) [Macro]
int mpfr_init_set_z (mpfr_t rop, mpz_t op, mpfr_rnd_t rnd) [Macro]
int mpfr_init_set_q (mpfr_t rop, mpq_t op, mpfr_rnd_t rnd) [Macro]
int mpfr_init_set_f (mpfr_t rop, mpfr_t op, mpfr_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`.

```c
int mpfr_init_set_str (mpfr_t x, const char *s, int base, mpfr_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

```c
float mpfr_get_flt (mpfr_t op, mpfr_rnd_t rnd) [Function]
double mpfr_get_d (mpfr_t op, mpfr_rnd_t rnd) [Function]
long mpfr_get_z (mpfr_t op, mpfr_rnd_t rnd) [Function]
__float128 mpfr_get_float128 (mpfr_t op, mpfr_rnd_t rnd) [Function]
_Decimal64 mpfr_get_decimal64 (mpfr_t op, mpfr_rnd_t rnd) [Function]
```

Convert op to a float (respectively double, long double or _Decimal64), 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_float128` and `mpfr_get_decimal64` functions are built only under some conditions: see the documentation of `mpfr_set_float128` and `mpfr_set_decimal64` respectively.

```c
long mpfr_get_si (mpfr_t op, mpfr_rnd_t rnd) [Function]
unsigned long mpfr_get_ui (mpfr_t op, mpfr_rnd_t rnd) [Function]
intmax_t mpfr_get_si (mpfr_t op, mpfr_rnd_t rnd) [Function]
uintmax_t mpfr_get_uj (mpfr_t op, mpfr_rnd_t rnd) [Function]
```

Convert op to a long, an unsigned long, an intmax_t or an uintmax_t (respectively) after rounding it to an integer with respect to rnd. If op is NaN, 0 is returned and the `range` flag is set. If op is too big for the return type, the function returns the maximum or the minimum of the corresponding C type, depending on the direction of the overflow; the `range` flag is set too. When there is no such range error, if the return value differs from op, i.e., if op is not an integer, the inexact flag is set. See also `mpfr_fits_slong_p`, `mpfr_fits_ulong_p`, `mpfr_fits_intmax_p` and `mpfr_fits_uintmax_p`.
double mpfr_get_d_2exp (long *exp, mpfr_t op, mpfr_rnd_t rnd) [Function]
long double mpfr_get_ld_2exp (long *exp, mpfr_t op, mpfr_rnd_t rnd) [Function]

Return $d$ and set $\exp$ (formally, the value pointed to by $\exp$) such that $0.5 \leq |d| < 1$ 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.

int mpfr_frexp (mpfr_exp_t *exp, mpfr_t y, mpfr_t x, mpfr_rnd_t rnd) [Function]

Set $\exp$ (formally, the value pointed to by $\exp$) and $x$ such that $0.5 \leq |y| < 1$ and $y \times 2^\exp$ equals $x$ rounded to the precision of $y$, using the given rounding mode. If $x$ is zero, then $y$ is set to a zero of the same sign and $\exp$ is set to 0. If $x$ is NaN or an infinity, then $y$ is set to the same value and $\exp$ is undefined.

mpfr_exp_t mpfr_get_z_2exp (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 $rop \times 2^\exp$. If $op$ is zero, the minimal exponent $\text{emin}$ is returned. If $op$ is NaN or an infinity, the $\text{erange}$ flag is set, $rop$ is set to 0, and the the minimal exponent $\text{emin}$ is returned. The returned exponent may be less than the minimal exponent $\text{emin}$ of MPFR numbers in the current exponent range; in case the exponent is not representable in the mpfr_exp_t type, the $\text{erange}$ flag is set and the minimal value of the mpfr_exp_t type is returned.

int mpfr_get_z (mpz_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]

Convert $op$ to a mpz_t, after rounding it with respect to $\text{rnd}$. If $op$ is NaN or an infinity, the $\text{erange}$ flag is set, $rop$ is set to 0, and 0 is returned. Otherwise the return value is zero when $rop$ is equal to $op$ (i.e., when $op$ is an integer), positive when it is greater than $op$, and negative when it is smaller than $op$; moreover, if $rop$ differs from $op$, i.e., if $op$ is not an integer, the inexact flag is set.

void mpfr_get_q (mpq_t rop, mpfr_t op) [Function]

Convert $op$ to a mpq_t. If $op$ is NaN or an infinity, the $\text{erange}$ flag is set and $rop$ is set to 0. Otherwise the conversion is always exact.

int mpfr_get_f (mpf_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]

Convert $op$ to a mpf_t, after rounding it with respect to $\text{rnd}$. The $\text{erange}$ flag is set if $op$ is NaN or an infinity, which do not exist in MPF. If $op$ is NaN, then $rop$ is undefined. If $op$ is +Inf (resp. −Inf), then $rop$ is set to the maximum (resp. minimum) value in the precision of the MPF number; if a future MPF version supports infinities, this behavior will be considered incorrect and will change (portable programs should assume that $rop$ is set either to this finite number or to an infinite number). Note that since MPFR currently has the same exponent type as MPF (but not with the same radix), the range of values is much larger in MPF than in MPFR, so that an overflow or underflow is not possible.

char * mpfr_get_str (char *str, mpfr_exp_t *exppt, int b, size_t n, mpfr_t op, mpfr_rnd_t rnd) [Function]

Convert $op$ to a string of digits in base $b$, with rounding in the direction $\text{rnd}$, where $n$ is either zero (see below) or the number of significant digits output in the string; in the latter case, $n$ must be greater or equal to 2. The base may vary from 2 to 62; otherwise the function does nothing and immediately returns a null pointer.
If the input is NaN, then the returned string is ‘@NaN@’ and the NaN flag is set. If the input is +Inf (resp. −Inf), then the returned string is ‘@Inf@’ (resp. ‘−@Inf@’).

If the input number is a finite number, the exponent is written through the pointer expptr (for input 0, the current minimal exponent is written); the type mpfr_exp_t is large enough to hold the exponent in all cases.

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 consecutive possible outputs, the one with an even significand is chosen, where both significands are considered with the exponent of op. Note that for an odd base, this may not correspond to an even last digit: for example with 2 digits in base 7, (14) and a half is rounded to (15) which is 12 in decimal, (16) and a half is rounded to (20) which is 14 in decimal, and (26) and a half is rounded to (26) which is 20 in decimal.

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 \( m \) depending only on \( p = \text{PREC}(op) \) and \( b \) that satisfies the above property, i.e., \( m = 1 + \left\lceil \frac{p \log 2}{\log b} \right\rceil \), with \( p \) replaced by \( p − 1 \) if \( b \) is a power of 2, but in some very rare cases, it might be \( m + 1 \) (the smallest case for bases up to 62 is when \( p \) equals 186564318007 for bases 7 and 49).

If str is a null pointer, space for the significand is allocated using the allocation function (see Section 4.7 [Memory Handling], page 10) and a pointer to the string is returned (unless the base is invalid). 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. A safe block size (sufficient for any value) is \( \max(n + 2, 7) \) if \( n \) is not zero; if \( n \) is zero, replace it by \( m + 1 \), as discussed above. The extra two bytes are for a possible minus sign, and for the terminating null character, and the value 7 accounts for ‘−@Inf@’ plus the terminating null character. The pointer to the string str is returned (unless the base is invalid).

Like in usual functions, the inexact flag is set iff the result is inexact.

```c
void mpfr_free_str (char *str) [Function]
Free a string allocated by mpfr_get_str using the unallocation function (see Section 4.7 [Memory Handling], page 10) and a pointer to the string is returned (unless the base is invalid). To free the returned string, you must use mpfr_free_str.

int mpfr_fits_ulong_p (mpfr_t op, mpfr rnd_t rnd) [Function]
int mpfr_fits_slong_p (mpfr_t op, mpfr rnd_t rnd) [Function]
int mpfr_fits_uint_p (mpfr_t op, mpfr rnd_t rnd) [Function]
int mpfr_fits_sint_p (mpfr_t op, mpfr rnd_t rnd) [Function]
int mpfr_fits_ushort_p (mpfr_t op, mpfr rnd_t rnd) [Function]
int mpfr_fits_sshort_p (mpfr_t op, mpfr rnd_t rnd) [Function]
int mpfr_fits_uintmax_p (mpfr_t op, mpfr rnd_t rnd) [Function]
int mpfr_fits_intmax_p (mpfr_t op, mpfr rnd_t rnd) [Function]
```
that is representable in the corresponding type. As a consequence, for MPFR_RNDF, mpfr_fits_ulong_p will return non-zero for a non-negative number less or equal to ULONG_MAX.

### 5.5 Basic Arithmetic Functions

```c
int mpfr_add (mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_rnd_t rnd)  [Function]
int mpfr_add_ui (mpfr_t rop, mpfr_t op1, unsigned long int op2, mpfr_rnd_t rnd)  [Function]
int mpfr_add_si (mpfr_t rop, mpfr_t op1, long int op2, mpfr_rnd_t rnd)  [Function]
int mpfr_add_d (mpfr_t rop, mpfr_t op1, double op2, mpfr_rnd_t rnd)  [Function]
int mpfr_add_z (mpfr_t rop, mpfr_t op1, mpz_t op2, mpfr_rnd_t rnd)  [Function]
int mpfr_add_q (mpfr_t rop, mpfr_t op1, mpq_t op2, mpfr_rnd_t rnd)  [Function]
```

Set rop to op1 + op2 rounded in the direction rnd. The IEEE 754 rules are used, in particular for signed zeros. But for types having no signed zeros, 0 is considered unsigned (i.e., (+0) + 0 = (+0) and (−0) + 0 = (−0)). The mpfr_add_d function assumes that the radix of the double type is a power of 2, with a precision at most that declared by the C implementation (macro IEEE_DBL_MANT_DIG, and if not defined 53 bits).

```c
int mpfr_sub (mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_rnd_t rnd)  [Function]
int mpfr_ui_sub (mpfr_t rop, unsigned long int op1, mpfr_t op2, mpfr_rnd_t rnd)  [Function]
int mpfr_sub_ui (mpfr_t rop, mpfr_t op1, unsigned long int op2, mpfr_rnd_t rnd)  [Function]
int mpfr_si_sub (mpfr_t rop, long int op1, mpfr_t op2, mpfr_rnd_t rnd)  [Function]
int mpfr_sub_si (mpfr_t rop, mpfr_t op1, long int op2, mpfr_rnd_t rnd)  [Function]
int mpfr_d_sub (mpfr_t rop, double op1, mpfr_t op2, mpfr_rnd_t rnd)  [Function]
int mpfr_sub_d (mpfr_t rop, mpfr_t op1, double op2, mpfr_rnd_t rnd)  [Function]
int mpfr_z_sub (mpfr_t rop, mpz_t op1, mpfr_t op2, mpfr_rnd_t rnd)  [Function]
int mpfr_sub_z (mpfr_t rop, mpfr_t op1, mpz_t op2, mpfr_rnd_t rnd)  [Function]
int mpfr_sub_q (mpfr_t rop, mpfr_t op1, mpq_t op2, mpfr_rnd_t rnd)  [Function]
```

Set rop to op1 − op2 rounded in the direction rnd. The IEEE 754 rules are used, in particular for signed zeros. But for types having no signed zeros, 0 is considered unsigned (i.e., (+0) − 0 = (+0), (−0) − 0 = (−0), 0 − (+0) = (−0) and 0 − (−0) = (+0)). The same restrictions than for mpfr_add_d apply to mpfr_d_sub and mpfr_sub_d.

```c
int mpfr_mul (mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_rnd_t rnd)  [Function]
int mpfr_mul_ui (mpfr_t rop, mpfr_t op1, unsigned long int op2, mpfr_rnd_t rnd)  [Function]
int mpfr_mul_si (mpfr_t rop, mpfr_t op1, long int op2, mpfr_rnd_t rnd)  [Function]
int mpfr_mul_d (mpfr_t rop, mpfr_t op1, double op2, mpfr_rnd_t rnd)  [Function]
int mpfr_mul_z (mpfr_t rop, mpfr_t op1, mpz_t op2, mpfr_rnd_t rnd)  [Function]
int mpfr_mul_q (mpfr_t rop, mpfr_t op1, mpq_t op2, mpfr_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 zeros, 0 is considered positive). The same restrictions than for mpfr_add_d apply to mpfr_mul_d.

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

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

```c
int mpfr_div (mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_rnd_t rnd)  [Function]
int mpfr_ui_div (mpfr_t rop, unsigned long int op1, mpfr_t op2, mpfr_rnd_t rnd)  [Function]
```
int mpfr_div_ui (mpfr_t rop, mpfr_t op1, unsigned long int op2, mpfr_rnd_t rnd) [Function]
int mpfr_si_div (mpfr_t rop, long int op1, mpfr_t op2, mpfr_rnd_t rnd) [Function]
int mpfr_div_si (mpfr_t rop, mpfr_t op1, long int op2, mpfr_rnd_t rnd) [Function]
int mpfr_d_div (mpfr_t rop, double op1, mpfr_t op2, mpfr_rnd_t rnd) [Function]
int mpfr_div_d (mpfr_t rop, mpfr_t op1, double op2, mpfr_rnd_t rnd) [Function]
int mpfr_div_z (mpfr_t rop, mpfr_t op1, mpz_t op2, mpfr_rnd_t rnd) [Function]
int mpfr_div_q (mpfr_t rop, mpfr_t op1, mpq_t op2, mpfr_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 zeros, 0 is considered positive; but note that if op1 is non-zero and op2 is zero, the result might change from ±Inf to NaN in future MPFR versions if there is an opposite decision on the IEEE 754 side. The same restrictions than for mpfr_add_d apply to mpfr_d_div and mpfr_div_d.

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

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

int mpfr_rec_sqrt (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]

Set rop to \(1/\sqrt{op}\) rounded in the direction rnd. Set rop to +Inf if op is ±0, +0 if op is +Inf, and NaN if op is negative. Warning! Therefore the result on \(-0\) is different from the one of the rSqrt function recommended by the IEEE 754-2008 standard (Section 9.2.1), which is \(-Inf\) instead of +Inf.

int mpfr_cbrt (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]
int mpfr_rootn_ui (mpfr_t rop, mpfr_t op, unsigned long int k, mpfr_rnd_t rnd) [Function]

Set rop to the cubic root (resp. the kth root) of op rounded in the direction rnd. For \(k = 0\), set rop to NaN. For \(k\) odd (resp. even) and op negative (including \(-Inf\)), set rop to a negative number (resp. NaN). If op is zero, set rop to zero with the sign obtained by the usual limit rules, i.e., the same sign as op if k is odd, and positive if k is even.

These functions agree with the rootn function of the IEEE 754-2008 standard (Section 9.2).

int mpfr_root (mpfr_t rop, mpfr_t op, unsigned long int k, mpfr_rnd_t rnd) [Function]

This function is the same as mpfr_rootn_ui except when op is \(-0\) and k is even: the result is \(-0\) instead of +0 (the reason was to be consistent with mpfr_sqrt). Said otherwise, if op is zero, set rop to op.

This function predates the IEEE 754-2008 standard and behaves differently from its rootn function. It is marked as deprecated and will be removed in a future release.

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

Set rop to \( op1^{op2} \), rounded in the direction rnd. Special values are handled as described in the ISO C99 and IEEE 754-2008 standards 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 y, even a NaN.
- pow(x, ±0) 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 or minus infinity for y negative, and plus infinity for y positive.

Note: When 0 is of integer type, it is regarded as +0 by these functions. We do not use the usual limit rules in this case, as these rules are not used for pow.

int mpfr_neg (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)

int mpfr_abs (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)

Set rop to \(-op\) and the absolute value of op respectively, rounded in the direction rnd. Just changes or adjusts the sign if rop and op are the same variable, otherwise a rounding might occur if the precision of rop is less than that of op.

The sign rule also applies to NaN in order to mimic the IEEE 754 negate and abs operations, i.e., for mpfr_neg, the sign is reversed, and for mpfr_abs, the sign is set to positive. But contrary to IEEE 754, the NaN flag is set as usual.

int mpfr_dim (mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_rnd_t rnd)

Set rop to the positive difference of op1 and op2, i.e., \( op1 - op2 \) rounded in the direction rnd if \( op1 > op2 \), +0 if \( op1 \leq op2 \), and NaN if \( op1 \) or \( op2 \) is NaN.

int mpfr_mul_2ui (mpfr_t rop, mpfr_t op1, unsigned long int op2, mpfr_rnd_t rnd)

int mpfr_mul_2si (mpfr_t rop, mpfr_t op1, long int op2, mpfr_rnd_t rnd)

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, mpfr_rnd_t rnd)

int mpfr_div_2si (mpfr_t rop, mpfr_t op1, long int op2, mpfr_rnd_t rnd)

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

```c
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, 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).

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

Compare op1 and op2 × 2^e. Similar as above.

```c
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.

```c
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]
int mpfr_regular_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), zero, or a regular number (i.e., neither NaN nor an infinity nor zero). Return zero otherwise.

```c
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. This is equivalent to mpfr_cmp_ui (op, 0), but more efficient.

```c
int mpfr_greater_p (mpfr_t op1, mpfr_t op2) [Function]
int mpfr_greaterequal_p (mpfr_t op1, mpfr_t op2) [Function]
int mpfr_less_p (mpfr_t op1, mpfr_t op2) [Function]
int mpfr_lessequal_p (mpfr_t op1, mpfr_t op2) [Function]
int mpfr_equal_p (mpfr_t op1, mpfr_t op2) [Function]
```

Return non-zero if op1 > op2, op1 ≥ op2, op1 < op2, op1 ≤ op2, op1 = op2 respectively, and zero otherwise. Those functions return zero whenever op1 and/or op2 is NaN.
5.7 Special Functions

All those functions, except explicitly stated (for example mpfr_sin_cos), return a [ternary value], page 8, i.e., 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 (even more with correct rounding) is expensive, even for small precision, for example the trigonometric and Bessel functions for large argument. For some functions, the memory usage might depend not only on the output precision: it is the case of the mpfr_rootn_ui function where the memory usage is also linear in the argument \( k \), and of the incomplete Gamma function (dependence on the precision of \( op \)).

```
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 \( op2 \) is NaN, or \( op1 = op2 \)).

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.
```

```
int mpfr_log (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]
int mpfr_log_ui (mpfr_t rop, unsigned long op, mpfr_rnd_t rnd) [Function]
int mpfr_log2 (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]
int mpfr_log10 (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]
   Set \( rop \) to the natural logarithm of \( op \), rounded in the direction \( rnd \). Set \( rop \) to +0 if \( op \) is 1 (in all rounding modes), for consistency with the ISO C99 and IEEE 754-2008 standards. Set \( rop \) to −Inf if \( op \) is ±0 (i.e., the sign of the zero has no influence on the result).

int mpfr_log1p (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]
   Set \( rop \) to the logarithm of one plus \( op \), rounded in the direction \( rnd \). Set \( rop \) to −Inf if \( op \) is −1.

int mpfr_exp (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]
int mpfr_exp2 (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]
int mpfr_exp10 (mpfr_t rop, mpfr_t op, mpfr_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_expm1 (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]
   Set \( rop \) to \( e^{op} - 1 \), rounded in the direction \( rnd \).

int mpfr_cos (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]
int mpfr_sin (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]
int mpfr_tan (mpfr_t rop, mpfr_t op, mpfr_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, mpfr_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 if both results are exact, more precisely it returns \( s + 4c \) where \( s = 0 \) if \( sop \) is exact, \( s = 1 \) if \( sop \) is larger than the sine of \( op \), \( s = 2 \) if \( sop \) is smaller than the sine of \( op \), and similarly for \( c \) and the cosine of \( op \).

int mpfr_sec (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]
int mpfr_csc (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd) [Function]```
int mpfr_cot (mpfr_t rop, mpfr_t op, mpfr_t rnd_t rnd)  [Function]
Set rop to the secant of op, cosecant of op, cotangent of op, rounded in the direction rnd.

int mpfr_acos (mpfr_t rop, mpfr_t op, mpfr_t rnd_t rnd)  [Function]
int mpfr_asin (mpfr_t rop, mpfr_t op, mpfr_t rnd_t rnd)  [Function]
int mpfr_atan (mpfr_t rop, mpfr_t op, mpfr_t 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 $\pi$ according to the given rounding mode, this number might not be in the output range $0 \leq \text{rop} < \pi$ 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) or for atan(op) with large op and small precision of rop.

int mpfr_atan2 (mpfr_t rop, mpfr_t y, mpfr_t x, mpfr_t rnd_t rnd)  [Function]
Set rop to the arc-tangent2 of y and x, rounded in the direction rnd: if $x > 0$, \(\text{atan2}(y, x) = \text{atan}(y/x)\); if $x < 0$, \(\text{atan2}(y, x) = \text{sign}(y) \cdot (\pi - \text{atan}(|y/x|))\), thus a number from $-\pi$ to $\pi$. 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 handled as described in the ISO C99 and IEEE 754-2008 standards for the atan2 function:

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

int mpfr_cosh (mpfr_t rop, mpfr_t op, mpfr_t rnd_t rnd)  [Function]
int mpfr_sinh (mpfr_t rop, mpfr_t op, mpfr_t rnd_t rnd)  [Function]
int mpfr_tanh (mpfr_t rop, mpfr_t op, mpfr_t rnd_t rnd)  [Function]
Set rop to the hyperbolic cosine, sine or tangent of op, rounded in the direction rnd.
int mpfr_sinh_cosh (mpfr_t sop, mpfr_t cop, mpfr_t op, mpfr_rnd_t rnd)
Set simultaneously sop to the hyperbolic sine of op and cop to the hyperbolic cosine of op, rounded in the direction rnd with the corresponding precision of sop and cop, which must be different variables. Return 0 iff both results are exact (see mpfr_sin_cos for a more detailed description of the return value).

int mpfr_sech (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)
int mpfr_csch (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)
int mpfr_coth (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)
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, mpfr_rnd_t rnd)
int mpfr_asinh (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)
int mpfr_atanh (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)
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, mpfr_rnd_t rnd)
Set rop to the factorial of op, rounded in the direction rnd.

int mpfr_eint (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)
Set rop to the exponential integral of op, rounded in the direction rnd. This is the sum of Euler’s constant, of the logarithm of the absolute value of op, and of the sum for k from 1 to infinity of op^k/(k · k!). For positive op, it corresponds to the E1 function at op (see formula 5.1.10 from the Handbook of Mathematical Functions from Abramowitz and Stegun), and for negative op, to the opposite of the E1 function (sometimes called eint1) at −op (formula 5.1.1 from the same reference).

int mpfr_li2 (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)
Set rop to real part of the dilogarithm of op, rounded in the direction rnd. MPFR defines the dilogarithm function as − \int_{t=0}^{op} \log(1−t)/t \, dt.

int mpfr_gamma (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)
int mpfr_gamma_inc (mpfr_t rop, mpfr_t op, mpfr_rnd_t op2, mpfr_rnd_t rnd)
Set rop to the value of the Gamma function on op, resp. the incomplete Gamma function on op and op2, rounded in the direction rnd. (In the literature, mpfr_gamma_inc is called upper incomplete Gamma function, or sometimes complementary incomplete Gamma function.) For mpfr_gamma (and mpfr_gamma_inc when op2 is zero), when op is a negative integer, rop is set to NaN.

Note: the current implementation of mpfr_gamma_inc is slow for large values of rop or op, in which case some internal overflow might also occur.

int mpfr_lngamma (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)
Set rop to the value of the logarithm of the Gamma function on op, rounded in the direction rnd. When op is 1 or 2, set rop to +0 (in all rounding modes). When op is an infinity or a nonpositive integer, set rop to +Inf, following the general rules on special values. When −2k − 1 < op < −2k, k being a nonnegative integer, set rop to NaN. See also mpfr_lgamma.
int mpfr_lgamma (mpfr_t rop, int *signp, mpfr_t op, mpfr_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 1 or 2, set rop to +0 (in all rounding modes). When op  
is an infinity or a nonpositive integer, set rop to +Inf. 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_digamma (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)  
[Function]
Set rop to the value of the Digamma (sometimes also called Psi) function on op, rounded in  
the direction rnd. When op is a negative integer, set rop to NaN.

int mpfr_beta (mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_rnd_t rnd)  
[Function]
Set rop to the value of the Beta function at arguments op1 and op2. Note: the current code  
does not try to avoid internal overflow or underflow, and might use a huge internal precision  
in some cases.

int mpfr_zeta (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)  
[Function]
int mpfr_zeta_ui (mpfr_t rop, unsigned long op, mpfr_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, mpfr_rnd_t rnd)  
[Function]
int mpfr_erfc (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)  
[Function]
Set rop to the value of the error function on op (resp. the complementary error function on  
op) rounded in the direction rnd.

int mpfr_j0 (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)  
[Function]
int mpfr_j1 (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)  
[Function]
int mpfr_jn (mpfr_t rop, long n, mpfr_t op, mpfr_rnd_t rnd)  
[Function]
Set rop to the value of the first kind Bessel function of order 0, (resp. 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 set to +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, mpfr_rnd_t rnd)  
[Function]
int mpfr_y1 (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)  
[Function]
int mpfr_yn (mpfr_t rop, long n, mpfr_t op, mpfr_rnd_t rnd)  
[Function]
Set rop to the value of the second kind Bessel function of order 0 (resp. 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 set to +0. When op is zero, rop is set to +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,  
mpfr_rnd_t rnd)  
[Function]
int mpfr_fms (mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_t op3,  
mpfr_rnd_t rnd)  
[Function]
Set rop to (op1 × op2)+op3 (resp. (op1 × op2)−op3) rounded in the direction rnd. Concerning  
special values (signed zeros, infinities, NaN), these functions behave like a multiplication  
followed by a separate addition or subtraction. That is, the fused operation matters only for  
rounding.
int mpfr_fmma (mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_t op3, mpfr_t op4, mpfr_round_t rnd) [Function]
int mpfr_fmms (mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_t op3, mpfr_t op4, mpfr_round_t rnd) [Function]

Set rop to (op1 × op2) + (op3 × op4) (resp. (op1 × op2) − (op3 × op4)) rounded in the direction rnd. In case the computation of op1 × op2 overflows or underflows (or that of op3 × op4), the result rop is computed as if the two intermediate products were computed with rounding toward zero.

int mpfr_agm (mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr_round_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 and the other one is not zero, set rop to NaN. If any operand is zero and the other one is finite (resp. infinite), set rop to +0 (resp. NaN).

int mpfr_hypot (mpfr_t rop, mpfr_t x, mpfr_t y, mpfr_round_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 handled as described in the ISO C99 (Section F.9.4.3) and IEEE 754-2008 (Section 9.2.1) standards: If x or y is an infinity, then +Inf is returned in rop, even if the other number is NaN.

int mpfr_ai (mpfr_t rop, mpfr_t x, mpfr_round_t rnd) [Function]

Set rop to the value of the Airy function Ai on x, rounded in the direction rnd. When x is NaN, rop is always set to NaN. When x is +Inf or −Inf, rop is +0. The current implementation is not intended to be used with large arguments. It works with — x — typically smaller than 500. For larger arguments, other methods should be used and will be implemented in a future version.

int mpfr_const_log2 (mpfr_t rop, mpfr_round_t rnd) [Function]
int mpfr_const_pi (mpfr_t rop, mpfr_round_t rnd) [Function]
int mpfr_const_euler (mpfr_t rop, mpfr_round_t rnd) [Function]
int mpfr_const_catalan (mpfr_t rop, mpfr_round_t rnd) [Function]

Set rop to the logarithm of 2, the value of \( \pi \), 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 or mpfr_free_cache2.

void mpfr_free_cache (void) [Function]

Free all caches and pools used by MPFR internally (thoses local to the current thread and those shared by all threads). You should call this function before terminating a thread, even if you did not call mpfr_const_* functions directly (they could have been called internally).

void mpfr_free_cache2 (mpfr_free_cache_t way) [Function]

Free various caches and pools used by MPFR internally, as specified by way, which is a set of flags:

- those local to the current thread if flag MPFR_FREE_LOCAL_CACHE is set;
- those shared by all threads if flag MPFR_FREE_GLOBAL_CACHE is set.

The other bits of way are currently ignored and are reserved for future use; they should be zero.
Note: \texttt{mpfr\_free\_cache2(MPFR\_FREE\_LOCAL\_CACHE|MPFR\_FREE\_GLOBAL\_CACHE)} is currently equivalent to \texttt{mpfr\_free\_cache(\)}. 

\begin{quote}
\textbf{void mpfr\_free\_pool (void) [Function]}
Free the pools used by MPFR internally. Note: This function is automatically called after the thread-local caches are freed (with \texttt{mpfr\_free\_cache} or \texttt{mpfr\_free\_cache2}).
\end{quote}

\begin{quote}
\textbf{int mpfr\_mp\_memory\_cleanup (void) [Function]}
This function should be called before calling \texttt{mp\_set\_memory\_functions}. See Section 4.7 [Memory Handling], page 10, for more information. Zero is returned in case of success, non-zero in case of error. Errors are currently not possible, but checking the return value is recommended for future compatibility.
\end{quote}

\begin{quote}
\textbf{int mpfr\_sum (mpfr\_t rop, const mpfr\_ptr tab[], unsigned long int n, mpfr\_rnd\_t rnd) [Function]}
Set \textit{rop} to the sum of all elements of \textit{tab}, whose size is \textit{n}, correctly rounded in the direction \textit{rnd}. Warning: for efficiency reasons, \textit{tab} is an array of pointers to mpfr\_t, not an array of mpfr\_t. If \textit{n} = 0, then the result is +0, and if \textit{n} = 1, then the function is equivalent to mpfr\_set. For the special exact cases, the result is the same as the one obtained with a succession of additions (mpfr\_add) in infinite precision. In particular, if the result is an exact zero and \textit{n} \geq 1:
\begin{itemize}
  \item if all the inputs have the same sign (i.e., all +0 or all −0), then the result has the same sign as the inputs;
  \item otherwise, either because all inputs are zeros with at least a +0 and a −0, or because some inputs are non-zero (but they globally cancel), the result is +0, except for the MPFR\_RNDD rounding mode, where it is −0.
\end{itemize}
\end{quote}

### 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 \texttt{stream} to any of these functions will make them read from stdin and write to stdout, respectively.

When using a function that takes a \texttt{FILE *} argument, you must include the \texttt{<stdio.h>} standard header before mpfr\_h, to allow mpfr\_h to define prototypes for these functions.

\begin{quote}
\textbf{size\_t mpfr\_out\_str (FILE *stream, int base, size\_t n, mpfr\_t op, mpfr\_rnd\_t rnd) [Function]}
Output \textit{op} on stream \textit{stream}, as a string of digits in base \textit{base}, rounded in the direction \textit{rnd}. The base may vary from 2 to 62. Print \textit{n} significant digits exactly, or if \textit{n} is 0, enough digits so that \textit{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 \textit{base} is greater than 10, ‘@’ will be used instead of ‘e’ as exponent delimiter.

Return the number of characters written, or if an error occurred, return 0.
\end{quote}

\begin{quote}
\textbf{size\_t mpfr\_inp\_str (mpfr\_t rop, FILE *stream, int base, mpfr\_rnd\_t rnd) [Function]}
Input a string in base \textit{base} from stream \textit{stream}, rounded in the direction \textit{rnd}, and put the read float in \textit{rop}.
\end{quote}
This function reads a word (defined as a sequence of characters between whitespace) and parses it using `mpfr_set_str`. 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.

```c
int mpfr_fpif_export (FILE *stream, mpfr_t op) [Function]
Export the number `op` to the stream `stream` in a floating-point interchange format. In particular one can export on a 32-bit computer and import on a 64-bit computer, or export on a little-endian computer and import on a big-endian computer. The precision of `op` and the sign bit of a NaN are stored too. Return 0 iff the export was successful.

Note: this function is experimental and its interface might change in future versions.
```

```c
int mpfr_fpif_import (mpfr_t op, FILE *stream) [Function]
Import the number `op` from the stream `stream` in a floating-point interchange format (see `mpfr_fpif_export`). Note that the precision of `op` is set to the one read from the stream, and the sign bit is always retrieved (even for NaN). If the stored precision is zero or greater than `MPFR_PREC_MAX`, the function fails (it returns non-zero) and `op` is unchanged. If the function fails for another reason, `op` is set to NaN and it is unspecified whether the precision of `op` has changed to the one read from the file. Return 0 iff the import was successful.

Note: this function is experimental and its interface might change in future versions.
```

```c
void mpfr_dump (mpfr_t op) [Function]
Output `op` on stdout in some unspecified format, then a newline character. This function is mainly for debugging purpose. Thus invalid data may be supported. Everything that is not specified may change without breaking the ABI and may depend on the environment.

The current output format is the following: a minus sign if the sign bit is set (even for NaN); ‘@NaN@’, ‘@Inf@’ or ‘0’ if the argument is NaN, an infinity or zero, respectively; otherwise the remaining of the output is as follows: ‘0.’ then the `p` bits of the binary significand, where `p` is the precision of the number; if the trailing bits are not all zeros (which must not occur with valid data), they are output enclosed by square brackets; the character ‘E’ followed by the exponent written in base 10; in case of invalid data or out-of-range exponent, this function outputs three exclamation marks (‘!!!’), followed by flags, followed by three exclamation marks (‘!!!’) again. These flags are: ‘N’ if the most significant bit of the significand is 0 (i.e., the number is not normalized); ‘T’ if there are non-zero trailing bits; ‘U’ if this is a UBF number (internal use only); ‘<’ if the exponent is less than the current minimum exponent; ‘>’ if the exponent is greater than the current maximum exponent.
```

### 5.9 Formatted Output Functions

#### 5.9.1 Requirements

The class of `mpfr_printf` functions provides formatted output in a similar manner as the standard C `printf`. These functions are defined only if your system supports ISO C variadic functions and the corresponding argument access macros.

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.

#### 5.9.2 Format String

The format specification accepted by `mpfr_printf` is an extension of the `printf` one. The conversion specification is of the form:
% [flags] [width] [.precision] [type] [rounding] conv

‘flags’, ‘width’, and ‘precision’ have the same meaning as for the standard printf (in particular, notice that the ‘precision’ is related to the number of digits displayed in the base chosen by ‘conv’ and not related to the internal precision of the mpfr_t variable), but note that for ‘Re’, the default precision is not the same as the one for ‘e’. mpfr_printf accepts the same ‘type’ specifiers as GMP (except the non-standard and deprecated ‘q’, use ‘ll’ instead), namely the length modifiers defined in the C standard:

‘h’    short
‘hh’   char
‘j’    intmax_t or uintmax_t
‘l’    long or wchar_t
‘ll’   long long
‘L’    long double
‘t’    ptrdiff_t
‘z’    size_t

and the ‘type’ specifiers defined in GMP plus ‘R’ and ‘P’ specific to MPFR (the second column in the table below shows the type of the argument read in the argument list and the kind of ‘conv’ specifier to use after the ‘type’ specifier):

‘F’    mpfr_t, float conversions
‘Q’    mpq_t, integer conversions
‘M’    mp_limb_t, integer conversions
‘N’    mp_limb_t array, integer conversions
‘Z’    mpz_t, integer conversions
‘P’    mpfr_prec_t, integer conversions
‘R’    mpfr_t, float conversions

The ‘type’ specifiers have the same restrictions as those mentioned in the GMP documentation: see Section “Formatted Output Strings” in GNU MP. In particular, the ‘type’ specifiers (except ‘R’ and ‘P’) are supported only if they are supported by gmp_printf in your GMP build; this implies that the standard specifiers, such as ‘t’, must also be supported by your C library if you want to use them.

The ‘rounding’ field is specific to mpfr_t arguments and should not be used with other types.

With conversion specification not involving ‘P’ and ‘R’ types, mpfr_printf behaves exactly as gmp_printf.

The ‘P’ type specifies that a following ‘d’, ‘i’, ‘o’, ‘u’, ‘x’, or ‘X’ conversion specifier applies to a mpfr_prec_t argument. It is needed because the mpfr_prec_t type does not necessarily correspond to an int or any fixed standard type. The ‘precision’ field specifies the minimum number of digits to appear. The default ‘precision’ is 1. For example:

```c
mpfr_t x;
mpfr_prec_t p;
mpfr_init (x);
...
p = mpfr_get_prec (x);
mpfr_printf ("variable x with %Pu bits", p);
```
The ‘R’ type specifies that a following ‘a’, ‘A’, ‘b’, ‘e’, ‘E’, ‘f’, ‘F’, ‘g’, ‘G’, or ‘n’ conversion specifier applies to a mpfr_t argument. The ‘R’ type can be followed by a ‘rounding’ specifier denoted by one of the following characters:

- ‘U’  round toward plus infinity
- ‘D’  round toward minus infinity
- ‘Y’  round away from zero
- ‘Z’  round toward zero
- ‘N’  round to nearest (with ties to even)
- ‘*’  rounding mode indicated by the mpfr_rnd_t argument just before the corresponding mpfr_t variable.

The default rounding mode is rounding to nearest. The following three examples are equivalent:

```c
mpfr_t x;
mpfr_init (x);
...
mpfr_printf ("%.128Rf", x);
mpfr_printf ("%.128RNf", x);
mpfr_printf ("%.128R*f", MPFR_RNDN, x);
```

Note that the rounding away from zero mode is specified with ‘Y’ because ISO C reserves the ‘A’ specifier for hexadecimal output (see below).

The output ‘conv’ specifiers allowed with mpfr_t parameter are:

- ‘a’  ‘A’  hex float, C99 style
- ‘b’  binary output
- ‘e’  ‘E’  scientific-format float
- ‘f’  ‘F’  fixed-point float
- ‘g’  ‘G’  fixed-point or scientific float

The conversion specifier ‘b’ which displays the argument in binary is specific to mpfr_t arguments and should not be used with other types. Other conversion specifiers have the same meaning as for a double argument.

In case of non-decimal output, only the significand is written in the specified base, the exponent is always displayed in decimal. Special values are always displayed as nan, -inf, and inf for ‘a’, ‘b’, ‘e’, ‘f’, and ‘g’ specifiers and NAN, -INF, and INF for ‘A’, ‘E’, ‘F’, and ‘G’ specifiers.

If the ‘precision’ field is not empty, the mpfr_t number is rounded to the given precision in the direction specified by the rounding mode. If the precision is zero with rounding to nearest mode and one of the following ‘conv’ specifiers: ‘a’, ‘A’, ‘b’, ‘e’, ‘E’, tie case is rounded to even when it lies between two consecutive values at the wanted precision which have the same exponent, otherwise, it is rounded away from zero. For instance, 85 is displayed as "8e+1" and 95 is displayed as "1e+2" with the format specification "%.0RNe". This also applies when the ‘g’ (resp. ‘G’) conversion specifier uses the ‘e’ (resp. ‘E’) style. If the precision is set to a value greater than the maximum value for an int, it will be silently reduced down to INT_MAX.

If the ‘precision’ field is empty (as in %Re or %.RE) with ‘conv’ specifier ‘e’ and ‘E’, the number is displayed with enough digits so that it can be read back exactly, assuming that the input and output variables have the same precision and that the input and output rounding modes are both rounding to nearest (as for mpfr_get_str). The default precision for an empty ‘precision’ field with ‘conv’ specifiers ‘f’, ‘F’, ‘g’, and ‘G’ is 6.
5.9.3 Functions

For all the following functions, if the number of characters that ought to be written exceeds the maximum limit INT_MAX for an int, nothing is written in the stream (resp. to stdout, to buf, to str), the function returns \(-1\), sets the erange flag, and errno is set to EOVERFLOW if the EOVERFLOW macro is defined (such as on POSIX systems). Note, however, that errno might be changed to another value by some internal library call if another error occurs there (currently, this would come from the unallocation function).

\[
\begin{align*}
\text{int mpfr_fprintf (FILE *stream, const char *template, \ldots)} & \quad \text{[Function]} \\
\text{int mpfr_vfprintf (FILE *stream, const char *template, va_list ap)} & \quad \text{[Function]} \\
\end{align*}
\]

Print to the stream stream the optional arguments under the control of the template string template. Return the number of characters written or a negative value if an error occurred.

\[
\begin{align*}
\text{int mpfr_printf (const char *template, \ldots)} & \quad \text{[Function]} \\
\text{int mpfr_vprintf (const char *template, va_list ap)} & \quad \text{[Function]} \\
\end{align*}
\]

Print to stdout the optional arguments under the control of the template string template. Return the number of characters written or a negative value if an error occurred.

\[
\begin{align*}
\text{int mpfr_sprintf (char *buf, const char *template, \ldots)} & \quad \text{[Function]} \\
\text{int mpfr_vsprintf (char *buf, const char *template, va_list ap)} & \quad \text{[Function]} \\
\end{align*}
\]

Form a null-terminated string corresponding to the optional arguments under the control of the template string template, and print it in buf. No overlap is permitted between buf and the other arguments. Return the number of characters written in the array buf not counting the terminating null character or a negative value if an error occurred.

\[
\begin{align*}
\text{int mpfr_snprintf (char *buf, size_t n, const char *template, \ldots)} & \quad \text{[Function]} \\
\text{int mpfr_vsnprintf (char *buf, size_t n, const char *template, va_list ap)} & \quad \text{[Function]} \\
\end{align*}
\]

Form a null-terminated string corresponding to the optional arguments under the control of the template string template, and print it in buf. If \(n\) is zero, nothing is written and buf may be a null pointer, otherwise, the \((n-1)\) first characters are written in buf and the \(n\)-th is a null character. Return the number of characters that would have been written had \(n\) been sufficiently large, not counting the terminating null character, or a negative value if an error occurred.

\[
\begin{align*}
\text{int mpfr_asprintf (char **str, const char *template, \ldots)} & \quad \text{[Function]} \\
\text{int mpfr_vasprintf (char **str, const char *template, va_list ap)} & \quad \text{[Function]} \\
\end{align*}
\]

Write their output as a null terminated string in a block of memory allocated using the allocation function (see Section 4.7 [Memory Handling], page 10). A pointer to the block is stored in str. The block of memory must be freed using mpfr_free_str. The return value is the number of characters written in the string, excluding the null-terminator, or a negative value if an error occurred, in which case the contents of str are undefined.

5.10 Integer and Remainder Related Functions

\[
\begin{align*}
\text{int mpfr_rint (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)} & \quad \text{[Function]} \\
\text{int mpfr ceil (mpfr_t rop, mpfr_t op)} & \quad \text{[Function]} \\
\text{int mpfr_floor (mpfr_t rop, mpfr_t op)} & \quad \text{[Function]} \\
\text{int mpfr_round (mpfr_t rop, mpfr_t op)} & \quad \text{[Function]} \\
\text{int mpfr_roundeven (mpfr_t rop, mpfr_t op)} & \quad \text{[Function]} \\
\end{align*}
\]
void mpfr_trunc (mpfr_t rop, mpfr_t op)

Set rop to op rounded to an integer. mpfr_rint rounds to the nearest representable integer in the given direction rnd, and the other five functions behave in a similar way with some fixed rounding mode:

- mpfr ceil: to the next higher or equal representable integer (like mpfr_rint with MPFR_RNDU);
- mpfr floor: to the next lower or equal representable integer (like mpfr_rint with MPFR_RNDD);
- mpfr round: to the nearest representable integer, rounding halfway cases away from zero (as in the roundTiesToAway mode of IEEE 754-2008);
- mpfr roundeven: to the nearest representable integer, rounding halfway cases with the even-rounding rule (like mpfr_rint with MPFR_RNDN);
- mpfr trunc: to the next representable integer toward zero (like mpfr_rint with MPFR_RNDZ).

When op is a zero or an infinity, set rop to the same value (with the same sign).

The return 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 return 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.

When op is NaN, the NaN flag is set as usual. In the other cases, the inexact flag is set when rop differs from op, following the ISO C99 rule for the rint function. If you want the behavior to be more like IEEE 754 / ISO TS 18661-1, i.e., the usual behavior where the round-to-integer function is regarded as any other mathematical function, you should use one of the mpfr_rint_ functions instead.

Note that no double rounding is performed; for instance, 10.5 (1010.1 in binary) is rounded by mpfr_rint with rounding to nearest to 12 (1100 in binary) in 2-bit precision, because the two enclosing numbers representable on two bits are 8 and 12, and the closest is 12. (If one first rounded to an integer, one would round 10.5 to 10 with even rounding, and then 10 would be rounded to 8 again with even rounding.)
Contrary to \texttt{mpfr\_rint}, those functions do perform a double rounding: first \textit{op} is rounded to the nearest integer in the direction given by the function name, then this nearest integer (if not representable) is rounded in the given direction \textit{rnd}. Thus these round-to-integer functions behave more like the other mathematical functions, i.e., the returned result is the correct rounding of the exact result of the function in the real numbers.

For example, \texttt{mpfr\_rint\_round} with rounding to nearest and a precision of two bits rounds 6.5 to 7 (halfway cases away from zero), then 7 is rounded to 8 by the round-even rule, despite the fact that 6 is also representable on two bits, and is closer to 6.5 than 8.

\begin{verbatim}
int mpfr_frac (mpfr_t rop, mpfr_t op, mpfr_rnd_t rnd)
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). When op is an integer or an infinity, set rop to zero with the same sign as op.

int mpfr_modf (mpfr_t iop, mpfr_t fop, mpfr_t op, mpfr_rnd_t rnd)
Set simultaneously iop to the integral part of op and fop to the fractional part of op, rounded in the direction rnd with the corresponding precision of iop and fop (equivalent to mpfr\_trunc(iop, op, rnd) and mpfr\_frac(fop, op, rnd)). The variables iop and fop must be different. Return 0 iff both results are exact (see mpfr\_sin\_cos for a more detailed description of the return value).

int mpfr_fmod (mpfr_t r, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)
int mpfr_fmodquo (mpfr_t r, long* q, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)
int mpfr_remainder (mpfr_t r, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)
int mpfr_remquo (mpfr_t r, long* q, mpfr_t x, mpfr_t y, mpfr_rnd_t rnd)
Set \textit{r} to the value of \textit{x} − \textit{ny}, rounded according to the direction \textit{rnd}, where \textit{n} is the integer quotient of \textit{x} divided by \textit{y}, defined as follows: \textit{n} is rounded toward zero for mpfr\_fmod and mpfr\_fmodquo, and to the nearest integer (ties rounded to even) for mpfr\_remainder and mpfr\_remquo.

Special values are handled as described in Section F.9.7.1 of the ISO C99 standard: If \textit{x} is infinite or \textit{y} is zero, \textit{r} is NaN. If \textit{y} is infinite and \textit{x} is finite, \textit{r} is \textit{x} rounded to the precision of \textit{r}. If \textit{r} is zero, it has the sign of \textit{x}. The return value is the ternary value corresponding to \textit{r}.

Additionally, mpfr\_fmodquo and mpfr\_remquo store the low significant bits from the quotient \textit{n} in *q (more precisely the number of bits in a \texttt{long} minus one), with the sign of \textit{x} divided by \textit{y} (except if those low bits are all zero, in which case zero is returned). Note that \textit{x} may be so large in magnitude relative to \textit{y} that an exact representation of the quotient is not practical. The mpfr\_remainder and mpfr\_remquo functions are useful for additive argument reduction.

int mpfr_integer_p (mpfr_t op)
Return non-zero iff \textit{op} is an integer.
\end{verbatim}

5.11 Rounding-Related Functions

\begin{verbatim}
void mpfr_set_default_rounding_mode (mpfr_rnd_t rnd)
Set the default rounding mode to \textit{rnd}. The default rounding mode is to nearest initially.
\end{verbatim}
mpfr_rnd_t mpfr_get_default_rounding_mode (void)  
Get the default rounding mode.

int mpfr_prec_round (mpfr_t x, mpfr_prec_t prec, mpfr_rnd_t rnd)  
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.

Here is an example of how to use mpfr_prec_round to implement Newton’s algorithm to compute the inverse of a, assuming x is already an approximation to n bits:

mpfr_set_prec (t, 2 * n);
mpfr_set (t, a, MPFR_RNDN);  /* round a to 2n bits */
mpfr_mul (t, t, x, MPFR_RNDN);  /* t is correct to 2n bits */
mpfr_ui_sub (t, 1, t, MPFR_RNDN);  /* high n bits cancel with 1 */
mpfr_prec_round (t, n, MPFR_RNDN);  /* t is correct to n bits */
mpfr_mul (t, t, x, MPFR_RNDN);  /* t is correct to n bits */
mpfr_prec_round (x, 2 * n, MPFR_RNDN);  /* exact */
mpfr_add (x, x, t, MPFR_RNDN);  /* x is correct to 2n bits */

Warning! You must not use this function if x was initialized with MPFR_DECL_INIT or with mpfr_custom_init_set (see Section 5.15 [Custom Interface], page 45).

int mpfr_can_round (mpfr_t b, mpfr_exp_t err, mpfr_rnd_t rnd1,  
                   mpfr_rnd_t rnd2, mpfr_prec_t prec)  
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 assuming an unbounded exponent range, and 0 otherwise (including for NaN and Inf). In other words, if the error on b is bounded by two to the power k ulps, and b has precision prec, you should give err = prec − k. This function does not modify its arguments.

If rnd1 is MPFR_RNDN or MPFR_RNDF, the error is considered to be either positive or negative, thus the possible range is twice as large as with a directed rounding for rnd1 (with the same value of err).

When rnd2 is MPFR_RNDF, let rnd3 be the opposite direction if rnd1 is a directed rounding, and MPFR_RNDN if rnd1 is MPFR_RNDN or MPFR_RNDF. The returned value of mpfr_can_round (b, err, rnd1, MPFR_RNDF, prec) is non-zero iff after the call mpfr_set (y, b, rnd3) with y of precision prec, y is guaranteed to be a faithful rounding of x.

Note: The [ternary value], page 8, cannot be determined in general with this function. However, if it is known that the exact value is not exactly representable in precision prec, then one can use the following trick to determine the (non-zero) ternary value in any rounding mode rnd2 (note that MPFR_RNDZ below can be replaced by any directed rounding mode):

if (mpfr_can_round (b, err, MPFR_RNDN, MPFR_RNDZ,  
                   prec + (rnd2 == MPFR_RNDN)))
{
    /* round the approximation 'b' to the result 'r' of 'prec' bits  
       with rounding mode 'rnd2' and get the ternary value 'inex' */
    inex = mpfr_set (r, b, rnd2);
}
Indeed, if \texttt{rnd2} is \texttt{MPFR\_RNDN}, this will check if one can round to \texttt{prec+1} bits with a directed rounding: if so, one can surely round to nearest to \texttt{prec} bits, and in addition one can determine the correct ternary value, which would not be the case when \texttt{b} is near from a value exactly representable on \texttt{prec} bits.

A detailed example is available in the \texttt{examples} subdirectory, file \texttt{can\_round.c}.

\begin{verbatim}
mpfr_prec_t mpfr_min_prec (mpfr_t x)  \hspace{1cm} [Function]
\quad Return the minimal number of bits required to store the significand of \texttt{x}, and 0 for special values, including 0.

const char * mpfr_print_rnd_mode (mpfr_rnd_t rnd)  \hspace{1cm} [Function]
\quad Return a string ("MPFR\_RNDD", "MPFR\_RNDU", "MPFR\_RNDN", "MPFR\_RNDZ", "MPFR\_RNDZ") corresponding to the rounding mode \texttt{rnd}, or a null pointer if \texttt{rnd} is an invalid rounding mode.

int mpfr_round_nearest_away (int (foo)(mpfr_t, type1_t, ..., mpfr_rnd_t), mpfr_t rop, type1_t op, ...)  \hspace{1cm} [Macro]
\quad Given a function \texttt{foo} and one or more values \texttt{op} (which may be a mpfr_t, a long, a double, etc.), put in \texttt{rop} the round-to-nearest-away rounding of \texttt{foo(op,...)}. This rounding is defined in the same way as round-to-nearest-even, except in case of tie, where the value away from zero is returned. The function \texttt{foo} takes as input, from second to penultimate argument(s), the argument list given after \texttt{rop}, a rounding mode as final argument, puts in its first argument the value \texttt{foo(op,...)} rounded according to this rounding mode, and returns the corresponding ternary value (which is expected to be correct, otherwise \texttt{mpfr\_round\_nearest\_away} will not work as desired). Due to implementation constraints, this function must not be called when the minimal exponent \texttt{emin} is the smallest possible one. This macro has been made such that the compiler is able to detect mismatch between the argument list \texttt{op} and the function prototype of \texttt{foo}. Multiple input arguments \texttt{op} are supported only with C99 compilers. Otherwise, for C89 compilers, only one such argument is supported.

Note: this macro is experimental and its interface might change in future versions.

\begin{verbatim}
unsigned long ul;
mpfr_t f, r;
/* Code that inits and sets r, f, and ul, and if needed sets emin */
int i = mpfr_round_nearest_away (mpfr_add_ui, r, f, ul);
\end{verbatim}

\section{5.12 Miscellaneous Functions}

\begin{verbatim}
void mpfr_nexttoward (mpfr_t x, mpfr_t y)  \hspace{1cm} [Function]
\quad If \texttt{x} or \texttt{y} is NaN, set \texttt{x} to NaN; note that the NaN flag is set as usual. If \texttt{x} and \texttt{y} are equal, \texttt{x} is unchanged. Otherwise, if \texttt{x} is different from \texttt{y}, replace \texttt{x} by the next floating-point number (with the precision of \texttt{x} and the current exponent range) in the direction of \texttt{y} (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, overflow, or inexact exception is raised.

void mpfr_nextabove (mpfr_t x)  \hspace{1cm} [Function]
void mpfr_nextbelow (mpfr_t x)  \hspace{1cm} [Function]
\quad Equivalent to \texttt{mpfr\_nexttoward} where \texttt{y} is plus infinity (resp. minus infinity).
\end{verbatim}
int mpfr_min (mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr rnd_t rnd)  [Function]
int mpfr_max (mpfr_t rop, mpfr_t op1, mpfr_t op2, mpfr rnd_t rnd)  [Function]

Set rop to the minimum (resp. 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 (resp. +0).

int mpfr urandomb (mpfr_t rop, gmp randstate_t state)  [Function]

Generate a uniformly distributed random float in the interval $0 \leq \text{rop} < 1$. More precisely, the number can be seen as a float with a random non-normalized significand and exponent 0, which is then normalized (thus if $e$ denotes the exponent after normalization, then the least $-e$ significant bits of the significand are always 0).

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 (this should never happen in practice, except in very specific cases). The second argument is a gmp randstate_t structure which should be created using the GMP gmp_randinit function (see the GMP manual).

Note: for a given version of MPFR, the returned value of rop and the new value of state (which controls further random values) do not depend on the machine word size.

int mpfr urandom (mpfr_t rop, gmp randstate_t state, mpfr rnd_t rnd)  [Function]

Generate a uniformly distributed random float. The floating-point number rop can be seen as if a random real number is generated according to the continuous uniform distribution on the interval $[0, 1]$ and then rounded in the direction rnd.

The second argument is a gmp randstate_t structure which should be created using the GMP gmp_randinit function (see the GMP manual).

Note: the note for mpfr urandomb holds too. Moreover, the exact number (the random value to be rounded) and the next random state do not depend on the current exponent range and the rounding mode. However, they depend on the target precision: from the same state of the random generator, if the precision of the destination is changed, then the value may be completely different (and the state of the random generator is different too).

int mpfr nrandom (mpfr_t rop1, gmp randstate_t state, mpfr rnd_t rnd)
int mpfr grandom (mpfr_t rop1, mpfr_t rop2, gmp randstate_t state, mpfr rnd_t rnd)  [Function]

Generate one (possibly two for mpfr grandom) random floating-point number according to a standard normal Gaussian distribution (with mean zero and variance one). For mpfr grandom, if rop2 is a null pointer, then only one value is generated and stored in rop1.

The floating-point number rop1 (and rop2) can be seen as if a random real number were generated according to the standard normal Gaussian distribution and then rounded in the direction rnd.

The gmp randstate_t argument should be created using the GMP gmp randinit function (see the GMP manual).

For mpfr grandom, the combination of the ternary values is returned like with mpfr sin cos. If rop2 is a null pointer, the second ternary value is assumed to be 0 (note that the encoding of the only ternary value is not the same as the usual encoding for functions that return only one result). Otherwise the ternary value of a random number is always non-zero.
Note: the note for `mpfr_urandomb` holds too. In addition, the exponent range and the rounding mode might have a side effect on the next random state.

Note: `mpfr_nrandom` is much more efficient than `mpfr_grandom`, especially for large precision. Thus `mpfr_grandom` is marked as deprecated and will be removed in a future release.

```c
int mpfr_erandom (mpfr_t rop1, gmp_randstate_t state, mpfr_rnd_t rnd)
```
Generate one random floating-point number according to an exponential distribution, with mean one. Other characteristics are identical to `mpfr_nrandom`.

```c
mpfr_exp_t mpfr_get_exp (mpfr_t x)
```
Return the exponent of `x`, assuming that `x` is a non-zero ordinary number and the significand is considered in \([1/2,1)\). For this function, `x` is allowed to be outside of the current range of acceptable values. The behavior for NaN, infinity or zero is undefined.

```c
int mpfr_set_exp (mpfr_t x, mpfr_exp_t e)
```
Set the exponent of `x` to `e` if `x` is a non-zero ordinary number and `e` is in the current exponent range, and return 0; otherwise, return a non-zero value (`x` is not changed).

```c
int mpfr_signbit (mpfr_t op)
```
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).

```c
int mpfr_setsign (mpfr.t rop, mpfr.t op, int s, mpfr_rnd.t rnd)
```
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.

```c
int mpfr_copysign (mpfr.t rop, mpfr.t op1, mpfr.t op2, mpfr_rnd.t rnd)
```
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)`.

```c
const char * mpfr_get_version (void)
```
Return the MPFR version, as a null-terminated string.

```
MPFR_VERSION
MPFR_VERSION_MAJOR
MPFR_VERSION_MINOR
MPFR_VERSION_PATCHLEVEL
MPFR_VERSION_STRING
```
`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:

```c
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).

```c
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:

```c
    #if (!defined(MPFR_VERSION) || (MPFR_VERSION<MPFR_VERSION_NUM(3,0,0)))
    # error "Wrong MPFR version."
    #endif
```

```c
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 identifiers of the patches applied to the old (compile-time) MPFR version are not available (however this information should not have much interest in general).
```

```c
int mpfr_buildopt_threadsafe_p (void) [Function]
Return a non-zero value if MPFR was compiled as thread safe using compiler-level Thread Local Storage (that is, MPFR was built with the ‘--enable-thread-safe’ configure option, see INSTALL file), return zero otherwise.
```

```c
int mpfr_buildopt_float128_p (void) [Function]
Return a non-zero value if MPFR was compiled with ‘__float128’ support (that is, MPFR was built with the ‘--enable-float128’ configure option), return zero otherwise.
```

```c
int mpfr_buildopt_decimal_p (void) [Function]
Return a non-zero value if MPFR was compiled with decimal float support (that is, MPFR was built with the ‘--enable-decimal-float’ configure option), return zero otherwise.
```

```c
int mpfr_buildopt_gmpinternals_p (void) [Function]
Return a non-zero value if MPFR was compiled with GMP internals (that is, MPFR was built with either ‘--with-gmp-build’ or ‘--enable-gmp-internals’ configure option), return zero otherwise.
```

```c
int mpfr_buildopt_sharedcache_p (void) [Function]
Return a non-zero value if MPFR was compiled so that all threads share the same cache for one MPFR constant, like mpfr_const_pi or mpfr_const_log2 (that is, MPFR was built with the ‘--enable-shared-cache’ configure option), return zero otherwise. If the return value is non-zero, MPFR applications may need to be compiled with the ‘-pthread’ option.
```

```c
const char * mpfr_buildopt_tune_case (void) [Function]
Return a string saying which thresholds file has been used at compile time. This file is normally selected from the processor type.
```

### 5.13 Exception Related Functions

```c
mpfr_exp_t mpfr_get_emin (void) [Function]
```
mpfr_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 $1/2 \times 2^{\text{emin}}$ and the largest value has the
form $(1 - \varepsilon) \times 2^{\text{emax}}$, where $\varepsilon$ depends on the precision of the considered variable.

int mpfr_set_emin (mpfr_exp_t exp)                           [Function]
int mpfr_set_emax (mpfr_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.

For the subsequent operations, it is the user’s responsibility to check that any floating-point
value used as an input is in the new exponent range (for example using mpfr_check_range).
If a floating-point value outside the new exponent range is used as an input, the default
behavior is undefined, in the sense of the ISO C standard; the behavior may also be explicitly
documented, such as for mpfr_check_range.

Note: Caches may still have values outside the current exponent range. This is not an issue
as the user cannot use these caches directly via the API (MPFR extends the exponent range
internally when need be).

If emin > emax and a floating-point value needs to be produced as output, the behavior is
undefined (mpfr_set_emin and mpfr_set_emax do not check this condition as it might occur
between successive calls to these two functions).

mpfr_exp_t mpfr_get_emin_min (void)                           [Function]
mpfr_exp_t mpfr_get_emin_max (void)                           [Function]
mpfr_exp_t mpfr_get_emax_min (void)                           [Function]
mpfr_exp_t mpfr_get_emax_max (void)                           [Function]
Return the minimum and maximum of the exponents allowed for mpfr_set_emin and
mpfr_set_emax respectively. These values are implementation dependent, thus a program
using mpfr_set_emax(mpfr_get_emax_max()) or mpfr_set_emin(mpfr_get_emin_min())
may not be portable.

int mpfr_check_range (mpfr_t x, int t, mpfr_rnd_t rnd)           [Function]
This function assumes that x is the correctly rounded value of some real value y in the
direction rnd and some extended exponent range, and that t is the corresponding [ternary
value], page 8. For example, one performed $t = \text{mpfr_log} (x, u, \text{rnd})$, and y is the exact
logarithm of u. Thus t is negative if x is smaller than y, positive if x is larger than y, and
zero if x equals y. This function modifies x if needed to be in the current range of acceptable
values: 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 new value of x equals the exact one y; a positive value if that new value is larger
than y, and a negative value if it is smaller than y. Note that unlike most functions, the new
result x is compared to the (unknown) exact one y, not the input value x, i.e., the ternary
value is propagated.

Note: If x is an infinity and t is different from zero (i.e., if the rounded result is an inexact
infinity), then the overflow flag is set. This is useful because mpfr_check_range is typically
called (at least in MPFR functions) after restoring the flags that could have been set due to
internal computations.
This function rounds \( x \) emulating subnormal number arithmetic: if \( x \) is outside the subnormal exponent range of the emulated floating-point system, this function just propagates the [ternary value], page 8, \( t \); otherwise, it rounds \( x \) to precision \( \text{EXP}(x)-\text{emin}+1 \) 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 \( \text{emin} \), \( x \) 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.

\( \text{PREC}(x) \) is not modified by this function. \( rnd \) and \( t \) must be the rounding mode and the returned ternary value used when computing \( x \) (as in \texttt{mpfr_check_range}). The subnormal exponent range is from \( \text{emin} \) to \( \text{emin}+\text{PREC}(x)-1 \). If the result cannot be represented in the current exponent range of MPFR (due to a too small \( \text{emax} \)), 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.

As usual, if the returned ternary value is non zero, the inexact flag is set. Moreover, if a second rounding occurred (because the input \( x \) was in the subnormal range), the underflow flag is set.

Warning! If you change \( \text{emin} \) (with \texttt{mpfr_set_emin}) just before calling \texttt{mpfr_subnormalize}, you need to make sure that the value is in the current exponent range of MPFR. But it is better to change \( \text{emin} \) before any computation, if possible.

This is an example of how to emulate binary double IEEE 754 arithmetic (binary64 in IEEE 754-2008) using MPFR:

```c
{  
    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, MPFR_RNDN);
    a = 0x1.1235P-1021; mpfr_set_d (xa, a, MPFR_RNDN);

    a /= b;  
    i = mpfr_div (xa, xa, xb, MPFR_RNDN);
    i = mpfr_subnormalize (xa, i, MPFR_RNDN); /* new ternary value */

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

Note that \texttt{mpfr_set_emin} and \texttt{mpfr_set_emax} are called early enough in order to make sure that all computed values are in the current exponent range. Warning! This emulates a double IEEE 754 arithmetic with correct rounding in the subnormal range, which may not be the case for your hardware.

Below is another example showing how to emulate fixed-point arithmetic in a specific case. Here we compute the sine of the integers 1 to 17 with a result in a fixed-point arithmetic rounded at \( 2^{-42} \) (using the fact that the result is at most 1 in absolute value):

```c
{  
```
mpfr_t x; int i, inex;

mpfr_set_emin (-41);
mpfr_init2 (x, 42);
for (i = 1; i <= 17; i++)
{
    mpfr_set_ui (x, i, MPFR_RNDN);
    inex = mpfr_sin (x, x, MPFR_RNDZ);
    mpfr_subnormalize (x, inex, MPFR_RNDZ);
    mpfr_dump (x);
}
mpfr_clear (x);

void mpfr_clear_underflow (void)
void mpfr_clear_overflow (void)
void mpfr_clear_divby0 (void)
void mpfr_clear_nanflag (void)
void mpfr_clear_inexflag (void)
void mpfr_clear_erangeflag (void)
    Clear (lower) the underflow, overflow, divide-by-zero, invalid, inexact and \textit{erange} flags.

void mpfr_clear_flags (void)
    Clear (lower) all global flags (underflow, overflow, divide-by-zero, invalid, inexact, \textit{erange}).
    Note: a group of flags can be cleared by using mpfr_flags_clear.

void mpfr_set_underflow (void)
void mpfr_set_overflow (void)
void mpfr_set_divby0 (void)
void mpfr_set_nanflag (void)
void mpfr_set_inexflag (void)
void mpfr_set_erangeflag (void)
    Set (raise) the underflow, overflow, divide-by-zero, invalid, inexact and \textit{erange} flags.

int mpfr_underflow_p (void)
int mpfr_overflow_p (void)
int mpfr_divby0_p (void)
int mpfr_nanflag_p (void)
int mpfr_inexflag_p (void)
int mpfr_erangeflag_p (void)
    Return the corresponding (underflow, overflow, divide-by-zero, invalid, inexact, \textit{erange}) flag,
    which is non-zero iff the flag is set.

The mpfr_flags_ functions below that take an argument mask can operate on any subset of the
exception flags: a flag is part of this subset (or group) if and only if the corresponding bit of the
argument mask is set. The MPFR_FLAGS macros will normally be used to build this argument.
See Section 4.6 [Exceptions], page 9.

void mpfr_flags_clear (mpfr_flags_t mask)
    Clear (lower) the group of flags specified by mask.

void mpfr_flags_set (mpfr_flags_t mask)
    Set (raise) the group of flags specified by mask.
mpfr_flags_t mpfr_flags_test (mpfr_flags_t mask)  [Function]
Return the flags specified by mask. To test whether any flag from mask is set, compare the return value to 0. You can also test individual flags by AND’ing the result with MPFR_FLAGS_ macros. Example:

```c
mpfr_flags_t t = mpfr_flags_test (MPFR_FLAGS_UNDERFLOW| 
MPFR_FLAGS_OVERFLOW)
```

```c
... 
if (t) /* underflow and/or overflow (unlikely) */
{
    if (t & MPFR_FLAGS_UNDERFLOW) { /* handle underflow */ }
    if (t & MPFR_FLAGS_OVERFLOW) { /* handle overflow */ }
}
```

mpfr_flags_t mpfr_flags_save (void)  [Function]
Return all the flags. It is equivalent to mpfr_flags_test(MPFR_FLAGS_ALL).

void mpfr_flags_restore (mpfr_flags_t flags, mpfr_flags_t mask)  [Function]
Restore the flags specified by mask to their state represented in flags.

### 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. By inserting the following two lines after the #include <gmp.h> line,

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

many programs written for MPF can be compiled directly against 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! There are some differences. In particular:

- The precision is different: MPFR rounds to the exact number of bits (zeroing trailing bits in the internal representation). Users may need to increase the precision of their variables.
- The exponent range is also different.
- The formatted output functions (gmp_printf, etc.) will not work for arguments of arbitrary-precision floating-point type (mpf_t, which mpf2mpfr.h redefines as mpfr_t).

void mpfr_set_prec_raw (mpfr_t x, mpfr_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, we do not recommend to use it otherwise. Do not use it either 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, mpfr_rnd_t rnd)  
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 precision of rop and the rounding mode rnd for all operations.

int mpfr_mul_2exp (mpfr_t rop, mpfr_t op1, unsigned long int op2, mpfr_rnd_t rnd)  
int mpfr_div_2exp (mpfr_t rop, mpfr_t op1, unsigned long int op2, mpfr_rnd_t rnd)  
These functions are identical to mpfr_mul_2ui and mpfr_div_2ui respectively. These functions are only kept for compatibility with MPF, one should prefer mpfr_mul_2ui and mpfr_div_2ui otherwise.

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 one to use MPFR in two ways:

- Either directly store a floating-point number as a mpfr_t on the stack.
- Either store its own representation on the stack and construct a new temporary mpfr_t each time it is needed.

Nothing has to be done to destroy the floating-point numbers except garbageing the used memory: all the memory management (allocating, destroying, garbageing) is left 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 floating-point numbers using mpfr_init and similar functions. 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.

size_t mpfr_custom_get_size (mpfr_prec_t prec)  
Return the needed size in bytes to store the significand of a floating-point number of precision prec.

void mpfr_custom_init (void *significand, mpfr_prec_t prec)  
Initialize a significand of precision prec, where 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 (GMP type, see Section 5.16 [Internals], page 46).

void mpfr_custom_init_set (mpfr_t x, int kind, mpfr_exp_t exp, mpfr_prec_t prec, void *significand)  
Perform a dummy initialization of a mpfr_t and set it to:

- if ABS(kind) == MPFR_NAN_KIND, x is set to NaN;
• if \( \text{ABS(kind)} == \text{MPFR_INF_KIND} \), \( x \) is set to the infinity of sign \( \text{sign(kind)} \);
• if \( \text{ABS(kind)} == \text{MPFR_ZERO_KIND} \), \( x \) is set to the zero of sign \( \text{sign(kind)} \);
• if \( \text{ABS(kind)} == \text{MPFR_REGULAR_KIND} \), \( x \) is set to a regular number: \( x = \text{sign(kind)} \text{significand} \times 2^\text{exp} \).

In all cases, it uses \( \text{significand} \) directly for further computing involving \( x \). It will not allocate anything. A floating-point number initialized with this function cannot be resized using \text{mpfr_set_prec} or \text{mpfr_prec_round}, or cleared using \text{mpfr_clear}! The \( \text{significand} \) must have been initialized with \text{mpfr_custom_init} using the same precision \( \text{prec} \).

\[ \text{Function} \]
\text{int mpfr_custom_get_kind (mpfr_t x)}

Return the current kind of a \( \text{mpfr_t} \) as created by \text{mpfr_custom_init_set}. The behavior of this function for any \( \text{mpfr_t} \) not initialized with \text{mpfr_custom_init_set} is undefined.

\[ \text{Function} \]
\text{void * mpfr_custom_get_significand (mpfr_t x)}

Return a pointer to the significand used by a \( \text{mpfr_t} \) initialized with \text{mpfr_custom_init_set}. The behavior of this function for any \( \text{mpfr_t} \) not initialized with \text{mpfr_custom_init_set} is undefined.

\[ \text{Function} \]
\text{mpfr_exp_t mpfr_custom_get_exp (mpfr_t x)}

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 does not produce any trap. The behavior of this function for any \( \text{mpfr_t} \) not initialized with \text{mpfr_custom_init_set} is undefined.

\[ \text{Function} \]
\text{void mpfr_custom_move (mpfr_t x, void *new_position)}

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

5.16 Internals

A \text{limb} means the part of a multi-precision number that fits in a single word. Usually a limb contains 32 or 64 bits. The C data type for a limb is \text{mp_limb_t}.

The \( \text{mpfr_t} \) type is internally defined as a one-element array of a structure, and \text{mpfr_ptr} is the C data type representing a pointer to this structure. The \( \text{mpfr_t} \) 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 \text{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 special values of the exponent field.
• Finally, the \_mpfr_d field is a pointer to the limbs, least significant limbs stored first. The number of limbs in use is controlled by \_mpfr_prec, namely \( \text{ceil(\_mpfr_prec/mp_bits_per_limb)} \). Non-singular (i.e., different from NaN, Infinity or 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 zeros.
6 API Compatibility

The goal of this section is to describe some API changes that occurred from one version of MPFR to another, and how to write code that can be compiled and run with older MPFR versions. The minimum MPFR version that is considered here is 2.2.0 (released on 20 September 2005).

API changes can only occur between major or minor versions. Thus the patchlevel (the third number in the MPFR version) will be ignored in the following. If a program does not use MPFR internals, changes in the behavior between two versions differing only by the patchlevel should only result from what was regarded as a bug or unspecified behavior.

As a general rule, a program written for some MPFR version should work with later versions, possibly except at a new major version, where some features (described as obsolete for some time) can be removed. In such a case, a failure should occur during compilation or linking. If a result becomes incorrect because of such a change, please look at the various changes below (they are minimal, and most software should be unaffected), at the FAQ and at the MPFR web page for your version (a bug could have been introduced and be already fixed); and if the problem is not mentioned, please send us a bug report (see Chapter 3 [Reporting Bugs], page 5).

However, a program written for the current MPFR version (as documented by this manual) may not necessarily work with previous versions of MPFR. This section should help developers to write portable code.

Note: Information given here may be incomplete. API changes are also described in the NEWS file (for each version, instead of being classified like here), together with other changes.

6.1 Type and Macro Changes

The official type for exponent values changed from mp_exp_t to mpfr_exp_t in MPFR 3.0. The type mp_exp_t will remain available as it comes from GMP (with a different meaning). These types are currently the same (mpfr_exp_t is defined as mp_exp_t with typedef), so that programs can still use mp_exp_t; but this may change in the future. Alternatively, using the following code after including mpfr.h will work with official MPFR versions, as mpfr_exp_t was never defined in MPFR 2.x:

```c
#if MPFR_VERSION_MAJOR < 3
typedef mp_exp_t mpfr_exp_t;
#endif
```

The official types for precision values and for rounding modes respectively changed from mp_prec_t and mp_rnd_t to mpfr_prec_t and mpfr_rnd_t in MPFR 3.0. This change was actually done a long time ago in MPFR, at least since MPFR 2.2.0, with the following code in mpfr.h:

```c
#ifndef mp_rnd_t
#define mp_rnd_t mpfr_rnd_t
#endif
#ifndef mp_prec_t
#define mp_prec_t mpfr_prec_t
#endif
```

This means that it is safe to use the new official types mpfr_prec_t and mpfr_rnd_t in your programs. The types mp_prec_t and mp_rnd_t (defined in MPFR only) may be removed in the future, as the prefix mp_ is reserved by GMP.

The precision type mpfr_prec_t (mp_prec_t) was unsigned before MPFR 3.0; it is now signed. MPFR_PREC_MAX has not changed, though. Indeed the MPFR code requires that MPFR_PREC_MAX
be representable in the exponent type, which may have the same size as mpfr_prec_t but has always been signed. The consequence is that valid code that does not assume anything about the signedness of mpfr_prec_t should work with past and new MPFR versions. This change was useful as the use of unsigned types tends to convert signed values to unsigned ones in expressions due to the usual arithmetic conversions, which can yield incorrect results if a negative value is converted in such a way. Warning! A program assuming (intentionally or not) that mpfr_prec_t is signed may be affected by this problem when it is built and run against MPFR 2.x.

The rounding modes GMP_RNDx were renamed to MPFR_RNDx in MPFR 3.0. However the old names GMP_RNDx have been kept for compatibility (this might change in future versions), using:

```c
#define GMP_RNDN MPFR_RNDN
#define GMP_RNZ MPFR_RNDZ
#define GMP_RNDU MPFR_RNU
#define GMP_RNDD MPFR_RNDD
```

The rounding mode “round away from zero” (MPFR_RNDA) was added in MPFR 3.0 (however no rounding mode GMP_RNDA exists). Faithful rounding (MPFR_RNDF) was added in MPFR 4.0, but currently, it is partially supported.

The flags-related macros, whose name starts with MPFR_FLAGS_, were added in MPFR 4.0 (for the new functions mpfr_flags_clear, mpfr_flags_restore, mpfr_flags_set and mpfr_flags_test, in particular).

### 6.2 Added Functions

We give here in alphabetical order the functions (and function-like macros) that were added after MPFR 2.2, and in which MPFR version.

- `mpfr_add_d` in MPFR 2.4.
- `mpfr_ai` in MPFR 3.0 (incomplete, experimental).
- `mpfr_asprintf` in MPFR 2.4.
- `mpfr_beta` in MPFR 4.0 (incomplete, experimental).
- `mpfr_buildopt_decimal_p` in MPFR 3.0.
- `mpfr_buildopt_float128_p` in MPFR 4.0.
- `mpfr_buildopt_gmpinternals_p` in MPFR 3.1.
- `mpfr_buildopt_sharedcache_p` in MPFR 4.0.
- `mpfr_buildopt_tls_p` in MPFR 3.0.
- `mpfr_buildopt_tune_case` in MPFR 3.1.
- `mpfr_clear_divby0` in MPFR 3.1 (new divide-by-zero exception).
- `mpfr_copysign` in MPFR 2.3. Note: MPFR 2.2 had a mpfr_copysign function that was available, but not documented, and with a slight difference in the semantics (when the second input operand is a NaN).
- `mpfr_custom_get_significand` in MPFR 3.0. This function was named mpfr_custom_get_mantissa in previous versions; mpfr_custom_get_mantissa is still available via a macro in mpfr.h:

  ```c
  #define mpfr_custom_get_mantissa mpfr_custom_get_significand
  ```

Thus code that needs to work with both MPFR 2.x and MPFR 3.x should use mpfr_custom_get_mantissa.

- `mpfr_d_div` and `mpfr_d_sub` in MPFR 2.4.
- `mpfr_digamma` in MPFR 3.0.
• mpfr_divby0_p in MPFR 3.1 (new divide-by-zero exception).
• mpfr_div_d in MPFR 2.4.
• mpfr_erandom in MPFR 4.0.
• mpfr_flags_clear, mpfr_flags_restore, mpfr_flags_save, mpfr_flags_set and mpfr_flags_test in MPFR 4.0.
• mpfr_fmma and mpfr_fmms in MPFR 4.0.
• mpfr_fmod in MPFR 2.4.
• mpfr_fmod quo in MPFR 4.0.
• mpfr_fms in MPFR 2.3.
• mpfr_fpif_export and mpfr_fpif_import in MPFR 4.0.
• mpfr_fprintf in MPFR 2.4.
• mpfr_free_cache2 in MPFR 4.0.
• mpfr_free_pool in MPFR 4.0.
• mpfr_frexp in MPFR 3.1.
• mpfr_gamma_inc in MPFR 4.0.
• mpfr_get_float128 in MPFR 4.0 if configured with ‘--enable-float128’.
• mpfr_get_flt in MPFR 3.0.
• mpfr_get_patches in MPFR 2.3.
• mpfr_get_q in MPFR 4.0.
• mpfr_get_z_2exp in MPFR 3.0. This function was named mpfr_get_z_exp in previous versions; mpfr_get_z_exp is still available via a macro in mpfr.h:

#define mpfr_get_z_exp mpfr_get_z_2exp

Thus code that needs to work with both MPFR 2.x and MPFR 3.x should use mpfr_get_z_exp.
• mpfr_grandom in MPFR 3.1.
• mpfr_j0, mpfr_j1 and mpfr_jn in MPFR 2.3.
• mpfr_lgamma in MPFR 2.3.
• mpfr_li2 in MPFR 2.4.
• mpfr_log_ui in MPFR 4.0.
• mpfr_min_prec in MPFR 3.0.
• mpfr_modf in MPFR 2.4.
• mpfr_mp_memory_cleanup in MPFR 4.0.
• mpfr_mul_d in MPFR 2.4.
• mpfr_nrandom in MPFR 4.0.
• mpfr_fprintf in MPFR 2.4.
• mpfr_rec_sqrt in MPFR 2.4.
• mpfr_regular_p in MPFR 3.0.
• mpfr_remainder and mpfr_remquo in MPFR 2.3.
• mpfr_rint_roundeven and mpfr_roundeven in MPFR 4.0.
• mpfr_round_nearest_away in MPFR 4.0.
• mpfr_rotn_ui in MPFR 4.0.
• mpfr_set_divby0 in MPFR 3.1 (new divide-by-zero exception).
• mpfr_set_float128 in MPFR 4.0 if configured with ‘--enable-float128’.
• mpfr_set_flt in MPFR 3.0.
• mpfr_set_z_2exp in MPFR 3.0.
• mpfr_set_zero in MPFR 3.0.
• mpfr_setsign in MPFR 2.3.
• mpfr_signbit in MPFR 2.3.
• mpfr_sinh_cosh in MPFR 2.4.
• mpfr_snprintf and mpfr_sprintf in MPFR 2.4.
• mpfr_sub_d in MPFR 2.4.
• mpfr_urandom in MPFR 3.0.
• mpfr_vasprintf, mpfr_vfprintf, mpfr_vprintf, mpfr_vsprintf and mpfr_vsnprintf in MPFR 2.4.
• mpfr_y0, mpfr_y1 and mpfr_yn in MPFR 2.3.
• mpfr_z_sub in MPFR 3.1.

6.3 Changed Functions

The following functions have changed after MPFR 2.2. Changes can affect the behavior of code written for some MPFR version when built and run against another MPFR version (older or newer), as described below.

• mpfr_abs, mpfr_neg and mpfr_set changed in MPFR 4.0. In previous MPFR versions, the sign bit of a NaN was unspecified; however, in practice, it was set as now specified except for mpfr_neg with a reused argument: mpfr_neg(x, x, rnd).
• mpfr_check_range changed in MPFR 2.3.2 and MPFR 2.4. If the value is an inexact infinity, the overflow flag is now set (in case it was lost), while it was previously left unchanged. This is really what is expected in practice (and what the MPFR code was expecting), so that the previous behavior was regarded as a bug. Hence the change in MPFR 2.3.2.
• mpfr_eint changed in MPFR 4.0. This function now returns the value of the E1/eint1 function for negative argument (before MPFR 4.0, it was returning NaN).
• mpfr_get_f changed in MPFR 3.0. This function was returning zero, except for NaN and Inf, which do not exist in MPF. The erange flag is now set in these cases, and mpfr_get_f now returns the usual ternary value.
• mpfr_get_si, mpfr_get_sj, mpfr_get_ui and mpfr_get_uj changed in MPFR 3.0. In previous MPFR versions, the cases where the erange flag is set were unspecified.
• mpfr_get_str changed in MPFR 4.0. This function now sets the NaN flag on NaN input (to follow the usual MPFR rules on NaN and IEEE 754-2008 recommendations on string conversions from Subclause 5.12.1) and sets the inexact flag when the conversion is inexact.
• mpfr_get_z changed in MPFR 3.0. The return type was void; it is now int, and the usual ternary value is returned. Thus programs that need to work with both MPFR 2.x and 3.x must not use the return value. Even in this case, C code using mpfr_get_z as the second or third term of a conditional operator may also be affected. For instance, the following is correct with MPFR 3.0, but not with MPFR 2.x:
  ```c
  bool ? mpfr_get_z(...) : mpfr_add(...);
  ```
  On the other hand, the following is correct with MPFR 2.x, but not with MPFR 3.0:
  ```c
  bool ? mpfr_get_z(...) : (void) mpfr_add(...);
  ```
  Portable code should cast mpfr_get_z(...) to void to use the type void for both terms of the conditional operator, as in:
  ```c
  bool ? (void) mpfr_get_z(...) : (void) mpfr_add(...);
  ```
  Alternatively, if ... else can be used instead of the conditional operator.

Moreover the cases where the erange flag is set were unspecified in MPFR 2.x.
• mpfr_get_z_exp changed in MPFR 3.0. In previous MPFR versions, the cases where the 
erange flag is set were unspecified. Note: this function has been renamed to mpfr_get_z_2exp in MPFR 3.0, but mpfr_get_z_exp is still available for compatibility reasons.

• mpfr_set_exp changed in MPFR 4.0. Before MPFR 4.0, the exponent was set whatever 
the contents of the MPFR object in argument. In practice, this could be useful as a low-
level function when the MPFR number was being constructed by setting the fields of its 
internal structure, but the API does not provide a way to do this except by using internals. 
Thus, for the API, this behavior was useless and could quickly lead to undefined behavior 
due to the fact that the generated value could have an invalid format if the MPFR object 
contained a special value (NaN, infinity or zero).

• mpfr_strtofr changed in MPFR 2.3.1 and MPFR 2.4. This was actually a bug fix since 
the code and the documentation did not match. But both were changed in order to have 
a more consistent and useful behavior. The main changes in the code are as follows. The 
binary exponent is now accepted even without the 0b or 0x prefix. Data corresponding to 
NaN can now have an optional sign (such data were previously invalid).

• mpfr_strtofr changed in MPFR 3.0. This function now accepts bases from 37 to 62 (no 
changes for the other bases). Note: if an unsupported base is provided to this function, the 
behavior is undefined; more precisely, in MPFR 2.3.1 and later, providing an unsupported 
base yields an assertion failure (this behavior may change in the future).

• mpfr_subnormalize changed in MPFR 3.1. This was actually regarded as a bug fix. The 
mpfr_subnormalize implementation up to MPFR 3.0.0 did not change the flags. In partic-
ular, it did not follow the generic rule concerning the inexact flag (and no special behavior 
was specified). The case of the underflow flag was more a lack of specification.

• mpfr_sum changed in MPFR 4.0. The mpfr_sum function has completely been rewritten for 
MPFR 4.0, with an update of the specification: the sign of an exact zero result is now speci-
fied, and the return value is now the usual ternary value. The old mpfr_sum implementation 
could also take all the memory and crash on inputs of very different magnitude.

• mpfr_urandom and mpfr_urandomb changed in MPFR 3.1. Their behavior no longer de-
pends on the platform (assuming this is also true for GMP’s random generator, which is not 
the case between GMP 4.1 and 4.2 if gmp_randinit_default is used). As a consequence, 
the returned values can be different between MPFR 3.1 and previous MPFR versions. Note: 
as the reproducibility of these functions was not specified before MPFR 3.1, the MPFR 3.1 
behavior is not regarded as backward incompatible with previous versions.

• mpfr_urandom changed in MPFR 4.0. The next random state no longer depends on the 
current exponent range and the rounding mode. The exceptions due to the rounding of 
the random number are now correctly generated, following the uniform distribution. As a 
consequence, the returned values can be different between MPFR 4.0 and previous MPFR 
versions.

6.4 Removed Functions

Functions mpfr_random and mpfr_random2 have been removed in MPFR 3.0 (this only affects 
old code built against MPFR 3.0 or later). (The function mpfr_random had been deprecated 
since at least MPFR 2.2.0, and mpfr_random2 since MPFR 2.4.0.)

Macros mpfr_add_one_ulp and mpfr_sub_one_ulp have been removed in MPFR 4.0. They 
were no longer documented since MPFR 2.1.0 and were announced as deprecated since MPFR 
3.1.0.

Function mpfr_grandom is marked as deprecated in MPFR 4.0. It will be removed in a future 
release.
6.5 Other Changes

For users of a C++ compiler, the way how the availability of \texttt{intmax\_t} is detected has changed in MPFR 3.0. In MPFR 2.x, if a macro \texttt{INTMAX\_C} or \texttt{UINTMAX\_C} was defined (e.g. when the \texttt{__STDC_CONSTANT_MACROS} macro had been defined before \texttt{<stdint.h>} or \texttt{<inttypes.h>} has been included), \texttt{intmax\_t} was assumed to be defined. However this was not always the case (more precisely, \texttt{intmax\_t} can be defined only in the namespace \texttt{std}, as with Boost), so that compilations could fail. Thus the check for \texttt{INTMAX\_C} or \texttt{UINTMAX\_C} is now disabled for C++ compilers, with the following consequences:

- Programs written for MPFR 2.x that need \texttt{intmax\_t} may no longer be compiled against MPFR 3.0: a \texttt{#define MPFR\_USE\_INTMAX\_T} may be necessary before \texttt{mpfr.h} is included.
- The compilation of programs that work with MPFR 3.0 may fail with MPFR 2.x due to the problem described above. Workarounds are possible, such as defining \texttt{intmax\_t} and \texttt{uintmax\_t} in the global namespace, though this is not clean.

The divide-by-zero exception is new in MPFR 3.1. However it should not introduce incompatible changes for programs that strictly follow the MPFR API since the exception can only be seen via new functions.

As of MPFR 3.1, the \texttt{mpfr.h} header can be included several times, while still supporting optional functions (see Section 4.1 [Headers and Libraries], page 6).

The way memory is allocated by MPFR should be regarded as well-specified only as of MPFR 4.0.
7 MPFR and the IEEE 754 Standard

This section describes differences between MPFR and the IEEE 754 standard, and behaviors that are not specified yet in IEEE 754.

The MPFR numbers do not include subnormals. The reason is that subnormals are less useful than in IEEE 754 as the default exponent range in MPFR is large and they would have made the implementation more complex. However, subnormals can be emulated using \texttt{mpfr\_subnormalize}.

MPFR has a single NaN. The behavior is similar either to a signaling NaN or to a quiet NaN, depending on the context. For any function returning a NaN (either produced or propagated), the NaN flag is set, while in IEEE 754, some operations are quiet (even on a signaling NaN).

The \texttt{mpfr\_rec\_sqrt} function differs from IEEE 754 on $-0$, where it gives $+\text{Inf}$ (like for $+0$), following the usual limit rules, instead of $-\text{Inf}$.

The \texttt{mpfr\_root} function predates IEEE 754-2008 and behaves differently from its rootn operation. It is deprecated and \texttt{mpfr\_rootn\_ui} should be used instead.

Operations with an unsigned zero: For functions taking an argument of integer or rational type, a zero of such a type is unsigned unlike the floating-point zero (this includes the zero of type \texttt{unsigned long}, which is a mathematical, exact zero, as opposed to a floating-point zero, which may come from an underflow and whose sign would correspond to the sign of the real non-zero value). Unless documented otherwise, this zero is regarded as $+0$, as if it were first converted to a MPFR number with \texttt{mpfr\_set\_ui} or \texttt{mpfr\_set\_si} (thus the result may not agree with the usual limit rules applied to a mathematical zero). This is not the case of addition and subtraction (\texttt{mpfr\_add\_ui}, etc.), but for these functions, only the sign of a zero result would be affected, with $+0$ and $-0$ considered equal. Such operations are currently out of the scope of the IEEE 754 standard, and at the time of specification in MPFR, the Floating-Point Working Group in charge of the revision of IEEE 754 did not want to discuss issues with non-floating-point types in general.

Note also that some obvious differences may come from the fact that in MPFR, each variable has its own precision. For instance, a subtraction of two numbers of the same sign may yield an overflow; idem for a call to \texttt{mpfr\_set}, \texttt{mpfr\_neg} or \texttt{mpfr\_abs}, if the destination variable has a smaller precision.
Contributors

The main developers of MPFR are Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, Philippe Théveny and Paul Zimmermann.

Sylvie Boldo from ENS-Lyon, France, contributed the functions mpfr_agm and mpfr_log. Sylvain Chevillard contributed the mpfr_ai function. David Daney contributed the hyperbolic and inverse hyperbolic functions, the base-2 exponential, and the factorial function. Alain Delplanque contributed the new version of the mpfr_get_str function. Mathieu Dutour contributed the functions mpfr_acos, mpfr_asin and mpfr_atan, and a previous version of mpfr_gamma. Sylvain Chevillard contributed the mpfr_ai function. David Daney contributed the hyperbolic and inverse hyperbolic functions, the base-2 exponential, and the factorial function. Alain Delplanque contributed the new version of the mpfr_get_str function. Mathieu Dutour contributed the functions mpfr_acos, mpfr_asin and mpfr_atan, and a previous version of mpfr_gamma. Laurent Fousse contributed the original version of the mpfr_sum function (used up to MPFR 3.1). Emmanuel Jeandel, from ENS-Lyon too, contributed the generic hypergeometric code, as well as the internal function mpfr_exp3, a first implementation of the sine and cosine, and improved versions of mpfr_const_log2 and mpfr_const_pi. Ludovic Meunier helped in the design of the mpfr_erf code. Jean-Luc Rémy contributed the mpfr_zeta code. Fabrice Rouillier contributed the mpfr_xxx_z and mpfr_xxx_q functions, and helped to the Microsoft Windows porting. Damien Stehlé contributed the mpfr_get_1d_2exp function. Charles Karney contributed the mpfr_nrinstance and mpfr_erandom functions.

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. In particular Kevin Ryde did a tremendous job for the portability of MPFR in 2002-2004.

The development of the MPFR library would not have been possible without the continuous support of INRIA, and of the LORIA (Nancy, France) and LIP (Lyon, France) laboratories. In particular the main authors were or are members of the PolKA, Spaces, Cacao, Caramel and Caramba project-teams at LORIA and of the Arénaire and AriC project-teams at LIP. This project was started during the Fiabile (reliable in French) action supported by INRIA, and continued during the AOC action. The development of MPFR was also supported by a grant (202F0659 00 MPN 121) from the Conseil Régional de Lorraine in 2002, from INRIA by an "associate engineer" grant (2003-2005), an "opération de développement logiciel" grant (2007-2009), and the post-doctoral grant of Sylvain Chevillard in 2009-2010. The MPFR-MPC workshop in June 2012 was partly supported by the ERC grant ANTICS of Andreas Enge. The MPFR-MPC workshop in January 2013 was partly supported by the ERC grant ANTICS, the GDR IM and the Caramel project-team, during which Mickaël Gastineau contributed the MPFRbench program, and Fredrik Johansson a faster version of mpfr_const_euler.
References

Appendix A  GNU Free Documentation License

Version 1.2, November 2002

51 Franklin St, Fifth Floor, Boston, MA  02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

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 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

### A
- Accuracy ............................................. 12
- Arithmetic functions ............................ 20
- Assignment functions ............................ 14

### B
- Basic arithmetic functions ......................... 20

### C
- Combined initialization and assignment functions .............................................................. 17
- Comparison functions ................................ 22
- Compatibility with MPF ............................ 44
- Conditions for copying MPFR ........................ 1
- Conversion functions ................................ 17
- Copying conditions .................................. 1
- Custom interface .................................... 45

### E
- Exception related functions ......................... 40

### F
- Float arithmetic functions .......................... 20
- Float comparisons functions ...................... 22
- Float functions ...................................... 12
- Float input and output functions .................. 29
- Float output functions .............................. 30
- Floating-point functions ............................ 12
- Floating-point number .............................. 7

### G
- GNU Free Documentation License .................. 56
- Group of flags ....................................... 7

### I
- I/O functions ......................................... 29, 30
- Initialization functions ............................. 12
- Input functions ...................................... 29
- Installation .......................................... 3
- Integer related functions ........................... 33
- Internals ............................................. 46
- intmax_t ............................................ 46
- inttypes.h ........................................... 6

### L
- libmpfr ............................................... 6
- Libraries ............................................. 6
- Libtool ............................................... 6
- Limb .................................................. 46
- Linking ............................................... 6

### M
- Miscellaneous float functions ...................... 37
- mpfr.h .............................................. 6

### O
- Output functions .................................... 29, 30

### P
- Precision ............................................. 7, 12

### R
- Remainder related functions ....................... 33
- Reporting bugs ...................................... 5
- Rounding mode related functions .................. 35
- Rounding Modes ..................................... 7

### S
- Special functions ..................................... 24
- stdarg.h ............................................ 6
- stdint.h ............................................ 6
- stdio.h ............................................. 6

### T
- Ternary value ....................................... 8

### U
- uintmax_t ............................................ 6
Function and Type Index

mpfr_abs ........................................ 22
mpfr_acos ........................................ 25
mpfr_acosh ........................................ 26
mpfr_add ........................................... 20
mpfr_add_d ......................................... 20
mpfr_add_q ......................................... 20
mpfr_add_si ......................................... 20
mpfr_add_u1 ......................................... 20
mpfr_add_z ......................................... 20
mpfr_agm ........................................... 28
mpfr_al .............................................. 28
mpfr_asin ........................................... 25
mpfr_asinh .......................................... 26
mpfr_asprintf ...................................... 33
mpfr_atan .......................................... 25
mpfr_atan2 .......................................... 25
mpfr_atanh .......................................... 26
mpfr_atan ........................................... 27
mpfr_buildopt_decimal_p ......................... 40
mpfr_buildopt_float128_p ......................... 40
mpfr_buildopt_getinternals_p ...................... 40
mpfr_buildopt_sharedcache_p ...................... 40
mpfr_buildopt_tls_p ................................ 40
mpfr_buildopt_tune_case ......................... 40
mpfr_can_round .................................. 36
mpfr_cbrt .......................................... 21
mpfr ceil .......................................... 33
mpfr_check_range ................................ 41
mpfr clear ......................................... 12
mpfr clear_divby0 ................................ 43
mpfr clear_erangeflag ............................. 43
mpfr clear_flags .................................. 43
mpfr clear_inexflag ................................ 43
mpfr clear_nanflag ................................ 43
mpfr clear_overflow ................................. 43
mpfr clears ........................................ 12
mpfr cmp ............................................ 23
mpfr cmp_d .......................................... 23
mpfr cmp_f .......................................... 23
mpfr cmp ld ......................................... 23
mpfr cmp q .......................................... 23
mpfr cmp si ......................................... 23
mpfr cmp si_2exp .................................. 23
mpfr cmp ui ......................................... 23
mpfr cmp ui_2exp .................................. 23
mpfr cmp z .......................................... 23
mpfr capsabs ..................................... 23
mpfr const_catalan ................................ 28
mpfr const_euler ................................... 28
mpfr const_log2 ................................... 28
mpfr const_pi ...................................... 28
mpfr copy sign ...................................... 39
mpfr cos ............................................ 24
mpfr cosh .......................................... 25
mpfr cot ............................................ 24
mpfr coth .......................................... 26
mpfr csch .......................................... 24
mpfr csch .......................................... 26
mpfr custom_get_exp .................. 46
mpfr custom_get_kind ..................... 46
mpfr custom_get_significand .......... 46
mpfr custom_get_size .................... 45
mpfr custom_init ......................... 45
mpfr custom_init_set .................... 45
mpfr custom_move ......................... 46
mpfr d_div ......................................... 21
mpfr d_sub ......................................... 20
mpfr diagamma .................................... 27
mpfr dim ............................................ 22
mpfr div ............................................. 20
mpfr div_2exp .................................... 45
mpfr div_2si ...................................... 22
mpfr dui ............................................ 22
mpfr d .............................................. 21
mpfr div ............................................. 21
mpfr divby0 ...................................... 43
mpfr dump .......................................... 30
mpfr eint .......................................... 26
mpfr eq ............................................... 44
mpfr equal .......................................... 23
mpfr erand .......................................... 39
mpfr erangeflag .................................. 43
mpfr erf .............................................. 27
mpfr erfc .......................................... 27
mpfr exp ............................................. 24
mpfr exp10 .......................................... 24
mpfr exp2 ........................................... 24
mpfr expm1 .......................................... 24
mpfr fac ............................................. 26
mpfr fits_intmax .................................. 19
mpfr fits_sint ..................................... 19
mpfr fits_ulong .................................... 19
mpfr fits_sshort .................................. 19
mpfr fits_uint ..................................... 19
mpfr fits_uintmax ................................ 19
mpfr fits_ulong .................................... 19
mpfr fits_uhshort ................................ 19
mpfr flags .......................................... 43
mpfr flags_restore .............................. 44
mpfr flags_save .................................. 44
mpfr flags_set .................................... 43
mpfr flags_t ........................................ 7
mpfr flags_test ................................... 44
mpfr floor .......................................... 33
mpfr fma ............................................ 27
mpfr fmans ......................................... 28
mpfr fmod ............................................ 35
mpfr fmodquo ...................................... 35
mpfr fms ............................................. 27
mpfr fpif_export .................................. 30
mpfr fpif_import .................................. 30
mpfr fprintf ...................................... 33
mpfr frac ............................................ 35
mpfr free_cache ................................... 28
mpfr free_cache2 .................................. 28
mpfr free_pool ..................................... 29
mpfr free_str ...................................... 19
Function and Type Index

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpfr_set_inexflag</td>
<td>43</td>
</tr>
<tr>
<td>mpfr_set_inf</td>
<td>16</td>
</tr>
<tr>
<td>mpfr_set_ld</td>
<td>14</td>
</tr>
<tr>
<td>mpfr_set_nan</td>
<td>16</td>
</tr>
<tr>
<td>mpfr_set_nanflag</td>
<td>43</td>
</tr>
<tr>
<td>mpfr_set_overflow</td>
<td>43</td>
</tr>
<tr>
<td>mpfr_set_prec</td>
<td>14</td>
</tr>
<tr>
<td>mpfr_set_prec_raw</td>
<td>44</td>
</tr>
<tr>
<td>mpfr_set_q</td>
<td>15</td>
</tr>
<tr>
<td>mpfr_set_si</td>
<td>14</td>
</tr>
<tr>
<td>mpfr_set_si_2exp</td>
<td>15</td>
</tr>
<tr>
<td>mpfr_set_sj</td>
<td>14</td>
</tr>
<tr>
<td>mpfr_set_sj_2exp</td>
<td>15</td>
</tr>
<tr>
<td>mpfr_set_str</td>
<td>15</td>
</tr>
<tr>
<td>mpfr_set_ui</td>
<td>14</td>
</tr>
<tr>
<td>mpfr_set_ui_2exp</td>
<td>15</td>
</tr>
<tr>
<td>mpfr_set_uj</td>
<td>14</td>
</tr>
<tr>
<td>mpfr_set_uj_2exp</td>
<td>15</td>
</tr>
<tr>
<td>mpfr_set_underflow</td>
<td>43</td>
</tr>
<tr>
<td>mpfr_set_z</td>
<td>14</td>
</tr>
<tr>
<td>mpfr_set_z_2exp</td>
<td>15</td>
</tr>
<tr>
<td>mpfr_set_zero</td>
<td>16</td>
</tr>
<tr>
<td>mpfr_setsign</td>
<td>39</td>
</tr>
<tr>
<td>mpfr_sgn</td>
<td>23</td>
</tr>
<tr>
<td>mpfr_sj_div</td>
<td>21</td>
</tr>
<tr>
<td>mpfr_si_sub</td>
<td>20</td>
</tr>
<tr>
<td>mpfr_signbit</td>
<td>39</td>
</tr>
<tr>
<td>mpfr_sin</td>
<td>24</td>
</tr>
<tr>
<td>mpfr_sin_cos</td>
<td>24</td>
</tr>
<tr>
<td>mpfr_sinh</td>
<td>25</td>
</tr>
<tr>
<td>mpfr_sinh_cosh</td>
<td>26</td>
</tr>
<tr>
<td>mpfr_snprintf</td>
<td>33</td>
</tr>
<tr>
<td>mpfr_sprintf</td>
<td>33</td>
</tr>
<tr>
<td>mpfr_sqr</td>
<td>20</td>
</tr>
<tr>
<td>mpfr_sqr Ui</td>
<td>21</td>
</tr>
<tr>
<td>mpfr_sqr_roundoff</td>
<td>21</td>
</tr>
<tr>
<td>mpfr_strtoff</td>
<td>16</td>
</tr>
<tr>
<td>mpfr_sub</td>
<td>20</td>
</tr>
<tr>
<td>mpfr_sub_d</td>
<td>20</td>
</tr>
<tr>
<td>mpfr_sub_q</td>
<td>20</td>
</tr>
<tr>
<td>mpfr_sub_si</td>
<td>20</td>
</tr>
<tr>
<td>mpfr_sub_ui</td>
<td>20</td>
</tr>
<tr>
<td>mpfr_sub_z</td>
<td>20</td>
</tr>
<tr>
<td>mpfr_subnormal</td>
<td>42</td>
</tr>
<tr>
<td>mpfr_sum</td>
<td>29</td>
</tr>
<tr>
<td>mpfr_swap</td>
<td>17</td>
</tr>
<tr>
<td>mpfr_t</td>
<td>7</td>
</tr>
<tr>
<td>mpfr_tan</td>
<td>24</td>
</tr>
<tr>
<td>mpfr_tanh</td>
<td>25</td>
</tr>
<tr>
<td>mpfr_trunc</td>
<td>33</td>
</tr>
<tr>
<td>mpfr_ui_div</td>
<td>20</td>
</tr>
<tr>
<td>mpfr_ui_pow</td>
<td>21</td>
</tr>
<tr>
<td>mpfr_ui_pow_ui</td>
<td>21</td>
</tr>
<tr>
<td>mpfr_ui_sub</td>
<td>20</td>
</tr>
<tr>
<td>mpfr_underflow_p</td>
<td>43</td>
</tr>
<tr>
<td>mpfr_unordered_p</td>
<td>24</td>
</tr>
<tr>
<td>mpfr_urandom</td>
<td>38</td>
</tr>
<tr>
<td>mpfr_urandomb</td>
<td>38</td>
</tr>
<tr>
<td>mpfr_vasprintf</td>
<td>33</td>
</tr>
<tr>
<td>mpfr_vasprintff</td>
<td>33</td>
</tr>
<tr>
<td>mpfr_vasprintf</td>
<td>33</td>
</tr>
<tr>
<td>mpfr_vsprintf</td>
<td>33</td>
</tr>
<tr>
<td>mpfr_vsprintff</td>
<td>33</td>
</tr>
<tr>
<td>mpfr_y0</td>
<td>27</td>
</tr>
<tr>
<td>mpfr_y1</td>
<td>27</td>
</tr>
<tr>
<td>mpfr_yn</td>
<td>27</td>
</tr>
<tr>
<td>mpfr_z_sub</td>
<td>20</td>
</tr>
<tr>
<td>mpfr_zero_p</td>
<td>23</td>
</tr>
<tr>
<td>mpfr_zeta</td>
<td>27</td>
</tr>
<tr>
<td>mpfr_zeta_ui</td>
<td>27</td>
</tr>
<tr>
<td>MPFR_DECL_INIT</td>
<td>13</td>
</tr>
<tr>
<td>MPFR_VERSION</td>
<td>39</td>
</tr>
<tr>
<td>MPFR_VERSION_MAJOR</td>
<td>39</td>
</tr>
<tr>
<td>MPFR_VERSION_MINOR</td>
<td>39</td>
</tr>
<tr>
<td>MPFR_VERSION_NUM</td>
<td>40</td>
</tr>
<tr>
<td>MPFR_VERSION_PATCHLEVEL</td>
<td>39</td>
</tr>
<tr>
<td>MPFR_VERSION_STRING</td>
<td>39</td>
</tr>
</tbody>
</table>