GNU MP

The GNU Multiple Precision Arithmetic Library
Edition 3.1.1
18 September 2000

by Torbjörn Granlund, Swox AB
tege@swox.com

Copyright © 1991, 1993, 1994, 1995, 1996, 1997, 1998, 1999, 2000 Free Software Foundation, Inc.

Published by the Free Software Foundation
59 Temple Place - Suite 330
Boston, MA 02111-1307, USA

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Foundation.

GNU MP Copying Conditions

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

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

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the GNU MP 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 MP 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 MP library are found in the Lesser General Public License that accompany the source code.

1 Introduction to GNU MP

GNU MP is a portable library written in C for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. It aims to provide the fastest possible arithmetic for all applications that need higher precision than is directly supported by the basic C types.

Many applications use just a few hundred bits of precision; but some applications may need thousands or even millions of bits. GMP is designed to give good performance for both, by choosing algorithms based on the sizes of the operands, and by carefully keeping the overhead at a minimum.

The speed of GMP is achieved by using fullwords as the basic arithmetic type, by using sophisticated algorithms, by including carefully optimized assembly code for the most common inner loops for many different CPUs, and by a general emphasis on speed (as opposed to simplicity or elegance).

There is carefully optimized assembly code for these CPUs: ARM, DEC Alpha 21064, 21164, and 21264, AMD 29000, AMD K6 and Athlon, Hitachi SuperH and SH-2, HPPA 1.0, 1.1 and 2.0, Intel Pentium, Pentium Pro/Pentium II, generic x86, Intel i960, Motorola MC68000, MC68020, MC88100, and MC88110, Motorola/IBM PowerPC 32 and 64, National NS32000, IBM POWER, MIPS R3000, R4000, SPARCv7, SuperSPARC, generic SPARCv8, UltraSPARC, DEC VAX, and Zilog Z8000. Some optimizations also for Clipper, IBM ROMP (RT), and Pyramid AP/XP.

There is a mailing list for GMP users. To join it, send a mail to `gmp-request@swox.com` with the word '`subscribe`' in the message **body** (not in the subject line).

For up-to-date information on GMP, please see the GMP Home Pages at `http://www.swox.com/gmp/`.

1.1 How to use this Manual

Everyone should read Chapter 3 [GMP Basics], page 9. If you need to install the library yourself, you need to read Chapter 2 [Installing GMP], page 3, too.

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

2 Installing GMP

GMP has an autoconf/automake/libtool based configuration system. On a Unix-like system a basic build can be done with

```
./configure
make
```

Some self-tests can be run with

```
make check
```

And you can install (under '/usr/local' by default) with

```
make install
```

If you experience problems, please report them to bug-gmp@gnu.org. (See Chapter 4 [Reporting Bugs], page 13, for information on what to include in useful bug reports.)

2.1 Build Options

All the usual autoconf configure options are available, run './configure --help' for a summary.

Non-Unix Systems
: 'configure' needs various Unix-like tools installed. On an MS-DOS system cygwin or djgpp should work. It might be possible to build without the help of 'configure', certainly all the code is there, but unfortunately you'll be on your own.

Object Directory
: To compile in a separate object directory, cd to that directory, and prefix the configure command with the path to the GMP source directory. For example '../src/gmp/configure'. Not all 'make' programs have the necessary features (VPATH) to support this. In particular, SunOS and Slowaris make have bugs that make them unable to build from a separate object directory. Use GNU make instead.

'--disable-shared', '--disable-static'
: By default both shared and static libraries are built (where possible), but one or other can be disabled. Shared libraries are very slightly slower, having a small cost on each function call, but result in smaller executables and permit code sharing between separate running processes.

'--target=CPU-VENDOR-OS'
: The build target can be specified in the usual way, for either native or cross compilation.

If '--target' isn't given, './configure' builds for the host system as determined by './config.guess'. On some systems this can't distinguish between different CPUs in a family, and you should check the guess. Running './config.guess' on the target system will also show the relevant 'VENDOR-OS', if you don't already know what it should be.

In general, if you want a library that runs as fast as possible, you should configure GMP for the exact CPU type your system uses. However, this may mean the binaries won't run on older members of the family, and might run slower on other members, older or newer. The best idea is always to build GMP for the exact machine type you intend to run it on.

The following CPU targets have specific assembly code support. See '`configure.in`' for which '`mpn`' subdirectories get used by each.

- Alpha: '`alpha`', '`alphaev5`', '`alphaev6`'
- Hitachi: '`sh`', '`sh2`'
- HPPA: '`hppa1.0`', '`hppa1.1`', '`hppa2.0`', '`hppa2.0w`'
- MIPS: '`mips`', '`mips3`',
- Motorola: '`m68000`', '`m68k`', '`m88k`', '`m88110`'
- POWER: '`power1`', '`power2`', '`power2sc`', '`powerpc`', '`powerpc64`'
- SPARC: '`sparc`', '`sparcv8`', '`microsparc`', '`supersparc`', '`sparcv9`', '`ultrasparc`', '`sparc64`'
- 80x86 family: '`i386`', '`i486`', '`i586`', '`pentium`', '`pentiummmx`', '`pentiumpro`', '`pentium2`', '`pentium3`', '`k6`', '`k62`', '`k63`', '`athlon`'
- Other: '`a29k`', '`arm`', '`clipper`', '`i960`', '`ns32k`', '`pyramid`', '`vax`', '`z8k`'

CPUs not listed use generic C code. If some of the assembly code causes problems, the generic C code can be selected with CPU '`none`'.

'`CC`', '`CFLAGS`'

The C compiler used is chosen from among some likely candidates, with GCC normally preferred if it's present. The usual '`CC=whatever`' can be passed to '`./configure`' to choose something different.

For some configurations specific compiler flags are set based on the target CPU and compiler, see '`CFLAGS`' in the generated '`Makefile`'s. The usual '`CFLAGS="-whatever"`' can be passed to '`./configure`' to use something different or to set good flags for systems GMP doesn't otherwise know.

Note that if '`CC`' is set then '`CFLAGS`' must also be set. This applies even if '`CC`' is merely one of the choices GMP would make itself. This may change in a future release.

'`--disable-alloca`'

By default, GMP allocates temporary workspace using `alloca` if that function is available, or `malloc` if not. If you're working with large numbers and `alloca` overflows the available stack space, you can build with '`--disable-alloca`' to use `malloc` instead. `malloc` will probably be slightly slower than `alloca`.

When not using `alloca`, it's actually the allocation function selected with `mp_set_memory_functions` that's used, this being `malloc` by default. See Chapter 11 [Custom Allocation], page 50.

Depending on your system, the only indication of stack overflow might be a segmentation violation. It might be possible to increase available stack space with `limit`, `ulimit` or `setrlimit`, or under DJGPP with `stubedit` or `_stklen`.

'`--enable-fft`'

By default multiplications are done using Karatsuba and 3-way Toom-Cook algorithms, but a Fermat FFT can be enabled, for use on large to very large operands. Currently the FFT is recommended only for knowledgeable users who check the algorithm thresholds for their CPU.

'--enable-mpbsd'
: The Berkeley MP compatibility library ('libmp.a') and header file ('mp.h') are built and installed only if '--enable-mpbsd' is used. See Chapter 10 [BSD Compatible Functions], page 48.

'MPN_PATH'
: Various assembler versions of mpn subroutines are provided, and, for a given CPU target, a search is made though a path to choose a version of each. For example 'sparcv8' has path '"sparc32/v8 sparc32 generic"', which means it looks first for v8 code, falls back on plain sparc32, and finally falls back on generic C. Knowledgeable users with special requirements can specify a path with 'MPN_PATH="dir list"'. This will normally be unnecessary because all sensible paths should be available under one or other CPU target.

Demonstration Programs
: The 'demos' subdirectory has some sample programs using GMP. These aren't built or installed, but there's a 'Makefile' with rules for them. For instance, 'make pexpr' and then './pexpr 68^975+10'.

Documentation
: The document you're now reading is 'gmp.texi'. The usual automake targets are available to make 'gmp.ps' and/or 'gmp.dvi'. Some supplementary notes can be found in the 'doc' subdirectory.

2.2 ABI and ISA

ABI (Application Binary Interface) refers to the calling conventions between functions, meaning what registers are used and what sizes the various C data types are. ISA (Instruction Set Architecture) refers to the instructions and registers a CPU has available.

Some 64-bit ISA CPUs have both a 64-bit ABI and a 32-bit ABI defined, the latter for compatibility with older CPUs in the family. GMP chooses the best ABI available for a given target system, and this generally gives significantly greater speed.

The burden is on application programs and cooperating libraries to ensure they match the ABI chosen by GMP. Fortunately this presents a difficulty only on a few systems, and if you have one of them then the performance gains are enough to make it worth the trouble.

Some of what's described in this section may change in future releases of GMP.

HPPA 2.0
: CPU target 'hppa2.0' uses the hppa2.0n 32-bit ABI, but either a 32-bit or 64-bit limb.

  A 64-bit limb is available on HP-UX 10 or up when using c89. No gcc support is planned for 64-bit operations in this ABI. Applications must be compiled with the same options as GMP, which means

  ```
  c89  +DA2.0 +e -D_LONG_LONG_LIMB
  ```

  A 32-bit limb is used in other cases, and no special compiler options are needed.

  CPU target 'hppa2.0w' uses the hppa2.0w 64-bit ABI, which is available on HP-UX 11 or up when using c89. gcc support for this is in progress. Applications must be compiled for the same ABI, which means

```
c89  +DD64
```

MIPS 3 and 4 under IRIX 6

Targets '`mips*-*-irix6*`' use the n32 ABI and a 64-bit limb. Applications must be compiled for the same ABI, which means either

```
gcc  -mabi=n32
cc   -n32
```

PowerPC 64

CPU target '`powerpc64`' uses either the 32-bit ABI or the AIX 64-bit ABI. The latter is used on targets '`powerpc64-*-aix*`' and applications must be compiled using either

```
gcc  -maix64
xlc  -q64
```

On other systems the 32-bit ABI is used, but with 64-bit limbs provided by `long long` in `gcc`. Applications must be compiled using

```
gcc  -D_LONG_LONG_LIMB
```

Sparc V9

On a sparc v9 CPU, either the v8plus 32-bit ABI or v9 64-bit ABI is used. Targets '`ultrasparc*-*-solaris2.[7-9]`', '`sparcv9-*-solaris2.[7-9]`' and '`sparc64-*-linux*`' use the v9 ABI, if the compiler supports it. Other targets use the v8plus ABI (but with as much of the v9 ISA as possible in the circumstances). Note that Solaris prior to 2.7 doesn't save all registers properly, and hence uses the v8plus ABI.

For the v8plus ABI, applications can be compiled with either

```
gcc  -mv8plus
cc   -xarch=v8plus
```

For the v9 ABI, applications must be compiled with either

```
gcc  -m64 -mptr64 -Wa,-xarch=v9 -mcpu=v9
cc   -xarch=v9
```

Don't be confused by the names of these options, they're called '`arch`' but they effectively control the ABI.

2.3 Notes for Package Builds

GMP should present no great difficulties for packaging in a binary distribution.

Libtool is used to build the library and '`-version-info`' is set appropriately, having started from '`3:0:0`' in GMP 3.0. The GMP 3 series will be upwardly binary compatible in each release, but may be adding additional function interfaces. On systems where libtool versioning is not fully checked by the loader, an auxiliary mechanism may be needed to express that a dynamic linked application depends on a new enough minor version of GMP.

When building a package for a CPU family, care should be taken to use '`--target`' to choose the least common denominator among the CPUs which might use the package. For example this might necessitate '`i386`' for x86s, or plain '`sparc`' (meaning V7) for SPARCs.

Users who care about speed will want GMP built for their exact CPU type, to make use of the available optimizations. Providing a way to suitably rebuild a package may be useful. This could be as simple as making it possible for a user to omit '--target' in a build so './config.guess' will detect the CPU. But a way to manually specify a '--target' will be wanted for systems where './config.guess' is inexact.

2.4 Notes for Particular Systems

AIX 4.3

Targets '*-*-aix4.[3-9]*' have shared libraries disabled since they seem to fail on AIX 4.3.

OpenBSD 2.6

`m4` in this release of OpenBSD has a bug in `eval` that makes it unsuitable for '.asm' file processing. './configure' will detect the problem and either abort or choose another m4 in the `PATH`. The bug is fixed in OpenBSD 2.7, so either upgrade or use GNU m4.

Sparc V8

Using CPU target 'sparcv8' or 'supersparc' on relevant systems will give a significant performance increase over the V7 code.

SunOS 4

`/usr/bin/m4` lacks various features needed to process '.asm' files, and instead './configure' will automatically use `/usr/5bin/m4`, which we believe is always available (if not then use GNU m4).

x86 Pentium and PentiumPro

The Intel Pentium P5 code is good for its intended P5, but quite slow when run on Intel P6 class chips (PPro, P-II, P-III). 'i386' is a better choice if you're making binaries that must run on both.

x86 MMX and old GAS

Old versions of GAS don't support MMX instructions, in particular version 1.92.3 that comes with FreeBSD 2.2.8 doesn't (and unfortunately there's no newer assembler for that system).

If the target CPU has MMX code but the assembler doesn't support it, a warning is given and non-MMX code is used instead. This will be an inferior build, since the MMX code that's present is there because it's faster than the corresponding plain integer code.

x86 GCC 2.95.2 '-march=pentiumpro'

GCC 2.95.2 miscompiles 'mpz/powm.c' when '-march=pentiumpro' is used, so that option is omitted from the `CFLAGS` chosen for relevant CPUs. The problem is believed to be fixed in GCC 2.96.

2.5 Known Build Problems

You might find more up-to-date information at http://www.swox.com/gmp/.

Generic C on a 64-bit system

When making a generic C build using '--target=none' on a 64-bit system (meaning where unsigned long is 64 bits), BITS_PER_MP_LIMB, BITS_PER_LONGINT and BYTES_PER_MP_LIMB in 'mpn/generic/gmp-mparam.h' need to be changed to 64 and 8. This will hopefully be automated in a future version of GMP.

NeXT prior to 3.3

The system compiler on old versions of NeXT was a massacred and old GCC, even if it called itself 'cc'. This compiler cannot be used to build GMP, you need to get a real GCC, and install that before you compile GMP. (NeXT may have fixed this in release 3.3 of their system.)

POWER and PowerPC

Bugs in GCC 2.7.2 (and 2.6.3) mean it can't be used to compile GMP on POWER or PowerPC. If you want to use GCC for these machines, get GCC 2.7.2.1 (or later).

Sequent Symmetry

Use the GNU assembler instead of the system assembler, since the latter has serious bugs.

Stripped Libraries

GNU binutils 'strip' should not be used on the static libraries 'libgmp.a' and 'libmp.a', neither directly nor via 'make install-strip'. It can be used on the shared libraries 'libgmp.so' and 'libmp.so' though.

Currently (binutils 2.10.0), 'strip' extracts archives into a single directory, but GMP contains multiple object files of the same name (eg. three versions of 'init.o'), and they overwrite each other, leaving only the one that happens to be last.

If stripped static libraries are wanted, the suggested workaround is to build normally, strip the separate object files, and do another 'make all' to rebuild. Alternately 'CFLAGS' with '-g' omitted can always be used if it's just debugging which is unwanted.

SunOS 4 Native Tools

The setting for GSYM_PREFIX in 'config.m4' may be incorrectly determined when using the native grep, leading at link-time to undefined symbols like ___gmpn_add_n. To fix this, after running './configure', change the relevant line in 'config.m4' to 'define(<GSYM_PREFIX>, <_>)'.

The ranlib command will need to be run manually when building a static library with the native ar. After 'make', run 'ranlib .libs/libgmp.a', and when using '--enable-mpbsd' run 'ranlib .libs/libmp.a' too.

VAX running Ultrix

You need to build and install the GNU assembler before you compile GMP. The VAX assembly in GMP uses an instruction (jsobgtr) that cannot be assembled by the Ultrix assembler.

3 GMP Basics

All declarations needed to use GMP are collected in the include file '`gmp.h`'. It is designed to work with both C and C++ compilers.

Using functions, macros, data types, etc. not documented in this manual is strongly discouraged. If you do so your application is guaranteed to be incompatible with future versions of GMP.

3.1 Nomenclature and Types

In this manual, *integer* usually means a multiple precision integer, as defined by the GMP library. The C data type for such integers is `mpz_t`. Here are some examples of how to declare such integers:

```
mpz_t sum;

struct foo { mpz_t x, y; };

mpz_t vec[20];
```

Rational number means a multiple precision fraction. The C data type for these fractions is `mpq_t`. For example:

```
mpq_t quotient;
```

Floating point number or *Float* for short, is an arbitrary precision mantissa with a limited precision exponent. The C data type for such objects is `mpf_t`.

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

3.2 Function Classes

There are six classes of functions in the GMP library:

1. Functions for signed integer arithmetic, with names beginning with `mpz_`. The associated type is `mpz_t`. There are about 100 functions in this class.
2. Functions for rational number arithmetic, with names beginning with `mpq_`. The associated type is `mpq_t`. There are about 20 functions in this class, but the functions in the previous class can be used for performing arithmetic on the numerator and denominator separately.
3. Functions for floating-point arithmetic, with names beginning with `mpf_`. The associated type is `mpf_t`. There are about 50 functions is this class.
4. Functions compatible with Berkeley GMP, such as `itom`, `madd`, and `mult`. The associated type is `MINT`.
5. Fast low-level functions that operate on natural numbers. These are used by the functions in the preceding groups, and you can also call them directly from very time-critical user programs. These functions' names begin with `mpn_`. There are about 30 (hard-to-use) functions in this class.

   The associated type is array of `mp_limb_t`.

6. Miscellaneous functions. Functions for setting up custom allocation and functions for generating random numbers.

3.3 GMP Variable Conventions

As a general rule, all GMP functions expect output arguments before input arguments. This notation is based on an analogy with the assignment operator. (The BSD MP compatibility functions disobey this rule, having the output argument(s) last.)

GMP lets you use the same variable for both input and output in one call. For example, the main function for integer multiplication, `mpz_mul`, can be used to square `x` and put the result back in `x` with

```
mpz_mul (x, x, x);
```

Before you can assign to a GMP variable, you need to initialize it by calling one of the special initialization functions. When you're done with a variable, you need to clear it out, using one of the functions for that purpose. Which function to use depends on the type of variable. See the chapters on integer functions, rational number functions, and floating-point functions for details.

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 initializing and clearing out a GMP variable in a loop. Instead, initialize it before entering the loop, and clear it out after the loop has exited.

GMP variables are small, containing only a couple of sizes, and pointers to allocated data. Once you have initialized a GMP variable, you don't need to worry about space allocation. All functions in GMP automatically allocate additional space when a variable does not already have enough. They do not, however, reduce the space when a smaller value is stored. Most of the time this policy is best, since it avoids frequent re-allocation.

When a variable of type `mpz_t` is used as a function parameter, it's effectively a call-by-reference, meaning anything the function does to it will be be done to the original in the caller. When a function is going to return an `mpz_t` result, it should provide a separate parameter or parameters that it sets, like the GMP library functions do. A `return` of an `mpz_t` doesn't return the object, only a pointer to it, and this is almost certainly not what you want. All this applies to `mpq_t` and `mpf_t` too.

Here's an example function accepting an `mpz_t` parameter, doing a certain calculation, and returning a result.

```
void
myfunction (mpz_t result, mpz_t param, unsigned long n)
{
  unsigned long  i;

  mpz_mul_ui (result, param, n);
  for (i = 1; i < n; i++)
    mpz_add_ui (result, result, i*7);
}
```

```
int
main (void)
{
  mpz_t  r, n;
  mpz_init (r);
  mpz_init_set_str (n, "123456", 0);

  myfunction (r, n, 20L);
  mpz_out_str (stdout, 10, r); printf ("\n");

  return 0;
}
```

This example will work if `result` and `param` are the same variable, just like the library functions. But sometimes this is tricky to arrange, and an application might not want to bother for its own subroutines.

`mpz_t` is actually implemented as a one-element array of a certain structure type. This is why using it to declare a variable gives an object with the fields GMP needs, but then using it as a parameter passes a pointer to the object. Note that the actual contents of an `mpz_t` are for internal use only and you should not access them directly if you want your code to be compatible with future GMP releases.

3.4 GMP and Reentrancy

The GMP code is reentrant and thread-safe, with some exceptions:

- The function `mpf_set_default_prec` saves the selected precision in a global variable.
- The function `mp_set_memory_functions` uses several global variables for storing the selected memory allocation functions.
- If the memory allocation functions set by a call to `mp_set_memory_functions` (or `malloc` and friends by default) are not reentrant, GMP will not be reentrant either.
- The old random number functions (`mpz_random`, etc) use a random number generator from the C library, usually `mrand48` or `random`. These routines are not reentrant, since they rely on global state. (However the newer random number functions that accept a `gmp_randstate_t` parameter are reentrant.)
- If `alloca` is not available, or GMP is configured with '`--disable-alloca`', the library is not reentrant, due to the current implementation of '`stack-alloc.c`'. In the generated '`config.h`', `USE_STACK_ALLOC` set to 1 will mean not reentrant.

3.5 Useful Macros and Constants

`const int` **mp_bits_per_limb** Global Constant

The number of bits per limb.

__GNU_MP_VERSION Macro
__GNU_MP_VERSION_MINOR Macro
__GNU_MP_VERSION_PATCHLEVEL Macro

The major and minor GMP version, and patch level, respectively, as integers. For GMP i.j, these numbers will be i, j, and 0, respectively. For GMP i.j.k, these numbers will be i, j, and k, respectively.

3.6 Compatibility with older versions

This version of GMP is upwardly binary compatible with versions 3.0 and 3.0.1, and upwardly compatible at the source level with versions 2.0, 2.0.1, and 2.0.2, with the following exceptions.

- `mpn_gcd` had its source arguments swapped as of GMP 3.0 for consistency with other `mpn` functions.
- `mpf_get_prec` counted precision slightly differently in GMP 3.0 and 3.0.1, but in 3.1 has reverted to the 2.0.x style.

There are a number of compatibility issues between GMP 1 and GMP 2 that of course also apply when porting applications from GMP 1 to GMP 3. Please see the GMP 2 manual for details.

3.7 Getting the Latest Version of GMP

The latest version of the GMP library is available at `ftp://ftp.gnu.org/pub/gnu/gmp`. Many sites around the world mirror '`ftp.gnu.org`'; please use a mirror site near you, see `http://www.gnu.org/order/ftp.html`.

4 Reporting Bugs

If you think you have found a bug in the GMP library, please investigate it and report it. We have made this library available to you, and it is not too much to ask you to report the bugs you find. Before you report a bug, you may want to check `http://www.swox.com/gmp/` for patches for this release.

Please include the following in any report,

- The GMP version number, and if pre-packaged or patched then say so.
- A test program that makes it possible for us to reproduce the bug. Include instructions on how to run the program.
- A description of what is wrong. If the results are incorrect, in what way. If you get a crash, say so.
- If you get a crash, include a stack backtrace from the debugger if it's informative ('`where`' in `gdb`, or '`$C`' in `adb`).
- **Please do not send core dumps, executables or `strace`s.**
- The configuration options you used when building GMP, if any.
- The name of the compiler and its version. For `gcc`, get the version with '`gcc -v`', otherwise perhaps '``what `which cc` ``', or similar.
- The output from running '`uname -a`'.
- The output from running '`./config.guess`'.
- If the bug is related to '`configure`', then the contents of '`config.log`'.
- If the bug is related to an '`asm`' file not assembling, then the contents of '`config.m4`'.

It is not uncommon that an observed problem is actually due to a bug in the compiler; the GMP code tends to explore interesting corners in compilers.

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

Send your report to: `bug-gmp@gnu.org`.

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.

5 Integer Functions

This chapter describes the GMP functions for performing integer arithmetic. These functions start with the prefix `mpz_`.

GMP integers are stored in objects of type `mpz_t`.

5.1 Initialization Functions

The functions for integer arithmetic assume that all integer objects are initialized. You do that by calling the function `mpz_init`.

`void` **mpz_init** (`mpz_t` *integer*) Function

Initialize *integer* with limb space and set the initial numeric value to 0. Each variable should normally only be initialized once, or at least cleared out (using `mpz_clear`) between each initialization.

Here is an example of using `mpz_init`:

```
{
  mpz_t integ;
  mpz_init (integ);
  ...
  mpz_add (integ, ...);
  ...
  mpz_sub (integ, ...);

  /* Unless the program is about to exit, do ... */
  mpz_clear (integ);
}
```

As you can see, you can store new values any number of times, once an object is initialized.

`void` **mpz_clear** (`mpz_t` *integer*) Function

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

`void *` **_mpz_realloc** (`mpz_t` *integer*, `mp_size_t` *new_alloc*) Function

Change the limb space allocation to *new_alloc* limbs. This function is not normally called from user code, but it can be used to give memory back to the heap, or to increase the space of a variable to avoid repeated automatic re-allocation.

`void` **mpz_array_init** (`mpz_t` *integer_array*[], `size_t` *array_size*, `mp_size_t` *fixed_num_bits*) Function

Allocate **fixed** limb space for all *array_size* integers in *integer_array*. The fixed allocation for each integer in the array is enough to store *fixed_num_bits*. If the fixed space will be insufficient for storing the result of a subsequent calculation, the result is unpredictable.

This function is useful for decreasing the working set for some algorithms that use large integer arrays.

There is no way to de-allocate the storage allocated by this function. Don't call `mpz_clear`!

5.2 Assignment Functions

These functions assign new values to already initialized integers (see Section 5.1 [Initializing Integers], page 14).

void **mpz_set** (mpz_t *rop*, mpz_t *op*) Function
void **mpz_set_ui** (mpz_t *rop*, unsigned long int *op*) Function
void **mpz_set_si** (mpz_t *rop*, signed long int *op*) Function
void **mpz_set_d** (mpz_t *rop*, double *op*) Function
void **mpz_set_q** (mpz_t *rop*, mpq_t *op*) Function
void **mpz_set_f** (mpz_t *rop*, mpf_t *op*) Function

Set the value of *rop* from *op*.

int **mpz_set_str** (mpz_t *rop*, char **str*, int *base*) Function

Set the value of *rop* from *str*, a '\0'-terminated C string in base *base*. White space is allowed in the string, and is simply ignored. The base may vary from 2 to 36. If *base* is 0, the actual base is determined from the leading characters: if the first two characters are '0x' or '0X', hexadecimal is assumed, otherwise if the first character is '0', octal is assumed, otherwise decimal is assumed.

This function returns 0 if the entire string up to the '\0' is a valid number in base *base*. Otherwise it returns -1.

[It turns out that it is not entirely true that this function ignores white-space. It does ignore it between digits, but not after a minus sign or within or after "0x". We are considering changing the definition of this function, making it fail when there is any white-space in the input, since that makes a lot of sense. Please tell us your opinion about this change. Do you really want it to accept "3 14" as meaning 314 as it does now?]

void **mpz_swap** (mpz_t *rop1*, mpz_t *rop2*) Function

Swap the values *rop1* and *rop2* efficiently.

5.3 Combined Initialization and Assignment Functions

For convenience, GMP provides a parallel series of initialize-and-set functions which initialize the output and then store the value there. These functions' names have the form `mpz_init_set...`

Here is an example of using one:

```
{
  mpz_t pie;
  mpz_init_set_str (pie, "3141592653589793238462643383279502884", 10);
  ...
  mpz_sub (pie, ...);
```

```
  ...
  mpz_clear (pie);
}
```

Once the integer has been initialized by any of the `mpz_init_set...` functions, it can be used as the source or destination operand for the ordinary integer functions. Don't use an initialize-and-set function on a variable already initialized!

void **mpz_init_set** (mpz_t *rop*, mpz_t *op*) — Function
void **mpz_init_set_ui** (mpz_t *rop*, unsigned long int *op*) — Function
void **mpz_init_set_si** (mpz_t *rop*, signed long int *op*) — Function
void **mpz_init_set_d** (mpz_t *rop*, double *op*) — Function

Initialize *rop* with limb space and set the initial numeric value from *op*.

int **mpz_init_set_str** (mpz_t *rop*, char **str*, int *base*) — Function

Initialize *rop* and set its value like `mpz_set_str` (see its documentation above for details).

If the string is a correct base *base* number, the function returns 0; if an error occurs it returns −1. *rop* is initialized even if an error occurs. (I.e., you have to call `mpz_clear` for it.)

5.4 Conversion Functions

This section describes functions for converting GMP integers to standard C types. Functions for converting *to* GMP integers are described in Section 5.2 [Assigning Integers], page 15 and Section 5.12 [I/O of Integers], page 24.

mp_limb_t **mpz_getlimbn** (mpz_t *op*, mp_size_t *n*) — Function

Return limb #*n* from *op*. This function allows for very efficient decomposition of a number in its limbs.

The function `mpz_size` can be used to determine the useful range for *n*.

unsigned long int **mpz_get_ui** (mpz_t *op*) — Function

Return the least significant part from *op*. This function combined with `mpz_tdiv_q_2exp(..., op, CHAR_BIT*sizeof(unsigned long int))` can be used to decompose an integer into unsigned longs.

signed long int **mpz_get_si** (mpz_t *op*) — Function

If *op* fits into a `signed long int` return the value of *op*. Otherwise return the least significant part of *op*, with the same sign as *op*.

If *op* is too large to fit in a `signed long int`, the returned result is probably not very useful. To find out if the value will fit, use the function `mpz_fits_slong_p`.

double **mpz_get_d** (mpz_t *op*) — Function

Convert *op* to a double.

`char * mpz_get_str (char *str, int base, mpz_t op)` Function

Convert *op* to a string of digits in base *base*. The base may vary from 2 to 36.

If *str* is `NULL`, space for the result string is allocated using the default allocation function.

If *str* is not `NULL`, it should point to a block of storage enough large for the result. To find out the right amount of space to provide for *str*, use `mpz_sizeinbase (op, base) + 2`. The two extra bytes are for a possible minus sign, and for the terminating null character.

A pointer to the result string is returned. This pointer will will either equal *str*, or if that is `NULL`, will point to the allocated storage.

5.5 Arithmetic Functions

`void mpz_add (mpz_t rop, mpz_t op1, mpz_t op2)` Function
`void mpz_add_ui (mpz_t rop, mpz_t op1, unsigned long int op2)` Function

Set *rop* to $op1 + op2$.

`void mpz_sub (mpz_t rop, mpz_t op1, mpz_t op2)` Function
`void mpz_sub_ui (mpz_t rop, mpz_t op1, unsigned long int op2)` Function

Set *rop* to $op1 - op2$.

`void mpz_mul (mpz_t rop, mpz_t op1, mpz_t op2)` Function
`void mpz_mul_si (mpz_t rop, mpz_t op1, long int op2)` Function
`void mpz_mul_ui (mpz_t rop, mpz_t op1, unsigned long int op2)` Function

Set *rop* to $op1 \times op2$.

`void mpz_addmul_ui (mpz_t rop, mpz_t op1, unsigned long int op2)` Function

Set *rop* to $rop + op1 \times op2$.

`void mpz_mul_2exp (mpz_t rop, mpz_t op1, unsigned long int op2)` Function

Set *rop* to $op1 \times 2^{op2}$. This operation can also be defined as a left shift, *op2* steps.

`void mpz_neg (mpz_t rop, mpz_t op)` Function

Set *rop* to $-op$.

`void mpz_abs (mpz_t rop, mpz_t op)` Function

Set *rop* to the absolute value of *op*.

5.6 Division Functions

Division is undefined if the divisor is zero, and passing a zero divisor to the divide or modulo functions, as well passing a zero mod argument to the `mpz_powm` and `mpz_powm_ui` functions, will make these functions intentionally divide by zero. This lets the user handle arithmetic exceptions in these functions in the same manner as other arithmetic exceptions.

There are three main groups of division functions:

- Functions that truncate the quotient towards 0. The names of these functions start with `mpz_tdiv`. The '`t`' in the name is short for '`truncate`'.
- Functions that round the quotient towards $-\infty$ The names of these routines start with `mpz_fdiv`. The '`f`' in the name is short for '`floor`'.
- Functions that round the quotient towards $+\infty$ The names of these routines start with `mpz_cdiv`. The '`c`' in the name is short for '`ceil`'.

For each rounding mode, there are a couple of variants. Here '`q`' means that the quotient is computed, while '`r`' means that the remainder is computed. Functions that compute both the quotient and remainder have '`qr`' in the name.

void **mpz_tdiv_q** (mpz_t *q*, mpz_t *n*, mpz_t *d*) Function
unsigned long int **mpz_tdiv_q_ui** (mpz_t *q*, mpz_t *n*, unsigned long int *d*) Function

Set q to $[n/d]$, truncated towards 0.

The function `mpz_tdiv_q_ui` returns the absolute value of the true remainder.

void **mpz_tdiv_r** (mpz_t *r*, mpz_t *n*, mpz_t *d*) Function
unsigned long int **mpz_tdiv_r_ui** (mpz_t *r*, mpz_t *n*, unsigned long int *d*) Function

Set r to $(n - [n/d] \times d)$, where the quotient is truncated towards 0. Unless r becomes zero, it will get the same sign as n.

The function `mpz_tdiv_r_ui` returns the absolute value of the remainder.

void **mpz_tdiv_qr** (mpz_t *q*, mpz_t *r*, mpz_t *n*, mpz_t *d*) Function
unsigned long int **mpz_tdiv_qr_ui** (mpz_t *q*, mpz_t *r*, mpz_t *n*, unsigned long int *d*) Function

Set q to $[n/d]$, truncated towards 0. Set r to $(n - [n/d] \times d)$. Unless r becomes zero, it will get the same sign as n. If q and r are the same variable, the results are undefined.

The function `mpz_tdiv_qr_ui` returns the absolute value of the remainder.

unsigned long int **mpz_tdiv_ui** (mpz_t *n*, unsigned long int *d*) Function

Like `mpz_tdiv_r_ui`, but the remainder is not stored anywhere; its absolute value is just returned.

void **mpz_fdiv_q** (mpz_t q, mpz_t n, mpz_t d) Function
unsigned long int **mpz_fdiv_q_ui** (mpz_t q, mpz_t n, unsigned long int d) Function

Set q to $\lfloor n/d \rfloor$.

The function mpz_fdiv_q_ui returns the remainder.

void **mpz_fdiv_r** (mpz_t r, mpz_t n, mpz_t d) Function
unsigned long int **mpz_fdiv_r_ui** (mpz_t r, mpz_t n, unsigned long int d) Function

Set r to $(n - \lfloor n/d \rfloor \times d)$. Unless r becomes zero, it will get the same sign as d.

The function mpz_fdiv_r_ui returns the remainder.

void **mpz_fdiv_qr** (mpz_t q, mpz_t r, mpz_t n, mpz_t d) Function
unsigned long int **mpz_fdiv_qr_ui** (mpz_t q, mpz_t r, mpz_t n, unsigned long int d) Function

Set q to $\lfloor n/d \rfloor$. Set r to $(n - \lfloor n/d \rfloor \times d)$. Unless r becomes zero, it will get the same sign as d. If q and r are the same variable, the results are undefined.

The function mpz_fdiv_qr_ui returns the remainder.

unsigned long int **mpz_fdiv_ui** (mpz_t n, unsigned long int d) Function

Like mpz_fdiv_r_ui, but the remainder is not stored anywhere; it is just returned.

void **mpz_cdiv_q** (mpz_t q, mpz_t n, mpz_t d) Function
unsigned long int **mpz_cdiv_q_ui** (mpz_t q, mpz_t n, unsigned long int d) Function

Set q to $\lceil n/d \rceil$.

The function mpz_cdiv_q_ui returns the negated remainder.

void **mpz_cdiv_r** (mpz_t r, mpz_t n, mpz_t d) Function
unsigned long int **mpz_cdiv_r_ui** (mpz_t r, mpz_t n, unsigned long int d) Function

Set r to $(n - \lceil n/d \rceil \times d)$. Unless r becomes zero, it will get the opposite sign as d.

The function mpz_cdiv_r_ui returns the negated remainder.

void **mpz_cdiv_qr** (mpz_t q, mpz_t r, mpz_t n, mpz_t d) Function
unsigned long int **mpz_cdiv_qr_ui** (mpz_t q, mpz_t r, mpz_t n, unsigned long int d) Function

Set q to $\lceil n/d \rceil$. Set r to $(n - \lceil n/d \rceil \times d)$. Unless r becomes zero, it will get the opposite sign as d. If q and r are the same variable, the results are undefined.

The function mpz_cdiv_qr_ui returns the negated remainder.

unsigned long int **mpz_cdiv_ui** (mpz_t n, unsigned long int d) Function

Like mpz_tdiv_r_ui, but the remainder is not stored anywhere; its negated value is just returned.

void **mpz_mod** (mpz_t *r*, mpz_t *n*, mpz_t *d*) Function
unsigned long int **mpz_mod_ui** (mpz_t *r*, mpz_t *n*, unsigned long int *d*) Function

Set r to n `mod` d. The sign of the divisor is ignored; the result is always non-negative.

The function `mpz_mod_ui` returns the remainder.

void **mpz_divexact** (mpz_t *q*, mpz_t *n*, mpz_t *d*) Function

Set q to n/d. This function produces correct results only when it is known in advance that d divides n.

Since mpz_divexact is much faster than any of the other routines that produce the quotient (see [References], page 53 Jebelean), it is the best choice for instances in which exact division is known to occur, such as reducing a rational to lowest terms.

void **mpz_tdiv_q_2exp** (mpz_t *q*, mpz_t *n*, unsigned long int *d*) Function

Set q to $[n/2^d]$, truncated towards 0.

void **mpz_tdiv_r_2exp** (mpz_t *r*, mpz_t *n*, unsigned long int *d*) Function

Set r to $n - [n/2^d] \times 2^d$, where [ ] indicates rounding towards 0. Unless it is zero, r will have the same sign as n.

void **mpz_fdiv_q_2exp** (mpz_t *q*, mpz_t *n*, unsigned long int *d*) Function

Set q to $\lfloor n/2^d \rfloor$. This operation can also be defined as arithmetic right shift d bit positions.

void **mpz_fdiv_r_2exp** (mpz_t *r*, mpz_t *n*, unsigned long int *d*) Function

Set r to $n - \lfloor n/2^d \rfloor \times 2^d$. The sign of r will always be positive. This operation can also be defined as masking of the d least significant bits.

5.7 Exponentiation Functions

void **mpz_powm** (mpz_t *rop*, mpz_t *base*, mpz_t *exp*, mpz_t *mod*) Function
void **mpz_powm_ui** (mpz_t *rop*, mpz_t *base*, unsigned long int *exp*, mpz_t *mod*) Function

Set *rop* to $base^{exp}$ mod *mod*. If *exp* is negative, the result is undefined.

void **mpz_pow_ui** (mpz_t *rop*, mpz_t *base*, unsigned long int *exp*) Function
void **mpz_ui_pow_ui** (mpz_t *rop*, unsigned long int *base*, unsigned long int *exp*) Function

Set *rop* to $base^{exp}$. The case of 0^0 yields 1.

5.8 Root Extraction Functions

`int` **mpz_root** (`mpz_t` *rop*, `mpz_t` *op*, `unsigned long int` *n*) Function

Set *rop* to $\lfloor \sqrt[n]{op} \rfloor$, the truncated integer part of the *n*th root of *op*. Return non-zero if the computation was exact, i.e., if *op* is *rop* to the *n*th power.

`void` **mpz_sqrt** (`mpz_t` *rop*, `mpz_t` *op*) Function

Set *rop* to $\lfloor \sqrt{op} \rfloor$, the truncated integer part of the square root of *op*.

`void` **mpz_sqrtrem** (`mpz_t` *rop1*, `mpz_t` *rop2*, `mpz_t` *op*) Function

Set *rop1* to $\lfloor \sqrt{op} \rfloor$, like `mpz_sqrt`. Set *rop2* to $(op - rop1^2)$, (i.e., zero if *op* is a perfect square).

If *rop1* and *rop2* are the same variable, the results are undefined.

`int` **mpz_perfect_power_p** (`mpz_t` *op*) Function

Return non-zero if *op* is a perfect power, i.e., if there exist integers a and b, with $b > 1$, such that $op = a^b$. Return zero otherwise.

`int` **mpz_perfect_square_p** (`mpz_t` *op*) Function

Return non-zero if *op* is a perfect square, i.e., if the square root of *op* is an integer. Return zero otherwise.

5.9 Number Theoretic Functions

`int` **mpz_probab_prime_p** (`mpz_t` *n*, `int` *reps*) Function

If this function returns 0, n is definitely not prime. If it returns 1, then n is 'probably' prime. If it returns 2, then n is surely prime. Reasonable values of reps vary from 5 to 10; a higher value lowers the probability for a non-prime to pass as a 'probable' prime.

The function uses Miller-Rabin's probabilistic test.

`int` **mpz_nextprime** (`mpz_t` *rop*, `mpz_t` *op*) Function

Set *rop* to the next prime greater than *op*.

This function uses a probabilistic algorithm to identify primes, but for for practical purposes it's adequate, since the chance of a composite passing will be extremely small.

`void` **mpz_gcd** (`mpz_t` *rop*, `mpz_t` *op1*, `mpz_t` *op2*) Function

Set *rop* to the greatest common divisor of *op1* and *op2*. The result is always positive even if either of or both input operands are negative.

`unsigned long int` **mpz_gcd_ui** (`mpz_t` *rop*, `mpz_t` *op1*, `unsigned long int` *op2*) Function

Compute the greatest common divisor of *op1* and *op2*. If *rop* is not `NULL`, store the result there.

If the result is small enough to fit in an `unsigned long int`, it is returned. If the result does not fit, 0 is returned, and the result is equal to the argument *op1*. Note that the result will always fit if *op2* is non-zero.

`void` **mpz_gcdext** (`mpz_t` *g*, `mpz_t` *s*, `mpz_t` *t*, `mpz_t` *a*, `mpz_t` *b*) Function

Compute *g*, *s*, and *t*, such that $as + bt = g =$ `gcd`(a, b). If *t* is `NULL`, that argument is not computed.

`void` **mpz_lcm** (`mpz_t` *rop*, `mpz_t` *op1*, `mpz_t` *op2*) Function

Set *rop* to the least common multiple of *op1* and *op2*.

`int` **mpz_invert** (`mpz_t` *rop*, `mpz_t` *op1*, `mpz_t` *op2*) Function

Compute the inverse of *op1* modulo *op2* and put the result in *rop*. Return non-zero if an inverse exists, zero otherwise. When the function returns zero, *rop* is undefined.

`int` **mpz_jacobi** (`mpz_t` *op1*, `mpz_t` *op2*) Function
`int` **mpz_legendre** (`mpz_t` *op1*, `mpz_t` *op2*) Function

Compute the Jacobi and Legendre symbols, respectively. *op2* should be odd and must be positive.

`int` **mpz_si_kronecker** (`long` *a*, `mpz_t` *b*); Function
`int` **mpz_ui_kronecker** (`unsigned long` *a*, `mpz_t` *b*); Function
`int` **mpz_kronecker_si** (`mpz_t` *a*, `long` *b*); Function
`int` **mpz_kronecker_ui** (`mpz_t` *a*, `unsigned long` *b*); Function

Calculate the value of the Kronecker/Jacobi symbol $\left(\frac{a}{b}\right)$, with the Kronecker extension $\left(\frac{a}{2}\right) = \left(\frac{2}{a}\right)$ when a odd, or $\left(\frac{a}{2}\right) = 0$ when a even. All values of a and b give a well-defined result. See Henri Cohen, section 1.4.2, for more information (see [References], page 53). See also the example program '`demos/qcn.c`' which uses `mpz_kronecker_ui`.

`unsigned long int` **mpz_remove** (`mpz_t` *rop*, `mpz_t` *op*, `mpz_t` *f*) Function

Remove all occurrences of the factor *f* from *op* and store the result in *rop*. Return the multiplicity of *f* in *op*.

`void` **mpz_fac_ui** (`mpz_t` *rop*, `unsigned long int` *op*) Function

Set *rop* to *op*!, the factorial of *op*.

`void` **mpz_bin_ui** (`mpz_t` *rop*, `mpz_t` *n*, `unsigned long int` *k*) Function
`void` **mpz_bin_uiui** (`mpz_t` *rop*, `unsigned long int` *n*, `unsigned long int` *k*) Function

Compute the binomial coefficient $\binom{n}{k}$ and store the result in *rop*. Negative values of *n* are supported by `mpz_bin_ui`, using the identity $\binom{-n}{k} = (-1)^k \binom{n+k-1}{k}$ (see Knuth volume 1 section 1.2.6 part G).

`void` **mpz_fib_ui** (`mpz_t` *rop*, `unsigned long int` *n*) Function

Compute the *n*th Fibonacci number and store the result in *rop*.

5.10 Comparison Functions

`int` **mpz_cmp** (`mpz_t` *op1*, `mpz_t` *op2*) Function

Compare *op1* and *op2*. Return a positive value if $op1 > op2$, zero if $op1 = op2$, and a negative value if $op1 < op2$.

`int` **mpz_cmp_ui** (`mpz_t` *op1*, `unsigned long int` *op2*) Macro
`int` **mpz_cmp_si** (`mpz_t` *op1*, `signed long int` *op2*) Macro

Compare *op1* and *op2*. Return a positive value if $op1 > op2$, zero if $op1 = op2$, and a negative value if $op1 < op2$.

These functions are actually implemented as macros. They evaluate their arguments multiple times.

`int` **mpz_cmpabs** (`mpz_t` *op1*, `mpz_t` *op2*) Function
`int` **mpz_cmpabs_ui** (`mpz_t` *op1*, `unsigned long int` *op2*) Function

Compare the absolute values of *op1* and *op2*. Return a positive value if $|op1| > |op2|$, zero if $|op1| = |op2|$, and a negative value if $|op1| < |op2|$.

`int` **mpz_sgn** (`mpz_t` *op*) Macro

Return $+1$ if $op > 0$, 0 if $op = 0$, and -1 if $op < 0$.

This function is actually implemented as a macro. It evaluates its arguments multiple times.

5.11 Logical and Bit Manipulation Functions

These functions behave as if two's complement arithmetic were used (although sign-magnitude is used by the actual implementation).

`void` **mpz_and** (`mpz_t` *rop*, `mpz_t` *op1*, `mpz_t` *op2*) Function

Set *rop* to *op1* logical-and *op2*.

`void` **mpz_ior** (`mpz_t` *rop*, `mpz_t` *op1*, `mpz_t` *op2*) Function

Set *rop* to *op1* inclusive-or *op2*.

`void` **mpz_xor** (`mpz_t` *rop*, `mpz_t` *op1*, `mpz_t` *op2*) Function

Set *rop* to *op1* exclusive-or *op2*.

`void` **mpz_com** (`mpz_t` *rop*, `mpz_t` *op*) Function

Set *rop* to the one's complement of *op*.

`unsigned long int` **mpz_popcount** (`mpz_t` *op*) Function

For non-negative numbers, return the population count of *op*. For negative numbers, return the largest possible value (*MAX_ULONG*).

`unsigned long int` **mpz_hamdist** (`mpz_t` *op1*, `mpz_t` *op2*) Function

If *op1* and *op2* are both non-negative, return the hamming distance between the two operands. Otherwise, return the largest possible value (*MAX_ULONG*).

It is possible to extend this function to return a useful value when the operands are both negative, but the current implementation returns *MAX_ULONG* in this case. **Do not depend on this behavior, since it will change in a future release.**

`unsigned long int` **mpz_scan0** (`mpz_t` *op*, `unsigned long int` *starting_bit*) Function

Scan *op*, starting with bit *starting_bit*, towards more significant bits, until the first clear bit is found. Return the index of the found bit.

`unsigned long int` **mpz_scan1** (`mpz_t` *op*, `unsigned long int` *starting_bit*) Function

Scan *op*, starting with bit *starting_bit*, towards more significant bits, until the first set bit is found. Return the index of the found bit.

`void` **mpz_setbit** (`mpz_t` *rop*, `unsigned long int` *bit_index*) Function

Set bit *bit_index* in *rop*.

`void` **mpz_clrbit** (`mpz_t` *rop*, `unsigned long int` *bit_index*) Function

Clear bit *bit_index* in *rop*.

`int` **mpz_tstbit** (`mpz_t` *op*, `unsigned long int` *bit_index*) Function

Check bit *bit_index* in *op* and return 0 or 1 accordingly.

5.12 Input and Output Functions

Functions that perform input from a stdio stream, and functions that output to a stdio stream. Passing a `NULL` pointer for a *stream* argument to any of these functions will make them read from `stdin` and write to `stdout`, respectively.

When using any of these functions, it is a good idea to include '`stdio.h`' before '`gmp.h`', since that will allow '`gmp.h`' to define prototypes for these functions.

`size_t` **mpz_out_str** (`FILE` **stream*, `int` *base*, `mpz_t` *op*) Function

Output *op* on stdio stream *stream*, as a string of digits in base *base*. The base may vary from 2 to 36.

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

size_t **mpz_inp_str** (mpz_t *rop*, FILE **stream*, int *base*) Function

Input a possibly white-space preceded string in base *base* from stdio stream *stream*, and put the read integer in *rop*. The base may vary from 2 to 36. If *base* is 0, the actual base is determined from the leading characters: if the first two characters are '0x' or '0X', hexadecimal is assumed, otherwise if the first character is '0', octal is assumed, otherwise decimal is assumed.

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

size_t **mpz_out_raw** (FILE **stream*, mpz_t *op*) Function

Output *op* on stdio stream *stream*, in raw binary format. The integer is written in a portable format, with 4 bytes of size information, and that many bytes of limbs. Both the size and the limbs are written in decreasing significance order (i.e., in big-endian).

The output can be read with `mpz_inp_raw`.

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

The output of this can not be read by `mpz_inp_raw` from GMP 1, because of changes necessary for compatibility between 32-bit and 64-bit machines.

size_t **mpz_inp_raw** (mpz_t *rop*, FILE **stream*) Function

Input from stdio stream *stream* in the format written by `mpz_out_raw`, and put the result in *rop*. Return the number of bytes read, or if an error occurred, return 0.

This routine can read the output from `mpz_out_raw` also from GMP 1, in spite of changes necessary for compatibility between 32-bit and 64-bit machines.

5.13 Random Number Functions

The random number functions of GMP come in two groups; older function that rely on a global state, and newer functions that accept a state parameter that is read and modified. Please see the Chapter 9 [Random Number Functions], page 46 for more information on how to use and not to use random number functions.

void **mpz_urandomb** (mpz_t *rop*, gmp_randstate_t *state*, Function

unsigned long int *n*) Generate a uniformly distributed random integer in the range 0 to $2^n - 1$, inclusive.

The variable *state* must be initialized by calling one of the `gmp_randinit` functions (Section 9.1 [Random State Initialization], page 46) before invoking this function.

void **mpz_urandomm** (mpz_t *rop*, gmp_randstate_t *state*, Function

mpz_t *n*) Generate a uniform random integer in the range 0 to $n - 1$, inclusive.

The variable *state* must be initialized by calling one of the `gmp_randinit` functions (Section 9.1 [Random State Initialization], page 46) before invoking this function.

void **mpz_rrandomb** (mpz_t *rop*, gmp_randstate_t *state*, unsigned long int *n*) Function

Generate a random integer with long strings of zeros and ones in the binary representation. Useful for testing functions and algorithms, since this kind of random numbers have proven to be more likely to trigger corner-case bugs. The random number will be in the range 0 to $2^n - 1$, inclusive.

The variable *state* must be initialized by calling one of the `gmp_randinit` functions (Section 9.1 [Random State Initialization], page 46) before invoking this function.

void **mpz_random** (mpz_t *rop*, mp_size_t *max_size*) Function

Generate a random integer of at most *max_size* limbs. The generated random number doesn't satisfy any particular requirements of randomness. Negative random numbers are generated when *max_size* is negative.

This function is obsolete. Use `mpz_urandomb` or `mpz_urandomm` instead.

void **mpz_random2** (mpz_t *rop*, mp_size_t *max_size*) Function

Generate a random integer of at most *max_size* limbs, with long strings of zeros and ones in the binary representation. Useful for testing functions and algorithms, since this kind of random numbers have proven to be more likely to trigger corner-case bugs. Negative random numbers are generated when *max_size* is negative.

This function is obsolete. Use `mpz_rrandomb` instead.

5.14 Miscellaneous Functions

int **mpz_fits_ulong_p** (mpz_t *op*) Function
int **mpz_fits_slong_p** (mpz_t *op*) Function
int **mpz_fits_uint_p** (mpz_t *op*) Function
int **mpz_fits_sint_p** (mpz_t *op*) Function
int **mpz_fits_ushort_p** (mpz_t *op*) Function
int **mpz_fits_sshort_p** (mpz_t *op*) Function

Return non-zero iff the value of *op* fits in an `unsigned long int`, `signed long int`, `unsigned int`, `signed int`, `unsigned short int`, or `signed short int`, respectively. Otherwise, return zero.

int **mpz_odd_p** (mpz_t *op*) Macro
int **mpz_even_p** (mpz_t *op*) Macro

Determine whether *op* is odd or even, respectively. Return non-zero if yes, zero if no. These macros evaluate their arguments more than once.

size_t **mpz_size** (mpz_t *op*) Function

Return the size of *op* measured in number of limbs. If *op* is zero, the returned value will be zero.

`size_t` **mpz_sizeinbase** (`mpz_t` *op*, `int` *base*) Function

Return the size of *op* measured in number of digits in base *base*. The base may vary from 2 to 36. The returned value will be exact or 1 too big. If *base* is a power of 2, the returned value will always be exact.

This function is useful in order to allocate the right amount of space before converting *op* to a string. The right amount of allocation is normally two more than the value returned by `mpz_sizeinbase` (one extra for a minus sign and one for the terminating '\0').

6 Rational Number Functions

This chapter describes the GMP functions for performing arithmetic on rational numbers. These functions start with the prefix `mpq_`.

Rational numbers are stored in objects of type `mpq_t`.

All rational arithmetic functions assume operands have a canonical form, and canonicalize their result. The canonical from means that the denominator and the numerator have no common factors, and that the denominator is positive. Zero has the unique representation 0/1.

Pure assignment functions do not canonicalize the assigned variable. It is the responsibility of the user to canonicalize the assigned variable before any arithmetic operations are performed on that variable. **Note that this is an incompatible change from version 1 of the library.**

`void` **mpq_canonicalize** (`mpq_t` *op*) Function

Remove any factors that are common to the numerator and denominator of *op*, and make the denominator positive.

6.1 Initialization and Assignment Functions

`void` **mpq_init** (`mpq_t` *dest_rational*) Function

Initialize *dest_rational* and set it to 0/1. Each variable should normally only be initialized once, or at least cleared out (using the function `mpq_clear`) between each initialization.

`void` **mpq_clear** (`mpq_t` *rational_number*) Function

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

`void` **mpq_set** (`mpq_t` *rop*, `mpq_t` *op*) Function
`void` **mpq_set_z** (`mpq_t` *rop*, `mpz_t` *op*) Function

Assign *rop* from *op*.

`void` **mpq_set_ui** (`mpq_t` *rop*, `unsigned long int` *op1*, `unsigned long int` *op2*) Function
`void` **mpq_set_si** (`mpq_t` *rop*, `signed long int` *op1*, `unsigned long int` *op2*) Function

Set the value of *rop* to *op1*/*op2*. Note that if *op1* and *op2* have common factors, *rop* has to be passed to `mpq_canonicalize` before any operations are performed on *rop*.

`void` **mpq_swap** (`mpq_t` *rop1*, `mpq_t` *rop2*) Function

Swap the values *rop1* and *rop2* efficiently.

6.2 Arithmetic Functions

void **mpq_add** (`mpq_t` *sum*, `mpq_t` *addend1*, `mpq_t` *addend2*) Function

Set *sum* to *addend1* + *addend2*.

void **mpq_sub** (`mpq_t` *difference*, `mpq_t` *minuend*, `mpq_t` *subtrahend*) Function

Set *difference* to *minuend* − *subtrahend*.

void **mpq_mul** (`mpq_t` *product*, `mpq_t` *multiplier*, `mpq_t` *multiplicand*) Function

Set *product* to *multiplier* × *multiplicand*.

void **mpq_div** (`mpq_t` *quotient*, `mpq_t` *dividend*, `mpq_t` *divisor*) Function

Set *quotient* to *dividend*/*divisor*.

void **mpq_neg** (`mpq_t` *negated_operand*, `mpq_t` *operand*) Function

Set *negated_operand* to −*operand*.

void **mpq_inv** (`mpq_t` *inverted_number*, `mpq_t` *number*) Function

Set *inverted_number* to 1/*number*. If the new denominator is zero, this routine will divide by zero.

6.3 Comparison Functions

int **mpq_cmp** (`mpq_t` *op1*, `mpq_t` *op2*) Function

Compare *op1* and *op2*. Return a positive value if $op1 > op2$, zero if $op1 = op2$, and a negative value if $op1 < op2$.

To determine if two rationals are equal, `mpq_equal` is faster than `mpq_cmp`.

int **mpq_cmp_ui** (`mpq_t` *op1*, `unsigned long int` *num2*, `unsigned long int` *den2*) Macro

Compare *op1* and *num2*/*den2*. Return a positive value if $op1 > num2/den2$, zero if $op1 = num2/den2$, and a negative value if $op1 < num2/den2$.

This routine allows that *num2* and *den2* have common factors.

This function is actually implemented as a macro. It evaluates its arguments multiple times.

int **mpq_sgn** (`mpq_t` *op*) Macro

Return $+1$ if $op > 0$, 0 if $op = 0$, and -1 if $op < 0$.

This function is actually implemented as a macro. It evaluates its arguments multiple times.

int **mpq_equal** (`mpq_t` *op1*, `mpq_t` *op2*) Function

Return non-zero if *op1* and *op2* are equal, zero if they are non-equal. Although `mpq_cmp` can be used for the same purpose, this function is much faster.

6.4 Applying Integer Functions to Rationals

The set of `mpq` functions is quite small. In particular, there are few functions for either input or output. But there are two macros that allow us to apply any `mpz` function on the numerator or denominator of a rational number. If these macros are used to assign to the rational number, `mpq_canonicalize` normally need to be called afterwards.

`mpz_t` **mpq_numref** (`mpq_t` *op*) Macro
`mpz_t` **mpq_denref** (`mpq_t` *op*) Macro

Return a reference to the numerator and denominator of *op*, respectively. The `mpz` functions can be used on the result of these macros.

6.5 Input and Output Functions

Functions that perform input from a stdio stream, and functions that output to a stdio stream. Passing a `NULL` pointer for a *stream* argument to any of these functions will make them read from `stdin` and write to `stdout`, respectively.

When using any of these functions, it is a good idea to include '`stdio.h`' before '`gmp.h`', since that will allow '`gmp.h`' to define prototypes for these functions.

`size_t` **mpq_out_str** (`FILE *`*stream*, `int` *base*, `mpq_t` *op*) Function

Output *op* on stdio stream *stream*, as a string of digits in base *base*. The base may vary from 2 to 36. Output is in the form '`num/den`' or if the denominator is 1 then just '`num`'.

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

6.6 Miscellaneous Functions

`double` **mpq_get_d** (`mpq_t` *op*) Function

Convert *op* to a double.

`void` **mpq_set_d** (`mpq_t` *rop*, `double` *d*) Function

Set *rop* to the value of d, without rounding.

These functions assign between either the numerator or denominator of a rational, and an integer. Instead of using these functions, it is preferable to use the more general mechanisms `mpq_numref` and `mpq_denref`, together with `mpz_set`.

`void` **mpq_set_num** (`mpq_t` *rational*, `mpz_t` *numerator*) Function

Copy *numerator* to the numerator of *rational*. When this risks to make the numerator and denominator of *rational* have common factors, you have to pass *rational* to `mpq_canonicalize` before any operations are performed on *rational*.

This function is equivalent to `mpz_set (mpq_numref (`*rational*`), `*numerator*`)`.

void **mpq_set_den** (mpq_t *rational*, mpz_t *denominator*) Function

Copy *denominator* to the denominator of *rational*. When this risks to make the numerator and denominator of *rational* have common factors, or if the denominator might be negative, you have to pass *rational* to `mpq_canonicalize` before any operations are performed on *rational*.

In version 1 of the library, negative denominators were handled by copying the sign to the numerator. That is no longer done.

This function is equivalent to `mpz_set (mpq_denref (`*rational*`), `*denominators*`)`.

void **mpq_get_num** (mpz_t *numerator*, mpq_t *rational*) Function

Copy the numerator of *rational* to the integer *numerator*, to prepare for integer operations on the numerator.

This function is equivalent to `mpz_set (`*numerator*`, mpq_numref (`*rational*`))`.

void **mpq_get_den** (mpz_t *denominator*, mpq_t *rational*) Function

Copy the denominator of *rational* to the integer *denominator*, to prepare for integer operations on the denominator.

This function is equivalent to `mpz_set (`*denominator*`, mpq_denref (`*rational*`))`.

7 Floating-point Functions

This chapter describes the GMP functions for performing floating point arithmetic. These functions start with the prefix `mpf_`.

GMP floating point numbers are stored in objects of type `mpf_t`.

The GMP floating-point functions have an interface that is similar to the GMP integer functions. The function prefix for floating-point operations is `mpf_`.

There is one significant characteristic of floating-point numbers that has motivated a difference between this function class and other GMP function classes: the inherent inexactness of floating point arithmetic. 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 precision of variables used as input is ignored.

The precision of a calculation is defined as follows: Compute the requested operation exactly (with "infinite precision"), and truncate the result to the destination variable precision. Even if the user has asked for a very high precision, GMP will not calculate with superfluous digits. For example, if two low-precision numbers of nearly equal magnitude are added, the precision of the result will be limited to what is required to represent the result accurately.

The GMP floating-point functions are *not* intended as a smooth extension to the IEEE P754 arithmetic. Specifically, the results obtained on one computer often differs from the results obtained on a computer with a different word size.

7.1 Initialization Functions

`void` **mpf_set_default_prec** (`unsigned long int` *prec*) Function

Set the default precision to be **at least** *prec* bits. All subsequent calls to `mpf_init` will use this precision, but previously initialized variables are unaffected.

An `mpf_t` object must be initialized before storing the first value in it. The functions `mpf_init` and `mpf_init2` are used for that purpose.

`void` **mpf_init** (`mpf_t` *x*) Function

Initialize *x* to 0. Normally, a variable should be initialized once only or at least be cleared, using `mpf_clear`, between initializations. The precision of *x* is undefined unless a default precision has already been established by a call to `mpf_set_default_prec`.

`void` **mpf_init2** (`mpf_t` *x*, `unsigned long int` *prec*) Function

Initialize *x* to 0 and set its precision to be **at least** *prec* bits. Normally, a variable should be initialized once only or at least be cleared, using `mpf_clear`, between initializations.

`void` **mpf_clear** (`mpf_t` *x*) Function

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

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

```
{
  mpf_t x, y;
  mpf_init (x); /* use default precision */
  mpf_init2 (y, 256); /* precision at least 256 bits */
  ...
  /* Unless the program is about to exit, do ... */
  mpf_clear (x);
  mpf_clear (y);
}
```

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

`void` **mpf_set_prec** (`mpf_t` *rop*, `unsigned long int` *prec*) Function

Set the precision of *rop* to be **at least** *prec* bits. Since changing the precision involves calls to `realloc`, this routine should not be called in a tight loop.

`unsigned long int` **mpf_get_prec** (`mpf_t` *op*) Function

Return the precision actually used for assignments of *op*.

`void` **mpf_set_prec_raw** (`mpf_t` *rop*, `unsigned long int` *prec*) Function

Set the precision of *rop* to be **at least** *prec* bits. This is a low-level function that does not change the allocation. The *prec* argument must not be larger that the precision previously returned by `mpf_get_prec`. It is crucial that the precision of *rop* is ultimately reset to exactly the value returned by `mpf_get_prec` before the first call to `mpf_set_prec_raw`.

7.2 Assignment Functions

These functions assign new values to already initialized floats (see Section 7.1 [Initializing Floats], page 32).

`void` **mpf_set** (`mpf_t` *rop*, `mpf_t` *op*) Function
`void` **mpf_set_ui** (`mpf_t` *rop*, `unsigned long int` *op*) Function
`void` **mpf_set_si** (`mpf_t` *rop*, `signed long int` *op*) Function
`void` **mpf_set_d** (`mpf_t` *rop*, `double` *op*) Function
`void` **mpf_set_z** (`mpf_t` *rop*, `mpz_t` *op*) Function
`void` **mpf_set_q** (`mpf_t` *rop*, `mpq_t` *op*) Function

Set the value of *rop* from *op*.

`int` **mpf_set_str** (`mpf_t` *rop*, `char` **str*, `int` *base*) Function

Set the value of *rop* from the string in *str*. The string is of the form '`M@N`' or, if the base is 10 or less, alternatively '`MeN`'. '`M`' is the mantissa and '`N`' is the exponent. The mantissa is always in the specified base. The exponent is either in the specified base or, if *base* is negative, in decimal.

The argument *base* may be in the ranges 2 to 36, or −36 to −2. Negative values are used to specify that the exponent is in decimal.

Unlike the corresponding `mpz` function, the base will not be determined from the leading characters of the string if *base* is 0. This is so that numbers like '`0.23`' are not interpreted as octal.

White space is allowed in the string, and is simply ignored. [This is not really true; white-space is ignored in the beginning of the string and within the mantissa, but not in other places, such as after a minus sign or in the exponent. We are considering changing the definition of this function, making it fail when there is any white-space in the input, since that makes a lot of sense. Please tell us your opinion about this change. Do you really want it to accept "3 14" as meaning 314 as it does now?]

This function returns 0 if the entire string up to the '\0' is a valid number in base *base*. Otherwise it returns −1.

`void` **mpf_swap** (`mpf_t` *rop1*, `mpf_t` *rop2*) Function

Swap the values *rop1* and *rop2* efficiently.

7.3 Combined Initialization and Assignment Functions

For convenience, GMP provides a parallel series of initialize-and-set functions which initialize the output and then store the value there. These functions' names have the form `mpf_init_set...`

Once the float has been initialized by any of the `mpf_init_set...` functions, it can be used as the source or destination operand for the ordinary float functions. Don't use an initialize-and-set function on a variable already initialized!

`void` **mpf_init_set** (`mpf_t` *rop*, `mpf_t` *op*) Function
`void` **mpf_init_set_ui** (`mpf_t` *rop*, `unsigned long int` *op*) Function
`void` **mpf_init_set_si** (`mpf_t` *rop*, `signed long int` *op*) Function
`void` **mpf_init_set_d** (`mpf_t` *rop*, `double` *op*) Function

Initialize *rop* and set its value from *op*.

The precision of *rop* will be taken from the active default precision, as set by `mpf_set_default_prec`.

`int` **mpf_init_set_str** (`mpf_t` *rop*, `char` **str*, `int` *base*) Function

Initialize *rop* and set its value from the string in *str*. See `mpf_set_str` above for details on the assignment operation.

Note that *rop* is initialized even if an error occurs. (I.e., you have to call `mpf_clear` for it.)

The precision of *rop* will be taken from the active default precision, as set by `mpf_set_default_prec`.

7.4 Conversion Functions

`double` **mpf_get_d** (`mpf_t` *op*) Function

Convert *op* to a double.

`char *` **mpf_get_str** (`char` **str*, `mp_exp_t` **expptr*, `int` *base*, `size_t` *n_digits*, `mpf_t` *op*) Function

Convert *op* to a string of digits in base *base*. The base may vary from 2 to 36. Generate at most *n_digits* significant digits, or if *n_digits* is 0, the maximum number of digits accurately representable by *op*.

If *str* is `NULL`, space for the mantissa is allocated using the default allocation function.

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

The exponent is written through the pointer *expptr*.

If *n_digits* is 0, the maximum number of digits meaningfully achievable from the precision of *op* will be generated. Note that the space requirements for *str* in this case will be impossible for the user to predetermine. Therefore, you need to pass `NULL` for the string argument whenever *n_digits* is 0.

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*.

A pointer to the result string is returned. This pointer will will either equal *str*, or if that is `NULL`, will point to the allocated storage.

7.5 Arithmetic Functions

`void` **mpf_add** (`mpf_t` *rop*, `mpf_t` *op1*, `mpf_t` *op2*) Function
`void` **mpf_add_ui** (`mpf_t` *rop*, `mpf_t` *op1*, `unsigned long int` *op2*) Function

Set *rop* to $op1 + op2$.

`void` **mpf_sub** (`mpf_t` *rop*, `mpf_t` *op1*, `mpf_t` *op2*) Function
`void` **mpf_ui_sub** (`mpf_t` *rop*, `unsigned long int` *op1*, `mpf_t` *op2*) Function
`void` **mpf_sub_ui** (`mpf_t` *rop*, `mpf_t` *op1*, `unsigned long int` *op2*) Function

Set *rop* to $op1 - op2$.

`void` **mpf_mul** (`mpf_t` *rop*, `mpf_t` *op1*, `mpf_t` *op2*) Function
`void` **mpf_mul_ui** (`mpf_t` *rop*, `mpf_t` *op1*, `unsigned long int` *op2*) Function

Set *rop* to $op1 \times op2$.

Division is undefined if the divisor is zero, and passing a zero divisor to the divide functions will make these functions intentionally divide by zero. This lets the user handle arithmetic exceptions in these functions in the same manner as other arithmetic exceptions.

void **mpf_div** (mpf_t *rop*, mpf_t *op1*, mpf_t *op2*) Function
void **mpf_ui_div** (mpf_t *rop*, unsigned long int *op1*, mpf_t *op2*) Function
void **mpf_div_ui** (mpf_t *rop*, mpf_t *op1*, unsigned long int *op2*) Function

Set *rop* to $op1/op2$.

void **mpf_sqrt** (mpf_t *rop*, mpf_t *op*) Function
void **mpf_sqrt_ui** (mpf_t *rop*, unsigned long int *op*) Function

Set *rop* to $\sqrt{op}$.

void **mpf_pow_ui** (mpf_t *rop*, mpf_t *op1*, unsigned long int *op2*) Function

Set *rop* to $op1^{op2}$.

void **mpf_neg** (mpf_t *rop*, mpf_t *op*) Function

Set *rop* to $-op$.

void **mpf_abs** (mpf_t *rop*, mpf_t *op*) Function

Set *rop* to the absolute value of *op*.

void **mpf_mul_2exp** (mpf_t *rop*, mpf_t *op1*, unsigned long int *op2*) Function

Set *rop* to $op1 \times 2^{op2}$.

void **mpf_div_2exp** (mpf_t *rop*, mpf_t *op1*, unsigned long int *op2*) Function

Set *rop* to $op1/2^{op2}$.

7.6 Comparison Functions

int **mpf_cmp** (mpf_t *op1*, mpf_t *op2*) Function
int **mpf_cmp_ui** (mpf_t *op1*, unsigned long int *op2*) Function
int **mpf_cmp_si** (mpf_t *op1*, signed long int *op2*) Function

Compare *op1* and *op2*. Return a positive value if $op1 > op2$, zero if $op1 = op2$, and a negative value if $op1 < op2$.

int **mpf_eq** (mpf_t *op1*, mpf_t *op2*, unsigned long int op3) Function

Return non-zero if the first *op3* bits of *op1* and *op2* are equal, zero otherwise. I.e., test of *op1* and *op2* are approximately equal.

void **mpf_reldiff** (mpf_t *rop*, mpf_t *op1*, mpf_t *op2*) Function

Compute the relative difference between *op1* and *op2* and store the result in *rop*.

int **mpf_sgn** (mpf_t *op*) Macro

Return $+1$ if $op > 0$, 0 if $op = 0$, and -1 if $op < 0$.

This function is actually implemented as a macro. It evaluates its arguments multiple times.

7.7 Input and Output Functions

Functions that perform input from a stdio stream, and functions that output to a stdio stream. Passing a `NULL` pointer for a *stream* argument to any of these functions will make them read from `stdin` and write to `stdout`, respectively.

When using any of these functions, it is a good idea to include '`stdio.h`' before '`gmp.h`', since that will allow '`gmp.h`' to define prototypes for these functions.

`size_t` **mpf_out_str** (`FILE` **stream*, `int` *base*, `size_t` *n_digits*, `mpf_t` *op*) — Function

Output *op* on stdio stream *stream*, as a string of digits in base *base*. The base may vary from 2 to 36. Print at most *n_digits* significant digits, or if *n_digits* is 0, the maximum number of digits accurately representable by *op*.

In addition to the significant digits, a leading '`0.`' and a trailing exponent, in the form '`eNNN`', are printed. If *base* is greater than 10, '`@`' will be used instead of '`e`' as exponent delimiter.

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

`size_t` **mpf_inp_str** (`mpf_t` *rop*, `FILE` **stream*, `int` *base*) — Function

Input a string in base *base* from stdio stream *stream*, and put the read float in *rop*. The string is of the form '`M@N`' or, if the base is 10 or less, alternatively '`MeN`'. '`M`' is the mantissa and '`N`' is the exponent. The mantissa is always in the specified base. The exponent is either in the specified base or, if *base* is negative, in decimal.

The argument *base* may be in the ranges 2 to 36, or -36 to -2. Negative values are used to specify that the exponent is in decimal.

Unlike the corresponding `mpz` function, the base will not be determined from the leading characters of the string if *base* is 0. This is so that numbers like '`0.23`' are not interpreted as octal.

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

7.8 Miscellaneous Functions

`void` **mpf_ceil** (`mpf_t` *rop*, `mpf_t` *op*) — Function
`void` **mpf_floor** (`mpf_t` *rop*, `mpf_t` *op*) — Function
`void` **mpf_trunc** (`mpf_t` *rop*, `mpf_t` *op*) — Function

Set *rop* to *op* rounded to an integer. `mpf_ceil` rounds to the next higher integer, `mpf_floor` to the next lower, and `mpf_trunc` to the integer towards zero.

`void` **mpf_urandomb** (`mpf_t` *rop*, `gmp_randstate_t` *state*, `unsigned long int` *nbits*) — Function

Generate a uniformly distributed random float in *rop*, such that $0 <= rop < 1$, with *nbits* significant bits in the mantissa.

The variable *state* must be initialized by calling one of the `gmp_randinit` functions (Section 9.1 [Random State Initialization], page 46) before invoking this function.

`void` **mpf_random2** (`mpf_t` *rop*, `mp_size_t` *max_size*, `mp_exp_t` *max_exp*) Function

Generate a random float of at most *max_size* limbs, with long strings of zeros and ones in the binary representation. The exponent of the number is in the interval $-exp$ to exp. This function is useful for testing functions and algorithms, since this kind of random numbers have proven to be more likely to trigger corner-case bugs. Negative random numbers are generated when *max_size* is negative.

8 Low-level Functions

This chapter describes low-level GMP functions, used to implement the high-level GMP functions, but also intended for time-critical user code.

These functions start with the prefix `mpn_`.

The `mpn` functions are designed to be as fast as possible, **not** to provide a coherent calling interface. The different functions have somewhat similar interfaces, but there are variations that make them hard to use. These functions do as little as possible apart from the real multiple precision computation, so that no time is spent on things that not all callers need.

A source operand is specified by a pointer to the least significant limb and a limb count. A destination operand is specified by just a pointer. It is the responsibility of the caller to ensure that the destination has enough space for storing the result.

With this way of specifying operands, it is possible to perform computations on subranges of an argument, and store the result into a subrange of a destination.

A common requirement for all functions is that each source area needs at least one limb. No size argument may be zero. Unless otherwise stated, in-place operations are allowed where source and destination are the same, but not where they only partly overlap.

The `mpn` functions are the base for the implementation of the `mpz_`, `mpf_`, and `mpq_` functions.

This example adds the number beginning at *s1p* and the number beginning at *s2p* and writes the sum at *destp*. All areas have *size* limbs.

```
cy = mpn_add_n (destp, s1p, s2p, size)
```

In the notation used here, a source operand is identified by the pointer to the least significant limb, and the limb count in braces. For example, {s1p, s1size}.

`mp_limb_t` **mpn_add_n** (`mp_limb_t *`*rp*, `const mp_limb_t *`*s1p*, `const mp_limb_t *`*s2p*, `mp_size_t` *size*) — Function

Add {*s1p*, *size*} and {*s2p*, *size*}, and write the *size* least significant limbs of the result to *rp*. Return carry, either 0 or 1.

This is the lowest-level function for addition. It is the preferred function for addition, since it is written in assembly for most targets. For addition of a variable to itself (i.e., *s1p* equals *s2p*, use `mpn_lshift` with a count of 1 for optimal speed.

`mp_limb_t` **mpn_add_1** (`mp_limb_t *`*rp*, `const mp_limb_t *`*s1p*, `mp_size_t` *size*, `mp_limb_t` *s2limb*) — Function

Add {*s1p*, *size*} and *s2limb*, and write the *size* least significant limbs of the result to *rp*. Return carry, either 0 or 1.

`mp_limb_t` **mpn_add** (`mp_limb_t *`*rp*, `const mp_limb_t *`*s1p*, `mp_size_t` *s1size*, `const mp_limb_t *`*s2p*, `mp_size_t` *s2size*) — Function

Add {*s1p*, *s1size*} and {*s2p*, *s2size*}, and write the *s1size* least significant limbs of the result to *rp*. Return carry, either 0 or 1.

This function requires that *s1size* is greater than or equal to *s2size*.

`mp_limb_t` **mpn_sub_n** (`mp_limb_t` **rp*, `const mp_limb_t` **s1p*, `const mp_limb_t` **s2p*, `mp_size_t` *size*) Function

Subtract {*s2p*, *s2size*} from {*s1p*, *size*}, and write the *size* least significant limbs of the result to *rp*. Return borrow, either 0 or 1.

This is the lowest-level function for subtraction. It is the preferred function for subtraction, since it is written in assembly for most targets.

`mp_limb_t` **mpn_sub_1** (`mp_limb_t` **rp*, `const mp_limb_t` **s1p*, `mp_size_t` *size*, `mp_limb_t` *s2limb*) Function

Subtract *s2limb* from {*s1p*, *size*}, and write the *size* least significant limbs of the result to *rp*. Return borrow, either 0 or 1.

`mp_limb_t` **mpn_sub** (`mp_limb_t` **rp*, `const mp_limb_t` **s1p*, `mp_size_t` *s1size*, `const mp_limb_t` **s2p*, `mp_size_t` *s2size*) Function

Subtract {*s2p*, *s2size*} from {*s1p*, *s1size*}, and write the *s1size* least significant limbs of the result to *rp*. Return borrow, either 0 or 1.

This function requires that *s1size* is greater than or equal to *s2size*.

`void` **mpn_mul_n** (`mp_limb_t` **rp*, `const mp_limb_t` **s1p*, `const mp_limb_t` **s2p*, `mp_size_t` *size*) Function

Multiply {*s1p*, *size*} and {*s2p*, *size*}, and write the **entire** result to *rp*.

The destination has to have space for 2**size* limbs, even if the significant result might be one limb smaller.

`mp_limb_t` **mpn_mul_1** (`mp_limb_t` **rp*, `const mp_limb_t` **s1p*, `mp_size_t` *size*, `mp_limb_t` *s2limb*) Function

Multiply {*s1p*, *size*} and *s2limb*, and write the *size* least significant limbs of the product to *rp*. Return the most significant limb of the product.

This is a low-level function that is a building block for general multiplication as well as other operations in GMP. It is written in assembly for most targets.

Don't call this function if *s2limb* is a power of 2; use `mpn_lshift` with a count equal to the logarithm of *s2limb* instead, for optimal speed.

`mp_limb_t` **mpn_addmul_1** (`mp_limb_t` **rp*, `const mp_limb_t` **s1p*, `mp_size_t` *size*, `mp_limb_t` *s2limb*) Function

Multiply {*s1p*, *size*} and *s2limb*, and add the *size* least significant limbs of the product to {*rp*, *size*} and write the result to *rp*. Return the most significant limb of the product, plus carry-out from the addition.

This is a low-level function that is a building block for general multiplication as well as other operations in GMP. It is written in assembly for most targets.

mp_limb_t **mpn_submul_1** (mp_limb_t **rp*, const mp_limb_t **s1p*, mp_size_t *size*, mp_limb_t *s2limb*) Function

Multiply {*s1p*, *size*} and *s2limb*, and subtract the *size* least significant limbs of the product from {*rp*, *size*} and write the result to *rp*. Return the most significant limb of the product, minus borrow-out from the subtraction.

This is a low-level function that is a building block for general multiplication and division as well as other operations in GMP. It is written in assembly for most targets.

mp_limb_t **mpn_mul** (mp_limb_t **rp*, const mp_limb_t **s1p*, mp_size_t *s1size*, const mp_limb_t **s2p*, mp_size_t *s2size*) Function

Multiply {*s1p*, *s1size*} and {*s2p*, *s2size*}, and write the result to *rp*. Return the most significant limb of the result.

The destination has to have space for *s1size* + *s2size* limbs, even if the result might be one limb smaller.

This function requires that *s1size* is greater than or equal to *s2size*. The destination must be distinct from either input operands.

void **mpn_tdiv_qr** (mp_limb_t **qp*, mp_limb_t **rp*, mp_size_t *qxn*, const mp_limb_t **np*, mp_size_t *nn*, const mp_limb_t **dp*, mp_size_t *dn*) Function

Divide {*np*, *nn*} by {*dp*, *dn*}. Write the quotient at *qp* and the remainder at *rp*.

The quotient written at *qp* will be $nn - dn + 1$ limbs. The remainder written at *rp* will be *dn* limbs.

It is required that *nn* is greater than or equal to *dn*. The *qxn* operand must be zero.

The quotient is rounded towards 0.

No overlap between arguments is permitted.

mp_limb_t **mpn_divrem** (mp_limb_t **r1p*, mp_size_t *xsize*, mp_limb_t **rs2p*, mp_size_t *rs2size*, const mp_limb_t **s3p*, mp_size_t *s3size*) Function

[This function is obsolete. Please call `mpn_tdiv_qr` instead for best performance.]

Divide {*rs2p*, *rs2size*} by {*s3p*, *s3size*}, and write the quotient at *r1p*, with the exception of the most significant limb, which is returned. The remainder replaces the dividend at *rs2p*; it will be *s3size* limbs long (i.e., as many limbs as the divisor).

In addition to an integer quotient, *xsize* fraction limbs are developed, and stored after the integral limbs. For most usages, *xsize* will be zero.

It is required that *rs2size* is greater than or equal to *s3size*. It is required that the most significant bit of the divisor is set.

If the quotient is not needed, pass *rs2p* + *s3size* as *r1p*. Aside from that special case, no overlap between arguments is permitted.

Return the most significant limb of the quotient, either 0 or 1.

The area at *r1p* needs to be $rs2size - s3size + xsize$ limbs large.

mp_limb_t **mpn_divrem_1** (mp_limb_t **r1p*, mp_size_t *xsize*, mp_limb_t **s2p*, mp_size_t *s2size*, mp_limb_t *s3limb*) — Function

mp_limb_t **mpn_divmod_1** (mp_limb_t **r1p*, mp_limb_t **s2p*, mp_size_t *s2size*, mp_limb_t *s3limb*) — Macro

Divide {*s2p*, *s2size*} by *s3limb*, and write the quotient at *r1p*. Return the remainder.

The integer quotient is written to {*r1p*+*xsize*, *s2size*} and in addition *xsize* fraction limbs are developed and written to {*r1p*, *xsize*}. Either or both *s2size* and *xsize* can be zero. For most usages, *xsize* will be zero.

`mpn_divmod_1` exists for upward source compatibility and is simply a macro calling `mpn_divrem_1` with an *xsize* of 0.

The areas at *r1p* and *s2p* have to be identical or completely separate, not partially overlapping.

mp_limb_t **mpn_divmod** (mp_limb_t **r1p*, mp_limb_t **rs2p*, mp_size_t *rs2size*, const mp_limb_t **s3p*, mp_size_t *s3size*) — Function

This interface is obsolete. It will disappear from future releases. Use `mpn_divrem` in its stead.

mp_limb_t **mpn_divexact_by3** (mp_limb_t **rp*, mp_limb_t **sp*, mp_size_t *size*) — Macro

mp_limb_t **mpn_divexact_by3c** (mp_limb_t **rp*, mp_limb_t **sp*, mp_size_t *size*, mp_limb_t *carry*) — Function

Divide {*sp*, *size*} by 3, expecting it to divide exactly, and writing the result to {*rp*, *size*}. If 3 divides exactly, the return value is zero and the result is the quotient. If not, the return value is non-zero and the result won't be anything useful.

`mpn_divexact_by3c` takes an initial carry parameter, which can be the return value from a previous call, so a large calculation can be done piece by piece. `mpn_divexact_by3` is simply a macro calling `mpn_divexact_by3c` with a 0 carry parameter.

These routines use a multiply-by-inverse and will be faster than `mpn_divrem_1` on CPUs with fast multiplication but slow division.

The source a, result q, size n, initial carry i, and return value c satisfy $cb^n + a - i = 3q$, where b is the size of a limb (2^{32} or 2^{64}). c is always 0, 1 or 2, and the initial carry must also be 0, 1 or 2 (these are both borrows really). When $c = 0$, clearly $q = (a - i)/3$. When $c \neq 0$, the remainder $(a - i) \mathit{mod}\, 3$ is given by $3 - c$, because $b \equiv 1 \mathit{mod}\, 3$.

mp_limb_t **mpn_mod_1** (mp_limb_t **s1p*, mp_size_t *s1size*, mp_limb_t *s2limb*) — Function

Divide {*s1p*, *s1size*} by *s2limb*, and return the remainder. *s1size* can be zero.

mp_limb_t **mpn_preinv_mod_1** (mp_limb_t **s1p*, mp_size_t *s1size*, mp_limb_t *s2limb*, mp_limb_t *s3limb*) — Function

This interface is obsolete. It will disappear from future releases. Use `mpn_mod_1` in its stead.

mp_limb_t **mpn_bdivmod** (mp_limb_t *rp, mp_limb_t *s1p, mp_size_t s1size, const mp_limb_t *s2p, mp_size_t s2size, unsigned long int d) Function

The function puts the low [*d*/*BITS_PER_MP_LIMB*] limbs of *q* = {*s1p*, *s1size*}/{*s2p*, *s2size*} mod 2^*d* at *rp*, and returns the high *d* mod *BITS_PER_MP_LIMB* bits of *q*.

{*s1p*, *s1size*} - *q* * {*s2p*, *s2size*} mod 2^(*s1size***BITS_PER_MP_LIMB*) is placed at *s1p*. Since the low [*d*/*BITS_PER_MP_LIMB*] limbs of this difference are zero, it is possible to overwrite the low limbs at *s1p* with this difference, provided *rp* <= *s1p*.

This function requires that *s1size* * *BITS_PER_MP_LIMB* >= *D*, and that {*s2p*, *s2size*} is odd.

This interface is preliminary. It might change incompatibly in future revisions.

mp_limb_t **mpn_lshift** (mp_limb_t *rp, const mp_limb_t *src_ptr, mp_size_t src_size, unsigned long int count) Function

Shift {*src_ptr*, *src_size*} *count* bits to the left, and write the *src_size* least significant limbs of the result to *rp*. *count* might be in the range 1 to $n-1$, on an n-bit machine. The bits shifted out to the left are returned.

Overlapping of the destination space and the source space is allowed in this function, provided *rp* >= *src_ptr*.

This function is written in assembly for most targets.

mp_limp_t **mpn_rshift** (mp_limb_t *rp, const mp_limb_t *src_ptr, mp_size_t src_size, unsigned long int count) Function

Shift {*src_ptr*, *src_size*} *count* bits to the right, and write the *src_size* most significant limbs of the result to *rp*. *count* might be in the range 1 to $n-1$, on an n-bit machine. The bits shifted out to the right are returned.

Overlapping of the destination space and the source space is allowed in this function, provided *rp* <= *src_ptr*.

This function is written in assembly for most targets.

int **mpn_cmp** (const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t size) Function

Compare {*s1p*, *size*} and {*s2p*, *size*} and return a positive value if s1 > src2, 0 of they are equal, and a negative value if s1 < src2.

mp_size_t **mpn_gcd** (mp_limb_t *rp, mp_limb_t *s1p, mp_size_t s1size, mp_limb_t *s2p, mp_size_t s2size) Function

Puts at *rp* the greatest common divisor of {*s1p*, *s1size*} and {*s2p*, *s2size*}; both source operands are destroyed by the operation. The size in limbs of the greatest common divisor is returned.

{*s1p*, *s1size*} must have at least as many bits as {*s2p*, *s2size*}, and {*s2p*, *s2size*} must be odd.

`mp_limb_t` **mpn_gcd_1** (`const mp_limb_t *`*s1p*, `mp_size_t` *s1size*, `mp_limb_t` *s2limb*) Function

Return the greatest common divisor of {*s1p*, *s1size*} and *s2limb*, where *s2limb* (as well as *s1size*) must be different from 0.

`mp_size_t` **mpn_gcdext** (`mp_limb_t *`*r1p*, `mp_limb_t *`*r2p*, `mp_size_t *`*r2size*, `mp_limb_t *`*s1p*, `mp_size_t` *s1size*, `mp_limb_t *`*s2p*, `mp_size_t` *s2size*) Function

Compute the greatest common divisor of {*s1p*, *s1size*} and {*s2p*, *s2size*}. Store the gcd at *r1p* and return its size in limbs. Write the first cofactor at *r2p* and store its size in **r2size*. If the cofactor is negative, **r2size* is negative and *r2p* is the absolute value of the cofactor.

{*s1p*, *s1size*} must be greater than or equal to {*s2p*, *s2size*}. Neither operand may equal 0. Both source operands are destroyed, plus one limb past the end of each, ie. {*s1p*, *s1size*+1} and {*s2p*, *s2size*+1}.

`mp_size_t` **mpn_sqrtrem** (`mp_limb_t *`*r1p*, `mp_limb_t *`*r2p*, `const mp_limb_t *`*sp*, `mp_size_t` *size*) Function

Compute the square root of {*sp*, *size*} and put the result at *r1p*. Write the remainder at *r2p*, unless *r2p* is `NULL`.

Return the size of the remainder, whether *r2p* was `NULL` or non-`NULL`. Iff the operand was a perfect square, the return value will be 0.

The areas at *r1p* and *sp* have to be distinct. The areas at *r2p* and *sp* have to be identical or completely separate, not partially overlapping.

The area at *r1p* needs to have space for $\lceil size/2 \rceil$ limbs. The area at *r2p* needs to be *size* limbs large.

`mp_size_t` **mpn_get_str** (`unsigned char *`*str*, `int` *base*, `mp_limb_t *`*s1p*, `mp_size_t` *s1size*) Function

Convert {*s1p*, *s1size*} to a raw unsigned char array in base *base*. The string is not in ASCII; to convert it to printable format, add the ASCII codes for '`0`' or '`A`', depending on the base and range. There may be leading zeros in the string.

The area at *s1p* is clobbered.

Return the number of characters in *str*.

The area at *str* has to have space for the largest possible number represented by a *s1size* long limb array, plus one extra character.

`mp_size_t` **mpn_set_str** (`mp_limb_t *`*r1p*, `const char *`*str*, `size_t` *strsize*, `int` *base*) Function

Convert the raw unsigned char array at *str* of length *strsize* to a limb array {*s1p*, *s1size*}. The base of *str* is *base*.

Return the number of limbs stored in *r1p*.

unsigned long int **mpn_scan0** (const mp_limb_t **s1p*, unsigned long int *bit*) — Function

Scan *s1p* from bit position *bit* for the next clear bit.

It is required that there be a clear bit within the area at *s1p* at or beyond bit position *bit*, so that the function has something to return.

unsigned long int **mpn_scan1** (const mp_limb_t **s1p*, unsigned long int *bit*) — Function

Scan *s1p* from bit position *bit* for the next set bit.

It is required that there be a set bit within the area at *s1p* at or beyond bit position *bit*, so that the function has something to return.

void **mpn_random** (mp_limb_t **r1p*, mp_size_t *r1size*) — Function
void **mpn_random2** (mp_limb_t **r1p*, mp_size_t *r1size*) — Function

Generate a random number of length *r1size* and store it at *r1p*. The most significant limb is always non-zero. `mpn_random` generates uniformly distributed limb data, `mpn_random2` generates long strings of zeros and ones in the binary representation.

`mpn_random2` is intended for testing the correctness of the `mpn` routines.

unsigned long int **mpn_popcount** (const mp_limb_t **s1p*, unsigned long int *size*) — Function

Count the number of set bits in {*s1p*, *size*}.

unsigned long int **mpn_hamdist** (const mp_limb_t **s1p*, const mp_limb_t **s2p*, unsigned long int *size*) — Function

Compute the hamming distance between {*s1p*, *size*} and {*s2p*, *size*}.

int **mpn_perfect_square_p** (const mp_limb_t **s1p*, mp_size_t *size*) — Function

Return non-zero iff {*s1p*, *size*} is a perfect square.

9 Random Number Functions

There are two groups of random number functions in GNU MP; older functions that call C library random number generators, rely on a global state, and aren't very random; and newer functions that don't have these problems. The newer functions are self-contained, they accept a random state parameter that supplants global state, and generate good random numbers.

The random state parameter is of the type `gmp_randstate_t`. It must be initialized by a call to one of the `gmp_randinit` functions (Section 9.1 [Random State Initialization], page 46). The initial seed is set using one of the `gmp_randseed` functions (Section 9.1 [Random State Initialization], page 46).

The size of the seed determines the number of different sequences of random numbers that is possible to generate. The "quality" of the seed is the randomness of a given seed compared to the previous seed used and affects the randomness of separate number sequences.

The algorithm for assigning seed is critical if the generated random numbers are to be used for important applications, such as generating cryptographic keys.

The traditional method is to use the current system time for seeding. One has to be careful when using the current time though. If the application seeds the random functions very often, say several times per second, and the resolution of the system clock is comparatively low, like one second, the same sequence of numbers will be generated until the system clock ticks. Furthermore, the current system time is quite easy to guess, so a system depending on any unpredictability of the random number sequence should absolutely not use that as its only source for a seed value.

On some systems there is a special device, often called `/dev/random`, which provides a source of somewhat random numbers more usable as seed.

The functions actually generating random functions are documented under "Miscellaneous Functions" in their respective function class: Section 5.14 [Miscellaneous Integer Functions], page 26, Section 7.8 [Miscellaneous Float Functions], page 37.

9.1 Random State Initialization

See Chapter 9 [Random Number Functions], page 46 for a discussion on how to choose the initial seed value passed to these functions.

`void` **gmp_randinit** (`gmp_randstate_t` *state*, `gmp_randalg_t` *alg*, ...) Function

Initialize random state variable *state*.

alg denotes what algorithm to use for random number generation. Use one of

- GMP_RAND_ALG_LC — Linear congruential.

  A fast generator defined by $X = (aX+c) mod m$.

  A third argument *size* of type unsigned long int is required. *size* is the size of the largest good quality random number to be generated, expressed in number of bits. If the random generation functions are asked for a bigger random number than indicated by this parameter, two or more numbers of *size* bits will be generated and concatenated, resulting in a “bad” random number. This can be used to generate big random numbers relatively cheap if the quality of randomness isn't of great importance.

a, c, and m are picked from a table where the modulus (m) is a power of 2 and the multiplier is congruent to 5 (mod 8). The choice is based on the *size* parameter. The maximum *size* supported by this algorithm is 128. If you need bigger random numbers, use your own scheme and call one of the other `gmp_randinit` functions.

If *alg* is 0 or GMP_RAND_ALG_DEFAULT, the default algorithm is used. The default algorithm is typically a fast algorithm like the linear congruential and requires a third *size* argument (see GMP_RAND_ALG_LC).

When you're done with a *state* variable, call `gmp_randclear` to deallocate any memory allocated by this function.

`gmp_randinit` may set the following bits in *gmp_errno*:

GMP_ERROR_UNSUPPORTED_ARGUMENT — *alg* is unsupported

GMP_ERROR_INVALID_ARGUMENT — *size* is too big

`void` **gmp_randinit_lc_2exp** (`gmp_randstate_t` *state*, `mpz_t` *a*, unsigned long int *c*, unsigned long int *m2exp*) Function

Initialize random state variable *state* with given linear congruential scheme.

Parameters *a*, *c*, and *m2exp* are the multiplier, adder, and modulus for the linear congruential scheme to use, respectively. The modulus is expressed as a power of 2, so that $m = 2^{m2exp}$.

The least significant bits of a random number generated by the linear congruential algorithm where the modulus is a power of two are not very random. Therefore, the lower half of a random number generated by an LC scheme initialized with this function is discarded. Thus, the size of a random number is *m2exp* / 2 (rounded upwards) bits when this function has been used for initializing the random state.

When you're done with a *state* variable, call `gmp_randclear` to deallocate any memory allocated by this function.

`void` **gmp_randseed** (`gmp_randstate_t` *state*, `mpz_t` *seed*) Function
`void` **gmp_randseed_ui** (`gmp_randstate_t` *state*, `unsigned long int` *seed*) Function

Set the initial seed value.

Parameter *seed* is the initial random seed. The function `gmp_randseed_ui` takes the *seed* as an unsigned long int rather than as an mpz_t.

`void` **gmp_randclear** (`gmp_randstate_t` *state*) Function

Free all memory occupied by *state*. Make sure to call this function for all `gmp_randstate_t` variables when you are done with them.

10 Berkeley MP Compatible Functions

These functions are intended to be fully compatible with the Berkeley MP library which is available on many BSD derived U*ix systems. The '`--enable-mpbsd`' option must be used when building GNU MP to make these available (see Chapter 2 [Installing GMP], page 3).

The original Berkeley MP library has a usage restriction: you cannot use the same variable as both source and destination in a single function call. The compatible functions in GNU MP do not share this restriction—inputs and outputs may overlap.

It is not recommended that new programs are written using these functions. Apart from the incomplete set of functions, the interface for initializing `MINT` objects is more error prone, and the `pow` function collides with `pow` in '`libm.a`'.

Include the header '`mp.h`' to get the definition of the necessary types and functions. If you are on a BSD derived system, make sure to include GNU '`mp.h`' if you are going to link the GNU '`libmp.a`' to your program. This means that you probably need to give the -I<dir> option to the compiler, where <dir> is the directory where you have GNU '`mp.h`'.

`MINT *` **itom** (`signed short int` *initial_value*) — Function

Allocate an integer consisting of a `MINT` object and dynamic limb space. Initialize the integer to *initial_value*. Return a pointer to the `MINT` object.

`MINT *` **xtom** (`char *`*initial_value*) — Function

Allocate an integer consisting of a `MINT` object and dynamic limb space. Initialize the integer from *initial_value*, a hexadecimal, '\0'-terminate C string. Return a pointer to the `MINT` object.

`void` **move** (`MINT *`*src*, `MINT *`*dest*) — Function

Set *dest* to *src* by copying. Both variables must be previously initialized.

`void` **madd** (`MINT *`*src_1*, `MINT *`*src_2*, `MINT *`*destination*) — Function

Add *src_1* and *src_2* and put the sum in *destination*.

`void` **msub** (`MINT *`*src_1*, `MINT *`*src_2*, `MINT *`*destination*) — Function

Subtract *src_2* from *src_1* and put the difference in *destination*.

`void` **mult** (`MINT *`*src_1*, `MINT *`*src_2*, `MINT *`*destination*) — Function

Multiply *src_1* and *src_2* and put the product in *destination*.

`void` **mdiv** (`MINT *`*dividend*, `MINT *`*divisor*, `MINT *`*quotient*, `MINT *`*remainder*) — Function

`void` **sdiv** (`MINT *`*dividend*, `signed short int` *divisor*, `MINT *`*quotient*, `signed short int *`*remainder*) — Function

Set *quotient* to *dividend*/*divisor*, and *remainder* to *dividend* mod *divisor*. The quotient is rounded towards zero; the remainder has the same sign as the dividend unless it is zero.

Some implementations of these functions work differently—or not at all—for negative arguments.

void **msqrt** (MINT **operand*, MINT **root*, MINT **remainder*) Function

Set *root* to $\lfloor\sqrt{operand}\rfloor$, like `mpz_sqrt`. Set *remainder* to $(operand - root^2)$, (i.e., zero if *operand* is a perfect square).

If *root* and *remainder* are the same variable, the results are undefined.

void **pow** (MINT **base*, MINT **exp*, MINT **mod*, MINT **dest*) Function

Set *dest* to (*base* raised to *exp*) modulo *mod*.

void **rpow** (MINT **base*, signed short int *exp*, MINT **dest*) Function

Set *dest* to *base* raised to *exp*.

void **gcd** (MINT **operand1*, MINT **operand2*, MINT **res*) Function

Set *res* to the greatest common divisor of *operand1* and *operand2*.

int **mcmp** (MINT **operand1*, MINT **operand2*) Function

Compare *operand1* and *operand2*. Return a positive value if *operand1* > *operand2*, zero if *operand1* = *operand2*, and a negative value if *operand1* < *operand2*.

void **min** (MINT **dest*) Function

Input a decimal string from `stdin`, and put the read integer in *dest*. SPC and TAB are allowed in the number string, and are ignored.

void **mout** (MINT **src*) Function

Output *src* to `stdout`, as a decimal string. Also output a newline.

char * **mtox** (MINT **operand*) Function

Convert *operand* to a hexadecimal string, and return a pointer to the string. The returned string is allocated using the default memory allocation function, `malloc` by default.

void **mfree** (MINT **operand*) Function

De-allocate, the space used by *operand*. **This function should only be passed a value returned by `itom` or `xtom`.**

11 Custom Allocation

By default, GMP uses `malloc`, `realloc` and `free` for memory allocation. If `malloc` or `realloc` fails, GMP prints a message to the standard error output and terminates execution.

Some applications might want to allocate memory in other ways, or might not want a fatal error when there is no more memory available. To accomplish this, you can specify alternative memory allocation functions.

This can be done in the Berkeley compatibility library as well as the main GMP library.

`void` **mp_set_memory_functions** (— Function
`void *(*`*alloc_func_ptr*`) (size_t),`
`void *(*`*realloc_func_ptr*`) (void *, size_t, size_t),`
`void (*`*free_func_ptr*`) (void *, size_t))`

Replace the current allocation functions from the arguments. If an argument is `NULL`, the corresponding default function is retained.

Be sure to call this function only when there are no active GMP objects allocated using the previous memory functions! Usually, that means that you have to call this function before any other GMP function.

The functions you supply should fit the following declarations:

`void *` **allocate_function** (`size_t` *alloc_size*) — Function

This function should return a pointer to newly allocated space with at least *alloc_size* storage units.

`void *` **reallocate_function** (`void *`*ptr*, `size_t` *old_size*, `size_t` *new_size*) — Function

This function should return a pointer to newly allocated space of at least *new_size* storage units, after copying at least the first *old_size* storage units from *ptr*. It should also deallocate the space at *ptr*.

You can assume that the space at *ptr* was formerly returned from `allocate_function` or `reallocate_function`, for a request for *old_size* storage units.

`void` **deallocate_function** (`void *`*ptr*, `size_t` *size*) — Function

De-allocate the space pointed to by *ptr*.

You can assume that the space at *ptr* was formerly returned from `allocate_function` or `reallocate_function`, for a request for *size* storage units.

(A *storage unit* is the unit in which the `sizeof` operator returns the size of an object, normally an 8 bit byte.)

Contributors

Torbjorn Granlund wrote the original GMP library and is still developing and maintaining it. Several other individuals and organizations have contributed to GMP in various ways. Here is a list in chronological order:

Gunnar Sjoedin and Hans Riesel helped with mathematical problems in early versions of the library.

Richard Stallman contributed to the interface design and revised the first version of this manual.

Brian Beuning and Doug Lea helped with testing of early versions of the library and made creative suggestions.

John Amanatides of York University in Canada contributed the function `mpz_probab_prime_p`.

Paul Zimmermann of Inria sparked the development of GMP 2, with his comparisons between bignum packages.

Ken Weber (Kent State University, Universidade Federal do Rio Grande do Sul) contributed `mpz_gcd`, `mpz_divexact`, `mpn_gcd`, and `mpn_bdivmod`, partially supported by CNPq (Brazil) grant 301314194-2.

Per Bothner of Cygnus Support helped to set up GMP to use Cygnus' configure. He has also made valuable suggestions and tested numerous intermediary releases.

Joachim Hollman was involved in the design of the `mpf` interface, and in the `mpz` design revisions for version 2.

Bennet Yee contributed the functions `mpz_jacobi` and `mpz_legendre`.

Andreas Schwab contributed the files '`mpn/m68k/lshift.S`' and '`mpn/m68k/rshift.S`'.

The development of floating point functions of GNU MP 2, were supported in part by the ESPRIT-BRA (Basic Research Activities) 6846 project POSSO (POlynomial System SOlving).

GNU MP 2 was finished and released by SWOX AB (formerly known as TMG Datakonsult), Swedenborgsgatan 23, SE-118 27 STOCKHOLM, SWEDEN, in cooperation with the IDA Center for Computing Sciences, USA.

Robert Harley of Inria, France and David Seal of ARM, England, suggested clever improvements for population count.

Robert Harley also wrote highly optimized Karatsuba and 3-way Toom multiplication functions for GMP 3. He also contributed the ARM assembly code.

Torsten Ekedahl of the Mathematical department of Stockholm University provided significant inspiration during several phases of the GMP development. His mathematical expertise helped improve several algorithms.

Paul Zimmermann wrote the Burnikel-Ziegler division code, the REDC code, the REDC-based mpz_powm code, and the FFT multiply code. The ECMNET project Paul is organizing has been a driving force behind many of the optimization of GMP 3.

Linus Nordberg wrote the new configure system based on autoconf and implemented the new random functions.

Kent Boortz made the Macintosh port.

Kevin Ryde wrote a lot of very high quality x86 code, optimized for most CPU variants. He also made countless other valuable contributions.

Steve Root helped write the optimized alpha 21264 assembly code.

GNU MP 3.1 was finished and released by Torbjorn Granlund and Kevin Ryde. Torbjorn's work was partially funded by the IDA Center for Computing Sciences, USA.

(This list is chronological, not ordered after significance. If you have contributed to GMP but are not listed above, please tell `tege@swox.com` about the omission!)

References

- Donald E. Knuth, "The Art of Computer Programming", vol 2, "Seminumerical Algorithms", 3rd edition, Addison-Wesley, 1988.
- John D. Lipson, "Elements of Algebra and Algebraic Computing", The Benjamin Cummings Publishing Company Inc, 1981.
- Richard M. Stallman, "Using and Porting GCC", Free Software Foundation, 1999, available online `http://www.gnu.org/software/gcc/onlinedocs/`, and in the GCC package `ftp://ftp.gnu.org/pub/gnu/gcc/`.
- Peter L. Montgomery, "Modular Multiplication Without Trial Division", in Mathematics of Computation, volume 44, number 170, April 1985.
- Torbjorn Granlund and Peter L. Montgomery, "Division by Invariant Integers using Multiplication", in Proceedings of the SIGPLAN PLDI'94 Conference, June 1994. Available online,
  `ftp://ftp.cwi.nl/pub/pmontgom/divcnst.psa4.gz` (and .psl.gz too).
- Tudor Jebelean, "An algorithm for exact division", Journal of Symbolic Computation, v. 15, 1993, pp. 169-180. Research report version available online
  `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1992/92-35.ps.gz`
- Kenneth Weber, "The accelerated integer GCD algorithm", ACM Transactions on Mathematical Software, v. 21 (March), 1995, pp. 111-122.
- Christoph Burnikel and Joachim Ziegler, "Fast Recursive Division", Max-Planck-Institut fuer Informatik Research Report MPI-I-98-1-022,
  `http://www.mpi-sb.mpg.de/~ziegler/TechRep.ps.gz`.
- Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone, "Handbook of Applied Cryptography", `http://cacr.math.uwaterloo.ca/hac/`.
- Henri Cohen, "A Course in Computational Algebraic Number Theory", Graduate Texts in Mathematics number 138, Springer-Verlag, 1993. Errata available online
  `http://www.math.u-bordeaux.fr/~cohen`

Concept Index

J

K

L

M

N

O

P

R

S

T

U

V

W

Function and Type Index

P

R

S

X

Table of Contents