MPIR

The Multiple Precision Integers and Rationals Library
Edition 3.0.0
1 March 2017

Original Authors: Torbjorn Granlund and the GMP Development Team
Subsequent modifications: William Hart and the MPIR Team

This manual describes how to install and use MPIR, the Multiple Precision Integers and Rationals library, version 3.0.0.

Table of Contents

MPIR 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 MPIR 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 MPIR 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 MPIR library are found in the Lesser General Public License version 3 that accompanies the source code, see `COPYING.LIB`.

1 Introduction to MPIR

MPIR 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. MPIR 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 MPIR 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 assembly code for these CPUs: ARM, DEC Alpha 21064, 21164, and 21264, AMD K6, K6-2, Athlon, K8 and K10, Intel Pentium, Pentium Pro/II/III, Pentium 4, generic x86, Intel IA-64, Core 2, i7, Atom, Motorola/IBM PowerPC 32 and 64, MIPS R3000, R4000, SPARCv7, SuperSPARC, generic SPARCv8, UltraSPARC,

For up-to-date information on, and latest version of, MPIR, please see the MPIR web pages at

`http://www.mpir.org/`

There are a number of public mailing lists of interest. The development list is

`http://groups.google.com/group/mpir-devel/`.

The proper place for bug reports is `http://groups.google.com/group/mpir-devel`. See Chapter 4 [Reporting Bugs], page 28, for information about reporting bugs.

1.1 How to use this Manual

Everyone should read Chapter 3 [MPIR Basics], page 16. If you need to install the library yourself, then read Chapter 2 [Installing MPIR], page 3. If you have a system with multiple ABIs, then read Section 2.2 [ABI and ISA], page 8, for the compiler options that must be used on applications. In addition to usual compilation tools, MPIR depends on Yasm to be built. If yasm is not available on your system, you can download its sources at

`http://yasm.tortall.net/Download.html`

build your own version and make it available to MPIR configuration system by passing its path to `configure` through the '`--with-yasm`' option. See Section 2.1 [Build Options], page 3, for further details.

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

2 Installing MPIR

MPIR 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
```

Important note: by default MPIR produces libraries named libmpir, etc., and the header file `mpir.h`. If you wish to have MPIR to build a library named libgmp as well, etc., and a `gmp.h` header file, so that you can use mpir with programs designed to only work with GMP, then use the '`--enable-gmpcompat`' option when invoking configure:

```
./configure --enable-gmpcompat
```

Note `gmp.h` is only created upon running make install.

MPIR is compatible with GMP when the '`--enable-gmpcompat`' option is used, except that the GMP secure cryptographic functions are not available.

Some deprecated GMP functionality may be unavailable if this option is not selected.

If you experience problems, please report them to

`http://groups.google.com/group/mpir-devel`.

See Chapter 4 [Reporting Bugs], page 28, 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. The file `INSTALL.autoconf` has some generic installation information too.

Tools
: '`configure`' requires various Unix-like tools. See Section 2.5 [Notes for Particular Systems], page 13, for some options on non-Unix systems.

  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.

Build Directory
: To compile in a separate build directory, `cd` to that directory, and prefix the configure command with the path to the MPIR source directory. For example

  ```
  cd /my/build/dir
  /my/sources/mpir-3.0.0/configure
  ```

  Not all '`make`' programs have the necessary features (`VPATH`) to support this. In particular, SunOS and Solaris `make` have bugs that make them unable to build in a separate directory. Use GNU `make` instead.

`--prefix` and `--exec-prefix`
: The `--prefix` option can be used in the normal way to direct MPIR to install under a particular tree. The default is '`/usr/local`'.

`--exec-prefix` can be used to direct architecture-dependent files like `libmpir.a` to a different location. This can be used to share architecture-independent parts like the documentation, but separate the dependent parts. Note however that `mpir.h` and `mp.h` are architecture-dependent since they encode certain aspects of `libmpir`, so it will be necessary to ensure both `$prefix/include` and `$exec_prefix/include` are available to the compiler.

`--enable-gmpcompat`
By default make builds libmpir library files (and libmpirxx if C++ headers are requested) and the `mpir.h` header file. This option allows you to specify that you want additional libraries created called libgmp (and libgmpxx), etc., for libraries and gmp.h for compatibility with GMP (except for GMP secure cryptograhic functions, which are not available in MPIR).

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

Native Compilation, `--build=CPU-VENDOR-OS`
For normal native compilation, the system can be specified with '`--build`'. By default '`./configure`' uses the output from running '`./config.guess`'. On some systems '`./config.guess`' can determine the exact CPU type, on others it will be necessary to give it explicitly. For example,

```
./configure --build=ultrasparc-sun-solaris2.7
```

In all cases the '`OS`' part is important, since it controls how libtool generates shared libraries. Running '`./config.guess`' is the simplest way to see what it should be, if you don't know already.

Cross Compilation, `--host=CPU-VENDOR-OS`
When cross-compiling, the system used for compiling is given by '`--build`' and the system where the library will run is given by '`--host`'. For example when using a FreeBSD Athlon system to build GNU/Linux m68k binaries,

```
./configure --build=athlon-pc-freebsd3.5 --host=m68k-mac-linux-gnu
```

Compiler tools are sought first with the host system type as a prefix. For example `m68k-mac-linux-gnu-ranlib` is tried, then plain `ranlib`. This makes it possible for a set of cross-compiling tools to co-exist with native tools. The prefix is the argument to '`--host`', and this can be an alias, such as '`m68k-linux`'. But note that tools don't have to be setup this way, it's enough to just have a `PATH` with a suitable cross-compiling `cc` etc.

Compiling for a different CPU in the same family as the build system is a form of cross-compilation, though very possibly this would merely be special options on a native compiler. In any case '`./configure`' avoids depending on being able to run code on the build system, which is important when creating binaries for a newer CPU since they very possibly won't run on the build system.

In all cases the compiler must be able to produce an executable (of whatever format) from a standard C `main`. Although only object files will go to make up `libmpir`, '`./configure`' uses linking tests for various purposes, such as determining what functions are available on the host system.

Currently a warning is given unless an explicit '`--build`' is used when cross-compiling, because it may not be possible to correctly guess the build system type if the `PATH` has only a cross-compiling `cc`.

Note that the '`--target`' option is not appropriate for MPIR. It's for use when building compiler tools, with '`--host`' being where they will run, and '`--target`' what they'll produce code for. Ordinary programs or libraries like MPIR are only interested in the '`--host`' part, being where they'll run.

CPU types

In general, if you want a library that runs as fast as possible, you should configure MPIR 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 MPIR for the exact machine type you intend to run it on.

The following CPUs have specific support. See `configure.in` for details of what code and compiler options they select.

- Alpha: '`alpha`', '`alphaev5`', '`alphaev56`', '`alphapca56`', '`alphapca57`', '`alphaev6`', '`alphaev67`', '`alphaev68`' '`alphaev7`'
- IA-64: '`ia64`', '`itanium`', '`itanium2`'
- MIPS: '`mips`', '`mips3`', '`mips64`'
- PowerPC: '`powerpc`', '`powerpc64`', '`powerpc401`', '`powerpc403`', '`powerpc405`', '`powerpc505`', '`powerpc601`', '`powerpc602`', '`powerpc603`', '`powerpc603e`', '`powerpc604`', '`powerpc604e`', '`powerpc620`', '`powerpc630`', '`powerpc740`', '`powerpc7400`', '`powerpc7450`', '`powerpc750`', '`powerpc801`', '`powerpc821`', '`powerpc823`', '`powerpc860`', '`powerpc970`'
- SPARC: '`sparc`', '`sparcv8`', '`microsparc`', '`supersparc`', '`sparcv9`', '`ultrasparc`', '`ultrasparc2`', '`ultrasparc2i`', '`ultrasparc3`', '`sparc64`'
- x86 family: '`pentium`', '`pentiummmx`', '`pentiumpro`', '`pentium2`', '`pentium3`', '`pentium4`', '`netburst`', '`netburstlahf`', '`prescott`', '`core`', '`core2`', '`penryn`', '`nehalem`', '`westmere`' '`sandybridge`' '`haswell`' '`nano`' '`atom`', '`k5`', '`k6`', '`k62`', '`k63`', '`k7`', '`k8`', '`k10`' '`k102`' '`piledriver`' '`bulldozer`' '`bobcat`' '`viac3`', '`viac32`'
- Other: '`arm`',

CPUs not listed will use generic C code.

Generic C Build

If some of the assembly code causes problems, or if otherwise desired, the generic C code can be selected with CPU '`none`'. For example,

```
./configure --host=none-unknown-freebsd3.5
```

Note that this will run quite slowly, but it should be portable and should at least make it possible to get something running if all else fails.

Fat binary, `--enable-fat`

Using `--enable-fat` selects a "fat binary" build on x86 or x86_64 systems, where optimized low level subroutines are chosen at runtime according to the CPU detected. This means more code, but gives reasonable performance from a single binary for all x86 chips, or similarly for all x86_64 chips. (This option might become available for more architectures in the future.)

`ABI` On some systems MPIR supports multiple ABIs (application binary interfaces), meaning data type sizes and calling conventions. By default MPIR chooses the best ABI available, but a particular ABI can be selected. For example

```
./configure --host=mips64-sgi-irix6 ABI=n32
```

See Section 2.2 [ABI and ISA], page 8, for the available choices on relevant CPUs, and what applications need to do.

`--with-yasm`

By default MPIR will look for a system-wide Yasm using the `which` command. Passing `--with-yasm` let MPIR use a version of Yasm at a non-standard location. This is useful if none is available in `PATH`. With this option a full path to Yasm's binary should be given, for example

```
./configure --with-yasm=/usr/local/bin/yasm
```

`CC`, `CFLAGS`

By default 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 various systems, default compiler flags are set based on the CPU and compiler. The usual '`CFLAGS="-whatever"`' can be passed to '`./configure`' to use something different or to set good flags for systems MPIR doesn't otherwise know.

The '`CC`' and '`CFLAGS`' used are printed during '`./configure`', and can be found in each generated `Makefile`. This is the easiest way to check the defaults when considering changing or adding something.

Note that when '`CC`' and '`CFLAGS`' are specified on a system supporting multiple ABIs it's important to give an explicit '`ABI=whatever`', since MPIR can't determine the ABI just from the flags and won't be able to select the correct assembler code.

If just '`CC`' is selected then normal default '`CFLAGS`' for that compiler will be used (if MPIR recognises it). For example '`CC=gcc`' can be used to force the use of GCC, with default flags (and default ABI).

`CPPFLAGS` Any flags like '`-D`' defines or '`-I`' includes required by the preprocessor should be set in '`CPPFLAGS`' rather than '`CFLAGS`'. Compiling is done with both '`CPPFLAGS`' and '`CFLAGS`', but preprocessing uses just '`CPPFLAGS`'. This distinction is because most preprocessors won't accept all the flags the compiler does. Preprocessing is done separately in some configure tests, and in the '`ansi2knr`' support for K&R compilers.

`CC_FOR_BUILD`

Some build-time programs are compiled and run to generate host-specific data tables. '`CC_FOR_BUILD`' is the compiler used for this. It doesn't need to be in any particular ABI or mode, it merely needs to generate executables that can run. The default is to try the selected '`CC`' and some likely candidates such as '`cc`' and '`gcc`', looking for something that works.

No flags are used with '`CC_FOR_BUILD`' because a simple invocation like '`cc foo.c`' should be enough. If some particular options are required they can be included as for instance '`CC_FOR_BUILD="cc -whatever"`'.

C++ Support, `--enable-cxx`

C++ support in MPIR can be enabled with '`--enable-cxx`', in which case a C++ compiler will be required. As a convenience '`--enable-cxx=detect`' can be used to enable C++ support only if a compiler can be found. The C++ support consists of a library `libmpirxx.la` and header file `mpirxx.h` (see Section 3.1 [Headers and Libraries], page 16).

A separate `libmpirxx.la` has been adopted rather than having C++ objects within `libmpir.la` in order to ensure dynamic linked C programs aren't bloated by a dependency on the C++ standard library, and to avoid any chance that the C++ compiler could be required when linking plain C programs.

`libmpirxx.la` will use certain internals from `libmpir.la` and can only be expected to work with `libmpir.la` from the same MPIR version. Future changes to the rele-

vant internals will be accompanied by renaming, so a mismatch will cause unresolved symbols rather than perhaps mysterious misbehaviour.

In general `libmpirxx.la` will be usable only with the C++ compiler that built it, since name mangling and runtime support are usually incompatible between different compilers.

`CXX`, `CXXFLAGS`

When C++ support is enabled, the C++ compiler and its flags can be set with variables '`CXX`' and '`CXXFLAGS`' in the usual way. The default for '`CXX`' is the first compiler that works from a list of likely candidates, with `g++` normally preferred when available. The default for '`CXXFLAGS`' is to try '`CFLAGS`', '`CFLAGS`' without '`-g`', then for `g++` either '`-g -O2`' or '`-O2`', or for other compilers '`-g`' or nothing. Trying '`CFLAGS`' this way is convenient when using '`gcc`' and '`g++`' together, since the flags for '`gcc`' will usually suit '`g++`'.

It's important that the C and C++ compilers match, meaning their startup and runtime support routines are compatible and that they generate code in the same ABI (if there's a choice of ABIs on the system). '`./configure`' isn't currently able to check these things very well itself, so for that reason '`--disable-cxx`' is the default, to avoid a build failure due to a compiler mismatch. Perhaps this will change in the future.

Incidentally, it's normally not good enough to set '`CXX`' to the same as '`CC`'. Although `gcc` for instance recognises `foo.cc` as C++ code, only `g++` will invoke the linker the right way when building an executable or shared library from C++ object files.

Temporary Memory, `--enable-alloca=<choice>`

MPIR allocates temporary workspace using one of the following three methods, which can be selected with for instance '`--enable-alloca=malloc-reentrant`'.

- '`alloca`' - C library or compiler builtin.
- '`malloc-reentrant`' - the heap, in a re-entrant fashion.
- '`malloc-notreentrant`' - the heap, with global variables.

For convenience, the following choices are also available. '`--disable-alloca`' is the same as '`no`'.

- '`yes`' - a synonym for '`alloca`'.
- '`no`' - a synonym for '`malloc-reentrant`'.
- '`reentrant`' - `alloca` if available, otherwise '`malloc-reentrant`'. This is the default.
- '`notreentrant`' - `alloca` if available, otherwise '`malloc-notreentrant`'.

`alloca` is reentrant and fast, and is recommended. It actually allocates just small blocks on the stack; larger ones use malloc-reentrant.

'`malloc-reentrant`' is, as the name suggests, reentrant and thread safe, but '`malloc-notreentrant`' is faster and should be used if reentrancy is not required.

The two malloc methods in fact use the memory allocation functions selected by `mp_set_memory_functions`, these being `malloc` and friends by default. See Chapter 14 [Custom Allocation], page 106.

An additional choice '`--enable-alloca=debug`' is available, to help when debugging memory related problems (see Section 3.12 [Debugging], page 23).

FFT Multiplication, `--disable-fft`

By default multiplications are done using Karatsuba, Toom, and FFT algorithms. The FFT is only used on large to very large operands and can be disabled to save code size if desired.

Assertion Checking, `--enable-assert`
: This option enables some consistency checking within the library. This can be of use while debugging, see Section 3.12 [Debugging], page 23.

Execution Profiling, `--enable-profiling=prof/gprof/instrument`
: Enable profiling support, in one of various styles, see Section 3.13 [Profiling], page 25.

`MPN_PATH`
: Various assembler versions of mpn subroutines are provided. For a given CPU, a search is made though a path to choose a version of each. For example '`sparcv8`' has

```
MPN_PATH="sparc32/v8 sparc32 generic"
```

: which means look first for v8 code, then plain sparc32 (which is v7), and finally fall back on generic C. Knowledgeable users with special requirements can specify a different path. Normally this is completely unnecessary.

Documentation
: The source for the document you're now reading is `doc/mpir.texi`, in Texinfo format, see *Texinfo*.

: Info format '`doc/mpir.info`' is included in the distribution. The usual automake targets are available to make PostScript, DVI, PDF and HTML (these will require various TEX and Texinfo tools).

: DocBook and XML can be generated by the Texinfo `makeinfo` program too, see Section "Options for `makeinfo`" in *Texinfo*.

: Some supplementary notes can also 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. MPIR supports some CPUs like this in both ABIs. In fact within MPIR '`ABI`' means a combination of chip ABI, plus how MPIR chooses to use it. For example in some 32-bit ABIs, MPIR may support a limb as either a 32-bit `long` or a 64-bit `long long`.

By default MPIR chooses the best ABI available for a given system, and this generally gives significantly greater speed. But an ABI can be chosen explicitly to make MPIR compatible with other libraries, or particular application requirements. For example,

```
./configure ABI=32
```

In all cases it's vital that all object code used in a given program is compiled for the same ABI.

Usually a limb is implemented as a `long`. When a `long long` limb is used this is encoded in the generated `mpir.h`. This is convenient for applications, but it does mean that `mpir.h` will vary, and can't be just copied around. `mpir.h` remains compiler independent though, since all compilers for a particular ABI will be expected to use the same limb type.

Currently no attempt is made to follow whatever conventions a system has for installing library or header files built for a particular ABI. This will probably only matter when installing multiple builds of MPIR, and it might be as simple as configuring with a special '`libdir`', or it might require more than that. Note that builds for different ABIs need to done separately, with a fresh (`make distclean`), `./configure` and `make`.

AMD64 (‘x86_64’)

On AMD64 systems supporting both 32-bit and 64-bit modes for applications, the following ABI choices are available.

‘ABI=64’ The 64-bit ABI uses 64-bit limbs and pointers and makes full use of the chip architecture. This is the default. Applications will usually not need special compiler flags, but for reference the option is

```
gcc  -m64
```

‘ABI=32’ The 32-bit ABI is the usual i386 conventions. This will be slower, and is not recommended except for inter-operating with other code not yet 64-bit capable. Applications must be compiled with

```
gcc  -m32
```

(In GCC 2.95 and earlier there’s no ‘-m32’ option, it’s the only mode.)

IA-64 under HP-UX (‘ia64*-*-hpux*’, ‘itanium*-*-hpux*’)

HP-UX supports two ABIs for IA-64. MPIR performance is the same in both.

‘ABI=32’ In the 32-bit ABI, pointers, `int`s and `long`s are 32 bits and MPIR uses a 64 bit `long long` for a limb. Applications can be compiled without any special flags since this ABI is the default in both HP C and GCC, but for reference the flags are

```
gcc  -milp32
cc   +DD32
```

‘ABI=64’ In the 64-bit ABI, `long`s and pointers are 64 bits and MPIR uses a `long` for a limb. Applications must be compiled with

```
gcc  -mlp64
cc   +DD64
```

On other IA-64 systems, GNU/Linux for instance, ‘ABI=64’ is the only choice.

PowerPC 64 (‘powerpc64’, ‘powerpc620’, ‘powerpc630’, ‘powerpc970’)

‘ABI=aix64’

The AIX 64 ABI uses 64-bit limbs and pointers and is the default on PowerPC 64 ‘*-*-aix*’ systems. Applications must be compiled with

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

‘ABI=mode32’

The ‘mode32’ ABI uses a 64-bit `long long` limb but with the chip still in 32-bit mode and using 32-bit calling conventions. This is the default on PowerPC 64 ‘*-*-darwin*’ systems. No special compiler options are needed for applications.

‘ABI=32’ This is the basic 32-bit PowerPC ABI, with a 32-bit limb. No special compiler options are needed for applications.

MPIR speed is greatest in ‘aix64’ and ‘mode32’. In ‘ABI=32’ only the 32-bit ISA is used and this doesn’t make full use of a 64-bit chip. On a suitable system we could perhaps use more of the ISA, but there are no plans to do so.

Sparc V9 (‘sparc64’, ‘sparcv9’, ‘ultrasparc*’)

‘ABI=64’ The 64-bit V9 ABI is available on the various BSD sparc64 ports, recent versions of Sparc64 GNU/Linux, and Solaris 2.7 and up (when the kernel is in 64-bit mode). GCC 3.2 or higher, or Sun cc is required. On GNU/Linux, depending on the default gcc mode, applications must be compiled with

```
gcc  -m64
```

On Solaris applications must be compiled with

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

On the BSD sparc64 systems no special options are required, since 64-bits is the only ABI available.

‘ABI=32’ For the basic 32-bit ABI, MPIR still uses as much of the V9 ISA as it can. In the Sun documentation this combination is known as “v8plus”. On GNU/Linux, depending on the default gcc mode, applications may need to be compiled with

```
gcc  -m32
```

On Solaris, no special compiler options are required for applications, though using something like the following is recommended. (gcc 2.8 and earlier only support ‘-mv8’ though.)

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

MPIR speed is greatest in ‘ABI=64’, so it’s the default where available. The speed is partly because there are extra registers available and partly because 64-bits is considered the more important case and has therefore had better code written for it.

Don’t be confused by the names of the ‘-m’ and ‘-x’ compiler options, they’re called ‘arch’ but effectively control both ABI and ISA.

On Solaris 2.6 and earlier, only ‘ABI=32’ is available since the kernel doesn’t save all registers.

On Solaris 2.7 with the kernel in 32-bit mode, a normal native build will reject ‘ABI=64’ because the resulting executables won’t run. ‘ABI=64’ can still be built if desired by making it look like a cross-compile, for example

```
./configure --build=none --host=sparcv9-sun-solaris2.7 ABI=64
```

2.3 Notes for Package Builds

MPIR 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 (see Section “Library interface versions” in *GNU Libtool*).

The GMP 4 series and MPIR 1 series will be upwardly binary compatible in each release and will be upwardly binary compatible with all of the GMP 3 series. Additional function interfaces may be added in each release, so 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 MPIR.

From MPIR 2.0.0 binary compatibility with the GMP 5 series will be maintained with the exception of the availability of secure functions for cryptography, which will not be supported in MPIR.

For full GMP compatibility, including deprecated functionality, the '`--enable-gmpcompat`' configuration option must be used.

An auxiliary mechanism may also be needed to express that `libmpirxx.la` (from `--enable-cxx`, see Section 2.1 [Build Options], page 3) requires `libmpir.la` from the same MPIR version, since this is not done by the libtool versioning, nor otherwise. A mismatch will result in unresolved symbols from the linker, or perhaps the loader.

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

For x86s, `--enable-fat` sets things up for a fat binary build, making a runtime selection of optimized low level routines. This is a good choice for packaging to run on a range of x86 chips.

Users who care about speed will want MPIR built for their exact CPU type, to make best 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 '`--build`' (and '`--host`') so '`./config.guess`' will detect the CPU. But a way to manually specify a '`--build`' will be wanted for systems where '`./config.guess`' is inexact.

On systems with multiple ABIs, a packaged build will need to decide which among the choices is to be provided, see Section 2.2 [ABI and ISA], page 8. A given run of '`./configure`' etc will only build one ABI. If a second ABI is also required then a second run of '`./configure`' etc must be made, starting from a clean directory tree ('`make distclean`').

As noted under "ABI and ISA", currently no attempt is made to follow system conventions for install locations that vary with ABI, such as `/usr/lib/sparcv9` for '`ABI=64`' as opposed to `/usr/lib` for '`ABI=32`'. A package build can override '`libdir`' and other standard variables as necessary.

Note that `mpir.h` is a generated file, and will be architecture and ABI dependent. When attempting to install two ABIs simultaneously it will be important that an application compile gets the correct `mpir.h` for its desired ABI. If compiler include paths don't vary with ABI options then it might be necessary to create a `/usr/include/mpir.h` which tests preprocessor symbols and chooses the correct actual `mpir.h`.

2.4 Building with Microsoft Visual Studio

MPIR can be built with the professional and higher versions of Visual Studio 2012, 2013, 2015 and 2017. It can also be built with the community editions of Visual Studio 2015 and 2017. If the assembler optimised versions of MPIR are required, then both Python 3 and the YASM assembler also need to be installed. MPIR can also be built with the Intel C/C++ compiler that can be integrated into versions of Visual Studio.

Python 3 can be obtained from:

```
https://www.python.org/
```

and the YASM assembler from:

```
http://yasm.tortall.net/Download.html
```

This assembler (`vsyasm.exe`, NOT `yasm.exe`) should be placed in the directory `C:\Program Files\yasm`.

Alternatively `vsyasm.exe` can be placed elsewhere provided that the environment variable '`YASMPATH`' gives its location.

Building MPIR

A build of MPIR is started by double clicking on the file `mpir.sln` in the appropriate sub-directory within the MPIR root directory:

```
Visual Studio 2012:  mpir/build.vc11/mpir.sln
Visual Studio 2013:  mpir/build.vc12/mpir.sln
Visual Studio 2015:  mpir/build.vc14/mpir.sln
Visual Studio 2017:  mpir/build.vc15/mpir.sln
```

Visual Studio will then display a list of individual build projects from which an appropriate version of MPIR can be built. For example, a typical list of projects is:

```
dll_mpir_gc     standard DLL, no assembler (win32 and x64)
dll_mpir_p3     assembly optimised DLL for pentium 3 (win32)
lib_mpir_p3     assembly optimised static library for
                pentium 3 (x64)
lib_mpir_core2  assembly optimised static library for
                core2 (x64)
dll_mpir_core2  assembly optimised DLL for core2 (x64)
```

MPIR can be built either as a static library or as a DLL. A DLL will include both the C and C++ features of MPIR but a static library will include only the C features so in this case the project:

```
lib_mpir_cxx     the MPIRXX C++ static library (win32 and x64)
```

should also be built to provide the MPIR C++ static library ('`MPIRXX`').

Before a project is built, Visual Studio should be set to the required configuration (Release or Debug) and the required target architecture (win32 or x64). The build process puts the output files into one of the two sub-directories:

```
mpir/lib
mpir/dll
```

depending on whether static library or DLL versions have been built.

Additional Assembler Optimised Versions

The Visual Studio builds for MPIR are initially provided with a small set of assembler optimised projects but many more are available and can be obtained by running the Python program `mpir_config.py <N>` that is in the mpir\build-vc directory. The value of `<N>` required depends on the version of Visual Studio in use as follows:

```
Visual Studio 2012:  11
Visual Studio 2013:  12
Visual Studio 2015:  14
Visual Studio 2017:  15
```

This program, which has to be run before Visual Studio, provides a list of all the assembler optimised versions of MPIR that are available. Any number of versions can be chosen and these builds will then be available when Visual Studio is subsequently opened by double clicking on `mpir.sln`.

Testing Visual Studio versions of MPIR

Testing a version of the library once it has been built is started by double clicking on the appropriate solution file:

```
Visual Studio 2012:  mpir/build.vc11/mpir-tests.sln
Visual Studio 2013:  mpir/build.vc12/mpir-tests.sln
Visual Studio 2015:  mpir/build.vc14/mpir-tests.sln
Visual Studio 2017:  mpir/build.vc15/mpir-tests.sln
```

The tests are always run on the last version of MPIR built and it is important that the configuration set for building the tests (Release or Debug, win32 or x64) is the

same as that used to build MPIR. When testing the static library versions of MPIR, both the C (mpir.lib) and C++ (mpirxx.lib) must be built. After loading there will be a large list of test projects starting:

```
Solution 'mpir-tests' (202 projects)
      add-test-lib
      bswap
      constants
      ....
```

The project '`add-test-lib`' should be selected and built first, after which the solution as a whole (i.e. the first line shown above) can be selected to build all the tests. After the build has completed, the tests are run by executing the Python program `run-tests.py` in the appropriate Visual Studio build sub-directory, for example, for Visual Studio 2017:

```
mpir/build.vc15/mpir-tests/run-tests.py
```

2.5 Notes for Particular Systems

ARM
: On systems '`arm*-*-*`', versions of GCC up to and including 2.95.3 have a bug in unsigned division, giving wrong results for some operands. MPIR '`./configure`' will demand GCC 2.95.4 or later.

Floating Point Mode
: On some systems, the hardware floating point has a control mode which can set all operations to be done in a particular precision, for instance single, double or extended on x86 systems (x87 floating point). The MPIR functions involving a `double` cannot be expected to operate to their full precision when the hardware is in single precision mode. Of course this affects all code, including application code, not just MPIR.

MS-DOS and MS Windows
: On an MS Windows system Cygwin and Cygwin64 and Msys2/Mingw can be used, they are ports of GCC and the various GNU tools.

```
http://www.cygwin.com/
https://msys2.github.io/
```

Both 32 and 64 bit versions of Msys2/Mingw and Cygwin are supported. Building on these systems is very similar to building on Linux.

We strongly recommend using recent versions of Cygwin/Msys2.

MS Windows DLLs
: On systems '`*-*-cygwin*`' and '`*-*-mingw*`' and '`*-*-msys`' static and DLL libraries can't both be built, since certain export directives in `mpir.h` must be different. Therefore you must specify whether you want a shared library or a static library. For example if you want just a shared library you can type the following.

```
./configure --disable-static --enable-shared
```

Libtool doesn't install a `.lib` format import library, but it can be created with MS `lib` as follows, and copied to the install directory. Similarly for `libmpir` and `libmpirxx`.

```
cd .libs
lib /def:libgmp-3.dll.def /out:libgmp-3.lib
```

Sparc CPU Types
: '`sparcv8`' or '`supersparc`' on relevant systems will give a significant performance increase over the V7 code selected by plain '`sparc`'.

Sparc App Regs
: The MPIR assembler code for both 32-bit and 64-bit Sparc clobbers the "application registers" `g2`, `g3` and `g4`, the same way that the GCC default '`-mapp-regs`' does (see Section "SPARC Options" in *Using the GNU Compiler Collection (GCC)*).

  This makes that code unsuitable for use with the special V9 '`-mcmodel=embmedany`' (which uses `g4` as a data segment pointer), and for applications wanting to use those registers for special purposes. In these cases the only suggestion currently is to build MPIR with CPU '`none`' to avoid the assembler code.

SPARC Solaris
: Building applications against MPIR on SPARC Solaris (including `make check`) requires the `LD_LIBRARY_PATH` to be set appropriately. In particular if one is building with `ABI=64` the linker needs to know where to find `libgcc` (often often `/usr/lib/sparcv9` or `/usr/local/lib/sparcv9` or `/lib/sparcv9`).

  It is not enough to specify the location in `LD_LIBRARY_PATH_64` unless `LD_LIBRARY_PATH_64` is added to `LD_LIBRARY_PATH`. Specifically the 64 bit `libgcc` path needs to be in `LD_LIBRARY_PATH`.

  The linker is able to automatically distinguish 32 and 64 bit libraries, so it is safe to include paths to both the 32 and 64 bit libraries in the `LD_LIBRARY_PATH`.

Solaris 10 First Release on SPARC
: MPIR fails to build with Solaris 10 first release. Patch 123647-01 for SPARC, released by Sun in August 2006 fixes this problem.

x86 CPU Types
: '`i586`', '`pentium`' or '`pentiummmx`' code is good for its intended P5 Pentium chips, but quite slow when run on Intel P6 class chips (PPro, P-II, P-III). '`i386`' is a better choice when making binaries that must run on both.

x86 MMX and SSE2 Code
: If the CPU selected 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. The same applies to SSE2.

  Old versions of '`gas`' don't support MMX instructions, in particular version 1.92.3 that comes with FreeBSD 2.2.8 or the more recent OpenBSD 3.1 doesn't.

  Solaris 2.6 and 2.7 `as` generate incorrect object code for register to register `movq` instructions, and so can't be used for MMX code. Install a recent `gas` if MMX code is wanted on these systems.

2.6 Known Build Problems

You might find more up-to-date information at `http://www.mpir.org/`.

GCC XOP issues
: GCC from version 4.6.0 to 4.8.x have a problem with the XOP instruction, especially with '`-O3`' on at least AMD Opteron '`62xx/63xx`', '`FX-(4,6,8)[13]xx`' and the Devil's Canyon '`9xxx`' and the Kaveri APUs.

  A workaround is to pass '`-mno-xop`' when compiling with '`-O3`'.

2.7 Performance optimization

For optimal performance, build MPIR for the exact CPU type of the target computer, see Section 2.1 [Build Options], page 3.

Unlike what is the case for most other programs, the compiler typically doesn't matter much, since MPIR uses assembly language for the most critical operations.

In particular for long-running MPIR applications, and applications demanding extremely large numbers, building and running the `tuneup` program in the `tune` subdirectory, can be important. For example,

```
cd tune
make tuneup
./tuneup
```

will generate better contents for the `gmp-mparam.h` parameter file.

To use the results, put the output in the file indicated in the '`Parameters for ...`' header. Then recompile from scratch.

The `tuneup` program takes one useful parameter, '`-f NNN`', which instructs the program how long to check FFT multiply parameters. If you're going to use MPIR for extremely large numbers, you may want to run `tuneup` with a large NNN value.

3 MPIR Basics

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 MPIR.

3.1 Headers and Libraries

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

```
#include <mpir.h>
```

Note however that prototypes for MPIR functions with `FILE *` parameters are only provided if `<stdio.h>` is included too.

```
#include <stdio.h>
#include <mpir.h>
```

Likewise `<stdarg.h>` (or `<varargs.h>`) is required for prototypes with `va_list` parameters, such as `gmp_vprintf`. And `<obstack.h>` for prototypes with `struct obstack` parameters, such as `gmp_obstack_printf`, when available.

All programs using MPIR must link against the `libmpir` library. On a typical Unix-like system this can be done with '`-lmpir`' respectively, for example

```
gcc myprogram.c -lmpir
```

MPIR C++ functions are in a separate `libmpirxx` library. This is built and installed if C++ support has been enabled (see Section 2.1 [Build Options], page 3). For example,

```
g++ mycxxprog.cc -lmpirxx -lmpir
```

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

If MPIR has been installed to a non-standard location then it may be necessary to use '`-I`' and '`-L`' compiler options to point to the right directories, and some sort of run-time path for a shared library.

3.2 Nomenclature and Types

In this manual, *integer* usually means a multiple precision integer, as defined by the MPIR 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`. For example:

```
mpf_t fp;
```

The floating point functions accept and return exponents in the C type `mp_exp_t`. Currently this is usually a `long`, but on some systems it's an `int` for efficiency.

A *limb* means the part of a multi-precision number that fits in a single machine 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 is 32 or 64 bits. The C data type for a limb is `mp_limb_t`.

Counts of limbs are represented in the C type `mp_size_t`. Currently this is normally a `long`, but on some systems it's an `int` for efficiency.

Counts of bits of a multi-precision number are represented in the C type `mp_bitcnt_t`. Currently this is always an `unsigned long`, but on some systems it will be an `unsigned long long` in the future .

Random state means an algorithm selection and current state data. The C data type for such objects is `gmp_randstate_t`. For example:

```
gmp_randstate_t rstate;
```

Also, in general `mp_bitcnt_t` is used for bit counts and ranges, and `size_t` is used for byte or character counts.

3.3 MPIR on Windows x64

Although Windows x64 is a 64-bit operating system, Microsoft has decided to make long integers 32-bits, which is inconsistent when compared with almost all other 64-bit operating systems. This has caused many subtle bugs when open source code is ported to Windows x64 because many developers reasonably expect to find that long integers on a 64-bit operating system will be 64 bits long.

MPIR contains functions with suffixes of `_ui` and `_si` that are used to input unsigned and signed integers into and convert them for use with MPIR's multiple precision integers (mpz types). For example, the following functions set an `mpz_t` integer from `unsigned` and `signed long` integers respectively.

`void mpz_set_ui` (*mpz_t, unsigned long int*) [Function]

`void mpz_set_si` (*mpz_t, signed long int*) [Function]

Also, the following functions obtain `unsigned` and `signed long int` values from an MPIR multiple precision integer (`mpz_t`).

`unsigned long` *int mpz_get_ui* (*mpz_t*) [Function]

`signed long` *int mpz_get_si* (*mpz_t*) [Function]

To bring MPIR on Windows x64 into line with other 64-bit operating systems two new types have been introduced throughout MPIR:

- `mpir_ui` defined as `unsigned long int` on all but Windows x64, defined as `unsigned long long int` on Windows x64
- `mpir_si` defined as `signed long int` on all but Windows x64, defined as `signed long long int` on Windows x64

The above prototypes in MPIR 2.6.0 are changed to:

`void mpz_set_ui` (*mpz_t, mpir_ui*) [Function]

`void mpz_set_si` (*mpz_t, mpir_si*) [Function]

`mpir_ui mpz_get_ui` (*mpz_t*) [Function]

`mpir_si mpz_get_si` (*mpz_t*) [Function]

These changes are applied to all MPIR functions with `_ui` and `_si` suffixes.

3.4 Function Classes

There are five classes of functions in the MPIR library:

1. Functions for signed integer arithmetic, with names beginning with `mpz_`. The associated type is `mpz_t`. There are about 150 functions in this class. (see Chapter 5 [Integer Functions], page 29)
2. Functions for rational number arithmetic, with names beginning with `mpq_`. The associated type is `mpq_t`. There are about 40 functions in this class, but the integer functions can be used for arithmetic on the numerator and denominator separately. (see Chapter 6 [Rational Number Functions], page 46)
3. Functions for floating-point arithmetic, with names beginning with `mpf_`. The associated type is `mpf_t`. There are about 60 functions is this class. (see Chapter 7 [Floating-point Functions], page 50)
4. 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_`. The associated type is array of `mp_limb_t`. There are about 30 (hard-to-use) functions in this class. (see Chapter 8 [Low-level Functions], page 58)
5. Miscellaneous functions. Functions for setting up custom allocation and functions for generating random numbers. (see Chapter 14 [Custom Allocation], page 106, and see Chapter 9 [Random Number Functions], page 67)

3.5 Variable Conventions

MPIR functions generally have output arguments before input arguments. This notation is by analogy with the assignment operator.

MPIR 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 an MPIR 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 between each initialization. After a variable has been initialized, it may be assigned to any number of times.

For efficiency reasons, avoid excessive initializing and clearing. In general, initialize near the start of a function and clear near the end. For example,

```
void
```

```
foo (void)
{
  mpz_t  n;
  int    i;
  mpz_init (n);
  for (i = 1; i < 100; i++)
    {
      mpz_mul (n, ...);
      mpz_fdiv_q (n, ...);
      ...
    }
  mpz_clear (n);
}
```

3.6 Parameter Conventions

When an MPIR variable is used as a function parameter, it's effectively a call-by-reference, meaning if the function stores a value there it will change the original in the caller. Parameters which are input-only can be designated `const` to provoke a compiler error or warning on attempting to modify them.

When a function is going to return an MPIR result, it should designate a parameter that it sets, like the library functions do. More than one value can be returned by having more than one output parameter, again like the library functions. A `return` of an `mpz_t` etc doesn't return the object, only a pointer, and this is almost certainly not what's wanted.

Here's an example accepting an `mpz_t` parameter, doing a calculation, and storing the result to the indicated parameter.

```
void
foo (mpz_t result, const mpz_t param, mpir_ui n)
{
  mpir_ui  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);
  foo (r, n, 20L);
  gmp_printf ("%Zd\n", r);
  return 0;
}
```

`foo` works even if the mainline passes the same variable for `param` and `result`, just like the library functions. But sometimes it's tricky to make that work, and an application might not want to bother supporting that sort of thing.

For interest, the MPIR types `mpz_t` etc are implemented as one-element arrays of certain structures. This is why declaring a variable creates an object with the fields MPIR needs, but then

using it as a parameter passes a pointer to the object. Note that the actual fields in each `mpz_t` etc are for internal use only and should not be accessed directly by code that expects to be compatible with future MPIR releases.

3.7 Memory Management

The MPIR types like `mpz_t` are small, containing only a couple of sizes, and pointers to allocated data. Once a variable is initialized, MPIR takes care of all space allocation. Additional space is allocated whenever a variable doesn't have enough.

`mpz_t` and `mpq_t` variables never reduce their allocated space. Normally this is the best policy, since it avoids frequent reallocation. Applications that need to return memory to the heap at some particular point can use `mpz_realloc2`, or clear variables no longer needed.

`mpf_t` variables, in the current implementation, use a fixed amount of space, determined by the chosen precision and allocated at initialization, so their size doesn't change.

All memory is allocated using `malloc` and friends by default, but this can be changed, see Chapter 14 [Custom Allocation], page 106. Temporary memory on the stack is also used (via `alloca`), but this can be changed at build-time if desired, see Section 2.1 [Build Options], page 3.

3.8 Reentrancy

MPIR is reentrant and thread-safe, with some exceptions:

- If configured with `--enable-alloca=malloc-notreentrant` (or with `--enable-alloca=notreentrant` when `alloca` is not available), then naturally MPIR is not reentrant.
- `mpf_set_default_prec` and `mpf_init` use a global variable for the selected precision. `mpf_init2` can be used instead, and in the C++ interface an explicit precision to the `mpf_class` constructor.
- `mp_set_memory_functions` uses global variables to store 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, then MPIR will not be reentrant either.
- If the standard I/O functions such as `fwrite` are not reentrant then the MPIR I/O functions using them will not be reentrant either.
- It's safe for two threads to read from the same MPIR variable simultaneously, but it's not safe for one to read while the another might be writing, nor for two threads to write simultaneously. It's not safe for two threads to generate a random number from the same `gmp_randstate_t` simultaneously, since this involves an update of that variable.

3.9 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.k, these numbers will be i, j, and k, respectively. These numbers represent the version of GMP fully supported by this version of MPIR.

`__MPIR_VERSION` [Macro]
`__MPIR_VERSION_MINOR` [Macro]
`__MPIR_VERSION_PATCHLEVEL` [Macro]
The major and minor MPIR version, and patch level, respectively, as integers. For MPIR i.j.k, these numbers will be i, j, and k, respectively.

`const char * const gmp_version` [Global Constant]
The GNU MP version number, as a null-terminated string, in the form "i.j.k".

`__GMP_CC` [Macro]
`__GMP_CFLAGS` [Macro]
The compiler and compiler flags, respectively, used when compiling GMP, as strings.

`const char * const mpir_version` [Global Constant]
The MPIR version number, as a null-terminated string, in the form "i.j.k". This release is `"3.0.0"`.

3.10 Compatibility with older versions

This version of MPIR is upwardly binary compatible with all GMP 5.x, 4.x and 3.x versions, and upwardly compatible at the source level with all 2.x versions, 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 reverted to the 2.x style.
- `mpn_bdivmod` provided provisionally in the past has been removed from MPIR 2.7.0.
- MPIR does not support the secure cryptographic functions provided by GMP.
- Full GMP compatibility is only available when the '`--enable-gmpcompat`' configure option is used.

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 4 and MPIR 1 and 2. Please see the GMP 2 manual for details.

3.11 Efficiency

Small Operands
: On small operands, the time for function call overheads and memory allocation can be significant in comparison to actual calculation. This is unavoidable in a general purpose variable precision library, although MPIR attempts to be as efficient as it can on both large and small operands.

Static Linking
: On some CPUs, in particular the x86s, the static `libmpir.a` should be used for maximum speed, since the PIC code in the shared `libmpir.so` will have a small overhead on each function call and global data address. For many programs this will be insignificant, but for long calculations there's a gain to be had.

Initializing and Clearing
: Avoid excessive initializing and clearing of variables, since this can be quite time consuming, especially in comparison to otherwise fast operations like addition.

  A language interpreter might want to keep a free list or stack of initialized variables ready for use. It should be possible to integrate something like that with a garbage collector too.

Reallocations
: An `mpz_t` or `mpq_t` variable used to hold successively increasing values will have its memory repeatedly `realloc`ed, which could be quite slow or could fragment memory, depending on the C library. If an application can estimate the final size then `mpz_init2` or `mpz_realloc2` can be called to allocate the necessary space from the beginning (see Section 5.1 [Initializing Integers], page 29).

  It doesn't matter if a size set with `mpz_init2` or `mpz_realloc2` is too small, since all functions will do a further reallocation if necessary. Badly overestimating memory required will waste space though.

`2exp` Functions
: It's up to an application to call functions like `mpz_mul_2exp` when appropriate. General purpose functions like `mpz_mul` make no attempt to identify powers of two or other special forms, because such inputs will usually be very rare and testing every time would be wasteful.

`ui` and `si` Functions
: The `ui` functions and the small number of `si` functions exist for convenience and should be used where applicable. But if for example an `mpz_t` contains a value that fits in an `unsigned long` (`unsigned long long` on Windows x64) there's no need extract it and call a `ui` function, just use the regular `mpz` function.

In-Place Operations
: `mpz_abs`, `mpq_abs`, `mpf_abs`, `mpz_neg`, `mpq_neg` and `mpf_neg` are fast when used for in-place operations like `mpz_abs(x,x)`, since in the current implementation only a single field of `x` needs changing. On suitable compilers (GCC for instance) this is inlined too.

  `mpz_add_ui`, `mpz_sub_ui`, `mpf_add_ui` and `mpf_sub_ui` benefit from an in-place operation like `mpz_add_ui(x,x,y)`, since usually only one or two limbs of `x` will need to be changed. The same applies to the full precision `mpz_add` etc if `y` is small. If `y` is big then cache locality may be helped, but that's all.

  `mpz_mul` is currently the opposite, a separate destination is slightly better. A call like `mpz_mul(x,x,y)` will, unless `y` is only one limb, make a temporary copy of `x` before forming the result. Normally that copying will only be a tiny fraction of the time for the multiply, so this is not a particularly important consideration.

  `mpz_set`, `mpq_set`, `mpq_set_num`, `mpf_set`, etc, make no attempt to recognise a copy of something to itself, so a call like `mpz_set(x,x)` will be wasteful. Naturally that would never be written deliberately, but if it might arise from two pointers to the same object then a test to avoid it might be desirable.

```
if (x != y)
  mpz_set (x, y);
```

  Note that it's never worth introducing extra `mpz_set` calls just to get in-place operations. If a result should go to a particular variable then just direct it there and let MPIR take care of data movement.

Divisibility Testing (Small Integers)
: `mpz_divisible_ui_p` and `mpz_congruent_ui_p` are the best functions for testing whether an `mpz_t` is divisible by an individual small integer. They use an algorithm which is faster than `mpz_tdiv_ui`, but which gives no useful information about the actual remainder, only whether it's zero (or a particular value).

  However when testing divisibility by several small integers, it's best to take a remainder modulo their product, to save multi-precision operations. For instance to test whether a number is divisible by any of 23, 29 or 31 take a remainder modulo $23 \times 29 \times 31 = 20677$ and then test that.

The division functions like `mpz_tdiv_q_ui` which give a quotient as well as a remainder are generally a little slower than the remainder-only functions like `mpz_tdiv_ui`. If the quotient is only rarely wanted then it's probably best to just take a remainder and then go back and calculate the quotient if and when it's wanted (`mpz_divexact_ui` can be used if the remainder is zero).

Rational Arithmetic

The `mpq` functions operate on `mpq_t` values with no common factors in the numerator and denominator. Common factors are checked-for and cast out as necessary. In general, cancelling factors every time is the best approach since it minimizes the sizes for subsequent operations.

However, applications that know something about the factorization of the values they're working with might be able to avoid some of the GCDs used for canonicalization, or swap them for divisions. For example when multiplying by a prime it's enough to check for factors of it in the denominator instead of doing a full GCD. Or when forming a big product it might be known that very little cancellation will be possible, and so canonicalization can be left to the end.

The `mpq_numref` and `mpq_denref` macros give access to the numerator and denominator to do things outside the scope of the supplied `mpq` functions. See Section 6.5 [Applying Integer Functions], page 48.

The canonical form for rationals allows mixed-type `mpq_t` and integer additions or subtractions to be done directly with multiples of the denominator. This will be somewhat faster than `mpq_add`. For example,

```
/* mpq increment */
mpz_add (mpq_numref(q), mpq_numref(q), mpq_denref(q));

/* mpq += unsigned long */
mpz_addmul_ui (mpq_numref(q), mpq_denref(q), 123UL);

/* mpq -= mpz */
mpz_submul (mpq_numref(q), mpq_denref(q), z);
```

Number Sequences

Functions like `mpz_fac_ui`, `mpz_fib_ui` and `mpz_bin_uiui` are designed for calculating isolated values. If a range of values is wanted it's probably best to call to get a starting point and iterate from there.

Text Input/Output

Hexadecimal or octal are suggested for input or output in text form. Power-of-2 bases like these can be converted much more efficiently than other bases, like decimal. For big numbers there's usually nothing of particular interest to be seen in the digits, so the base doesn't matter much.

Maybe we can hope octal will one day become the normal base for everyday use, as proposed by King Charles XII of Sweden and later reformers.

3.12 Debugging

Stack Overflow

Depending on the system, a segmentation violation or bus error might be the only indication of stack overflow. See '`--enable-alloca`' choices in Section 2.1 [Build Options], page 3, for how to address this.

In new enough versions of GCC, '`-fstack-check`' may be able to ensure an overflow is recognised by the system before too much damage is done, or

'`-fstack-limit-symbol`' or '`-fstack-limit-register`' may be able to add checking if the system itself doesn't do any (see Section "Options for Code Generation" in *Using the GNU Compiler Collection (GCC)*). These options must be added to the '`CFLAGS`' used in the MPIR build (see Section 2.1 [Build Options], page 3), adding them just to an application will have no effect. Note also they're a slowdown, adding overhead to each function call and each stack allocation.

Heap Problems

The most likely cause of application problems with MPIR is heap corruption. Failing to `init` MPIR variables will have unpredictable effects, and corruption arising elsewhere in a program may well affect MPIR. Initializing MPIR variables more than once or failing to clear them will cause memory leaks.

In all such cases a `malloc` debugger is recommended. On a GNU or BSD system the standard C library `malloc` has some diagnostic facilities, see Section "Allocation Debugging" in *The GNU C Library Reference Manual*, or '`man 3 malloc`'. Other possibilities, in no particular order, include

```
http://dmalloc.com/
http://www.perens.com/FreeSoftware/ (electric fence)
http://www.gnupdate.org/components/leakbug/
http://wwww.gnome.org/projects/memprof
```

The MPIR default allocation routines in `memory.c` also have a simple sentinel scheme which can be enabled with `#define DEBUG` in that file. This is mainly designed for detecting buffer overruns during MPIR development, but might find other uses.

Stack Backtraces

On some systems the compiler options MPIR uses by default can interfere with debugging. In particular on x86 and 68k systems '`-fomit-frame-pointer`' is used and this generally inhibits stack backtracing. Recompiling without such options may help while debugging, though the usual caveats about it potentially moving a memory problem or hiding a compiler bug will apply.

GDB, the GNU Debugger

A sample `.gdbinit` is included in the distribution, showing how to call some undocumented dump functions to print MPIR variables from within GDB. Note that these functions shouldn't be used in final application code since they're undocumented and may be subject to incompatible changes in future versions of MPIR.

Source File Paths

MPIR has multiple source files with the same name, in different directories. For example `mpz`, `mpq` and `mpf` each have an `init.c`. If the debugger can't already determine the right one it may help to build with absolute paths on each C file. One way to do that is to use a separate object directory with an absolute path to the source directory.

```
cd /my/build/dir
/my/source/dir/gmp-3.0.0/configure
```

This works via `VPATH`, and might require GNU `make`. Alternately it might be possible to change the `.c.lo` rules appropriately.

Assertion Checking

The build option `--enable-assert` is available to add some consistency checks to the library (see Section 2.1 [Build Options], page 3). These are likely to be of limited value to most applications. Assertion failures are just as likely to indicate memory corruption as a library or compiler bug.

Applications using the low-level `mpn` functions, however, will benefit from `--enable-assert` since it adds checks on the parameters of most such functions, many of which

have subtle restrictions on their usage. Note however that only the generic C code has checks, not the assembler code, so CPU '`none`' should be used for maximum checking.

Temporary Memory Checking
: The build option `--enable-alloca=debug` arranges that each block of temporary memory in MPIR is allocated with a separate call to `malloc` (or the allocation function set with `mp_set_memory_functions`).

  This can help a malloc debugger detect accesses outside the intended bounds, or detect memory not released. In a normal build, on the other hand, temporary memory is allocated in blocks which MPIR divides up for its own use, or may be allocated with a compiler builtin `alloca` which will go nowhere near any malloc debugger hooks.

Maximum Debuggability
: To summarize the above, an MPIR build for maximum debuggability would be

```
./configure --disable-shared --enable-assert \
  --enable-alloca=debug --host=none CFLAGS=-g
```

  For C++, add '`--enable-cxx CXXFLAGS=-g`'.

Checker
: The GCC checker (`http://savannah.gnu.org/projects/checker/`) can be used with MPIR. It contains a stub library which means MPIR applications compiled with checker can use a normal MPIR build.

  A build of MPIR with checking within MPIR itself can be made. This will run very very slowly. On GNU/Linux for example,

```
./configure --host=none-pc-linux-gnu CC=checkergcc
```

  '`--host=none`' must be used, since the MPIR assembler code doesn't support the checking scheme. The MPIR C++ features cannot be used, since current versions of checker (0.9.9.1) don't yet support the standard C++ library.

Valgrind
: The valgrind program (`http://valgrind.org/`) is a memory checker for x86s. It translates and emulates machine instructions to do strong checks for uninitialized data (at the level of individual bits), memory accesses through bad pointers, and memory leaks.

  Recent versions of Valgrind are getting support for MMX and SSE/SSE2 instructions, for past versions MPIR will need to be configured not to use those, ie. for an x86 without them (for instance plain '`i486`').

Other Problems
: Any suspected bug in MPIR itself should be isolated to make sure it's not an application problem, see Chapter 4 [Reporting Bugs], page 28.

3.13 Profiling

Running a program under a profiler is a good way to find where it's spending most time and where improvements can be best sought. The profiling choices for a MPIR build are as follows.

'`--disable-profiling`'
: The default is to add nothing special for profiling.

  It should be possible to just compile the mainline of a program with `-p` and use `prof` to get a profile consisting of timer-based sampling of the program counter. Most of the MPIR assembler code has the necessary symbol information.

  This approach has the advantage of minimizing interference with normal program operation, but on most systems the resolution of the sampling is quite low (10 milliseconds for instance), requiring long runs to get accurate information.

‘--enable-profiling=prof’
: Build with support for the system `prof`, which means ‘`-p`’ added to the ‘`CFLAGS`’.

  This provides call counting in addition to program counter sampling, which allows the most frequently called routines to be identified, and an average time spent in each routine to be determined.

  The x86 assembler code has support for this option, but on other processors the assembler routines will be as if compiled without ‘`-p`’ and therefore won't appear in the call counts.

  On some systems, such as GNU/Linux, ‘`-p`’ in fact means ‘`-pg`’ and in this case ‘`--enable-profiling=gprof`’ described below should be used instead.

‘--enable-profiling=gprof’
: Build with support for `gprof` (see *GNU gprof*), which means ‘`-pg`’ added to the ‘`CFLAGS`’.

  This provides call graph construction in addition to call counting and program counter sampling, which makes it possible to count calls coming from different locations. For example the number of calls to `mpn_mul` from `mpz_mul` versus the number from `mpf_mul`. The program counter sampling is still flat though, so only a total time in `mpn_mul` would be accumulated, not a separate amount for each call site.

  The x86 assembler code has support for this option, but on other processors the assembler routines will be as if compiled without ‘`-pg`’ and therefore not be included in the call counts.

  On x86 and m68k systems ‘`-pg`’ and ‘`-fomit-frame-pointer`’ are incompatible, so the latter is omitted from the default flags in that case, which might result in poorer code generation.

  Incidentally, it should be possible to use the `gprof` program with a plain ‘`--enable-profiling=prof`’ build. But in that case only the ‘`gprof -p`’ flat profile and call counts can be expected to be valid, not the ‘`gprof -q`’ call graph.

‘--enable-profiling=instrument’
: Build with the GCC option ‘`-finstrument-functions`’ added to the ‘`CFLAGS`’ (see Section “Options for Code Generation” in *Using the GNU Compiler Collection (GCC)*).

  This inserts special instrumenting calls at the start and end of each function, allowing exact timing and full call graph construction.

  This instrumenting is not normally a standard system feature and will require support from an external library, such as

  ```
  http://sourceforge.net/projects/fnccheck/
  ```

  This should be included in ‘`LIBS`’ during the MPIR configure so that test programs will link. For example,

  ```
  ./configure --enable-profiling=instrument LIBS=-lfc
  ```

  On a GNU system the C library provides dummy instrumenting functions, so programs compiled with this option will link. In this case it's only necessary to ensure the correct library is added when linking an application.

  The x86 assembler code supports this option, but on other processors the assembler routines will be as if compiled without ‘`-finstrument-functions`’ meaning time spent in them will effectively be attributed to their caller.

3.14 Autoconf

Autoconf based applications can easily check whether MPIR is installed. The only thing to be noted is that GMP/MPIR library symbols from version 3 of GMP and version 1 of MPIR onwards have prefixes like `__gmpz`. The following therefore would be a simple test,

```
AC_CHECK_LIB(mpir, __gmpz_init)
```

This just uses the default `AC_CHECK_LIB` actions for found or not found, but an application that must have MPIR would want to generate an error if not found. For example,

```
AC_CHECK_LIB(mpir, __gmpz_init, ,
  [AC_MSG_ERROR([MPIR not found, see http://www.mpir.org/])])
```

If functions added in some particular version of GMP/MPIR are required, then one of those can be used when checking. For example `mpz_mul_si` was added in GMP 3.1,

```
AC_CHECK_LIB(mpir, __gmpz_mul_si, ,
  [AC_MSG_ERROR(
  [GMP/MPIR not found, or not GMP 3.1 or up or MPIR 1.0 or up, see http://www.mpir.org/
```

An alternative would be to test the version number in `mpir.h` using say `AC_EGREP_CPP`. That would make it possible to test the exact version, if some particular sub-minor release is known to be necessary.

In general it's recommended that applications should simply demand a new enough MPIR rather than trying to provide supplements for features not available in past versions.

Occasionally an application will need or want to know the size of a type at configuration or preprocessing time, not just with `sizeof` in the code. This can be done in the normal way with `mp_limb_t` etc, but GMP 4.0 or up and MPIR 1.0 and up is best for this, since prior versions needed certain '`-D`' defines on systems using a `long long` limb. The following would suit Autoconf 2.50 or up,

```
AC_CHECK_SIZEOF(mp_limb_t, , [#include <mpir.h>])
```

3.15 Emacs

`C-h C-i` (`info-lookup-symbol`) is a good way to find documentation on C functions while editing (see Section "Info Documentation Lookup" in *The Emacs Editor*).

The MPIR manual can be included in such lookups by putting the following in your `.emacs`,

```
(eval-after-load "info-look"
  '(let ((mode-value (assoc 'c-mode (assoc 'symbol info-lookup-alist))))
     (setcar (nthcdr 3 mode-value)
             (cons '("(gmp)Function Index" nil "^ -.* " "\\>")
                   (nth 3 mode-value)))))
```

4 Reporting Bugs

If you think you have found a bug in the MPIR 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, check it's not already addressed in Section 2.6 [Known Build Problems], page 14, or perhaps Section 2.5 [Notes for Particular Systems], page 13. You may also want to check `http://www.mpir.org/` for patches for this release.

Please include the following in any report,

- The MPIR 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 MPIR, 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`', and from running '`./configfsf.guess`' (might be the same).
- 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` and the offending line or lines from the temporary `mpn/tmp-<file>.s`.

Please make an effort to produce a self-contained report, with something definite that can be tested or debugged. Vague queries or piecemeal messages are difficult to act on and don't help the development effort.

It is not uncommon that an observed problem is actually due to a bug in the compiler; the MPIR 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: `http://groups.google.com/group/mpir-devel`.

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 MPIR functions for performing integer arithmetic. These functions start with the prefix `mpz_`.

MPIR 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`. For example,

```
{
  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_init (`*mpz_t* ***integer***`)` [Function]
Initialize *integer*, and set its value to 0.

`void mpz_inits (`*mpz_t* `x`, ...`)` [Function]
Initialize a NULL-terminated list of `mpz_t` variables, and set their values to 0.

`void mpz_init2 (`*mpz_t* ***integer***, *mp_bitcnt_t* `n`) [Function]
Initialize *integer*, with space for *n* bits, and set its value to 0.

n is only the initial space, *integer* will grow automatically in the normal way, if necessary, for subsequent values stored. `mpz_init2` makes it possible to avoid such reallocations if a maximum size is known in advance.

`void mpz_clear (`*mpz_t* ***integer***`)` [Function]
Free the space occupied by *integer*. Call this function for all `mpz_t` variables when you are done with them.

`void mpz_clears (`*mpz_t* `x`, ...`)` [Function]
Free the space occupied by a NULL-terminated list of `mpz_t` variables.

`void mpz_realloc2 (`*mpz_t* ***integer***, *mp_bitcnt_t* `n`) [Function]
Change the space allocated for *integer* to *n* bits. The value in *integer* is preserved if it fits, or is set to 0 if not.

This function can be used to increase the space for a variable in order to avoid repeated automatic reallocations, or to decrease it to give memory back to the heap.

5.2 Assignment Functions

These functions assign new values to already initialized integers (see Section 5.1 [Initializing Integers], page 29).

`void mpz_set` (*mpz_t* **rop**, *mpz_t* **op**) [Function]
`void mpz_set_ui` (*mpz_t* **rop**, *mpir_ui* **op**) [Function]
`void mpz_set_si` (*mpz_t* **rop**, *mpir_si* **op**) [Function]
`void mpz_set_ux` (*mpz_t* **rop**, *uintmax_t* **op**) [Function]
`void mpz_set_sx` (*mpz_t* **rop**, *intmax_t* **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*. Note the intmax versions are only available if you include the `stdint.h` header before including `mpir.h`.

`mpz_set_d`, `mpz_set_q` and `mpz_set_f` truncate *op* to make it an integer.

`int mpz_set_str` (*mpz_t* **rop**, *char* ***str**, *int* **base**) [Function]
Set the value of *rop* from *str*, a null-terminated C string in base *base*. White space is allowed in the string, and is simply ignored.

The *base* may vary from 2 to 62, or if *base* is 0, then the leading characters are used: `0x` and `0X` for hexadecimal, `0b` and `0B` for binary, `0` for octal, or decimal otherwise.

For bases up to 36, case is ignored; upper-case and lower-case letters have the same value. For bases 37 to 62, upper-case letter represent the usual 10..35 while lower-case letter represent 36..61.

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

`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, MPIR 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**, *mpir_ui* **op**) [Function]
`void mpz_init_set_si` (*mpz_t* **rop**, *mpir_si* **op**) [Function]
`void mpz_init_set_ux` (*mpz_t* **rop**, *uintmax_t* **op**) [Function]
`void mpz_init_set_sx` (*mpz_t* **rop**, *intmax_t* **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*. Note the intmax versions are only available if you include the `stdint.h` header before including `mpir.h`.

`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 MPIR integers to standard C types. Functions for converting *to* MPIR integers are described in Section 5.2 [Assigning Integers], page 30, and Section 5.12 [I/O of Integers], page 40.

`mpir_ui mpz_get_ui` (*mpz_t* **op**) [Function]
Return the value of *op* as an `mpir_ui`.

If *op* is too big to fit an `mpir_ui` then just the least significant bits that do fit are returned. The sign of *op* is ignored, only the absolute value is used.

`mpir_si mpz_get_si` (*mpz_t* **op**) [Function]
If *op* fits into a `mpir_si` return the value of *op*. Otherwise return the least significant part of *op*, with the same sign as *op*.

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

`uintmax_t mpz_get_ux` (*mpz_t* **op**) [Function]
Return the value of *op* as an `uintmax_t`.

If *op* is too big to fit an `uintmax_t` then just the least significant bits that do fit are returned. The sign of *op* is ignored, only the absolute value is used. Note this function is only available if you include `stdint.h` before including `mpir.h`.

`intmax_t mpz_get_sx` (*mpz_t* **op**) [Function]
If *op* fits into a `intmax_t` return the value of *op*. Otherwise return the least significant part of *op*, with the same sign as *op*.

If *op* is too big to fit in a `intmax_t`, the returned result is probably not very useful. Note this function is only available if you include the `stdint.h` header before including `mpir.h`.

`double mpz_get_d` (*mpz_t* **op**) [Function]
Convert *op* to a `double`, truncating if necessary (ie. rounding towards zero).

If the exponent from the conversion is too big, the result is system dependent. An infinity is returned where available. A hardware overflow trap may or may not occur.

`double mpz_get_d_2exp` (*mpir_si *exp, mpz_t op*) [Function]
Convert *op* to a `double`, truncating if necessary (ie. rounding towards zero), and returning the exponent separately.

The return value is in the range $0.5 \le |d| < 1$ and the exponent is stored to `*exp`. $d * 2^{exp}$ is the (truncated) *op* value. If *op* is zero, the return is 0.0 and 0 is stored to `*exp`.

This is similar to the standard C `frexp` function (see Section "Normalization Functions" in *The GNU C Library Reference Manual*).

`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 or from −2 to −36.

For *base* in the range 2..36, digits and lower-case letters are used; for −2..−36, digits and upper-case letters are used; for 37..62, digits, upper-case letters, and lower-case letters (in that significance order) are used.

If *str* is `NULL`, the result string is allocated using the current allocation function (see Chapter 14 [Custom Allocation], page 106). The block will be `strlen(str)+1` bytes, that being exactly enough for the string and null-terminator.

If *str* is not `NULL`, it should point to a block of storage large enough for the result, that being `mpz_sizeinbase (op, base) + 2`. The two extra bytes are for a possible minus sign, and the null-terminator.

A pointer to the result string is returned, being either the allocated block, or the given *str*.

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, mpir_ui 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, mpir_ui op2*) [Function]
`void mpz_ui_sub` (*mpz_t rop, mpir_ui op1, mpz_t 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, mpir_si op2*) [Function]
`void mpz_mul_ui` (*mpz_t rop, mpz_t op1, mpir_ui op2*) [Function]
Set *rop* to *op1* × *op2*.

`void mpz_addmul` (*mpz_t rop, mpz_t op1, mpz_t op2*) [Function]
`void mpz_addmul_ui` (*mpz_t rop, mpz_t op1, mpir_ui op2*) [Function]
Set *rop* to *rop* + *op1* × *op2*.

`void mpz_submul` (*mpz_t rop, mpz_t op1, mpz_t op2*) [Function]
`void mpz_submul_ui` (*mpz_t rop, mpz_t op1, mpir_ui op2*) [Function]
Set *rop* to *rop* − *op1* × *op2*.

`void mpz_mul_2exp` (*mpz_t rop, mpz_t op1, mp_bitcnt_t op2*) [Function]
Set *rop* to $op1 \times 2^{op2}$. This operation can also be defined as a left shift by *op2* bits.

`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. Passing a zero divisor to the division or modulo functions (including the modular powering functions `mpz_powm` and `mpz_powm_ui`), will cause an intentional division by zero. This lets a program handle arithmetic exceptions in these functions the same way as for normal C `int` arithmetic.

`void mpz_cdiv_q` (*mpz_t* **q**, *mpz_t* **n**, *mpz_t* **d**) [Function]
`void mpz_cdiv_r` (*mpz_t* **r**, *mpz_t* **n**, *mpz_t* **d**) [Function]
`void mpz_cdiv_qr` (*mpz_t* **q**, *mpz_t* **r**, *mpz_t* **n**, *mpz_t* **d**) [Function]
`mpir_ui mpz_cdiv_q_ui` (*mpz_t* **q**, *mpz_t* **n**, *mpir_ui* **d**) [Function]
`mpir_ui mpz_cdiv_r_ui` (*mpz_t* **r**, *mpz_t* **n**, *mpir_ui* **d**) [Function]
`mpir_ui mpz_cdiv_qr_ui` (*mpz_t* **q**, *mpz_t* **r**, *mpz_t* **n**, *mpir_ui* **d**) [Function]
`mpir_ui mpz_cdiv_ui` (*mpz_t* **n**, *mpir_ui* **d**) [Function]
`void mpz_cdiv_q_2exp` (*mpz_t* **q**, *mpz_t* **n**, *mp_bitcnt_t* **b**) [Function]
`void mpz_cdiv_r_2exp` (*mpz_t* **r**, *mpz_t* **n**, *mp_bitcnt_t* **b**) [Function]

`void mpz_fdiv_q` (*mpz_t* **q**, *mpz_t* **n**, *mpz_t* **d**) [Function]
`void mpz_fdiv_r` (*mpz_t* **r**, *mpz_t* **n**, *mpz_t* **d**) [Function]
`void mpz_fdiv_qr` (*mpz_t* **q**, *mpz_t* **r**, *mpz_t* **n**, *mpz_t* **d**) [Function]
`mpir_ui mpz_fdiv_q_ui` (*mpz_t* **q**, *mpz_t* **n**, *mpir_ui* **d**) [Function]
`mpir_ui mpz_fdiv_r_ui` (*mpz_t* **r**, *mpz_t* **n**, *mpir_ui* **d**) [Function]
`mpir_ui mpz_fdiv_qr_ui` (*mpz_t* **q**, *mpz_t* **r**, *mpz_t* **n**, *mpir_ui* **d**) [Function]
`mpir_ui mpz_fdiv_ui` (*mpz_t* **n**, *mpir_ui* **d**) [Function]
`void mpz_fdiv_q_2exp` (*mpz_t* **q**, *mpz_t* **n**, *mp_bitcnt_t* **b**) [Function]
`void mpz_fdiv_r_2exp` (*mpz_t* **r**, *mpz_t* **n**, *mp_bitcnt_t* **b**) [Function]

`void mpz_tdiv_q` (*mpz_t* **q**, *mpz_t* **n**, *mpz_t* **d**) [Function]
`void mpz_tdiv_r` (*mpz_t* **r**, *mpz_t* **n**, *mpz_t* **d**) [Function]
`void mpz_tdiv_qr` (*mpz_t* **q**, *mpz_t* **r**, *mpz_t* **n**, *mpz_t* **d**) [Function]
`mpir_ui mpz_tdiv_q_ui` (*mpz_t* **q**, *mpz_t* **n**, *mpir_ui* **d**) [Function]
`mpir_ui mpz_tdiv_r_ui` (*mpz_t* **r**, *mpz_t* **n**, *mpir_ui* **d**) [Function]
`mpir_ui mpz_tdiv_qr_ui` (*mpz_t* **q**, *mpz_t* **r**, *mpz_t* **n**, *mpir_ui* **d**) [Function]
`mpir_ui mpz_tdiv_ui` (*mpz_t* **n**, *mpir_ui* **d**) [Function]
`void mpz_tdiv_q_2exp` (*mpz_t* **q**, *mpz_t* **n**, *mp_bitcnt_t* **b**) [Function]
`void mpz_tdiv_r_2exp` (*mpz_t* **r**, *mpz_t* **n**, *mp_bitcnt_t* **b**) [Function]

Divide n by d, forming a quotient q and/or remainder r. For the `2exp` functions, $d = 2^b$. The rounding is in three styles, each suiting different applications.

- `cdiv` rounds q up towards $+\infty$, and r will have the opposite sign to d. The `c` stands for "ceil".
- `fdiv` rounds q down towards $-\infty$, and r will have the same sign as d. The `f` stands for "floor".
- `tdiv` rounds q towards zero, and r will have the same sign as n. The `t` stands for "truncate".

In all cases q and r will satisfy $n = qd + r$, and r will satisfy $0 \le |r| < |d|$.

The q functions calculate only the quotient, the r functions only the remainder, and the qr functions calculate both. Note that for qr the same variable cannot be passed for both q and r, or results will be unpredictable.

For the ui variants the return value is the remainder, and in fact returning the remainder is all the div_ui functions do. For tdiv and cdiv the remainder can be negative, so for those the return value is the absolute value of the remainder.

For the 2exp variants the divisor is 2^b. These functions are implemented as right shifts and bit masks, but of course they round the same as the other functions.

For positive n both mpz_fdiv_q_2exp and mpz_tdiv_q_2exp are simple bitwise right shifts. For negative n, mpz_fdiv_q_2exp is effectively an arithmetic right shift treating n as twos complement the same as the bitwise logical functions do, whereas mpz_tdiv_q_2exp effectively treats n as sign and magnitude.

void mpz_mod (*mpz_t* r, *mpz_t* n, *mpz_t* d) [Function]
mpir_ui mpz_mod_ui (*mpz_t* r, *mpz_t* n, *mpir_ui* d) [Function]
Set r to n mod d. The sign of the divisor is ignored; the result is always non-negative.

mpz_mod_ui is identical to mpz_fdiv_r_ui above, returning the remainder as well as setting r. See mpz_fdiv_ui above if only the return value is wanted.

void mpz_divexact (*mpz_t* q, *mpz_t* n, *mpz_t* d) [Function]
void mpz_divexact_ui (*mpz_t* q, *mpz_t* n, *mpir_ui* d) [Function]
Set q to n/d. These functions produce correct results only when it is known in advance that d divides n.

These routines are much faster than the other division functions, and are the best choice when exact division is known to occur, for example reducing a rational to lowest terms.

int mpz_divisible_p (*mpz_t* n, *mpz_t* d) [Function]
int mpz_divisible_ui_p (*mpz_t* n, *mpir_ui* d) [Function]
int mpz_divisible_2exp_p (*mpz_t* n, *mp_bitcnt_t* b) [Function]
Return non-zero if n is exactly divisible by d, or in the case of mpz_divisible_2exp_p by 2^b.

n is divisible by d if there exists an integer q satisfying $n = qd$. Unlike the other division functions, $d = 0$ is accepted and following the rule it can be seen that only 0 is considered divisible by 0.

int mpz_congruent_p (*mpz_t* n, *mpz_t* c, *mpz_t* d) [Function]
int mpz_congruent_ui_p (*mpz_t* n, *mpir_ui* c, *mpir_ui* d) [Function]
int mpz_congruent_2exp_p (*mpz_t* n, *mpz_t* c, *mp_bitcnt_t* b) [Function]
Return non-zero if n is congruent to c modulo d, or in the case of mpz_congruent_2exp_p modulo 2^b.

n is congruent to c mod d if there exists an integer q satisfying $n = c + qd$. Unlike the other division functions, $d = 0$ is accepted and following the rule it can be seen that n and c are considered congruent mod 0 only when exactly equal.

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**, *mpir_ui* **exp**, *mpz_t* **mod**) [Function]
Set *rop* to $base^{exp}$ mod *mod*.

A negative *exp* is supported in `mpz_powm` if an inverse $base^{-1}$ mod *mod* exists (see `mpz_invert` in Section 5.9 [Number Theoretic Functions], page 36). If an inverse doesn't exist then a divide by zero is raised.

`void mpz_pow_ui` (*mpz_t* **rop**, *mpz_t* **base**, *mpir_ui* **exp**) [Function]
`void mpz_ui_pow_ui` (*mpz_t* **rop**, *mpir_ui* **base**, *mpir_ui* **exp**) [Function]
Set *rop* to $base^{exp}$. The case 0^0 yields 1.

5.8 Root Extraction Functions

`int mpz_root` (*mpz_t* **rop**, *mpz_t* **op**, *mpir_ui* **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_nthroot` (*mpz_t* **rop**, *mpz_t* **op**, *mpir_ui* **n**) [Function]
Set *rop* to $\lfloor \sqrt[n]{op} \rfloor$, the truncated integer part of the *n*th root of *op*.

`void mpz_rootrem` (*mpz_t* **root**, *mpz_t* **rem**, *mpz_t* **u**, *mpir_ui* **n**) [Function]
Set *root* to $\lfloor \sqrt[n]{u} \rfloor$, the truncated integer part of the *n*th root of *u*. Set *rem* to the remainder, $(u - root^n)$.

`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 the remainder $(op - rop1^2)$, which will be 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 $op = a^b$.

Under this definition both 0 and 1 are considered to be perfect powers. Negative values of *op* are accepted, but of course can only be odd perfect powers.

`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. Under this definition both 0 and 1 are considered to be perfect squares.

5.9 Number Theoretic Functions

`int mpz_probable_prime_p` (*mpz_t* `n`, *gmp_randstate_t* `state`, *int* `prob`, *mpir_ui* `div`) [Function]

Determine whether *n* is a probable prime with the chance of error being at most 1 in 2^prob. return value is 1 if *n* is probably prime, or 0 if *n* is definitely composite.

This function does some trial divisions to speed up the average case, then some probabilistic primality tests to achieve the desired level of error.

div can be used to inform the function that trial division up to *div* has already been performed on *n* and so *n* has NO divisors <= *div*.Use 0 to inform the function that no trial division has been done.

This function interface is preliminary and may change in the future.

`int mpz_likely_prime_p` (*mpz_t* `n`, *gmp_randstate_t* `state`, *mpir_ui* `div`) [Function]

Determine whether *n* is likely a prime, i.e. you can consider it a prime for practical purposes. return value is 1 if *n* can be considered prime, or 0 if *n* is definitely composite.

This function does some trial divisions to speed up the average case, then some probabilistic primality tests. The term "likely" refers to the fact that the number will not have small factors.

div can be used to inform the function that trial division up to *div* has already been performed on *n* and so *n* has NO divisors <= *div*

This function interface is preliminary and may change in the future.

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

Determine whether *n* is prime. Return 2 if *n* is definitely prime, return 1 if *n* is probably prime (without being certain), or return 0 if *n* is definitely composite.

This function does some trial divisions, then some Miller-Rabin probabilistic primality tests. *reps* controls how many such tests are done, 5 to 10 is a reasonable number, more will reduce the chances of a composite being returned as "probably prime".

Miller-Rabin and similar tests can be more properly called compositeness tests. Numbers which fail are known to be composite but those which pass might be prime or might be composite. Only a few composites pass, hence those which pass are considered probably prime.

This function is obsolete. It will disappear from future MPIR releases.

`void 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. For practical purposes it's adequate, the chance of a composite passing will be extremely small. However, despite the name, it does not guarantee primality.

This function is obsolete. It will disappear from future MPIR releases.

`void mpz_next_prime_candidate (`*mpz_t* **rop**, *mpz_t* **op**, *gmp_randstate_t* **state**`)` [Function]

Set *rop* to the next candidate prime greater than *op*. Note that this function will occasionally return composites. It is designed to give a quick method for generating numbers which do not have small prime factors (less than 1000) and which pass a small number of rounds of Miller-Rabin (just two rounds).The test is designed for speed, assuming that a high quality followup test can then be run to ensure primality.

The variable *state* must be initialized by calling one of the `gmp_randinit` functions (Section 9.1 [Random State Initialization], page 67) before invoking this function.

`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 one or both input operands are negative.

`mpir_ui mpz_gcd_ui (`*mpz_t* **rop**, *mpz_t* **op1**, *mpir_ui* **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 `mpir_ui`, 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**, *const mpz_t* **a**, *const mpz_t* **b**`)` [Function]

Set g to the greatest common divisor of a and b, and in addition set s and t to coefficients satisfying $as + bt = g$. The value in g is always positive, even if one or both of a and b are negative (or zero if both inputs are zero). The values in s and t are chosen such that normally, $|s| < |b|/(2g)$ and $|t| < |a|/(2g)$, and these relations define s and t uniquely. There are a few exceptional cases:

If $|a| = |b|$, then $s = 0$, $t = sgn(b)$.

Otherwise, $s = sgn(\mathrm{a})$ if $b = 0$ or $|b| = 2g$, and $t = sgn(b)$ if $\mathrm{a} = 0$ or $|\mathrm{a}| = 2g$.

In all cases, $s = 0$ if and only if $g = |b|$, i.e., if b divides a or $\mathrm{a} = b = 0$.

If t is `NULL` then that value is not computed.

`void mpz_lcm (`*mpz_t* **rop**, *mpz_t* **op1**, *mpz_t* **op2**`)` [Function]
`void mpz_lcm_ui (`*mpz_t* **rop**, *mpz_t* **op1**, *mpir_ui* **op2**`)` [Function]

Set *rop* to the least common multiple of *op1* and *op2*. *rop* is always positive, irrespective of the signs of *op1* and *op2*. *rop* will be zero if either *op1* or *op2* is zero.

`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*. If the inverse exists, the return value is non-zero and *rop* will satisfy $0 \le rop < op2$. If an inverse doesn't exist the return value is zero and *rop* is undefined.

`int mpz_jacobi (`*mpz_t* **a**, *mpz_t* **b**`)` [Function]

Calculate the Jacobi symbol $\left(\frac{a}{b}\right)$. This is defined only for b odd.

`int mpz_legendre (`*mpz_t* **a**, *mpz_t* **p**`)` [Function]

Calculate the Legendre symbol $\left(\frac{a}{p}\right)$. This is defined only for p an odd positive prime, and for such p it's identical to the Jacobi symbol.

`int mpz_kronecker` (*mpz_t* `a`, *mpz_t* `b`) [Function]
`int mpz_kronecker_si` (*mpz_t* `a`, *mpir_si* `b`) [Function]
`int mpz_kronecker_ui` (*mpz_t* `a`, *mpir_ui* `b`) [Function]
`int mpz_si_kronecker` (*mpir_si* `a`, *mpz_t* `b`) [Function]
`int mpz_ui_kronecker` (*mpir_ui* `a`, *mpz_t* `b`) [Function]
Calculate the Jacobi symbol $(\frac{a}{b})$ with the Kronecker extension $(\frac{a}{2}) = (\frac{2}{a})$ when a odd, or $(\frac{a}{2}) = 0$ when a even.

When b is odd the Jacobi symbol and Kronecker symbol are identical, so `mpz_kronecker_ui` etc can be used for mixed precision Jacobi symbols too.

For more information see Henri Cohen section 1.4.2 (see Appendix B [References], page 145), or any number theory textbook. See also the example program `demos/qcn.c` which uses `mpz_kronecker_ui` on the MPIR website.

`mp_bitcnt_t 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*. The return value is how many such occurrences were removed.

`void mpz_fac_ui` (*mpz_t* `rop`, *unsigned long int* `n`) [Function]
`void mpz_2fac_ui` (*mpz_t* `rop`, *unsigned long int* `n`) [Function]
`void mpz_mfac_uiui` (*mpz_t* `rop`, *unsigned long int* `n`, *unsigned long int* `m`) [Function]
Set *rop* to the factorial of n: `mpz_fac_ui` computes the plain factorial $n!$, `mpz_2fac_ui` computes the double-factorial $n!!$, and `mpz_mfac_uiui` the m-multi-factorial $n!^{(m)}$.

`void mpz_primorial_ui` (*mpz_t* `rop`, *unsigned long int* `n`) [Function]
Set *rop* to the primorial of n, i.e. the product of all positive prime numbers $\leq n$.

`void mpz_bin_ui` (*mpz_t* `rop`, *mpz_t* `n`, *mpir_ui* `k`) [Function]
`void mpz_bin_uiui` (*mpz_t* `rop`, *mpir_ui* `n`, *mpir_ui* `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* `fn`, *mpir_ui* `n`) [Function]
`void mpz_fib2_ui` (*mpz_t* `fn`, *mpz_t* `fnsub1`, *mpir_ui* `n`) [Function]
`mpz_fib_ui` sets *fn* to to F_n, the n'th Fibonacci number. `mpz_fib2_ui` sets *fn* to F_n, and *fnsub1* to F_{n-1}.

These functions are designed for calculating isolated Fibonacci numbers. When a sequence of values is wanted it's best to start with `mpz_fib2_ui` and iterate the defining $F_{n+1} = F_n + F_{n-1}$ or similar.

`void mpz_lucnum_ui` (*mpz_t* `ln`, *mpir_ui* `n`) [Function]
`void mpz_lucnum2_ui` (*mpz_t* `ln`, *mpz_t* `lnsub1`, *mpir_ui* `n`) [Function]
`mpz_lucnum_ui` sets *ln* to to L_n, the n'th Lucas number. `mpz_lucnum2_ui` sets *ln* to L_n, and *lnsub1* to L_{n-1}.

These functions are designed for calculating isolated Lucas numbers. When a sequence of values is wanted it's best to start with `mpz_lucnum2_ui` and iterate the defining $L_{n+1} = L_n + L_{n-1}$ or similar.

The Fibonacci numbers and Lucas numbers are related sequences, so it's never necessary to call both `mpz_fib2_ui` and `mpz_lucnum2_ui`. The formulas for going from Fibonacci to Lucas can be found in Section 16.7.5 [Lucas Numbers Algorithm], page 129, the reverse is straightforward too.

5.10 Comparison Functions

`int mpz_cmp` (*mpz_t op1*, *mpz_t op2*) [Function]
`int mpz_cmp_d` (*mpz_t op1*, *double op2*) [Function]
`int mpz_cmp_si` (*mpz_t op1*, *mpir_si op2*) [Macro]
`int mpz_cmp_ui` (*mpz_t op1*, *mpir_ui op2*) [Macro]
Compare *op1* and *op2*. Return a positive value if $op1 > op2$, zero if $op1 = op2$, or a negative value if $op1 < op2$.

`mpz_cmp_ui` and `mpz_cmp_si` are macros and will evaluate their arguments more than once. `mpz_cmp_d` can be called with an infinity, but results are undefined for a NaN.

`int mpz_cmpabs` (*mpz_t op1*, *mpz_t op2*) [Function]
`int mpz_cmpabs_d` (*mpz_t op1*, *double op2*) [Function]
`int mpz_cmpabs_ui` (*mpz_t op1*, *mpir_ui op2*) [Function]
Compare the absolute values of *op1* and *op2*. Return a positive value if $|op1| > |op2|$, zero if $|op1| = |op2|$, or a negative value if $|op1| < |op2|$.

`mpz_cmpabs_d` can be called with an infinity, but results are undefined for a NaN.

`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 argument multiple times.

5.11 Logical and Bit Manipulation Functions

These functions behave as if twos complement arithmetic were used (although sign-magnitude is the actual implementation). The least significant bit is number 0.

`void mpz_and` (*mpz_t rop*, *mpz_t op1*, *mpz_t op2*) [Function]
Set *rop* to *op1* bitwise-and *op2*.

`void mpz_ior` (*mpz_t rop*, *mpz_t op1*, *mpz_t op2*) [Function]
Set *rop* to *op1* bitwise inclusive-or *op2*.

`void mpz_xor` (*mpz_t rop*, *mpz_t op1*, *mpz_t op2*) [Function]
Set *rop* to *op1* bitwise exclusive-or *op2*.

`void mpz_com` (*mpz_t rop*, *mpz_t op*) [Function]
Set *rop* to the one's complement of *op*.

`mp_bitcnt_t mpz_popcount` (*mpz_t op*) [Function]
If $op \geq 0$, return the population count of *op*, which is the number of 1 bits in the binary representation. If $op < 0$, the number of 1s is infinite, and the return value is *ULONG_MAX*, the largest possible `mp_bitcnt_t`.

`mp_bitcnt_t mpz_hamdist` (*mpz_t op1*, *mpz_t op2*) [Function]
If *op1* and *op2* are both ≥ 0 or both < 0, return the hamming distance between the two operands, which is the number of bit positions where *op1* and *op2* have different bit values.

If one operand is ≥ 0 and the other < 0 then the number of bits different is infinite, and the return value is the largest possible `imp_bitcnt_t`.

`mp_bitcnt_t mpz_scan0` (*mpz_t op, mp_bitcnt_t starting_bit*) [Function]
`mp_bitcnt_t mpz_scan1` (*mpz_t op, mp_bitcnt_t starting_bit*) [Function]
Scan *op*, starting from bit *starting_bit*, towards more significant bits, until the first 0 or 1 bit (respectively) is found. Return the index of the found bit.

If the bit at *starting_bit* is already what's sought, then *starting_bit* is returned.

If there's no bit found, then the largest possible `mp_bitcnt_t` is returned. This will happen in `mpz_scan0` past the end of a positive number, or `mpz_scan1` past the end of a nonnegative number.

`void mpz_setbit` (*mpz_t rop, mp_bitcnt_t bit_index*) [Function]
Set bit *bit_index* in *rop*.

`void mpz_clrbit` (*mpz_t rop, mp_bitcnt_t bit_index*) [Function]
Clear bit *bit_index* in *rop*.

`void mpz_combit` (*mpz_t rop, mp_bitcnt_t bit_index*) [Function]
Complement bit *bit_index* in *rop*.

`int mpz_tstbit` (*mpz_t op, mp_bitcnt_t bit_index*) [Function]
Test 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 `mpir.h`, since that will allow `mpir.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 argument may vary from 2 to 62 or from -2 to -36.

For *base* in the range 2..36, digits and lower-case letters are used; for $-2..-36$, digits and upper-case letters are used; for 37..62, digits, upper-case letters, and lower-case letters (in that significance order) are used.

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 62, or if *base* is 0, then the leading characters are used: `0x` and `0X` for hexadecimal, `0b` and `0B` for binary, `0` for octal, or decimal otherwise.

For bases up to 36, case is ignored; upper-case and lower-case letters have the same value. For bases 37 to 62, upper-case letter represent the usual 10..35 while lower-case letter represent 36..61.

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 MPIR 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 67, for more information on how to use and not to use random number functions.

`void mpz_urandomb (mpz_t rop, gmp_randstate_t state, mp_bitcnt_t n)` [Function]
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 67) before invoking this function.

`void mpz_urandomm (mpz_t rop, gmp_randstate_t state, mpz_t n)` [Function]
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 67) before invoking this function.

`void mpz_rrandomb (mpz_t rop, gmp_randstate_t state, mp_bitcnt_t 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 67) before invoking this function.

5.14 Integer Import and Export

`mpz_t` variables can be converted to and from arbitrary words of binary data with the following functions.

`void mpz_import (`*mpz_t* `rop`, *size_t* `count`, *int* `order`, *size_t* `size`, *int* `endian`, *size_t* `nails`, *const void* `*op``)` [Function]

Set *rop* from an array of word data at *op*.

The parameters specify the format of the data. *count* many words are read, each *size* bytes. *order* can be 1 for most significant word first or -1 for least significant first. Within each word *endian* can be 1 for most significant byte first, -1 for least significant first, or 0 for the native endianness of the host CPU. The most significant *nails* bits of each word are skipped, this can be 0 to use the full words.

There is no sign taken from the data, *rop* will simply be a positive integer. An application can handle any sign itself, and apply it for instance with `mpz_neg`.

There are no data alignment restrictions on *op*, any address is allowed.

Here's an example converting an array of `mpir_ui` data, most significant element first, and host byte order within each value.

```
mpir_ui  a[20];
mpz_t          z;
mpz_import (z, 20, 1, sizeof(a[0]), 0, 0, a);
```

This example assumes the full `sizeof` bytes are used for data in the given type, which is usually true, and certainly true for `mpir_ui` everywhere we know of. However on Cray vector systems it may be noted that `short` and `int` are always stored in 8 bytes (and with `sizeof` indicating that) but use only 32 or 46 bits. The *nails* feature can account for this, by passing for instance `8*sizeof(int)-INT_BIT`.

`void * mpz_export (`*void* `*rop`, *size_t* `*countp`, *int* `order`, *size_t* `size`, *int* `endian`, *size_t* `nails`, *mpz_t* `op``)` [Function]

Fill *rop* with word data from *op*.

The parameters specify the format of the data produced. Each word will be *size* bytes and *order* can be 1 for most significant word first or -1 for least significant first. Within each word *endian* can be 1 for most significant byte first, -1 for least significant first, or 0 for the native endianness of the host CPU. The most significant *nails* bits of each word are unused and set to zero, this can be 0 to produce full words.

The number of words produced is written to `*countp`, or *countp* can be `NULL` to discard the count. *rop* must have enough space for the data, or if *rop* is `NULL` then a result array of the necessary size is allocated using the current MPIR allocation function (see Chapter 14 [Custom Allocation], page 106). In either case the return value is the destination used, either *rop* or the allocated block.

If *op* is non-zero then the most significant word produced will be non-zero. If *op* is zero then the count returned will be zero and nothing written to *rop*. If *rop* is `NULL` in this case, no block is allocated, just `NULL` is returned.

The sign of *op* is ignored, just the absolute value is exported. An application can use `mpz_sgn` to get the sign and handle it as desired. (see Section 5.10 [Integer Comparisons], page 39)

There are no data alignment restrictions on *rop*, any address is allowed.

When an application is allocating space itself the required size can be determined with a calculation like the following. Since `mpz_sizeinbase` always returns at least 1, `count` here will be at least one, which avoids any portability problems with `malloc(0)`, though if `z` is zero no space at all is actually needed (or written).

```
numb = 8*size - nail;
count = (mpz_sizeinbase (z, 2) + numb-1) / numb;
p = malloc (count * size);
```

5.15 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`, `long`, `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 argument more than once.

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

Return the size of *op* measured in number of digits in the given *base*. *base* can vary from 2 to 36. The sign of *op* is ignored, just the absolute value is used. The result will be either exact or 1 too big. If *base* is a power of 2, the result is always exact. If *op* is zero the return value is always 1.

This function can be used to determine the space required when 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 null-terminator.

It will be noted that `mpz_sizeinbase(op,2)` can be used to locate the most significant 1 bit in *op*, counting from 1. (Unlike the bitwise functions which start from 0, See Section 5.11 [Logical and Bit Manipulation Functions], page 39.)

5.16 Special Functions

The functions in this section are for various special purposes. Most applications will not need them.

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

This is a special type of initialization. **Fixed** space of *fixed_num_bits* is allocated to each of the *array_size* integers in *integer_array*. There is no way to free the storage allocated by this function. Don't call `mpz_clear`!

The *integer_array* parameter is the first `mpz_t` in the array. For example,

```
mpz_t  arr[20000];
mpz_array_init (arr[0], 20000, 512);
```

This function is only intended for programs that create a large number of integers and need to reduce memory usage by avoiding the overheads of allocating and reallocating lots of small blocks. In normal programs this function is not recommended.

The space allocated to each integer by this function will not be automatically increased, unlike the normal `mpz_init`, so an application must ensure it is sufficient for any value stored. The following space requirements apply to various routines,

- `mpz_abs`, `mpz_neg`, `mpz_set`, `mpz_set_si` and `mpz_set_ui` need room for the value they store.
- `mpz_add`, `mpz_add_ui`, `mpz_sub` and `mpz_sub_ui` need room for the larger of the two operands, plus an extra `mp_bits_per_limb`.
- `mpz_mul`, `mpz_mul_ui` and `mpz_mul_ui` need room for the sum of the number of bits in their operands, but each rounded up to a multiple of `mp_bits_per_limb`.
- `mpz_swap` can be used between two array variables, but not between an array and a normal variable.

For other functions, or if in doubt, the suggestion is to calculate in a regular `mpz_init` variable and copy the result to an array variable with `mpz_set`.

This function is obsolete. It will disappear from future MPIR releases.

`void * _mpz_realloc` (*mpz_t integer, mp_size_t new_alloc*) [Function]
Change the space for *integer* to *new_alloc* limbs. The value in *integer* is preserved if it fits, or is set to 0 if not. The return value is not useful to applications and should be ignored.

`mpz_realloc2` is the preferred way to accomplish allocation changes like this. `mpz_realloc2` and `_mpz_realloc` are the same except that `_mpz_realloc` takes its size in limbs.

`mp_limb_t mpz_getlimbn` (*mpz_t op, mp_size_t n*) [Function]
Return limb number *n* from *op*. The sign of *op* is ignored, just the absolute value is used. The least significant limb is number 0.

`mpz_size` can be used to find how many limbs make up *op*. `mpz_getlimbn` returns zero if *n* is outside the range 0 to `mpz_size(op)-1`.

`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.

`const mp_limb_t * mpz_limbs_read` (*const mpz_t x*) [Function]
Return a pointer to the limb array representing the absolute value of *x*. The size of the array is `mpz_size(x)`. Intended for read access only.

`mp_limb_t * mpz_limbs_write` (*mpz_t x, mp_size_t n*) [Function]
`mp_limb_t * mpz_limbs_modify` (*mpz_t x, mp_size_t n*) [Function]
Return a pointer to the limb array, intended for write access. The array is reallocated as needed, to make room for *n* limbs. Requires $n > 0$. The `mpz_limbs_modify` function returns an array that holds the old absolute value of *x*, while `mpz_limbs_write` may destroy the old value and return an array with unspecified contents.

`void mpz_limbs_finish` (*mpz_t x, mp_size_t s*) [Function]
Updates the internal size field of *x*. Used after writing to the limb array pointer returned by `mpz_limbs_write` or `mpz_limbs_modify` is completed. The array should contain $|s|$ valid limbs, representing the new absolute value for *x*, and the sign of *x* is taken from the sign of *s*. This function never reallocates *x*, so the limb pointer remains valid.

```
void foo (mpz_t x)
```

```
{
  mp_size_t n, i;
  mp_limb_t *xp;

  n = mpz_size (x);
  xp = mpz_limbs_modify (x, 2*n);
  for (i = 0; i < n; i++)
    xp[n+i] = xp[n-1-i];
  mpz_limbs_finish (x, mpz_sgn (x) < 0 ? - 2*n : 2*n);
}
```

`mpz_srcptr mpz_roinit_n` (*mpz_t x, const mp_limb_t *xp, mp_size_t xs*) [Function]

Special initialization of *x*, using the given limb array and size. *x* should be treated as read-only: it can be passed safely as input to any mpz function, but not as an output. The array *xp* must point to at least a readable limb, its size is $|xs|$, and the sign of *x* is the sign of *xs*. For convenience, the function returns *x*, but cast to a const pointer type.

```
void foo (mpz_t x)
{
  static const mp_limb_t y[3] = { 0x1, 0x2, 0x3 };
  mpz_t tmp;
  mpz_add (x, x, mpz_roinit_n (tmp, y, 3));
}
```

`mpz_t MPZ_ROINIT_N` (*mp_limb_t *xp, mp_size_t xs*) [Macro]

This macro expands to an initializer which can be assigned to an mpz_t variable. The limb array *xp* must point to at least a readable limb, moreover, unlike the `mpz_roinit_n` function, the array must be normalized: if *xs* is non-zero, then `xp[`$|xs|-1$`]` must be non-zero. Intended primarily for constant values. Using it for non-constant values requires a C compiler supporting C99.

```
void foo (mpz_t x)
{
  static const mp_limb_t ya[3] = { 0x1, 0x2, 0x3 };
  static const mpz_t y = MPZ_ROINIT_N ((mp_limb_t *) ya, 3);

  mpz_add (x, x, y);
}
```

6 Rational Number Functions

This chapter describes the MPIR 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.

`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_inits (`*mpq_t x, ...*`)` [Function]
Initialize a NULL-terminated list of `mpq_t` variables, and set their values to 0/1.

`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_clears (`*mpq_t x, ...*`)` [Function]
Free the space occupied by a NULL-terminated list of `mpq_t` variables.

`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, mpir_ui op1, mpir_ui op2*`)` [Function]
`void mpq_set_si (`*mpq_t rop, mpir_si op1, mpir_ui 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*.

`int mpq_set_str (`*mpq_t rop, char *str, int base*`)` [Function]
Set *rop* from a null-terminated string *str* in the given *base*.

The string can be an integer like "41" or a fraction like "41/152". The fraction must be in canonical form (see Chapter 6 [Rational Number Functions], page 46), or if not then `mpq_canonicalize` must be called.

The numerator and optional denominator are parsed the same as in `mpz_set_str` (see Section 5.2 [Assigning Integers], page 30). White space is allowed in the string, and is simply

ignored. The *base* can vary from 2 to 62, or if *base* is 0 then the leading characters are used: `0x` or `0X` for hex, `0b` or `0B` for binary, `0` for octal, or decimal otherwise. Note that this is done separately for the numerator and denominator, so for instance `0xEF/100` is 239/100, whereas `0xEF/0x100` is 239/256.

The return value is 0 if the entire string is a valid number, or −1 if not.

`void mpq_swap (`*mpq_t* ***rop1***, *mpq_t* ***rop2***`)` [Function]
Swap the values *rop1* and *rop2* efficiently.

6.2 Conversion Functions

`double mpq_get_d (`*mpq_t* ***op***`)` [Function]
Convert *op* to a `double`, truncating if necessary (ie. rounding towards zero).

If the exponent from the conversion is too big or too small to fit a `double` then the result is system dependent. For too big an infinity is returned when available. For too small 0.0 is normally returned. Hardware overflow, underflow and denorm traps may or may not occur.

`void mpq_set_d (`*mpq_t* ***rop***, *double* ***op***`)` [Function]
`void mpq_set_f (`*mpq_t* ***rop***, *mpf_t* ***op***`)` [Function]
Set *rop* to the value of *op*. There is no rounding, this conversion is exact.

`char * mpq_get_str (`*char* ***str***, *int* `base`, *mpq_t* ***op***`)` [Function]
Convert *op* to a string of digits in base *base*. The base may vary from 2 to 36. The string will be of the form '`num/den`', or if the denominator is 1 then just '`num`'.

If *str* is `NULL`, the result string is allocated using the current allocation function (see Chapter 14 [Custom Allocation], page 106). The block will be `strlen(str)+1` bytes, that being exactly enough for the string and null-terminator.

If *str* is not `NULL`, it should point to a block of storage large enough for the result, that being

```
mpz_sizeinbase (mpq_numref(op), base)
+ mpz_sizeinbase (mpq_denref(op), base) + 3
```

The three extra bytes are for a possible minus sign, possible slash, and the null-terminator.

A pointer to the result string is returned, being either the allocated block, or the given *str*.

6.3 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_mul_2exp (`*mpq_t* ***rop***, *mpq_t* ***op1***, *mp_bitcnt_t* ***op2***`)` [Function]
Set *rop* to $op1 \times 2^{op2}$.

`void mpq_div` (*mpq_t* **quotient**, *mpq_t* **dividend**, *mpq_t* **divisor**) [Function]
Set *quotient* to *dividend*/*divisor*.

`void mpq_div_2exp` (*mpq_t* **rop**, *mpq_t* **op1**, *mp_bitcnt_t* **op2**) [Function]
Set *rop* to $op1/2^{op2}$.

`void mpq_neg` (*mpq_t* **negated_operand**, *mpq_t* **operand**) [Function]
Set *negated_operand* to $-operand$.

`void mpq_abs` (*mpq_t* **rop**, *mpq_t* **op**) [Function]
Set *rop* to the absolute value of *op*.

`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.

6.4 Comparison Functions

`int mpq_cmp` (*mpq_t* **op1**, *mpq_t* **op2**) [Function]
`int mpq_cmp_z` (*const mpq_t* **op1**, *const mpz_t* **op2**) [Function]
Compare *op1* and *op2*. Return a positive value if $op1 > op2$, zero if $op1 = op2$, and a negative value if $op1 < op2$.

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

`int mpq_cmp_ui` (*mpq_t* **op1**, *mpir_ui* **num2**, *mpir_ui* **den2**) [Macro]
`int mpq_cmp_si` (*mpq_t* **op1**, *mpir_si* **num2**, *mpir_ui* **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 $op1 < num2/den2$.

num2 and *den2* are allowed to have common factors.

These functions are implemented as a macros and evaluate their 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.

6.5 Applying Integer Functions to Rationals

The set of `mpq` functions is quite small. In particular, there are few functions for either input or output. The following functions give direct access to the numerator and denominator of an `mpq_t`.

Note that if an assignment to the numerator and/or denominator could take an `mpq_t` out of the canonical form described at the start of this chapter (see Chapter 6 [Rational Number Functions], page 46) then `mpq_canonicalize` must be called before any other `mpq` functions are applied to that `mpq_t`.

`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.

`void mpq_get_num` (*mpz_t numerator*, *mpq_t rational*) [Function]
`void mpq_get_den` (*mpz_t denominator*, *mpq_t rational*) [Function]
`void mpq_set_num` (*mpq_t rational*, *mpz_t numerator*) [Function]
`void mpq_set_den` (*mpq_t rational*, *mpz_t denominator*) [Function]
Get or set the numerator or denominator of a rational. These functions are equivalent to calling `mpz_set` with an appropriate `mpq_numref` or `mpq_denref`. Direct use of `mpq_numref` or `mpq_denref` is recommended instead of these functions.

6.6 Input and Output Functions

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

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

`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.

`size_t mpq_inp_str` (*mpq_t rop*, *FILE *stream*, *int base*) [Function]
Read a string of digits from *stream* and convert them to a rational in *rop*. Any initial white-space characters are read and discarded. Return the number of characters read (including white space), or 0 if a rational could not be read.

The input can be a fraction like '`17/63`' or just an integer like '`123`'. Reading stops at the first character not in this form, and white space is not permitted within the string. If the input might not be in canonical form, then `mpq_canonicalize` must be called (see Chapter 6 [Rational Number Functions], page 46).

The *base* can be between 2 and 36, or can be 0 in which case the leading characters of the string determine the base, '`0x`' or '`0X`' for hexadecimal, '`0`' for octal, or decimal otherwise. The leading characters are examined separately for the numerator and denominator of a fraction, so for instance '`0x10/11`' is 16/11, whereas '`0x10/0x11`' is 16/17.

7 Floating-point Functions

MPIR floating point numbers are stored in objects of type `mpf_t` and functions operating on them have an `mpf_` prefix.

The mantissa of each float has a user-selectable precision, limited only by available memory. Each variable has its own precision, and that can be increased or decreased at any time.

The exponent of each float is a fixed precision, one machine word on most systems. In the current implementation the exponent is a count of limbs, so for example on a 32-bit system this means a range of roughly $2^{-68719476768}$ to $2^{68719476736}$, or on a 64-bit system this will be greater. Note however `mpf_get_str` can only return an exponent which fits an `mp_exp_t` and currently `mpf_set_str` doesn't accept exponents bigger than a `mpir_si`.

Each variable keeps a size for the mantissa data actually in use. This means that if a float is exactly represented in only a few bits then only those bits will be used in a calculation, even if the selected precision is high.

All calculations are performed to the precision of the destination variable. Each function is defined to calculate with "infinite precision" followed by a truncation to the destination precision, but of course the work done is only what's needed to determine a result under that definition.

The precision selected for a variable is a minimum value, MPIR may increase it a little to facilitate efficient calculation. Currently this means rounding up to a whole limb, and then sometimes having a further partial limb, depending on the high limb of the mantissa. But applications shouldn't be concerned by such details.

The mantissa in stored in binary, as might be imagined from the fact precisions are expressed in bits. One consequence of this is that decimal fractions like 0.1 cannot be represented exactly. The same is true of plain IEEE `double` floats. This makes both highly unsuitable for calculations involving money or other values that should be exact decimal fractions. (Suitably scaled integers, or perhaps rationals, are better choices.)

`mpf` functions and variables have no special notion of infinity or not-a-number, and applications must take care not to overflow the exponent or results will be unpredictable. This might change in a future release.

Note that the `mpf` functions are *not* intended as a smooth extension to IEEE P754 arithmetic. In particular results obtained on one computer often differ from the results on a computer with a different word size.

7.1 Initialization Functions

`void mpf_set_default_prec` (*mp_bitcnt_t 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.

`mp_bitcnt_t mpf_get_default_prec` (*void*) [Function]
Return the default precision actually used.

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, mp_bitcnt_t 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_inits` (*mpf_t x, ...*) [Function]
Initialize a NULL-terminated list of `mpf_t` variables, and set their values to 0. The precision of the initialized variables is undefined unless a default precision has already been established by a call to `mpf_set_default_prec`.

`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.

`void mpf_clears` (*mpf_t x, ...*) [Function]
Free the space occupied by a NULL-terminated list of `mpf_t` variables.

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.

`mp_bitcnt_t mpf_get_prec` (*mpf_t op*) [Function]
Return the current precision of *op*, in bits.

`void mpf_set_prec` (*mpf_t rop, mp_bitcnt_t prec*) [Function]
Set the precision of *rop* to be **at least** *prec* bits. The value in *rop* will be truncated to the new precision.

This function requires a call to `realloc`, and so should not be used in a tight loop.

`void mpf_set_prec_raw` (*mpf_t rop, mp_bitcnt_t prec*) [Function]
Set the precision of *rop* to be **at least** *prec* bits, without changing the memory allocated.

prec must be no more than the allocated precision for *rop*, that being the precision when *rop* was initialized, or in the most recent `mpf_set_prec`.

The value in *rop* is unchanged, and in particular if it had a higher precision than *prec* it will retain that higher precision. New values written to *rop* will use the new *prec*.

Before calling `mpf_clear` or the full `mpf_set_prec`, another `mpf_set_prec_raw` call must be made to restore *rop* to its original allocated precision. Failing to do so will have unpredictable results.

`mpf_get_prec` can be used before `mpf_set_prec_raw` to get the original allocated precision. After `mpf_set_prec_raw` it reflects the *prec* value set.

`mpf_set_prec_raw` is an efficient way to use an `mpf_t` variable at different precisions during a calculation, perhaps to gradually increase precision in an iteration, or just to use various different precisions for different purposes during a calculation.

7.2 Assignment Functions

These functions assign new values to already initialized floats (see Section 7.1 [Initializing Floats], page 50).

`void mpf_set (mpf_t rop, mpf_t op)` [Function]
`void mpf_set_ui (mpf_t rop, mpir_ui op)` [Function]
`void mpf_set_si (mpf_t rop, mpir_si 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 decimal point expected is taken from the current locale, on systems providing `localeconv`.

The argument *base* may be in the ranges 2 to 62, or −62 to −2. Negative values are used to specify that the exponent is in decimal.

For bases up to 36, case is ignored; upper-case and lower-case letters have the same value; for bases 37 to 62, upper-case letter represent the usual 10..35 while lower-case letter represent 36..61.

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 is a valid number in base *base*. Otherwise it returns −1.

`void mpf_swap (mpf_t rop1, mpf_t rop2)` [Function]
Swap *rop1* and *rop2* efficiently. Both the values and the precisions of the two variables are swapped.

7.3 Combined Initialization and Assignment Functions

For convenience, MPIR 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**, *mpir_ui* **op**) [Function]
`void mpf_init_set_si` (*mpf_t* **rop**, *mpir_si* **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`, truncating if necessary (ie. rounding towards zero).

If the exponent in *op* is too big or too small to fit a `double` then the result is system dependent. For too big an infinity is returned when available. For too small 0.0 is normally returned. Hardware overflow, underflow and denorm traps may or may not occur.

`double mpf_get_d_2exp` (*mpir_si* ***exp**, *mpf_t* **op**) [Function]
Convert *op* to a `double`, truncating if necessary (ie. rounding towards zero), and with an exponent returned separately.

The return value is in the range $0.5 \le |d| < 1$ and the exponent is stored to `*exp`. $d * 2^{exp}$ is the (truncated) *op* value. If *op* is zero, the return is 0.0 and 0 is stored to `*exp`.

This is similar to the standard C `frexp` function (see Section "Normalization Functions" in *The GNU C Library Reference Manual*).

`mpir_si mpf_get_si` (*mpf_t* **op**) [Function]
`mpir_ui mpf_get_ui` (*mpf_t* **op**) [Function]
Convert *op* to a `mpir_si` or `mpir_ui`, truncating any fraction part. If *op* is too big for the return type, the result is undefined.

See also `mpf_fits_slong_p` and `mpf_fits_ulong_p` (see Section 7.8 [Miscellaneous Float Functions], page 56).

`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*. *base* can vary from 2 to 362 or from −2 to −36. Up to *n_digits* digits will be generated. Trailing zeros are not returned. No more digits than can be accurately represented by *op* are ever generated. If *n_digits* is 0 then that accurate maximum number of digits are generated.

For *base* in the range 2..36, digits and lower-case letters are used; for −2..−36, digits and upper-case letters are used; for 37..62, digits, upper-case letters, and lower-case letters (in that significance order) are used.

If *str* is `NULL`, the result string is allocated using the current allocation function (see Chapter 14 [Custom Allocation], page 106). The block will be `strlen(str)+1` bytes, that being exactly enough for the string and null-terminator.

If *str* is not `NULL`, it should point to a block of *n_digits* + 2 bytes, that being enough for the mantissa, a possible minus sign, and a null-terminator. When *n_digits* is 0 to get all significant digits, an application won't be able to know the space required, and *str* should be `NULL` in that case.

The generated string is a fraction, with an implicit radix point immediately to the left of the first digit. The applicable exponent is written through the *expptr* pointer. For example, the number 3.1416 would be returned as string `"31416"` and exponent 1.

When *op* is zero, an empty string is produced and the exponent returned is 0.

A pointer to the result string is returned, being either the allocated block or the given *str*.

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, mpir_ui 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, mpir_ui op1, mpf_t op2*) [Function]
`void mpf_sub_ui` (*mpf_t rop, mpf_t op1, mpir_ui 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, mpir_ui op2*) [Function]

Set *rop* to *op1* × *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, mpir_ui op1, mpf_t op2*) [Function]
`void mpf_div_ui` (*mpf_t rop, mpf_t op1, mpir_ui op2*) [Function]

Set *rop* to *op1*/*op2*.

`void mpf_sqrt` (*mpf_t rop, mpf_t op*) [Function]
`void mpf_sqrt_ui` (*mpf_t rop, mpir_ui op*) [Function]

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

`void mpf_pow_ui` (*mpf_t* **rop**, *mpf_t* **op1**, *mpir_ui* **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**, *mp_bitcnt_t* **op2**) [Function]
Set *rop* to $op1 \times 2^{op2}$.

`void mpf_div_2exp` (*mpf_t* **rop**, *mpf_t* **op1**, *mp_bitcnt_t* **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_d` (*mpf_t* **op1**, *double* **op2**) [Function]
`int mpf_cmp_ui` (*mpf_t* **op1**, *mpir_ui* **op2**) [Function]
`int mpf_cmp_si` (*mpf_t* **op1**, *mpir_si* **op2**) [Function]
Compare *op1* and *op2*. Return a positive value if $op1 > op2$, zero if $op1 = op2$, and a negative value if $op1 < op2$.

`mpf_cmp_d` can be called with an infinity, but results are undefined for a NaN.

`int mpf_eq` (*mpf_t* **op1**, *mpf_t* **op2**, *mp_bitcnt_t op3*) [Function]
Return non-zero if the first *op3* bits of *op1* and *op2* are equal, zero otherwise. I.e., test if *op1* and *op2* are approximately equal.

In the future values like 1000 and 0111 may be considered the same to 3 bits (on the basis that their difference is that small).

`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*. This is $|op1 - op2|/op1$.

`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 `mpir.h`, since that will allow `mpir.h` to define prototypes for these functions.

`size_t mpf_out_str` (*FILE* ***stream**, *int* **base**, *size_t* **n_digits**, *mpf_t* **op**) [Function]
Print *op* to *stream*, as a string of digits. Return the number of bytes written, or if an error occurred, return 0.

The mantissa is prefixed with an '0.' and is in the given *base*, which may vary from 2 to 36. An exponent then printed, separated by an 'e', or if *base* is greater than 10 then by an '@'. The exponent is always in decimal. The decimal point follows the current locale, on systems providing `localeconv`.

For *base* in the range 2..36, digits and lower-case letters are used; for −2..−36, digits and upper-case letters are used; for 37..62, digits, upper-case letters, and lower-case letters (in that significance order) are used.

Up to *n_digits* will be printed from the mantissa, except that no more digits than are accurately representable by *op* will be printed. *n_digits* can be 0 to select that accurate maximum.

`size_t mpf_inp_str` (*mpf_t* `rop`, *FILE* `*stream`, *int* `base`) [Function]
Read a string in base *base* from *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 decimal point expected is taken from the current locale, on systems providing `localeconv`.

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.

`int mpf_integer_p` (*mpf_t* `op`) [Function]
Return non-zero if *op* is an integer.

`int mpf_fits_ulong_p` (*mpf_t* `op`) [Function]
`int mpf_fits_slong_p` (*mpf_t* `op`) [Function]
`int mpf_fits_uint_p` (*mpf_t* `op`) [Function]
`int mpf_fits_sint_p` (*mpf_t* `op`) [Function]
`int mpf_fits_ushort_p` (*mpf_t* `op`) [Function]
`int mpf_fits_sshort_p` (*mpf_t* `op`) [Function]
Return non-zero if *op* would fit in the respective C data type, when truncated to an integer.

`void mpf_urandomb` (*mpf_t* `rop`, *gmp_randstate_t* `state`, *mp_bitcnt_t* `nbits`) [Function]
Generate a uniformly distributed random float in *rop*, such that $0 \leq rop < 1$, 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 67) before invoking this function.

`void mpf_rrandomb` (*mpf_t* `rop`, *gmp_randstate_t* `state`, *mp_size_t* `max_size`, *mp_exp_t* `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 (in limbs). This function is useful for testing functions and algorithms, since these 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 interface is preliminary. It might change incompatibly in future revisions.

`void mpf_random2` (*mpf_t* `rop`, *mp_size_t* `max_size`, *mp_exp_t* `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 (in limbs). This function is useful for testing functions and algorithms, since these 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. It will disappear from future MPIR releases.

8 Low-level Functions

This chapter describes low-level MPIR functions, used to implement the high-level MPIR 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 *n* limbs.

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

It should be noted that the `mpn` functions make no attempt to identify high or low zero limbs on their operands, or other special forms. On random data such cases will be unlikely and it'd be wasteful for every function to check every time. An application knowing something about its data can take steps to trim or perhaps split its calculations.

In the notation used below, a source operand is identified by the pointer to the least significant limb, and the limb count in braces. For example, {*s1p*, *s1n*}.

`mp_limb_t mpn_add_n` (*mp_limb_t* ***rp**, const mp_limb_t* ***s1p**, const mp_limb_t* ***s2p**, mp_size_t* **n**) [Function]

Add {*s1p*, *n*} and {*s2p*, *n*}, and write the *n* 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 CPUs. 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* **n**, *mp_limb_t* **s2limb**) [Function]

Add {*s1p*, *n*} and *s2limb*, and write the *n* 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* **s1n**, *const mp_limb_t* ***s2p**, mp_size_t* **s2n**) [Function]

Add {*s1p*, *s1n*} and {*s2p*, *s2n*}, and write the *s1n* least significant limbs of the result to *rp*. Return carry, either 0 or 1.

This function requires that *s1n* is greater than or equal to *s2n*.

`mp_limb_t mpn_sub_n` (*mp_limb_t* *`rp`, *const mp_limb_t* *`s1p`, *const mp_limb_t* *`s2p`, *mp_size_t* `n`) [Function]

Subtract {*s2p*, *n*} from {*s1p*, *n*}, and write the *n* 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 CPUs.

`mp_limb_t mpn_sub_1` (*mp_limb_t* *`rp`, *const mp_limb_t* *`s1p`, *mp_size_t* `n`, *mp_limb_t* `s2limb`) [Function]

Subtract *s2limb* from {*s1p*, *n*}, and write the *n* 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* `s1n`, *const mp_limb_t* *`s2p`, *mp_size_t* `s2n`) [Function]

Subtract {*s2p*, *s2n*} from {*s1p*, *s1n*}, and write the *s1n* least significant limbs of the result to *rp*. Return borrow, either 0 or 1.

This function requires that *s1n* is greater than or equal to *s2n*.

`void mpn_neg` (*mp_limb_t* *`rp`, *const mp_limb_t* *`sp`, *mp_size_t* `n`) [Function]

Perform the negation of {*sp*, *n*}, and write the result to {*rp*, *n*}. Return carry-out.

`void mpn_mul_n` (*mp_limb_t* *`rp`, *const mp_limb_t* *`s1p`, *const mp_limb_t* *`s2p`, *mp_size_t* `n`) [Function]

Multiply {*s1p*, *n*} and {*s2p*, *n*}, and write the 2**n*-limb result to *rp*.

The destination has to have space for 2**n* limbs, even if the product's most significant limb is zero. No overlap is permitted between the destination and either source.

If the input operands are the same, `mpn_sqr` will generally be faster.

`mp_limb_t mpn_mul_1` (*mp_limb_t* *`rp`, *const mp_limb_t* *`s1p`, *mp_size_t* `n`, *mp_limb_t* `s2limb`) [Function]

Multiply {*s1p*, *n*} by *s2limb*, and write the *n* least significant limbs of the product to *rp*. Return the most significant limb of the product. {*s1p*, *n*} and {*rp*, *n*} are allowed to overlap provided $rp \leq s1p$.

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

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* `n`, *mp_limb_t* `s2limb`) [Function]

Multiply {*s1p*, *n*} and *s2limb*, and add the *n* least significant limbs of the product to {*rp*, *n*} 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 MPIR. It is written in assembly for most CPUs.

`mp_limb_t mpn_submul_1` (*mp_limb_t *rp, const mp_limb_t *s1p, mp_size_t n, mp_limb_t s2limb*) [Function]

Multiply {*s1p*, *n*} and *s2limb*, and subtract the *n* least significant limbs of the product from {*rp*, *n*} 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 MPIR. It is written in assembly for most CPUs.

`mp_limb_t mpn_mul` (*mp_limb_t *rp, const mp_limb_t *s1p, mp_size_t s1n, const mp_limb_t *s2p, mp_size_t s2n*) [Function]

Multiply {*s1p*, *s1n*} and {*s2p*, *s2n*}, and write the result to *rp*. Return the most significant limb of the result.

The destination has to have space for *s1n* + *s2n* limbs, even if the result might be one limb smaller.

This function requires that *s1n* is greater than or equal to *s2n*. The destination must be distinct from both input operands.

`void mpn_sqr` (*mp_limb_t *rp, const mp_limb_t *s1p, mp_size_t n*) [Function]

Compute the square of {*s1p*, *n*} and write the 2**n*-limb result to *rp*.

The destination has to have space for 2**n* limbs, even if the result's most significant limb is zero. No overlap is permitted between the destination and the source.

`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*} and put the quotient at {*qp*, *nn*−*dn*+1} and the remainder at {*rp*, *dn*}. The quotient is rounded towards 0.

No overlap is permitted between arguments. *nn* must be greater than or equal to *dn*. The most significant limb of *dp* must be non-zero. The *qxn* operand must be zero.

`mp_limb_t mpn_divrem` (*mp_limb_t *r1p, mp_size_t qxn, mp_limb_t *rs2p, mp_size_t rs2n, const mp_limb_t *s3p, mp_size_t s3n*) [Function]

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

Divide {*rs2p*, *rs2n*} by {*s3p*, *s3n*}, 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 *s3n* limbs long (i.e., as many limbs as the divisor).

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

It is required that *rs2n* is greater than or equal to *s3n*. It is required that the most significant bit of the divisor is set.

If the quotient is not needed, pass *rs2p* + *s3n* 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 *rs2n* − *s3n* + *qxn* limbs large.

`mp_limb_t mpn_divrem_1` (*mp_limb_t* ***r1p**, *mp_size_t* **qxn**, *mp_limb_t* ***s2p**, *mp_size_t* **s2n**, *mp_limb_t* **s3limb**) [Function]

`mp_limb_t mpn_divmod_1` (*mp_limb_t* ***r1p**, *mp_limb_t* ***s2p**, *mp_size_t* **s2n**, *mp_limb_t* **s3limb**) [Macro]

Divide {*s2p*, *s2n*} by *s3limb*, and write the quotient at *r1p*. Return the remainder.

The integer quotient is written to {*r1p*+*qxn*, *s2n*} and in addition *qxn* fraction limbs are developed and written to {*r1p*, *qxn*}. Either or both *s2n* and *qxn* can be zero. For most usages, *qxn* will be zero.

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

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

`mp_limb_t mpn_divexact_by3` (*mp_limb_t* ***rp**, *mp_limb_t* ***sp**, *mp_size_t* **n**) [Macro]

`mp_limb_t mpn_divexact_by3c` (*mp_limb_t* ***rp**, *mp_limb_t* ***sp**, *mp_size_t* **n**, *mp_limb_t* **carry**) [Function]

Divide {*sp*, *n*} by 3, expecting it to divide exactly, and writing the result to {*rp*, *n*}. 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 from low to high. `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 $b = 2^{\texttt{GMP\_NUMB\_BITS}}$. The return c is always 0, 1 or 2, and the initial carry i 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 $(a - i) \bmod 3$ is given by $3 - c$, because $b \equiv 1 \bmod 3$ (when `mp_bits_per_limb` is even, which is always so currently).

`mp_limb_t mpn_mod_1` (*mp_limb_t* ***s1p**, *mp_size_t* **s1n**, *mp_limb_t* **s2limb**) [Function]

Divide {*s1p*, *s1n*} by *s2limb*, and return the remainder. *s1n* can be zero.

`mp_limb_t mpn_lshift` (*mp_limb_t* ***rp**, *const mp_limb_t* ***sp**, *mp_size_t* **n**, *unsigned int* **count**) [Function]

Shift {*sp*, *n*} left by *count* bits, and write the result to {*rp*, *n*}. The bits shifted out at the left are returned in the least significant *count* bits of the return value (the rest of the return value is zero).

count must be in the range 1 to `mp_bits_per_limb`−1. The regions {*sp*, *n*} and {*rp*, *n*} may overlap, provided $rp \geq sp$.

This function is written in assembly for most CPUs.

`mp_limb_t mpn_rshift` (*mp_limb_t *rp, const mp_limb_t *sp, mp_size_t n, unsigned int count*) [Function]

Shift {*sp*, *n*} right by *count* bits, and write the result to {*rp*, *n*}. The bits shifted out at the right are returned in the most significant *count* bits of the return value (the rest of the return value is zero).

count must be in the range 1 to `mp_bits_per_limb`−1. The regions {*sp*, *n*} and {*rp*, *n*} may overlap, provided $rp \leq sp$.

This function is written in assembly for most CPUs.

`int mpn_cmp` (*const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t n*) [Function]

Compare {*s1p*, *n*} and {*s2p*, *n*} and return a positive value if $s1 > s2$, 0 if they are equal, or a negative value if $s1 < s2$.

`mp_size_t mpn_gcd` (*mp_limb_t *rp, mp_limb_t *s1p, mp_size_t s1n, mp_limb_t *s2p, mp_size_t s2n*) [Function]

Set {*rp*, *retval*} to the greatest common divisor of {*s1p*, *s1n*} and {*s2p*, *s2n*}. The result can be up to *s2n* limbs, the return value is the actual number produced. Both source operands are destroyed.

{*s1p*, *s1n*} must have at least as many bits as {*s2p*, *s2n*}. {*s2p*, *s2n*} must be odd. Both operands must have non-zero most significant limbs. No overlap is permitted between {*s1p*, *s1n*} and {*s2p*, *s2n*}.

`mp_limb_t mpn_gcd_1` (*const mp_limb_t *s1p, mp_size_t s1n, mp_limb_t s2limb*) [Function]

Return the greatest common divisor of {*s1p*, *s1n*} and *s2limb*. Both operands must be non-zero.

`mp_size_t mpn_gcdext` (*mp_limb_t *gp, mp_limb_t *sp, mp_size_t *sn, mp_limb_t *xp, mp_size_t xn, mp_limb_t *yp, mp_size_t yn*) [Function]

Let U be defined by {*xp*, *xn*} and let V be defined by {*yp*, *yn*}.

Compute the greatest common divisor G of U and V. Compute a cofactor S such that $G = US + VT$. The second cofactor T is not computed but can easily be obtained from $(G - US)/V$ (the division will be exact). It is required that $U \geq V > 0$.

S satisfies $S = 1$ or $|S| < V/(2G)$. $S = 0$ if and only if V divides U (i.e., $G = V$).

Store G at *gp* and let the return value define its limb count. Store S at *sp* and let `|*sn|` define its limb count. S can be negative; when this happens *sn will be negative. The areas at *gp* and *sp* should each have room for $xn + 1$ limbs.

The areas {*xp*, $xn+1$} and {*yp*, $yn+1$} are destroyed (i.e. the input operands plus an extra limb past the end of each).

Compatibility note: MPIR versions 1.3,2.0 and GMP versions 4.3.0,4.3.1 defined S less strictly. Earlier as well as later GMP releases define S as described here.

`mp_size_t mpn_sqrtrem` (*mp_limb_t *r1p, mp_limb_t *r2p, const mp_limb_t *sp, mp_size_t n*) [Function]

Compute the square root of {*sp*, *n*} and put the result at {*r1p*, $\lceil n/2 \rceil$} and the remainder at {*r2p*, *retval*}. *r2p* needs space for *n* limbs, but the return value indicates how many are produced.

The most significant limb of {*sp*, *n*} must be non-zero. The areas {*r1p*, $\lceil n/2 \rceil$} and {*sp*, *n*} must be completely separate. The areas {*r2p*, *n*} and {*sp*, *n*} must be either identical or completely separate.

If the remainder is not wanted then *r2p* can be `NULL`, and in this case the return value is zero or non-zero according to whether the remainder would have been zero or non-zero.

A return value of zero indicates a perfect square. See also `mpz_perfect_square_p`.

`mp_size_t mpn_get_str` (*unsigned char* ***str***, *int* **base**, *mp_limb_t* ***s1p***, *mp_size_t* ***s1n***) [Function]

Convert {*s1p*, *s1n*} to a raw unsigned char array at *str* in base *base*, and return the number of characters produced. There may be leading zeros in the string. 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. *base* can vary from 2 to 256.

The most significant limb of the input {*s1p*, *s1n*} must be non-zero. The input {*s1p*, *s1n*} is clobbered, except when *base* is a power of 2, in which case it's unchanged.

The area at *str* has to have space for the largest possible number represented by a *s1n* long limb array, plus one extra character.

`mp_size_t mpn_set_str` (*mp_limb_t* ***rp***, *const unsigned char* ***str***, *size_t* **strsize**, *int* **base**) [Function]

Convert bytes {*str*,*strsize*} in the given *base* to limbs at *rp*.

str[0] is the most significant byte and *str*[*strsize* − 1] is the least significant. Each byte should be a value in the range 0 to *base* − 1, not an ASCII character. *base* can vary from 2 to 256.

The return value is the number of limbs written to *rp*. If the most significant input byte is non-zero then the high limb at *rp* will be non-zero, and only that exact number of limbs will be required there.

If the most significant input byte is zero then there may be high zero limbs written to *rp* and included in the return value.

strsize must be at least 1, and no overlap is permitted between {*str*,*strsize*} and the result at *rp*.

`mp_bitcnt_t mpn_scan0` (*const mp_limb_t* ***s1p***, *imp_bitcnt_t* **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.

`mp_bitcnt_t mpn_scan1` (*const mp_limb_t* ***s1p***, *mp_bitcnt_t* **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* ***r1n***) [Function]
`void mpn_random2` (*mp_limb_t* ***r1p***, *mp_size_t* ***r1n***) [Function]

Generate a random number of length *r1n* 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.

These functions are obsolete. They will disappear from future MPIR releases.

`void mpn_urandomb` (*mp_limb_t* ***rp**, *gmp_randstate_t* **state**, *mpir_ui* **n**) [Function]
Generate a uniform random number of length *n* bits and store it at *rp*.

This function interface is preliminary and may change in the future.

`void mpn_urandomm` (*mp_limb_t* ***rp**, *gmp_randstate_t* **state**, *const mp_limb_t* ***mp**, *mp_size_t* **n**) [Function]
Generate a uniform random number modulo (*mp*,*n*) of length *n* limbs and store it at *rp*.

This function interface is preliminary and may change in the future.

`void mpn_randomb` (*mp_limb_t* ***rp**, *gmp_randstate_t* **state**, *mp_size_t* **n**) [Function]
Generate a random number of length *n* limbs and store it at *rp*. The most significant limb is always non-zero.

This function interface is preliminary and may change in the future.

`void mpn_rrandom` (*mp_limb_t* ***rp**, *gmp_randstate_t* **state**, *mp_size_t* **n**) [Function]
Generate a random number of length *n* limbs and store it at *rp*. The most significant limb is always non-zero. Generates long strings of zeros and ones in the binary representation and is intended for testing the correctness of the `mpn` routines.

This function interface is preliminary and may change in the future.

`mp_bitcnt_t mpn_popcount` (*const mp_limb_t* ***s1p**, *mp_size_t* **n**) [Function]
Count the number of set bits in {*s1p*, *n*}.

`mp_bitcnt_t mpn_hamdist` (*const mp_limb_t* ***s1p**, *const mp_limb_t* ***s2p**, *mp_size_t* **n**) [Function]
Compute the hamming distance between {*s1p*, *n*} and {*s2p*, *n*}, which is the number of bit positions where the two operands have different bit values.

`int mpn_perfect_square_p` (*const mp_limb_t* ***s1p**, *mp_size_t* **n**) [Function]
Return non-zero iff {*s1p*, *n*} is a perfect square.

`void mpn_and_n` (*mp_limb_t* ***rp**, *const mp_limb_t* ***s1p**, *const mp_limb_t* ***s2p**, *mp_size_t* **n**) [Function]
Perform the bitwise logical and of {*s1p*, *n*} and {*s2p*, *n*}, and write the result to {*rp*, *n*}.

`void mpn_ior_n` (*mp_limb_t* ***rp**, *const mp_limb_t* ***s1p**, *const mp_limb_t* ***s2p**, *mp_size_t* **n**) [Function]
Perform the bitwise logical inclusive or of {*s1p*, *n*} and {*s2p*, *n*}, and write the result to {*rp*, *n*}.

`void mpn_xor_n` (*mp_limb_t* ***rp**, *const mp_limb_t* ***s1p**, *const mp_limb_t* ***s2p**, *mp_size_t* **n**) [Function]
Perform the bitwise logical exclusive or of {*s1p*, *n*} and {*s2p*, *n*}, and write the result to {*rp*, *n*}.

`void mpn_andn_n` (*mp_limb_t *rp, const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t n*) [Function]
Perform the bitwise logical and of {*s1p*, *n*} and the bitwise complement of {*s2p*, *n*}, and write the result to {*rp*, *n*}.

`void mpn_iorn_n` (*mp_limb_t *rp, const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t n*) [Function]
Perform the bitwise logical inclusive or of {*s1p*, *n*} and the bitwise complement of {*s2p*, *n*}, and write the result to {*rp*, *n*}.

`void mpn_nand_n` (*mp_limb_t *rp, const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t n*) [Function]
Perform the bitwise logical and of {*s1p*, *n*} and {*s2p*, *n*}, and write the bitwise complement of the result to {*rp*, *n*}.

`void mpn_nior_n` (*mp_limb_t *rp, const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t n*) [Function]
Perform the bitwise logical inclusive or of {*s1p*, *n*} and {*s2p*, *n*}, and write the bitwise complement of the result to {*rp*, *n*}.

`void mpn_xnor_n` (*mp_limb_t *rp, const mp_limb_t *s1p, const mp_limb_t *s2p, mp_size_t n*) [Function]
Perform the bitwise logical exclusive or of {*s1p*, *n*} and {*s2p*, *n*}, and write the bitwise complement of the result to {*rp*, *n*}.

`void mpn_com` (*mp_limb_t *rp, const mp_limb_t *sp, mp_size_t n*) [Function]
Perform the bitwise complement of {*sp*, *n*}, and write the result to {*rp*, *n*}.

`void mpn_copyi` (*mp_limb_t *rp, const mp_limb_t *s1p, mp_size_t n*) [Function]
Copy from {*s1p*, *n*} to {*rp*, *n*}, increasingly.

`void mpn_copyd` (*mp_limb_t *rp, const mp_limb_t *s1p, mp_size_t n*) [Function]
Copy from {*s1p*, *n*} to {*rp*, *n*}, decreasingly.

`void mpn_zero` (*mp_limb_t *rp, mp_size_t n*) [Function]
Zero {*rp*, *n*}.

8.1 Nails

Everything in this section is highly experimental and may disappear or be subject to incompatible changes in a future version of MPIR.

N.B: Nails are currently disabled and not supported in MPIR. They may or may not return in a future version of MPIR.

Nails are an experimental feature whereby a few bits are left unused at the top of each `mp_limb_t`. This can significantly improve carry handling on some processors.

All the `mpn` functions accepting limb data will expect the nail bits to be zero on entry, and will return data with the nails similarly all zero. This applies both to limb vectors and to single limb arguments.

Nails can be enabled by configuring with '`--enable-nails`'. By default the number of bits will be chosen according to what suits the host processor, but a particular number can be selected with '`--enable-nails=N`'.

At the mpn level, a nail build is neither source nor binary compatible with a non-nail build, strictly speaking. But programs acting on limbs only through the mpn functions are likely to work equally well with either build, and judicious use of the definitions below should make any program compatible with either build, at the source level.

For the higher level routines, meaning `mpz` etc, a nail build should be fully source and binary compatible with a non-nail build.

`GMP_NAIL_BITS` [Macro]
`GMP_NUMB_BITS` [Macro]
`GMP_LIMB_BITS` [Macro]

`GMP_NAIL_BITS` is the number of nail bits, or 0 when nails are not in use. `GMP_NUMB_BITS` is the number of data bits in a limb. `GMP_LIMB_BITS` is the total number of bits in an `mp_limb_t`. In all cases

```
GMP_LIMB_BITS == GMP_NAIL_BITS + GMP_NUMB_BITS
```

`GMP_NAIL_MASK` [Macro]
`GMP_NUMB_MASK` [Macro]

Bit masks for the nail and number parts of a limb. `GMP_NAIL_MASK` is 0 when nails are not in use.

`GMP_NAIL_MASK` is not often needed, since the nail part can be obtained with `x >> GMP_NUMB_BITS`, and that means one less large constant, which can help various RISC chips.

`GMP_NUMB_MAX` [Macro]

The maximum value that can be stored in the number part of a limb. This is the same as `GMP_NUMB_MASK`, but can be used for clarity when doing comparisons rather than bit-wise operations.

The term "nails" comes from finger or toe nails, which are at the ends of a limb (arm or leg). "numb" is short for number, but is also how the developers felt after trying for a long time to come up with sensible names for these things.

In the future (the distant future most likely) a non-zero nail might be permitted, giving non-unique representations for numbers in a limb vector. This would help vector processors since carries would only ever need to propagate one or two limbs.

9 Random Number Functions

Sequences of pseudo-random numbers in MPIR are generated using a variable of type `gmp_randstate_t`, which holds an algorithm selection and a current state. Such a variable must be initialized by a call to one of the `gmp_randinit` functions, and can be seeded with one of the `gmp_randseed` functions.

The functions actually generating random numbers are described in Section 5.13 [Integer Random Numbers], page 41, and Section 7.8 [Miscellaneous Float Functions], page 56.

The older style random number functions don't accept a `gmp_randstate_t` parameter but instead share a global variable of that type. They use a default algorithm and are currently not seeded (though perhaps that will change in the future). The new functions accepting a `gmp_randstate_t` are recommended for applications that care about randomness.

9.1 Random State Initialization

`void gmp_randinit_default (`*gmp_randstate_t* `state``)` [Function]
Initialize *state* with a default algorithm. This will be a compromise between speed and randomness, and is recommended for applications with no special requirements. Currently this is `gmp_randinit_mt`.

`void gmp_randinit_mt (`*gmp_randstate_t* `state``)` [Function]
Initialize *state* for a Mersenne Twister algorithm. This algorithm is fast and has good randomness properties.

`void gmp_randinit_lc_2exp (`*gmp_randstate_t* `state`, *mpz_t* `a`, *mpir_ui* `c`, *mp_bitcnt_t* `m2exp``)` [Function]
Initialize *state* with a linear congruential algorithm $X = (aX + c) \bmod 2^{m2exp}$.

The low bits of X in this algorithm are not very random. The least significant bit will have a period no more than 2, and the second bit no more than 4, etc. For this reason only the high half of each X is actually used.

When a random number of more than *m2exp*/2 bits is to be generated, multiple iterations of the recurrence are used and the results concatenated.

`int gmp_randinit_lc_2exp_size (`*gmp_randstate_t* `state`, *mp_bitcnt_t* `size``)` [Function]
Initialize *state* for a linear congruential algorithm as per `gmp_randinit_lc_2exp`. *a*, *c* and *m2exp* are selected from a table, chosen so that *size* bits (or more) of each X will be used, ie. $m2exp/2 \geq size$.

If successful the return value is non-zero. If *size* is bigger than the table data provides then the return value is zero. The maximum *size* currently supported is 128.

`int gmp_randinit_set (`*gmp_randstate_t* `rop`, *gmp_randstate_t* `op``)` [Function]
Initialize *rop* with a copy of the algorithm and state from *op*.

`void gmp_randclear (`*gmp_randstate_t* `state``)` [Function]
Free all memory occupied by *state*.

9.2 Random State Seeding

`void gmp_randseed` (*gmp_randstate_t* `state`, *mpz_t* `seed`) [Function]
`void gmp_randseed_ui` (*gmp_randstate_t* `state`, *mpir_ui* `seed`) [Function]
Set an initial seed value into *state*.

The size of a seed determines how many different sequences of random numbers that it's possible to generate. The "quality" of the seed is the randomness of a given seed compared to the previous seed used, and this affects the randomness of separate number sequences. The method for choosing a seed is critical if the generated numbers are to be used for important applications, such as generating cryptographic keys.

Traditionally the system time has been used to seed, but care needs to be taken with this. If an application seeds often and the resolution of the system clock is low, then the same sequence of numbers might be repeated. Also, the system time is quite easy to guess, so if unpredictability is required then it should definitely not be the only source for the seed value. On some systems there's a special device `/dev/random` which provides random data better suited for use as a seed.

9.3 Random State Miscellaneous

`mpir_ui gmp_urandomb_ui` (*gmp_randstate_t* `state`, *mpir_ui* `n`) [Function]
Return a uniformly distributed random number of n bits, ie. in the range 0 to $2^n - 1$ inclusive. n must be less than or equal to the number of bits in an `mpir_ui`.

`mpir_ui gmp_urandomm_ui` (*gmp_randstate_t* `state`, *mpir_ui* `n`) [Function]
Return a uniformly distributed random number in the range 0 to $n - 1$, inclusive.

10 Formatted Output

10.1 Format Strings

`gmp_printf` and friends accept format strings similar to the standard C `printf` (see Section "Formatted Output" in *The GNU C Library Reference Manual*). A format specification is of the form

```
% [flags] [width] [.[precision]] [type] conv
```

MPIR adds types '`Z`', '`Q`' and '`F`' for `mpz_t`, `mpq_t` and `mpf_t` respectively, '`M`' for `mp_limb_t`, and '`N`' for an `mp_limb_t` array. '`Z`', '`Q`', '`M`' and '`N`' behave like integers. '`Q`' will print a '`/`' and a denominator, if needed. '`F`' behaves like a float. For example,

```
mpz_t z;
gmp_printf ("%s is an mpz %Zd\n", "here", z);

mpq_t q;
gmp_printf ("a hex rational: %#40Qx\n", q);

mpf_t f;
int   n;
gmp_printf ("fixed point mpf %.*Ff with %d digits\n", n, f, n);

mp_limb_t l;
gmp_printf ("limb %Mu\n", limb);

const mp_limb_t *ptr;
mp_size_t       size;
gmp_printf ("limb array %Nx\n", ptr, size);
```

For '`N`' the limbs are expected least significant first, as per the `mpn` functions (see Chapter 8 [Low-level Functions], page 58). A negative size can be given to print the value as a negative.

All the standard C `printf` types behave the same as the C library `printf`, and can be freely intermixed with the MPIR extensions. In the current implementation the standard parts of the format string are simply handed to `printf` and only the MPIR extensions handled directly.

The flags accepted are as follows. GLIBC style '`'`' is only for the standard C types (not the MPIR types), and only if the C library supports it.

`0`	pad with zeros (rather than spaces)
`#`	show the base with '`0x`', '`0X`' or '`0`'
`+`	always show a sign
(space)	show a space or a '`-`' sign
`'`	group digits, GLIBC style (not MPIR types)

The optional width and precision can be given as a number within the format string, or as a '`*`' to take an extra parameter of type `int`, the same as the standard `printf`.

The standard types accepted are as follows. '`h`' and '`l`' are portable, the rest will depend on the compiler (or include files) for the type and the C library for the output.

`h`	`short`
`hh`	`char`

j	intmax_t or uintmax_t
l	long or wchar_t
ll	long long
L	long double
q	quad_t or u_quad_t
t	ptrdiff_t
z	size_t

The MPIR types are

F	mpf_t, float conversions
Q	mpq_t, integer conversions
M	mp_limb_t, integer conversions
N	mp_limb_t array, integer conversions
Z	mpz_t, integer conversions

The conversions accepted are as follows. 'a' and 'A' are always supported for mpf_t but depend on the C library for standard C float types. 'm' and 'p' depend on the C library.

a A	hex floats, C99 style
c	character
d	decimal integer
e E	scientific format float
f	fixed point float
i	same as d
g G	fixed or scientific float
m	strerror string, GLIBC style
n	store characters written so far
o	octal integer
p	pointer
s	string
u	unsigned integer
x X	hex integer

'o', 'x' and 'X' are unsigned for the standard C types, but for types 'Z', 'Q' and 'N' they are signed. 'u' is not meaningful for 'Z', 'Q' and 'N'.

'M' is a proxy for the C library 'l' or 'L', according to the size of mp_limb_t. Unsigned conversions will be usual, but a signed conversion can be used and will interpret the value as a twos complement negative.

'n' can be used with any type, even the MPIR types.

Other types or conversions that might be accepted by the C library printf cannot be used through gmp_printf, this includes for instance extensions registered with GLIBC register_printf_function. Also currently there's no support for POSIX '$' style numbered arguments (perhaps this will be added in the future).

The precision field has it's usual meaning for integer 'Z' and float 'F' types, but is currently undefined for 'Q' and should not be used with that.

mpf_t conversions only ever generate as many digits as can be accurately represented by the operand, the same as mpf_get_str does. Zeros will be used if necessary to pad to the requested precision. This happens even for an 'f' conversion of an mpf_t which is an integer, for instance

2^{1024} in an `mpf_t` of 128 bits precision will only produce about 40 digits, then pad with zeros to the decimal point. An empty precision field like '`%.Fe`' or '`%.Ff`' can be used to specifically request just the significant digits.

The decimal point character (or string) is taken from the current locale settings on systems which provide `localeconv` (see Section "Locales and Internationalization" in *The GNU C Library Reference Manual*). The C library will normally do the same for standard float output.

The format string is only interpreted as plain `char`s, multibyte characters are not recognised. Perhaps this will change in the future.

10.2 Functions

Each of the following functions is similar to the corresponding C library function. The basic `printf` forms take a variable argument list. The `vprintf` forms take an argument pointer, see Section "Variadic Functions" in *The GNU C Library Reference Manual*, or '`man 3 va_start`'.

It should be emphasised that if a format string is invalid, or the arguments don't match what the format specifies, then the behaviour of any of these functions will be unpredictable. GCC format string checking is not available, since it doesn't recognise the MPIR extensions.

The file based functions `gmp_printf` and `gmp_fprintf` will return −1 to indicate a write error. Output is not "atomic", so partial output may be produced if a write error occurs. All the functions can return −1 if the C library `printf` variant in use returns −1, but this shouldn't normally occur.

`int gmp_printf (`*const char *`**fmt**, ...*`)` [Function]
`int gmp_vprintf (`*const char *`**fmt**, va_list `**ap**`)` [Function]
Print to the standard output `stdout`. Return the number of characters written, or −1 if an error occurred.

`int gmp_fprintf (`*FILE *`**fp**, const char \*`**fmt**, ...*`)` [Function]
`int gmp_vfprintf (`*FILE *`**fp**, const char \*`**fmt**, va_list `**ap**`)` [Function]
Print to the stream *fp*. Return the number of characters written, or −1 if an error occurred.

`int gmp_sprintf (`*char *`**buf**, const char \*`**fmt**, ...*`)` [Function]
`int gmp_vsprintf (`*char *`**buf**, const char \*`**fmt**, va_list `**ap**`)` [Function]
Form a null-terminated string in *buf*. Return the number of characters written, excluding the terminating null.

No overlap is permitted between the space at *buf* and the string *fmt*.

These functions are not recommended, since there's no protection against exceeding the space available at *buf*.

`int gmp_snprintf (`*char *`**buf**, size_t `**size**, const char *`**fmt**, ...*`)` [Function]
`int gmp_vsnprintf (`*char *`**buf**, size_t `**size**, const char *`**fmt**, va_list `**ap**`)` [Function]
Form a null-terminated string in *buf*. No more than *size* bytes will be written. To get the full output, *size* must be enough for the string and null-terminator.

The return value is the total number of characters which ought to have been produced, excluding the terminating null. If $retval \geq size$ then the actual output has been truncated to the first $size - 1$ characters, and a null appended.

No overlap is permitted between the region {*buf*,*size*} and the *fmt* string.

Notice the return value is in ISO C99 `snprintf` style. This is so even if the C library `vsnprintf` is the older GLIBC 2.0.x style.

`int gmp_asprintf` (*char **pp, const char *fmt, ...*) [Function]
`int gmp_vasprintf` (*char **pp, const char *fmt, va_list ap*) [Function]
Form a null-terminated string in a block of memory obtained from the current memory allocation function (see Chapter 14 [Custom Allocation], page 106). The block will be the size of the string and null-terminator. The address of the block in stored to **pp*. The return value is the number of characters produced, excluding the null-terminator.

Unlike the C library `asprintf`, `gmp_asprintf` doesn't return −1 if there's no more memory available, it lets the current allocation function handle that.

`int gmp_obstack_printf` (*struct obstack *ob, const char *fmt, ...*) [Function]
`int gmp_obstack_vprintf` (*struct obstack *ob, const char *fmt, va_list ap*) [Function]
Append to the current object in *ob*. The return value is the number of characters written. A null-terminator is not written.

fmt cannot be within the current object in *ob*, since that object might move as it grows.

These functions are available only when the C library provides the obstack feature, which probably means only on GNU systems, see Section "Obstacks" in *The GNU C Library Reference Manual*.

10.3 C++ Formatted Output

The following functions are provided in `libmpirxx` (see Section 3.1 [Headers and Libraries], page 16), which is built if C++ support is enabled (see Section 2.1 [Build Options], page 3). Prototypes are available from `<mpir.h>`.

`ostream& operator<<` (*ostream& stream, mpz_t op*) [Function]
Print *op* to *stream*, using its `ios` formatting settings. `ios::width` is reset to 0 after output, the same as the standard `ostream operator<<` routines do.

In hex or octal, *op* is printed as a signed number, the same as for decimal. This is unlike the standard `operator<<` routines on `int` etc, which instead give twos complement.

`ostream& operator<<` (*ostream& stream, mpq_t op*) [Function]
Print *op* to *stream*, using its `ios` formatting settings. `ios::width` is reset to 0 after output, the same as the standard `ostream operator<<` routines do.

Output will be a fraction like '`5/9`', or if the denominator is 1 then just a plain integer like '`123`'.

In hex or octal, *op* is printed as a signed value, the same as for decimal. If `ios::showbase` is set then a base indicator is shown on both the numerator and denominator (if the denominator is required).

`ostream& operator<<` (*ostream& stream, mpf_t op*) [Function]
Print *op* to *stream*, using its `ios` formatting settings. `ios::width` is reset to 0 after output, the same as the standard `ostream operator<<` routines do.

The decimal point follows the standard library float `operator<<`, which on recent systems means the `std::locale` imbued on *stream*.

Hex and octal are supported, unlike the standard `operator<<` on `double`. The mantissa will be in hex or octal, the exponent will be in decimal. For hex the exponent delimiter is an '`@`'. This is as per `mpf_out_str`.

`ios::showbase` is supported, and will put a base on the mantissa, for example hex '`0x1.8`' or '`0x0.8`', or octal '`01.4`' or '`00.4`'. This last form is slightly strange, but at least differentiates itself from decimal.

These operators mean that MPIR types can be printed in the usual C++ way, for example,

```
mpz_t  z;
int    n;
...
cout << "iteration " << n << " value " << z << "\n";
```

But note that `ostream` output (and `istream` input, see Section 11.3 [C++ Formatted Input], page 76) is the only overloading available for the MPIR types and that for instance using `+` with an `mpz_t` will have unpredictable results. For classes with overloading, see Chapter 12 [C++ Class Interface], page 78.

11 Formatted Input

11.1 Formatted Input Strings

`gmp_scanf` and friends accept format strings similar to the standard C `scanf` (see Section "Formatted Input" in *The GNU C Library Reference Manual*). A format specification is of the form

```
% [flags] [width] [type] conv
```

MPIR adds types '`Z`', '`Q`' and '`F`' for `mpz_t`, `mpq_t` and `mpf_t` respectively. '`Z`' and '`Q`' behave like integers. '`Q`' will read a '`/`' and a denominator, if present. '`F`' behaves like a float.

MPIR variables don't require an `&` when passed to `gmp_scanf`, since they're already "call-by-reference". For example,

```
/* to read say "a(5) = 1234" */
int   n;
mpz_t z;
gmp_scanf ("a(%d) = %Zd\n", &n, z);

mpq_t q1, q2;
gmp_sscanf ("0377 + 0x10/0x11", "%Qi + %Qi", q1, q2);

/* to read say "topleft (1.55,-2.66)" */
mpf_t x, y;
char  buf[32];
gmp_scanf ("%31s (%Ff,%Ff)", buf, x, y);
```

All the standard C `scanf` types behave the same as in the C library `scanf`, and can be freely intermixed with the MPIR extensions. In the current implementation the standard parts of the format string are simply handed to `scanf` and only the MPIR extensions handled directly.

The flags accepted are as follows. '`a`' and '`'`' will depend on support from the C library, and '`'`' cannot be used with MPIR types.

`*`	read but don't store
`a`	allocate a buffer (string conversions)
`'`	grouped digits, GLIBC style (not MPIR types)

The standard types accepted are as follows. '`h`' and '`l`' are portable, the rest will depend on the compiler (or include files) for the type and the C library for the input.

`h`	`short`
`hh`	`char`
`j`	`intmax_t` or `uintmax_t`
`l`	`long int`, `double` or `wchar_t`
`ll`	`long long`
`L`	`long double`
`q`	`quad_t` or `u_quad_t`
`t`	`ptrdiff_t`
`z`	`size_t`

The MPIR types are

`F`	`mpf_t`, float conversions
`Q`	`mpq_t`, integer conversions
`Z`	`mpz_t`, integer conversions

The conversions accepted are as follows. '`p`' and '`[`' will depend on support from the C library, the rest are standard.

`c`	character or characters
`d`	decimal integer
`e E f g G`	float
`i`	integer with base indicator
`n`	characters read so far
`o`	octal integer
`p`	pointer
`s`	string of non-whitespace characters
`u`	decimal integer
`x X`	hex integer
`[`	string of characters in a set

'`e`', '`E`', '`f`', '`g`' and '`G`' are identical, they all read either fixed point or scientific format, and either upper or lower case '`e`' for the exponent in scientific format.

C99 style hex float format (`printf %a`, see Section 10.1 [Formatted Output Strings], page 69) is always accepted for `mpf_t`, but for the standard float types it will depend on the C library.

'`x`' and '`X`' are identical, both accept both upper and lower case hexadecimal.

'`o`', '`u`', '`x`' and '`X`' all read positive or negative values. For the standard C types these are described as "unsigned" conversions, but that merely affects certain overflow handling, negatives are still allowed (per `strtoul`, see Section "Parsing of Integers" in *The GNU C Library Reference Manual*). For MPIR types there are no overflows, so '`d`' and '`u`' are identical.

'`Q`' type reads the numerator and (optional) denominator as given. If the value might not be in canonical form then `mpq_canonicalize` must be called before using it in any calculations (see Chapter 6 [Rational Number Functions], page 46).

'`Qi`' will read a base specification separately for the numerator and denominator. For example '`0x10/11`' would be 16/11, whereas '`0x10/0x11`' would be 16/17.

'`n`' can be used with any of the types above, even the MPIR types. '`*`' to suppress assignment is allowed, though in that case it would do nothing at all.

Other conversions or types that might be accepted by the C library `scanf` cannot be used through `gmp_scanf`.

Whitespace is read and discarded before a field, except for '`c`' and '`[`' conversions.

For float conversions, the decimal point character (or string) expected is taken from the current locale settings on systems which provide `localeconv` (see Section "Locales and International-ization" in *The GNU C Library Reference Manual*). The C library will normally do the same for standard float input.

The format string is only interpreted as plain `char`s, multibyte characters are not recognised. Perhaps this will change in the future.

11.2 Formatted Input Functions

Each of the following functions is similar to the corresponding C library function. The plain `scanf` forms take a variable argument list. The `vscanf` forms take an argument pointer, see Section "Variadic Functions" in *The GNU C Library Reference Manual*, or '`man 3 va_start`'.

It should be emphasised that if a format string is invalid, or the arguments don't match what the format specifies, then the behaviour of any of these functions will be unpredictable. GCC format string checking is not available, since it doesn't recognise the MPIR extensions.

No overlap is permitted between the *fmt* string and any of the results produced.

`int gmp_scanf` (*const char *`fmt`, ...*) [Function]
`int gmp_vscanf` (*const char *`fmt`, va_list `ap`*) [Function]
Read from the standard input `stdin`.

`int gmp_fscanf` (*FILE *`fp`, const char *`fmt`, ...*) [Function]
`int gmp_vfscanf` (*FILE *`fp`, const char *`fmt`, va_list `ap`*) [Function]
Read from the stream *fp*.

`int gmp_sscanf` (*const char *`s`, const char *`fmt`, ...*) [Function]
`int gmp_vsscanf` (*const char *`s`, const char *`fmt`, va_list `ap`*) [Function]
Read from a null-terminated string *s*.

The return value from each of these functions is the same as the standard C99 `scanf`, namely the number of fields successfully parsed and stored. '`%n`' fields and fields read but suppressed by '`*`' don't count towards the return value.

If end of input (or a file error) is reached before a character for a field or a literal, and if no previous non-suppressed fields have matched, then the return value is `EOF` instead of 0. A whitespace character in the format string is only an optional match and doesn't induce an `EOF` in this fashion. Leading whitespace read and discarded for a field don't count as characters for that field.

For the MPIR types, input parsing follows C99 rules, namely one character of lookahead is used and characters are read while they continue to meet the format requirements. If this doesn't provide a complete number then the function terminates, with that field not stored nor counted towards the return value. For instance with `mpf_t` an input '`1.23e-XYZ`' would be read up to the '`X`' and that character pushed back since it's not a digit. The string '`1.23e-`' would then be considered invalid since an '`e`' must be followed by at least one digit.

For the standard C types, in the current implementation MPIR calls the C library `scanf` functions, which might have looser rules about what constitutes a valid input.

Note that `gmp_sscanf` is the same as `gmp_fscanf` and only does one character of lookahead when parsing. Although clearly it could look at its entire input, it is deliberately made identical to `gmp_fscanf`, the same way C99 `sscanf` is the same as `fscanf`.

11.3 C++ Formatted Input

The following functions are provided in `libmpirxx` (see Section 3.1 [Headers and Libraries], page 16), which is built only if C++ support is enabled (see Section 2.1 [Build Options], page 3). Prototypes are available from `<mpir.h>`.

`istream& operator>>` (*istream& `stream`, mpz_t `rop`*) [Function]
Read *rop* from *stream*, using its `ios` formatting settings.

`istream& operator>>` (*istream& stream*, *mpq_t rop*) [Function]
An integer like '`123`' will be read, or a fraction like '`5/9`'. No whitespace is allowed around the '`/`'. If the fraction is not in canonical form then `mpq_canonicalize` must be called (see Chapter 6 [Rational Number Functions], page 46) before operating on it.

As per integer input, an '`0`' or '`0x`' base indicator is read when none of `ios::dec`, `ios::oct` or `ios::hex` are set. This is done separately for numerator and denominator, so that for instance '`0x10/11`' is 16/11 and '`0x10/0x11`' is 16/17.

`istream& operator>>` (*istream& stream*, *mpf_t rop*) [Function]
Read *rop* from *stream*, using its `ios` formatting settings.

Hex or octal floats are not supported, but might be in the future, or perhaps it's best to accept only what the standard float `operator>>` does.

Note that digit grouping specified by the `istream` locale is currently not accepted. Perhaps this will change in the future.

These operators mean that MPIR types can be read in the usual C++ way, for example,

```
mpz_t  z;
...
cin >> z;
```

But note that `istream` input (and `ostream` output, see Section 10.3 [C++ Formatted Output], page 72) is the only overloading available for the MPIR types and that for instance using `+` with an `mpz_t` will have unpredictable results. For classes with overloading, see Chapter 12 [C++ Class Interface], page 78.

12 C++ Class Interface

This chapter describes the C++ class based interface to MPIR.

All MPIR C language types and functions can be used in C++ programs, since `mpir.h` has `extern "C"` qualifiers, but the class interface offers overloaded functions and operators which may be more convenient.

Due to the implementation of this interface, a reasonably recent C++ compiler is required, one supporting namespaces, partial specialization of templates and member templates. For GCC this means version 2.91 or later.

Everything described in this chapter is to be considered preliminary and might be subject to incompatible changes if some unforeseen difficulty reveals itself.

12.1 C++ Interface General

All the C++ classes and functions are available with

```
#include <mpirxx.h>
```

Programs should be linked with the `libmpirxx` and `libmpir` libraries. For example,

```
g++ mycxxprog.cc -lmpirxx -lmpir
```

The classes defined are

`mpz_class` [Class]
`mpq_class` [Class]
`mpf_class` [Class]
The standard operators and various standard functions are overloaded to allow arithmetic with these classes. For example,

```
int
main (void)
{
  mpz_class a, b, c;

  a = 1234;
  b = "-5678";
  c = a+b;
  cout << "sum is " << c << "\n";
  cout << "absolute value is " << abs(c) << "\n";

  return 0;
}
```

An important feature of the implementation is that an expression like `a=b+c` results in a single call to the corresponding `mpz_add`, without using a temporary for the `b+c` part. Expressions which by their nature imply intermediate values, like `a=b*c+d*e`, still use temporaries though.

The classes can be freely intermixed in expressions, as can the classes and the standard types `mpir_si`, `mpir_ui` and `double`. Smaller types like `int` or `float` can also be intermixed, since C++ will promote them.

Note that `bool` is not accepted directly, but must be explicitly cast to an `int` first. This is because C++ will automatically convert any pointer to a `bool`, so if MPIR accepted `bool` it

would make all sorts of invalid class and pointer combinations compile but almost certainly not do anything sensible.

Conversions back from the classes to standard C++ types aren't done automatically, instead member functions like `get_si` are provided (see the following sections for details).

Also there are no automatic conversions from the classes to the corresponding MPIR C types, instead a reference to the underlying C object can be obtained with the following functions,

`mpz_t mpz_class::get_mpz_t ()` [Function]
`mpq_t mpq_class::get_mpq_t ()` [Function]
`mpf_t mpf_class::get_mpf_t ()` [Function]
These can be used to call a C function which doesn't have a C++ class interface. For example to set `a` to the GCD of `b` and `c`,

```
mpz_class a, b, c;
...
mpz_gcd (a.get_mpz_t(), b.get_mpz_t(), c.get_mpz_t());
```

In the other direction, a class can be initialized from the corresponding MPIR C type, or assigned to if an explicit constructor is used. In both cases this makes a copy of the value, it doesn't create any sort of association. For example,

```
mpz_t z;
// ... init and calculate z ...
mpz_class x(z);
mpz_class y;
y = mpz_class (z);
```

There are no namespace setups in `mpirxx.h`, all types and functions are simply put into the global namespace. This is what `mpir.h` has done in the past, and continues to do for compatibility. The extras provided by `mpirxx.h` follow MPIR naming conventions and are unlikely to clash with anything.

12.2 C++ Interface Integers

`void mpz_class::mpz_class (`*type* `n``)` [Function]
Construct an `mpz_class`. All the standard C++ types may be used `long double`, and all the MPIR C++ classes can be used. Any necessary conversion follows the corresponding C function, for example `double` follows `mpz_set_d` (see Section 5.2 [Assigning Integers], page 30).

`void mpz_class::mpz_class (`*mpz_t* `z``)` [Function]
Construct an `mpz_class` from an `mpz_t`. The value in *z* is copied into the new `mpz_class`, there won't be any permanent association between it and *z*.

`void mpz_class::mpz_class (`*const char* `*s``)` [Function]
`void mpz_class::mpz_class (`*const char* `*s`, *int* `base` = *0*`)` [Function]
`void mpz_class::mpz_class (`*const string&* `s``)` [Function]
`void mpz_class::mpz_class (`*const string&* `s`, *int* `base` = *0*`)` [Function]
Construct an `mpz_class` converted from a string using `mpz_set_str` (see Section 5.2 [Assigning Integers], page 30).

If the string is not a valid integer, an `std::invalid_argument` exception is thrown. The same applies to `operator=`.

`mpz_class operator"" _mpz` (*const char *str*) [Function]
With C++11 compilers, integers can be constructed with the syntax `123_mpz` which is equivalent to `mpz_class("123")`.

`mpz_class operator/` (*mpz_class a, mpz_class d*) [Function]
`mpz_class operator%` (*mpz_class a, mpz_class d*) [Function]
Divisions involving `mpz_class` round towards zero, as per the `mpz_tdiv_q` and `mpz_tdiv_r` functions (see Section 5.6 [Integer Division], page 33). This is the same as the C99 `/` and `%` operators.

The `mpz_fdiv...` or `mpz_cdiv...` functions can always be called directly if desired. For example,

```
mpz_class q, a, d;
...
mpz_fdiv_q (q.get_mpz_t(), a.get_mpz_t(), d.get_mpz_t());
```

`mpz_class abs` (*mpz_class op1*) [Function]
`int cmp` (*mpz_class op1, type op2*) [Function]
`int cmp` (*type op1, mpz_class op2*) [Function]
`bool mpz_class::fits_sint_p` (*void*) [Function]
`bool mpz_class::fits_slong_p` (*void*) [Function]
`bool mpz_class::fits_sshort_p` (*void*) [Function]
`bool mpz_class::fits_uint_p` (*void*) [Function]
`bool mpz_class::fits_ulong_p` (*void*) [Function]
`bool mpz_class::fits_ushort_p` (*void*) [Function]
`double mpz_class::get_d` (*void*) [Function]
`mpir_si mpz_class::get_si` (*void*) [Function]
`string mpz_class::get_str` (*int base = 10*) [Function]
`mpir_ui mpz_class::get_ui` (*void*) [Function]
`int mpz_class::set_str` (*const char *str, int base*) [Function]
`int mpz_class::set_str` (*const string& str, int base*) [Function]
`int sgn` (*mpz_class op*) [Function]
`mpz_class sqrt` (*mpz_class op*) [Function]
`void mpz_class::swap` (*mpz_class& op*) [Function]
`void swap` (*mpz_class& op1, mpz_class& op2*) [Function]
These functions provide a C++ class interface to the corresponding MPIR C routines.

`cmp` can be used with any of the classes or the standard C++ types, except `long double`.

Overloaded operators for combinations of `mpz_class` and `double` are provided for completeness, but it should be noted that if the given `double` is not an integer then the way any rounding is done is currently unspecified. The rounding might take place at the start, in the middle, or at the end of the operation, and it might change in the future.

Conversions between `mpz_class` and `double`, however, are defined to follow the corresponding C functions `mpz_get_d` and `mpz_set_d`. And comparisons are always made exactly, as per `mpz_cmp_d`.

12.3 C++ Interface Rationals

In all the following constructors, if a fraction is given then it should be in canonical form, or if not then `mpq_class::canonicalize` called.

`void mpq_class::mpq_class (`*type* `op``)` [Function]
`void mpq_class::mpq_class (`*integer* `num`, *integer* `den``)` [Function]
Construct an `mpq_class`. The initial value can be a single value of any type, or a pair of integers (`mpz_class` or standard C++ integer types) representing a fraction, except that `long double` is not supported. For example,

```
mpq_class q (99);
mpq_class q (1.75);
mpq_class q (1, 3);
```

`void mpq_class::mpq_class (`*mpq_t* `q``)` [Function]
Construct an `mpq_class` from an `mpq_t`. The value in q is copied into the new `mpq_class`, there won't be any permanent association between it and q.

`void mpq_class::mpq_class (`*const char* `*s``)` [Function]
`void mpq_class::mpq_class (`*const char* `*s`, *int* `base` = *0*`)` [Function]
`void mpq_class::mpq_class (`*const string&* `s``)` [Function]
`void mpq_class::mpq_class (`*const string&* `s`, *int* `base` = *0*`)` [Function]
Construct an `mpq_class` converted from a string using `mpq_set_str` (see Section 6.1 [Initializing Rationals], page 46).

If the string is not a valid rational, an `std::invalid_argument` exception is thrown. The same applies to `operator=`.

`mpq_class operator"" _mpq (`*const char* `*str``)` [Function]
With C++11 compilers, integral rationals can be constructed with the syntax `123_mpq` which is equivalent to `mpq_class(123_mpz)`. Other rationals can be built as `-1_mpq/2` or `0xb_mpq/123456_mpz`.

`void mpq_class::canonicalize ()` [Function]
Put an `mpq_class` into canonical form, as per Chapter 6 [Rational Number Functions], page 46. All arithmetic operators require their operands in canonical form, and will return results in canonical form.

`mpq_class abs (`*mpq_class* `op``)` [Function]
`int cmp (`*mpq_class* `op1`, *type* `op2``)` [Function]
`int cmp (`*type* `op1`, *mpq_class* `op2``)` [Function]
`double mpq_class::get_d (`*void*`)` [Function]
`string mpq_class::get_str (`*int* `base` = *10*`)` [Function]
`int mpq_class::set_str (`*const char* `*str`, *int* `base``)` [Function]
`int mpq_class::set_str (`*const string&* `str`, *int* `base``)` [Function]
`int sgn (`*mpq_class* `op``)` [Function]
`void mpq_class::swap (`*mpq_class&* `op``)` [Function]
`void swap (`*mpq_class&* `op1`, *mpq_class&* `op2``)` [Function]
These functions provide a C++ class interface to the corresponding MPIR C routines.

`cmp` can be used with any of the classes or the standard C++ types, except `long double`.

`mpz_class& mpq_class::get_num ()` [Function]
`mpz_class& mpq_class::get_den ()` [Function]
Get a reference to an `mpz_class` which is the numerator or denominator of an `mpq_class`. This can be used both for read and write access. If the object returned is modified, it modifies the original `mpq_class`.

If direct manipulation might produce a non-canonical value, then `mpq_class::canonicalize` must be called before further operations.

`mpz_t mpq_class::get_num_mpz_t ()` [Function]
`mpz_t mpq_class::get_den_mpz_t ()` [Function]
Get a reference to the underlying `mpz_t` numerator or denominator of an `mpq_class`. This can be passed to C functions expecting an `mpz_t`. Any modifications made to the `mpz_t` will modify the original `mpq_class`.

If direct manipulation might produce a non-canonical value, then `mpq_class::canonicalize` must be called before further operations.

`istream& operator>> (`*istream& stream, mpq_class& rop*`);` [Function]
Read *rop* from *stream*, using its `ios` formatting settings, the same as `mpq_t operator>>` (see Section 11.3 [C++ Formatted Input], page 76).

If the *rop* read might not be in canonical form then `mpq_class::canonicalize` must be called.

12.4 C++ Interface Floats

When an expression requires the use of temporary intermediate `mpf_class` values, like `f=g*h+x*y`, those temporaries will have the same precision as the destination `f`. Explicit constructors can be used if this doesn't suit.

`mpf_class::mpf_class (`*type op*`)` [Function]
`mpf_class::mpf_class (`*type op, mpir_ui prec*`)` [Function]
Construct an `mpf_class`. Any standard C++ type can be used, except `long double`, and any of the MPIR C++ classes can be used.

If *prec* is given, the initial precision is that value, in bits. If *prec* is not given, then the initial precision is determined by the type of *op* given. An `mpz_class`, `mpq_class`, or C++ builtin type will give the default `mpf` precision (see Section 7.1 [Initializing Floats], page 50). An `mpf_class` or expression will give the precision of that value. The precision of a binary expression is the higher of the two operands.

```
mpf_class f(1.5);         // default precision
mpf_class f(1.5, 500);    // 500 bits (at least)
mpf_class f(x);           // precision of x
mpf_class f(abs(x));      // precision of x
mpf_class f(-g, 1000);    // 1000 bits (at least)
mpf_class f(x+y);         // greater of precisions of x and y
```

`void mpf_class::mpf_class (`*const char *s*`)` [Function]
`void mpf_class::mpf_class (`*const char *s, mpir_ui prec, int base = 0*`)` [Function]
`void mpf_class::mpf_class (`*const string& s*`)` [Function]
`void mpf_class::mpf_class (`*const string& s, mpir_ui prec, int base = 0*`)` [Function]
Construct an `mpf_class` converted from a string using `mpf_set_str` (see Section 7.2 [Assigning Floats], page 52). If *prec* is given, the initial precision is that value, in bits. If not, the default `mpf` precision (see Section 7.1 [Initializing Floats], page 50) is used.

If the string is not a valid float, an `std::invalid_argument` exception is thrown. The same applies to `operator=`.

`mpf_class operator"" _mpf (const char *str)` [Function]
With C++11 compilers, floats can be constructed with the syntax `1.23e-1_mpf` which is equivalent to `mpf_class("1.23e-1")`.

`mpf_class& mpf_class::operator= (type op)` [Function]
Convert and store the given *op* value to an `mpf_class` object. The same types are accepted as for the constructors above.

Note that `operator=` only stores a new value, it doesn't copy or change the precision of the destination, instead the value is truncated if necessary. This is the same as `mpf_set` etc. Note in particular this means for `mpf_class` a copy constructor is not the same as a default constructor plus assignment.

```
mpf_class x (y);   // x created with precision of y

mpf_class x;       // x created with default precision
x = y;             // value truncated to that precision
```

Applications using templated code may need to be careful about the assumptions the code makes in this area, when working with `mpf_class` values of various different or non-default precisions. For instance implementations of the standard `complex` template have been seen in both styles above, though of course `complex` is normally only actually specified for use with the builtin float types.

`mpf_class abs (mpf_class op)` [Function]
`mpf_class ceil (mpf_class op)` [Function]
`int cmp (mpf_class op1, type op2)` [Function]
`int cmp (type op1, mpf_class op2)` [Function]
`bool mpf_class::fits_sint_p (void)` [Function]
`bool mpf_class::fits_slong_p (void)` [Function]
`bool mpf_class::fits_sshort_p (void)` [Function]
`bool mpf_class::fits_uint_p (void)` [Function]
`bool mpf_class::fits_ulong_p (void)` [Function]
`bool mpf_class::fits_ushort_p (void)` [Function]
`mpf_class floor (mpf_class op)` [Function]
`mpf_class hypot (mpf_class op1, mpf_class op2)` [Function]
`double mpf_class::get_d (void)` [Function]
`mpir_si mpf_class::get_si (void)` [Function]
`string mpf_class::get_str (mp_exp_t& exp, int base = 10, size_t digits = 0)` [Function]
`mpir_ui mpf_class::get_ui (void)` [Function]
`int mpf_class::set_str (const char *str, int base)` [Function]
`int mpf_class::set_str (const string& str, int base)` [Function]
`int sgn (mpf_class op)` [Function]
`mpf_class sqrt (mpf_class op)` [Function]
`void mpf_class::swap (mpf_class& op)` [Function]
`void swap (mpf_class& op1, mpf_class& op2)` [Function]
`mpf_class trunc (mpf_class op)` [Function]
These functions provide a C++ class interface to the corresponding MPIR C routines.

`cmp` can be used with any of the classes or the standard C++ types, except `long double`.

The accuracy provided by `hypot` is not currently guaranteed.

`mp_bitcnt_t mpf_class::get_prec ()` [Function]
`void mpf_class::set_prec (`*mp_bitcnt_t* `prec``)` [Function]
`void mpf_class::set_prec_raw (`*mp_bitcnt_t* `prec``)` [Function]
Get or set the current precision of an `mpf_class`.

The restrictions described for `mpf_set_prec_raw` (see Section 7.1 [Initializing Floats], page 50) apply to `mpf_class::set_prec_raw`. Note in particular that the `mpf_class` must be restored to it's allocated precision before being destroyed. This must be done by application code, there's no automatic mechanism for it.

12.5 C++ Interface Random Numbers

`gmp_randclass` [Class]
The C++ class interface to the MPIR random number functions uses `gmp_randclass` to hold an algorithm selection and current state, as per `gmp_randstate_t`.

`gmp_randclass::gmp_randclass (`*void* `(*`*`randinit`*`) (`*gmp_randstate_t*, ...`), ...)` [Function]
Construct a `gmp_randclass`, using a call to the given *randinit* function (see Section 9.1 [Random State Initialization], page 67). The arguments expected are the same as *randinit*, but with `mpz_class` instead of `mpz_t`. For example,

```
gmp_randclass r1 (gmp_randinit_default);
gmp_randclass r2 (gmp_randinit_lc_2exp_size, 32);
gmp_randclass r3 (gmp_randinit_lc_2exp, a, c, m2exp);
gmp_randclass r4 (gmp_randinit_mt);
```

`gmp_randinit_lc_2exp_size` will fail if the size requested is too big, an `std::length_error` exception is thrown in that case.

`void gmp_randclass::seed (`*mpir_ui* `s`)` [Function]
`void gmp_randclass::seed (`*mpz_class* `s`)` [Function]
Seed a random number generator. See see Chapter 9 [Random Number Functions], page 67, for how to choose a good seed.

`mpz_class gmp_randclass::get_z_bits (`*mpir_ui* `bits`)` [Function]
`mpz_class gmp_randclass::get_z_bits (`*mpz_class* `bits`)` [Function]
Generate a random integer with a specified number of bits.

`mpz_class gmp_randclass::get_z_range (`*mpz_class* `n`)` [Function]
Generate a random integer in the range 0 to $n-1$ inclusive.

`mpf_class gmp_randclass::get_f ()` [Function]
`mpf_class gmp_randclass::get_f (`*mpir_ui* `prec`)` [Function]
Generate a random float f in the range $0 <= f < 1$. f will be to *prec* bits precision, or if *prec* is not given then to the precision of the destination. For example,

```
gmp_randclass  r;
...
mpf_class  f (0, 512);   // 512 bits precision
f = r.get_f();           // random number, 512 bits
```

12.6 C++ Interface Limitations

`mpq_class` and Templated Reading

A generic piece of template code probably won't know that `mpq_class` requires a `canonicalize` call if inputs read with `operator>>` might be non-canonical. This can lead to incorrect results.

`operator>>` behaves as it does for reasons of efficiency. A canonicalize can be quite time consuming on large operands, and is best avoided if it's not necessary.

But this potential difficulty reduces the usefulness of `mpq_class`. Perhaps a mechanism to tell `operator>>` what to do will be adopted in the future, maybe a preprocessor define, a global flag, or an `ios` flag pressed into service. Or maybe, at the risk of inconsistency, the `mpq_class operator>>` could canonicalize and leave `mpq_t operator>>` not doing so, for use on those occasions when that's acceptable. Send feedback or alternate ideas to `http://groups.google.com/group/mpir-devel`.

Subclassing

Subclassing the MPIR C++ classes works, but is not currently recommended.

Expressions involving subclasses resolve correctly (or seem to), but in normal C++ fashion the subclass doesn't inherit constructors and assignments. There's many of those in the MPIR classes, and a good way to reestablish them in a subclass is not yet provided.

Templated Expressions

A subtle difficulty exists when using expressions together with application-defined template functions. Consider the following, with `T` intended to be some numeric type,

```
template <class T>
T fun (const T &, const T &);
```

When used with, say, plain `mpz_class` variables, it works fine: `T` is resolved as `mpz_class`.

```
mpz_class f(1), g(2);
fun (f, g);    // Good
```

But when one of the arguments is an expression, it doesn't work.

```
mpz_class f(1), g(2), h(3);
fun (f, g+h);  // Bad
```

This is because `g+h` ends up being a certain expression template type internal to `mpirxx.h`, which the C++ template resolution rules are unable to automatically convert to `mpz_class`. The workaround is simply to add an explicit cast.

```
mpz_class f(1), g(2), h(3);
fun (f, mpz_class(g+h));  // Good
```

Similarly, within `fun` it may be necessary to cast an expression to type `T` when calling a templated `fun2`.

```
template <class T>
void fun (T f, T g)
{
  fun2 (f, f+g);     // Bad
}

template <class T>
void fun (T f, T g)
{
```

```
  fun2 (f, T(f+g));  // Good
}
```

13 .Net Interface

This chapter describes the Microsoft.Net wrapper around MPIR.

If you are a .Net developer on MS Windows, using MPIR is possible via the basic managed-to-native interop tooling provided by .Net. While this would allow access to the full MPIR intreface, you would essentially be embedding C code inside whatever .Net language you are using. This would virtually require familiarity with C/C++, the interop artefacts in your code would be distractingly evident, and it would be hard to maintain a smooth code style around managed/native transitions.

MPIR offers an alternative that addresses these issues: **MPIR.Net**. MPIR.Net is a Microsoft Visual Studio solution that interoperates with MPIR and exposes a full managed interface built from scratch, for consumption in any .Net language. It internalizes all C-rooted idiosynchrasies and allows you to work with MPIR objects through managed classes that perform all necessary marshaling behind the scenes. It strives to provide maximum performance by implementing MPIR operations with direct calls to the native routines while not requiring you to sacrifice any of your code style. It eliminates any requirement of fluency in C, yet delivers the performance of native MPIR. In fact, it can consume any native MPIR build, including all supported processor-specific builds, and can thus take advantage of the entire wealth of assembly-optimized MPIR routines.

MPIR.Net is, however, limited to MS Windows and Visual Studio at this time. The managed interface is written in Microsoft C++/CLI, which ties you to that specific environment. If you use .Net on Linux and use a compiler other than Visual Studio, MPIR.Net will not work for you, but then again, you may already have better native interop facilities available to you than your Windows colleagues, making MPIR.Net rather moot.

MPIR.Net is bundled with MPIR as an optional feature. To build it, you still need to build the native MPIR library first. As you do, you can select the best processor architecture that matches your requirements. Then you build MPIR.Net, and it is linked statically to the native MPIR library, producing a managed assembly. Thus, to build MPIR.Net, you need to be familiar with the MPIR build process on Windows, and have a recent version of Visual Studio available (a community edition will suffice).

13.1 MPIR.Net Feature Overview

MPIR.Net exposes the following main classes:

`HugeInt` [Class]
`HugeRational` [Class]
`HugeFloat` [Class]
`MpirRandom` [Class]

The standard operators are overloaded to allow arithmetic with these classes. For example,

```
void Calculate()
{
  using (var a = new HugeInt(1234))
  using (var b = new HugeInt("-5678"))
  using (var c = new HugeInt(a + b))
  {
    Debug.WriteLine("Result: {0}", c);
  }
}
```

MPIR.Net's multi-precision classes implement `IDisposable`, and the recommended usage for local instances is as shown above, within a `using` clause to guarantee native memory clean-up when a variable is disposed.

References that go out of scope without having been disposed are subject to the normal .Net garbage collection, which in most cases invokes object finalizers, and those in turn deallocate native memory. Applications that don't have memory pressure should work just fine either way, although deterministic disposal is a best practice.

Like MPIR's native Chapter 12 [C++ Class Interface], page 78, MPIR.Net implements an expression like `a.Value = b + c` with a single call to the corresponding native `mpz_add`, without using a temporary for the `b + c` part. More complex expressions that do not have a single-call native implementation like `a.Value = b*c + d*e`, still use temporary variables. Importantly, `a.Value = a + b*c` and the like will utilize the native `mpz_addmul`, etc. Note that in all of the above cases the assignment syntax is to set the `Value` property; more on that below.

Another similarity of MPIR.Net with the C++ interface is the deferral of evaluation. All arithmetic operations and many methods produce an expression object rather than an immediate result. This allows expressions of arbitrary complexity to be built. They are not evaluated until the expression is assigned to a destination variable, or when calling a method that produces a primitive (non-MPIR.Net type) result. For example:

```
void Calculate()
{
  var a = new HugeInt(12345);
  var b = new HugeInt(67890);
  var sum = a + b;                // produces an expression
  var doubleSum = sum * 2;        // produces a new expression
  bool positive = doubleSum > 0;  // evaluates the doubleSum expression
  int sumSign = doubleSum.Sign(); // evaluates the doubleSum expression
  a.Value = doubleSum - 4;        // evaluates the doubleSum expression
}
```

Here the addition and multiplication in `(a + b) * 2` are computed three times because they are part of an expression that is consumed by three destinations, `positive`, `sumSign`, and `a`. To avoid the triple addition, this method should be re-written as:

```
void Calculate()
{
  var a = new HugeInt(12345);
  var b = new HugeInt(67890);
  var sum = a + b;                       // produces an expression
  var doubleSum = new HugeInt(sum * 2); // evaluates the expression
  bool positive = doubleSum > 0;         // evaluates the > comparison
  int sumSign = doubleSum.Sign();        // computes the sign
  a.Value = doubleSum - 4;               // computes the subtraction
}
```

Now the result of `(a + b) * 2` is computed once and stored in an intermediate variable, whose value is used in subsequent statements. This code can be shortened as follows without changing the internal calculation:

```
void Calculate()
{
  var a = new HugeInt(12345);
```

```
    var b = new HugeInt(67890);
    var doubleSum = new HugeInt((a + b) * 2); // evaluates the expression
    var positive = doubleSum > 0;             // evaluates the > comparison
    var sumSign = doubleSum.Sign();           // computes the sign
    a.Value = doubleSum - 4;                  // computes the subtraction
}
```

The main idiosyncrasy of MPIR.Net is its assignment pattern. MPIR.Net types are implemented as reference types with value semantics. Like .Net Strings, the objects themselves are just lightweight pointers to data allocated elsewhere. In this case, the data is in native memory. Unlike Strings, MPIR types are mutable.

Value semantics requires you to be able to code statements like `a = b + c`. However, .Net (outside of C++) does not allow overloading the assignment operator, while assigning references would necessitate some unnecessary duplication and extra memory allocations, require reliance on the garbage collector, and prevent the use of `mpz_addmul` and the like.

To solve this problem, MPIR.Net uses the property assignment. All MPIR.Net types have a `Value` property. The magic of this property is in its setter, which does what an overloaded assignment operator would do in C++. So you write `a.Value = b + c` to calculate the sum of `b` and `c` and store the result in the existing variable `a`. This seems to be as close to an overloaded assignment as you can get in .Net, but is fluent enough to become a quick habit, and additionally reinforces the concept that an existing object can change its value while reusing internally allocated memory.

Setting `Value` evaluates the expression being assigned. Since at this point the destination is known, `mpz_addmul` and similar can be recognized and invoked.

Reading this property is less interesting, as it's equivalent to but wordier than using the reference itself, i.e. `a + b` is equivalent to `a.Value + b.Value`. However it is still useful for making possible constructs such as `a.Value += 5`, `a.Value *= 10`, etc.

If you absent-mindedly type `a = b + c` or `a *= 10`, these will not compile because there is no implicit conversion from an expression. If an implicit conversion were defined, such code would incur an extra allocation plus garbage collection, making it potentially slower than performing the same operations on `a.Value`. It would also not compile if the destination were a local variable defined in a `using` clause, as is the recommended practice for method-local instances.

Care should be taken with the construct `var a = b;`. While perfectly legal (and cannot be made otherwise) in .Net, this only creates a copy of the managed reference to the same MPIR.Net object, without any copying of the data. If `b` is subsequently disposed, referencing `a` will throw an error.

MPIR classes can be intermixed in expressions to some degree. For example, most arithmetic operations with rational operands will accept integers. Where mixed operations are defined in MPIR, they are also implemented in MPIR.Net. Floats, on the other hand, typically don't accept operands of other types. There is some cost associated with creating a floating point instance out of an integer, which would not be evident if automatic promotion existed. Use explicit constructors to convert instances of one type to new instances of other types, or one of the `SetTo()` overloads to save the result into an existing instance.

MPIR classes can also be intermixed in expressions with primitive types. For 64-bit builds, this includes `long` and `ulong`, which correspond to an MPIR limb. For 32-bit builds, `int` and `uint` are the largest primitive types you can use. Smaller integer primitives can always be used because they will be promoted by .Net.

Conversions back from MPIR classes to primitive types aren't done automatically, instead methods `ToLong()`/`ToUlong()` for 64-bit builds or `ToInt()`/`ToUint()` are provided. Integers also implement `GetLimb()`.

13.2 Building MPIR.Net

To build MPIR.Net, follow the steps below:

1. Get the sources
2. Build MPIR
3. Run MPIR unit tests
4. Build MPIR.Net
5. Run MPIR.Net unit tests
6. Reference MPIR.Net in your managed project

Get the sources: Clone the MPIR repository on GitHub to get the latest stable MPIR release. This repository includes MPIR.Net. Or you can clone the MPIR.Net fork, which will get you the development repository.

Build MPIR: Once you have the sources, you will need to build MPIR first. Read the MPIR manual, available as a Documentation link on the MPIR page, for full details. Since MPIR.Net currently requires Windows, you will need to build MPIR for Windows using Microsoft Visual Studio. MPIR provides solutions for the three latest versions of Visual Studio, and includes full build instructions. You can select either a generic C build or an optimized build for a specific processor. You must also select the Windows architecture desired (32-bit or 64-bit), and build configuration (debug/release). You will need to build MPIR as Lib, not DLL, to use it with MPIR.Net.

Run MPIR unit tests: MPIR contains a full suite of unit tests that you can (and should) execute to validate your build. It is a large and complex project, and many things can go wrong while building from sources. Building and running the tests only takes a few of minutes and might save you a lot of headache. Note that you must also build MPIR's C++ interface to run unit tests, however it is not a dependency for MPIR.Net.

Build MPIR.Net: Next, load the MPIR.Net solution in Visual Studio. It is located in the MPIR.Net folder, under which there are folders for the different supported Visual Studio versions. The projects are set up to look for the previously built MPIR library in its normal location in the Lib folder. You will need to select the same architecture (x64 or x86) and configuration (debug/release) as when you built MPIR. Then simply build the solution, and you are good to go.

Run MPIR.Net unit tests: MPIR.Net includes its own suite of unit tests. Because MPIR.Net is a wrapper around MPIR, these tests simply ensure that the right routines in MPIR are being called, but do not validate the robustness of the MPIR build itself. Thus, it is necessary to run both MPIR tests and MPIR.Net tests. MPIR.Net tests, though, are easier to run because they are included right in the MPIR.Net solution.

Through binary compatibility with GMP 5.x, MPIR 2.x inherits a known issue that causes a few MPIR.Net tests (2 for x86, 3 for x64) to fail. The issue has been corrected in GMP 6.x, and is expected to be corrected correspondingly in MPIR 3.x. Because this behavior is not intuitive, these tests remain in their current failing state until this is resolved.

Reference MPIR.Net: With the MPIR.Net assembly built, you're ready to create your own project in a .Net language of your choice, add a reference to MPIR.Net, and take advantage of the great mathematical powers of MPIR!

13.3 MPIR.Net Integers

`HugeInt` *: IntegerExpression, IDisposable* [Class]
The MPIR.Net type for the MPIR multi-precision integer is `HugeInt`. A closely related type is `IntegerExpression`, which is returned from all operators and methods whose value semantics are to compute another number from the source instance and any arguments. `HugeInt` derives from `IntegerExpression`, and many operations are defined on the expression class. Operations defined on `HugeInt` but not on `IntegerExpression` are typically those that modify the value of the source number itself, and thus performing them on an expression is meaningless. Because through inheritance all operations are available on `HugeInt`, the descriptions below do not specifically indicate whether each operator or method is defined for expressions, or just for `HugeInt` instances. For the sake of brevity, they are listed as if they were methods of the `HugeInt` class. Visual Studio provides Intellisense and immediate feedback to help sort out which operations are available on expressions.

Below is a brief summary of the supported multi-precision integer methods and operators. To avoid repetition, implementation details are ommitted. Since MPIR native functions are called behind the scenes, review Chapter 5 [Integer Functions], page 29, for further details about the native implementations.

`HugeInt ()` [Constructor]
`HugeInt (`*int/long* `n)` [Constructor]
`HugeInt (`*uint/ulong* `n)` [Constructor]
`HugeInt (`*double* `n)` [Constructor]
Constructs a `HugeInt` object. Single-limb constructors vary by architecture, 32-bit builds take an `int` or `uint` argument, 64-bit builds take a `long` or `ulong`. Any necessary conversion follows the corresponding C function, for example `double` follows `mpz_set_d` (see Section 5.2 [Assigning Integers], page 30).

`HugeInt (`*string* `s)` [Constructor]
`HugeInt (`*string* `s`, *int* `base)` [Constructor]
Constructs a `HugeInt` converted from a string using `mpz_set_str` (see Section 5.2 [Assigning Integers], page 30). If the string is not a valid integer, an exception is thrown.

`HugeInt (`*IntegerExpression* `e)` [Constructor]
Evaluates the supplied expression and saves its result to the new instance. Because `HugeInt` is derived from `IntegerExpression`, this constructor can be used to make a copy of an existing variable, i.e. `HugeInt a = new HugeInt(b);` without creating any permanent association between them.

`static HugeInt Allocate (`*uint/ulong* `bits)` [Static Method]
`void Reallocate (`*uint/ulong* `bits)` [Method]
Controls the capacity in bits of the allocated integer.

`int AllocatedSize` [Property]
Returns the number of limbs currently allocated.

`ulong Size ()` [Method]
Returns the number of limbs currently used.

`long GetLimb (`*mp_size_t* `index)` [Method]
Returns the specified limb.

`bool FitsUlong ()` *//64-bit builds only* [Method]
`bool FitsLong ()` *//64-bit builds only* [Method]
`bool FitsUint ()` [Method]
`bool FitsInt ()` [Method]
`bool FitsUshort ()` [Method]
`bool FitsShort ()` [Method]
`long ApproximateSizeInBase (`*int* `base`) [Method]
Checks whether the number would fit in one of the built-in .Net types.

`string ToString ()` [Method]
`string ToString (`*int* `base`) [Method]
`string ToString (`*int* `base`, *bool* `lowercase`) [Method]
Returns the string representation of the number. The default `base` is 10, and the parameterless overload is limited to 256 least significant digits by default, producing a leading ellipsis (i.e. ...12345) when the number has more digits. This is done to prevent huge numbers from unexpectedly consuming large amounts of memory in the debugger. The maximum number of digits output is configurable via the `MpirSettings.ToStringDigits` property, where zero means unlimited. The other overloads always output all digits.

`int ToInt ()` *//32-bit builds* [Method]
`uint ToUint ()` *//32-bit builds* [Method]
`long ToLong ()` *//64-bit builds* [Method]
`ulong ToUlong ()` *//64-bit builds* [Method]
`double ToDouble ()` [Method]
`double ToDouble (out` *int/long* `exp`) [Method]
Converts the number to a primitive (built-in) .Net type, assuming it fits, which can be determined by calling one of the `Fits...` methods.

`IntegerExpression Value` [Property]
Getting this property is essentially a no-op, as it returns the object instance itself. This never needs to be done explicitly, but is used implicitly in statements like `a.Value += 5;`

Setting the `Value` property evaluates the assigned expression and saves the result to the object.

`void SetTo (`*int/long* `value`) *// 32/64-bit builds* [Method]
`void SetTo (`*uint/ulong* `value`) *// 32/64-bit builds* [Method]
`void SetTo (`*double* `value`) [Method]
`void SetTo (`*string* `value`) [Method]
`void SetTo (`*string* `value`, *int* `base`) [Method]
`void SetTo (`*RationalExpression* `value`) [Method]
`void SetTo (`*FloatExpression* `value`) [Method]
Sets the value of existing variable from types other than `IntegerExpression`.

`void Swap (`*HugeInt* `a`) [Method]
Swaps the values of the two objects. This is an $O(1)$ operation.

Arithmetic operators (`+`, `-`, `*`, `/`, `%`) are overloaded to allow integers to participate in expressions much like primitive integers can. Single-limb primitive types can be used. These operators will also accept `RationalExpression` arguments, producing a `RationalExpression` result. Some expression types expose additional methods, these are listed below. Invoking these methods does not prevent the expression from participating in further expressions.

Expressions resulting from division or computing a modulo allow setting an explicit rounding mode:

```
c.Value = (a / b).Rounding(RoundingModes.Ceiling) + 4;
d.Value = (a % b).Rounding(RoundingModes.Floor) + 4;
```

Division expressions optionally allow the remainder to be saved:

```
c.Value = (a / b).SavingRemainderTo(e) + 4;
```

When dividing by a limb, the remainder is a single limb and is saved to an unsigned limb variable. However, passing this variable as an `out` argument would not work because of the deferred evaluation. Instead, a delegate is passed which is called during evaluation:

```
ulong/uint remainder; // 64/32-bit builds
d.Value = (a / 100).SettingRemainderTo(x => remainder = x) + 4;
```

Symmetrically, the modulo expressions (`%`) allow the quotient to be saved:

```
c.Value = (a % b).SavingQuotientTo(e).RoundingMode(RoundingModes.Ceiling) + 4;
ulong/uint quotient; // 64/32-bit builds
d.Value = (a % 100).SettingQuotientTo(x => quotient = x) + 4;
```

`uint/ulong Mod` (*uint/ulong* ***`divisor`***) [Method]
`uint/ulong Mod` (*uint/ulong* ***`divisor`***, *RoundingModes* ***`roundingMode`***) [Method]
Computes the absolute value of the remainder from division of the source number by the specified `divisor`. This operation differs from using the `%` operator by where the result is saved. The `%` operator returns an expression, and a `HugeInt` variable is required to receive the result when the expression is assigned to its `Value` property. The `Mod` method, on the other hand, computes and returns the remainder immediately since it's a primitive type (single limb), and no destination `HugeInt` variable is needed.

Operator `^` serves dual purposes: when the right operand is a single limb, it raises the source number to a power, if the right operand is an `IntegerExpression` it performs a bitwise XOR.

Comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) accept `IntegerExpression`, single-limb, or double arguments, but do not accept `RationalExpression` because that would require an awkward explicit cast when comparing with null.

`int CompareTo` (*IntegerExpression* `a`) [Method]
`bool Equals` (*IntegerExpression* `a`) [Method]
Implement `IComparable<IntegerExpression>` and `IEquatable<IntegerExpression>` for strongly-typed comparisons.

`int CompareTo` (*object* `a`) [Method]
`bool Equals` (*object* `a`) [Method]
Implement `IComparable` and equality check for any object. These accept a `RationalExpression` as an argument, allowing cross-type comparisons not possible with operators.

`int GetHashCode ()` [Method]
This `object` override computes the hash code. This is an $O(N)$ operation where N is the number of limbs in use. Changing a number's `Value` changes its hash code, so this should not be done on any object that has been added to a hash table or dictionary.

`int CompareAbsTo` (*IntegerExpression* `a`) [Method]
`int CompareAbsTo` (*uint/ulong* `a`) [Method]
`int CompareAbsTo` (*double* `a`) [Method]
Compares the absolute value of the number with the operand.

`int Sign` () [Method]
Returns the number's sign.

Bit shift operators (`<<`, `>>`) accept an unsigned limb operand.

The right shift (`>>`) expression provides a method to compute the modulo, rather than the default quotient:

```
var a = new HugeInt("0x1357");
Debug.WriteLine((a >> 8).ToString(16)); //prints 13
Debug.WriteLine((a >> 8).Remainder().ToString(16)); //prints 57
```

Bitwize operators (`&`, `|`, `^`, `~`) are defined for `IntegerExpression` operands only. Note that operator `^` is also defined for a limb operand, and in that case computes a power.

`bool GetBit` (*uint/ulong* `position`) [Method]
`void SetBit` (*uint/ulong* `position`, *bool* `value`) [Method]
`void ComplementBit` (*uint/ulong* `position`) [Method]
Allows access to individual bits of the number, using a "virtual" two's complement representation.

`uint/ulong PopCount` () *// 32/64-bit builds* [Method]
Gets the number of set bits in the number.

`uint/ulong HammingDistance` (*IntegerExpression* `target`) *// 32/64-bit builds* [Method]
Gets the hamming distance between this number and `target`.

`uint/ulong FindBit` (*bool* `value`, *uint/ulong* `start`) *// 32/64-bit builds* [Method]
Scans the number for next set or cleared bit (depending on `value`).

`IntegerExpression Abs` () [Method]
Returns an expression that computes the absolute value of the number.

`IntegerExpression DivideExactly` (*IntegerExpression* `divisor`) [Method]
`IntegerExpression DivideExactly` (*uint/ulong* `divisor`) *// 32/64-bit builds* [Method]
Returns an expression that performs a fast division where it is known that there is no remainder.

`IntegerExpression PowerMod` (*IntegerExpression* `power`, *IntegerExpression* `modulo`) [Method]
`IntegerExpression PowerMod` (*uint/ulong* `power`, *IntegerExpression* `modulo`) *// 32/64-bit builds* [Method]
Returns an expression that raises the source to the specified `power` modulo `modulo`.

`bool IsDivisibleBy` (*IntegerExpression* `a`) [Method]
`bool IsDivisibleBy` (*uint/ulong* `a`) [Method]
`bool IsDivisibleByPowerOf2` (*uint/ulong* `power`) [Method]

`bool IsCongruentTo` (*IntegerExpression* `a`, *IntegerExpression* `modulo`) [Method]
`bool IsCongruentTo` (*uint/ulong* `a`, *uint/ulong* `modulo`) [Method]
`bool IsCongruentToModPowerOf2` (*IntegerExpression* `a`, *uint/ulong* `power`) [Method]
`bool IsPerfectPower` () [Method]
`bool IsPerfectSquare` () [Method]

Performs various divisibility checks. These methods return a bool result, and therefore are executed immediately. If they are called on an expression, the expression is evaluated to a temporary which is discarded immediately afterwards. If you will need this result again, assign the expression to a `HugeInt` variable and call the method on it.

`long Write` (*Stream* `stream`) [Method]
`long Read` (*Stream* `stream`) [Method]

Writes and reads integers to/from streams using the raw binary format.

`long Write` (*TextWriter* `writer`) [Method]
`long Write` (*TextWriter* `writer`, *int* `base`) [Method]
`long Write` (*TextWriter* `writer`, *int* `base`, *bool* `lowercase`) [Method]
`long Read` (*TextReader* `reader`) [Method]
`long Read` (*TextReader* `reader`, *int* `base`) [Method]

Writes and reads integers as text.

`void Import<T>` (*T*[] `data`, *long* `limbCount`, *int* `bytesPerLimb`, *LimbOrder* `limbOrder`, *Endianness* `endianness`, *int* `nails`) [Method]
`long Export<T>` (*T*[] `data`, *int* `bytesPerLimb`, *LimbOrder* `limbOrder`, *Endianness* `endianness`, *int* `nails`) [Method]
`T[] Export<T>` (*int* `bytesPerLimb`, *LimbOrder* `limbOrder`, *Endianness* `endianness`, *int* `nails`) [Method]

Imports/exports the absolute value of the number to/from arbitrary words of data.

`bool IsProbablePrime` (*MpirRandom* `random`, *int* `probability`, *ulong/uint* `pretested`) [Method]
`bool IsLikelyPrime` (*MpirRandom* `random`, *ulong/uint* `pretested`) [Method]
`static int Jacobi` (*HugeInteger* `a`, *HugeInteger* `b`) [Static Method]
`static int Legendre` (*HugeInteger* `a`, *HugeInteger* `b`) [Static Method]
`static int Kronecker` (*HugeInteger* `a`, *HugeInteger* `b`) [Static Method]
`static int Kronecker` (*HugeInteger* `a`, *int/long* `b`) [Static Method]
`static int Kronecker` (*HugeInteger* `a`, *uint/ulong* `b`) [Static Method]
`static int Kronecker` (*int/long* `a`, *HugeInteger* `b`) [Static Method]
`static int Kronecker` (*uint/ulong* `a`, *HugeInteger* `b`) [Static Method]
`static IntegerExpression Power` (*uint/ulong* `value`, *uint/ulong* `power`) [Static Method]
`static IntegerExpression Factorial` (*uint/ulong* `value`) [Static Method]
`static IntegerExpression Factorial` (*uint/ulong* `value`, *uint/ulong* `order`) [Static Method]
`static IntegerExpression Primorial` (*uint/ulong* `value`) [Static Method]
`static IntegerExpression Binomial` (*uint/ulong* `n`, *uint/ulong* `k`) [Static Method]
`static IntegerExpression Binomial` (*IntegerExpression* `n`, *uint/ulong* `k`) [Static Method]

Performs various number-theoretic computations.

`static IntegerSequenceExpression Fibonacci (`*int/long* `n`) [Static Method]
`static IntegerSequenceExpression Lucas (`*int/long* `n`) [Static Method]
These two methods return a specialized expression that provides an additional method to optionally save the previous number in the sequence, in addition to the number requested, for example:

```
var b = new HugeInt();
var c = new HugeInt(HugeInt.Fibonacci(300).SavingPreviousTo(b));
```

`IntegerSquareRootExpression SquareRoot ()` [Method]
Returns an expression that evaluates to the square root of the number. The expression provides a method to optionally save the remainder to a second variable:

```
a.Value = b.SquareRoot().SavingRemainderTo(c);
```

`IntegerRootExpression Root (`*ulong/uint* `power`) [Method]
Returns an expression that evaluates to the root of the specified `power` of the number. The expression provides two optional methods. One allows to save the remainder to a second variable, and the other allows to set a boolean flag indicating whether the root operation was exact. Note that computing the remainder is more costly than just getting an exact flag.

```
bool exact = false;
a.Value = b.Root(3).SavingRemainderTo(r);
c.Value = d.Root(4).SettingExactTo(x => exact = x);
e.Value = f.Root(5).SavingRemainderTo(r).SettingExactTo(x => exact = x);
```

`IntegerExpression NextPrimeCandidate (`*MpirRandom* `random`) [Method]
Returns an expression that looks for the next possible prime greater than the source number.

`uint/ulong Gcd (`*uint/ulong* `a`) [Method]
Computes the greatest common divisor with the specified single-limb number.

`IntegerGcdExpression Gcd (`*IntegerExpression* `a`) [Method]
Returns an expression that computes the greatest common divisor of the source number and `a`. Provides a method to optionally calculate the related Diophantine equation multiplier(s):

```
c.Value = a.Gcd(b).SavingDiophantineMultipliersTo(s, t);
```

If either `s` or `t` is null, that coefficient is not computed.

`IntegerExpression Lcm (`*IntegerExpression* `a`) [Method]
`IntegerExpression Lcm (`*uint/ulong* `a`) [Method]
Computes the least common multiple with `a`.

`IntegerExpression Invert (`*IntegerExpression* `modulo`) [Method]
Returns an expression to compute the inverse of the source number modulo `modulo`.

`IntegerRemoveFactorsExpression RemoveFactors (`*IntegerExpression* `factor`) [Method]
Returns an expression that evaluates to the result of removing all occurrences of the specified `factor` from the source number. Provides a method to optionally save the number of factors that were removed:

```
ulong/uint numberRemoved; // 64/32-bit builds
```

```
a.Value = b.RemoveFactors(c);
d.Value = e.RemoveFactors(f).SavingCountRemovedTo(x => numberRemoved = x);
```

13.4 MPIR.Net Rationals

`HugeRational` *: RationalExpression, IDisposable* [Class]

MPIR multi-precision rational numbers are represented by the `HugeRational` class, along with its corresponding expression class `RationalExpression`, which is returned from all operators and methods whose value semantics are to compute another number from the source instance and any arguments. Operations defined on `HugeRational` but not on `RationalExpression` are typically those that modify the value of the source number itself, and thus performing them on an expression is meaningless. Because through inheritance all operations are available on `HugeRational`, the descriptions below do not specifically indicate whether each operator or method is defined for expressions, or just for `HugeRational` instances. For the sake of brevity, they are listed as if they were methods of the `HugeRational` class. Visual Studio provides Intellisense and immediate feedback to help sort out which operations are available on expressions.

Below is a brief summary of the supported multi-precision rational methods and operators. To avoid repetition, implementation details are ommitted. Since MPIR native functions are called behind the scenes, review Chapter 6 [Rational Number Functions], page 46, for further details about the native implementations.

`HugeRational` () [Constructor]
`HugeRational` (*int/long* `numerator`, *uint/ulong* `denominator`) [Constructor]
`HugeRational` (*uint/ulong* `numerator`, *uint/ulong* `denominator`) [Constructor]
`HugeRational` (*IntegerExpression* `numerator`, *IntegerExpression* `demoninator`) [Constructor]
`HugeRational` (*double* `n`) [Constructor]

Constructs a `HugeRational` object. Single-limb constructors vary by architecture, 32-bit builds take `int` or `uint` arguments, 64-bit builds take `long` or `ulong`. Any necessary conversion follows the corresponding C function, for example `double` follows `mpq_set_d` (see Section 6.1 [Initializing Rationals], page 46).

`HugeRational` (*string* `s`) [Constructor]
`HugeRational` (*string* `s`, *int* `base`) [Constructor]

Constructs a `HugeRational` converted from a string using `mpq_set_str` (see Section 6.1 [Initializing Rationals], page 46). If the string is not a valid integer or rational, an exception is thrown.

When constructing a rational number from a numerator and denominator, including the string constructors where both numerator and denominator are specified, the fraction should be in canonical form, or if not then `Canonicalize()` should be called.

`HugeRational` (*IntegerExpression* `e`) [Constructor]
`HugeRational` (*RationalExpression* `e`) [Constructor]
`HugeRational` (*FloatExpression* `e`) [Constructor]

Evaluates the supplied expression and saves its result to the new instance. Because multi-precision classes are derived from their corresponding expression classes, these constructors can be used to make a copy of an existing variable, i.e. `HugeRational a = new HugeRational(b);` without creating any permanent association between them.

`static HugeRational Allocate` (*uint/ulong* `numeratorBits`, *uint/ulong* `denominatorBits`) [Static Method]
Controls the capacity in bits of the allocated integer. HugeRational does not have a `Reallocate` method, but its numerator and demonimator are derived from HugeInt and can thus be reallocated separately.

`void Canonicalize ()` [Method]
Puts a `HugeRational` into canonical form, as per Chapter 6 [Rational Number Functions], page 46. All arithmetic operators require their operands in canonical form, and will return results in canonical form.

`HugeInt Numerator` [Property]
`HugeInt Denominator` [Property]
These read-only properties expose the numerator and denominator for direct manipulation. They return specialized instances of the `HugeInt` class that do not own their limb data. They override the `Dispose()` method with a no-op, so they can be safely passed around as normal integers, even to code that tries to dispose of them.

Once a numerator or denominator is obtained, it remains valid for the life of the `HugeRational` instance. It references live data, so for example, if the `Value` of the rational is modified, it will be visible through a previously obtained numerator/denominator instance. Conversely, setting the `Value` of a numerator or denominator modifies the `Value` of its owning rational, and if this cannot be known to keep the rational in canonical form, `Canonicalize()` must be called before performing any further MPIR operations on the rational.

Multiple copies can be safely obtained, and reference the same internal structures. Once the `HugeRational` is disposed, any numerator and denominator instances obtained from it are no longer valid.

`long ApproximateSizeInBase` (*int* `base`) [Method]
Returns the number of digits the absolute value of number would take if written in the specified base. The result can be at most 2 characters too long, and allows for a numerator, a division sign, and a denominator, but excludes the leading minus sign.

`string ToString ()` [Method]
`string ToString` (*int* `base`) [Method]
`string ToString` (*int* `base`, *bool* `lowercase`) [Method]
Returns the string representation of the number. The default `base` is 10, and the parameterless overload is limited to 256 least significant digits by default, each for a numerator and a denominator, producing a leading ellipsis (i.e. ...12345) when either component has more digits. This is done to prevent huge numbers from unexpectedly consuming large amounts of memory in the debugger. The maximum number of digits output is configurable via the `MpirSettings.ToStringDigits` property, where zero means unlimited. The other overloads always output all digits.

`double ToDouble ()` [Method]
Converts the number to a double, possibly truncated.

`RationalExpression Value` [Property]
Getting this property is essentially a no-op, as it returns the object instance itself. This never needs to be done explicitly, but is used implicitly in statements like `a.Value += 5;`

Setting the `Value` property evaluates the assigned expression and saves the result to the object.

`void SetTo` (*int/long* `value`) *// 32/64-bit builds* [Method]
`void SetTo` (*int/long* `numerator`, *uint/ulong* `denominator`) [Method]
`void SetTo` (*uint/ulong* `value`) [Method]
`void SetTo` (*uint/ulong* `numerator`, *uint/ulong* `denominator`) [Method]
`void SetTo` (*double* `value`) [Method]
`void SetTo` (*string* `value`) [Method]
`void SetTo` (*string* `value`, *int* `base`) [Method]
`void SetTo` (*IntegerExpression* `value`) [Method]
`void SetTo` (*IntegerExpression* `numerator`, *IntegerExpression* `denominator`) [Method]
`void SetTo` (*FloatExpression* `value`) [Method]
Sets the value of existing variable from types other than `RationalExpression`. When setting both the numerator and denominator, canonicalization must be managed explicitly.

`void Swap` (*HugeRational* `a`) [Method]
Swaps the values of the two objects. This is an $O(1)$ operation. Any existing numerators and denominators remain associated with the object on which they were obtained, and reflect its new value.

Arithmetic operators (`+`, `-`, `*`, `/`) are overloaded to allow rationals to participate in expressions much like primitive integers can. Single-limb primitive types can be used. These operators will also accept `IntegerExpression` arguments, and will automatically promote them. In expressions, promotion of an `IntegerExpression` to a `RationalExpression` is an $O(1)$ operation. Of course, when constructing a rational from an integer, a copy is made so this becomes $O(N)$.

Due to the rationals' nature, division is always exact (there is no rounding) and the modulo operator (`%`) is not defined. Also not defined are the bit shift operators (`<<`, `>>`), and the bitwise operators (`&`, `|`, `^`, `~`).

Operator `^` raises the source number to the specified power.

Comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) accept `RationalExpression`, single-limb, or double arguments, but do not accept integer or float expressions because that would require an awkward explicit cast when comparing with null. Use the `CompareTo(object)` method for cross-comparisons.

`int CompareTo` (*RationalExpression* `a`) [Method]
`bool Equals` (*RationalExpression* `a`) [Method]
Implement `IComparable<RationalExpression>` and `IEquatable<RationalExpression>` for strongly-typed comparisons.

`int CompareTo` (*object* `a`) [Method]
`bool Equals` (*object* `a`) [Method]
Implement `IComparable` and equality check for any object. For rationals, these methods support any expression type (integer, rational, or float).

`bool Equals` (*int/long* `numerator`, *uint/ulong* `denominator`) [Method]
`bool Equals` (*uint/ulong* `numerator`, *uint/ulong* `denominator`) [Method]
`int CompareTo` (*int/long* `numerator`, *uint/ulong* `denominator`) [Method]
`int CompareTo` (*uint/ulong* `numerator`, *uint/ulong* `denominator`) [Method]
Single-limb comparisons for rationals take two arguments.

`int GetHashCode ()` [Method]
This `object` override computes the hash code. This is an $O(N)$ operation where N is the number of limbs in use in the numerator and denominator combined. Changing a number's `Value` changes its hash code, so this should not be done on any object that has been added to a hash table or dictionary.

`int Sign ()` [Method]
Returns the number's sign.

`RationalExpression Abs ()` [Method]
Returns an expression that computes the absolute value of the number.

`RationalExpression Invert ()` [Method]
Returns an expression that computes the inverse of the number.

`long Write (`*Stream* `stream``)` [Method]
`long Read (`*Stream* `stream``)` [Method]
Writes and reads rationals to/from streams using the raw binary format.

`long Write (`*TextWriter* `writer``)` [Method]
`long Write (`*TextWriter* `writer`, *int* `base``)` [Method]
`long Write (`*TextWriter* `writer`, *int* `base`, *bool* `lowercase``)` [Method]
`long Read (`*TextReader* `reader``)` [Method]
`long Read (`*TextReader* `reader`, *int* `base``)` [Method]
Writes and reads rationals as text.

There are no `Import/Export` methods, but they can of course be invoked on the numerator and/or denominator.

`RationalExpression` does not have any specialized subclasses, as there are no operations on the rational type that require additional inputs beyond the left and right operator operands.

13.5 MPIR.Net Floats

`HugeFloat` *: FloatExpression, IDisposable* [Class]
The MPIR.Net class for multi-precision floating point numbers is `HugeFloat`, and its corresponding expression class is `FloatExpression`, which is returned from all operators and methods whose value semantics are to compute another number from the source instance and any arguments. `HugeFloat` derives from `FloatExpression`, and many operations are defined on the expression class. Operations defined on `HugeFloat` but not on `FloatExpression` are typically those that modify the value of the source number itself, and thus performing them on an expression is meaningless. Because through inheritance all operations are available on HugeFloat, the descriptions below do not specifically indicate whether each operator or method is defined for expressions, or just for `HugeFloat` instances. For the sake of brevity, they are listed as if they were methods of the `HugeFloat` class. Visual Studio provides Intellisense and immediate feedback to help sort out which operations are available on expressions.

Below is a brief summary of the supported multi-precision rational methods and operators. To avoid repetition, implementation details are ommitted. Since MPIR native functions are called behind the scenes, review Chapter 7 [Floating-point Functions], page 50, for further details about the native implementations.

`static uint/ulong DefaultPrecision` [Static Property]
Gets or sets the default precision of the floating point mantissa, in bits. If the value is not a multiple of limb size, the actual precision will be rounded up. All newly constructed `HugeFloat` objects that don't explicitly specify precision will use this default. Previously constructed objects are unaffected. The initial default precision is 2 limbs.

When an expression is evaluated, it is either because it is being assigned to some destination variable (e.g. `a.Value = b + c;`) or a primitive-computing method is being called (e.g. `int s = (b + c).Sign();`) In the former case, the precision of the destination is used for all computations and temporaries during expression evaluation. In the latter case, there is no destination so the `DefaultPrecision` is used.

`HugeFloat ()` [Constructor]
`HugeFloat (`*int/long* `value)` [Constructor]
`HugeFloat (`*uint/ulong* `value)` [Constructor]
`HugeFloat (`*double* `n)` [Constructor]
Constructs a `HugeFloat` object. Single-limb constructors vary by architecture, 32-bit builds take `int` or `uint` arguments, 64-bit builds take `long` or `ulong`. Any necessary conversion follows the corresponding C function, for example `double` follows `mpf_set_d` (see Section 7.1 [Initializing Floats], page 50).

`HugeFloat (`*string* `s)` [Constructor]
`HugeFloat (`*string* `s`, *int* `base)` [Constructor]
`HugeFloat (`*string* `s`, *int* `base`, *bool* `exponentInDecimal)` [Constructor]
Constructs a `HugeFloat` converted from a string using `mpf_set_str` (see Section 7.1 [Initializing Floats], page 50). If the string is not a valid integer or floating point number, an exception is thrown.

`HugeFloat (`*IntegerExpression* `value)` [Constructor]
`HugeFloat (`*RationalExpression* `value)` [Constructor]
`HugeFloat (`*FloatExpression* `value)` [Constructor]
Evaluates the supplied expression and saves its result to the new instance. Because multi-precision classes are derived from their corresponding expression classes, these constructors can be used to make a copy of an existing variable, i.e. `HugeFloat a = new HugeFloat(b);` without creating any permanent association between them.

`static HugeRational Allocate (`*uint/ulong* `precision)` [Static Method]
`void Reallocate (`*uint/ulong* `precision)` [Method]
Controls the allocated precision in bits of the new or existing `HugeFloat`.

`uint/ulong AllocatedPrecision` [Property]
Gets the precision in bits that is currently allocated for internal storage of the mantissa. The precision actually in effect, used in calculations, is initially the same but may be reduced by setting the `Precision` property.

`uint/ulong Precision` [Property]
Gets or sets the effective precision of the number without changing the memory allocated. The number of bits cannot exceed the precision with which the variable was initialized or last reallocated. The value of the number is unchanged, and in particular if it previously had a higher precision it will retain that higher precision. New values assigned to the `Value` property will use the new precision. The number can be safely disposed after modifying its `Precision` (unlike the native MPIR, which requires you to restore the precision to the allocated value before the memory can be freed).

`bool FitsUlong ()` *//64-bit builds only* [Method]
`bool FitsLong ()` *//64-bit builds only* [Method]
`bool FitsUint ()` [Method]
`bool FitsInt ()` [Method]
`bool FitsUshort ()` [Method]
`bool FitsShort ()` [Method]
Checks whether the number would fit in one of the built-in .Net types.

`bool IsInteger ()` [Method]
Checks whether the number is a whole integer.

`string ToString ()` [Method]
`string ToString (`*int* `base)` [Method]
`string ToString (`*int* `base`, *bool* `lowercase)` [Method]
`string ToString (`*int* `base`, *bool* `lowercase`, *bool* `exponentInDecimal)` [Method]
Returns the string representation of the number. The default `base` is 10, and the parameterless overload is limited to 256 mantissa digits by default. This is done to prevent huge numbers from unexpectedly consuming large amounts of memory in the debugger. The maximum number of digits output is configurable via the `MpirSettings.ToStringDigits` property, where zero means unlimited. `MpirSettings.ToStringDigits` applies to integers and rationals as well. The other overloads always output all digits.

`int ToInt ()` *//32-bit builds* [Method]
`uint ToUint ()` *//32-bit builds* [Method]
`long ToLong ()` *//64-bit builds* [Method]
`ulong ToUlong ()` *//64-bit builds* [Method]
`double ToDouble ()` [Method]
`double ToDouble (out` *int/long* `exp)` [Method]
Converts the number to a primitive (built-in) .Net type, assuming it fits, which can be determined by calling one of the `Fits...` methods.

`FloatExpression Value` [Property]
Getting this property is essentially a no-op, as it returns the object instance itself. This never needs to be done explicitly, but is used implicitly in statements like `a.Value += 5;`

Setting the `Value` property evaluates the assigned expression and saves the result to the object.

`void SetTo (`*int/long* `value)` *// 32/64-bit builds* [Method]
`void SetTo (`*uint/ulong* `value)` *// 32/64-bit builds* [Method]
`void SetTo (`*double* `value)` [Method]
`void SetTo (`*string* `value)` [Method]
`void SetTo (`*string* `value`, *int* `base)` [Method]
`void SetTo (`*string* `value`, *int* `base`, *bool* `exponentInDecimal)` [Method]
`void SetTo (`*IntegerExpression* `value)` [Method]
`void SetTo (`*RationalExpression* `value)` [Method]
Sets the value of existing variable from types other than `FloatExpression`.

`void Swap (`*HugeFloat* `a)` [Method]
Swaps the values (and precisions) of the two objects. This is an $O(1)$ operation.

Arithmetic operators (`+`, `-`, `*`, `/`) and bit shifts (`<<`, `>>`) are overloaded to allow floats to participate in expressions much like primitive types can. Single-limb primitive types can be used.

These operators do not accept integer or rational expressions. There is some cost of instantiating a floating point number from another multi-precision type, so to make this point clear MPIR.Net forces you to use explicit constructors or assignments for this conversion.

The modulo operator (`%`) and the bitwise operators (`&`, `|`, `^`, `~`) are not defined.

Operator `^` raises the source number to the specified power.

Comparison operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) accept `FloatExpression`, single-limb, or double arguments, but do not accept integer or rational expressions because that would require an awkward explicit cast when comparing with null.

`int CompareTo` (*FloatExpression* `a`) [Method]
`bool Equals` (*FloatExpression* `a`) [Method]
Implement `IComparable<FloatExpression>` and `IEquatable<FloatExpression>` for strongly-typed comparisons.

`int CompareTo` (*object* `a`) [Method]
`bool Equals` (*object* `a`) [Method]
Implement `IComparable` and equality check for any object. These support only float expressions or .Net primitive types. When this method is called on a `HugeFloat` object, comparison is performed to the precision of the object. When called on an expression, comparison is performed to the default precision.

`int GetHashCode` () [Method]
This `object` override computes the hash code. This is an $O(N)$ operation where N is the number of limbs allocated. Changing a number's `Value` changes its hash code, so this should not be done on any object that has been added to a hash table or dictionary.

`bool Equals` (*object* `a`, *uint/ulong* `precision`) [Method]
Checks for equality using the specified precision. The argument `a` can be a `FloatExpression` or a primitive type.

`FloatExpression RelativeDifferenceFrom` (*FloatExpression* `a`) [Method]
Returns an expression that computes $|this - a|/this$

`FloatExpression Abs` () [Method]
`FloatExpression SquareRoot` () [Method]
`static FloatExpression SquareRoot` (*uint/ulong* `a`) [Static Method]
`FloatExpression Floor` () [Method]
`FloatExpression Ceiling` () [Method]
`FloatExpression Truncate` () [Method]
`int Sign` () [Method]
Perform various floating-point operations.

`long Write` (*TextWriter* `writer`) [Method]
`long Write` (*TextWriter* `writer`, *int* `base`) [Method]
`long Write` (*TextWriter* `writer`, *int* `base`, *int* `maxDigits`, *bool* `lowercase`, *bool* `exponentInDecimal`) [Method]
`long Read` (*TextReader* `reader`) [Method]
`long Read` (*TextReader* `reader`, *int* `base`) [Method]
`long Read` (*TextReader* `reader`, *int* `base`, *bool* `exponentInDecimal`) [Method]
Writes and reads floats as text.

13.6 MPIR.Net Random Numbers

`MpirRandom` *: IDisposable* [Class]
The MPIR.Net class that wraps the MPIR random number functions is `MpirRandom`. It holds an algorithm selection and current state, as per `gmp_randstate_t`. As the multi-precision classes, `MpirRandom` allocates unmanaged memory, and should be disposed of via its `IDisposable` implementation when no longer in use.

`static MpirRandom Default ()` [Static Method]
`static MpirRandom MersenneTwister ()` [Static Method]
`static MpirRandom LinearCongruential (`*HugeInt* `a`, *ulong/uint* `c`, *ulong/uint* `m``)` [Static Method]
`static MpirRandom LinearCongruential (`*ulong/uint* `size`) [Static Method]
In lieu of constructors, `MpirRandom` uses more descriptive static factory methods to create new instances of specific random number generator algorithms.

`MpirRandom Copy ()` [Method]
Creates a new random number generator with a copy of the algorithm and state from the source instance.

`void Seed (`*ulong/uint* `seed`) [Method]
`void Seed (`*HugeInt* `seed`) [Method]
Sets an initial seed value into the random number generator.

`ulong/uint GetLimbBits (`*ulong/uint* `bitCount`) [Method]
Generates a uniformly distributed random number of `bitCount` bits, i.e. in the range 0 to $2^{bitCount-1}$ inclusive.

`ulong/uint GetLimb (`*ulong/uint* `max`) [Method]
Generates a uniformly distributed random number in the range 0 to $max - 1$ inclusive.

`IntegerExpression GetIntBits (`*ulong/uint* `bitCount`) [Method]
`IntegerExpression GetIntBitsChunky (`*ulong/uint* `bitCount`) [Method]
Returns an expression that generates a uniformly distributed random integer in the range 0 to $2^{bitCount-1}$, inclusive.

`IntegerExpression GetInt (`*IntegerExpression* `max`) [Method]
Returns an expression that generates a uniformly distributed random number in the range 0 to $max - 1$ inclusive.

`FloatExpression GetFloat ()` [Method]
Returns an expression that generates a uniformly distributed random float in the range $0 \leq n < 1$. As with all float expressions, precision of the destination is used when available.

`FloatExpression GetFloatBits (`*ulong/uint* `bitCount`) [Method]
Returns an expression that generates a uniformly distributed random float in the range $0 \leq n < 1$, with the specified number of significant mantissa bits.

`FloatExpression GetFloatChunky (`*int* `maxExponent`) [Method]
Returns an expression that generates a random float with long strings of zeros and ones in the binary representation, using the precision of the destination. The argument is the maximum absolute value for the exponent of the generated number, expressed in limbs.

`FloatExpression GetFloatLimbsChunky (`*long `limbCount`, int `maxExponent`*`)` [Method]
Returns an expression that generates a random float with long strings of zeros and ones in the binary representation, and the specified number of significant limbs in the mantissa.

13.7 MPIR.Net Settings

`MpirSettings` [Static Class]
This static class contains several members that describe or control various default behaviors of the other MPIR.Net classes.

`int BITS_PER_LIMB` [Constant]
Represents the total number of bits in a single MPIR limb, including data bits and nail bits. This will be either 32 or 64, depending on your selected build architecture.

`int NAIL_BITS_PER_LIMB` [Constant]
Represents the number of nail bits in a single MPIR limb. Nail bits are used internally by MPIR.

`int USABLE_BITS_PER_LIMB` [Constant]
Represents the number of data bits in a single MPIR limb.

`Version MPIR_VERSION` [Constant]
Represents the version of the underlying MPIR library

`Version GMP_VERSION` [Constant]
Represents the version of GMP with which the underlying MPIR library is compatible

`RoundingModes RoundingMode` [Static Property]
Gets or sets the default rounding mode used for MPIR integer division operations that don't explicitly specify a rounding mode. Does not affect rational or float operations. The default value is `Truncate`.

`int ToStringDigits` [Static Property]
Gets or sets the maximum number of digits the `object.ToString()` method override will output. If an integer number is longer than this number of digits, it will be output as `"[-]...NNNNN"` with the least significant digits shown. Rational numbers apply the limit separately to the numerator and denominator. Floats output the most significant digits, and there is no ellipsis.

The primary purpose of this setting is to prevent accidental allocation of large memory blocks while inspecting variables in the debugger. The default value is 256. Setting this property to 0 causes all digits to be output.

14 Custom Allocation

By default MPIR uses `malloc`, `realloc` and `free` for memory allocation, and if they fail MPIR prints a message to the standard error output and terminates the program.

Alternate functions can be specified, to allocate memory in a different way or to have a different error action on running out of memory.

`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 used.

These functions will be used for all memory allocation done by MPIR, apart from temporary space from `alloca` if that function is available and MPIR is configured to use it (see Section 2.1 [Build Options], page 3).

Be sure to call `mp_set_memory_functions` only when there are no active MPIR objects allocated using the previous memory functions! Usually that means calling it before any other MPIR function.

The functions supplied should fit the following declarations:

`void * allocate_function (`*size_t `alloc_size`*`)` [Function]

Return a pointer to newly allocated space with at least *alloc_size* bytes.

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

Resize a previously allocated block *ptr* of *old_size* bytes to be *new_size* bytes.

The block may be moved if necessary or if desired, and in that case the smaller of *old_size* and *new_size* bytes must be copied to the new location. The return value is a pointer to the resized block, that being the new location if moved or just *ptr* if not.

ptr is never `NULL`, it's always a previously allocated block. *new_size* may be bigger or smaller than *old_size*.

`void free_function (`*void *`ptr`, size_t `size`*`)` [Function]

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

ptr is never `NULL`, it's always a previously allocated block of *size* bytes.

A *byte* here means the unit used by the `sizeof` operator.

The *old_size* parameters to *reallocate_function* and *free_function* are passed for convenience, but of course can be ignored if not needed. The default functions using `malloc` and friends for instance don't use them.

No error return is allowed from any of these functions, if they return then they must have performed the specified operation. In particular note that *allocate_function* or *reallocate_function* mustn't return `NULL`.

Getting a different fatal error action is a good use for custom allocation functions, for example giving a graphical dialog rather than the default print to `stderr`. How much is possible when genuinely out of memory is another question though.

There's currently no defined way for the allocation functions to recover from an error such as out of memory, they must terminate program execution. A `longjmp` or throwing a C++ exception will have undefined results. This may change in the future.

MPIR may use allocated blocks to hold pointers to other allocated blocks. This will limit the assumptions a conservative garbage collection scheme can make.

Any custom allocation functions must align pointers to limb boundaries. Thus if a limb is eight bytes (e.g. on x86_64), then all blocks must be aligned to eight byte boundaries. Check the configuration options for the custom allocation library in use. It is not necessary to align blocks to SSE boundaries even when SSE code is used. All MPIR assembly routines assume limb boundary alignment only (which is the default for most standard memory managers).

Since the default MPIR allocation uses `malloc` and friends, those functions will be linked in even if the first thing a program does is an `mp_set_memory_functions`. It's necessary to change the MPIR sources if this is a problem.

`void mp_get_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))*

Get the current allocation functions, storing function pointers to the locations given by the arguments. If an argument is `NULL`, that function pointer is not stored.

For example, to get just the current free function,

```
void (*freefunc) (void *, size_t);

mp_get_memory_functions (NULL, NULL, &freefunc);
```

15 Language Bindings

The following packages and projects offer access to MPIR from languages other than C, though perhaps with varying levels of functionality and efficiency.

C++

- MPIR C++ class interface, see Chapter 12 [C++ Class Interface], page 78, Straightforward interface, expression templates to eliminate temporaries.
- ALP `http://www-sop.inria.fr/saga/logiciels/ALP/`
  Linear algebra and polynomials using templates.
- CLN `http://www.ginac.de/CLN/`
  High level classes for arithmetic.
- LiDIA `http://www.informatik.tu-darmstadt.de/TI/LiDIA/`
  A C++ library for computational number theory.
- Linbox `http://www.linalg.org/`
  Sparse vectors and matrices.
- NTL `http://www.shoup.net/ntl/`
  A C++ number theory library.

Eiffel

- Eiffel Interface `http://www.eiffelroom.org/node/407`
  An Eiffel Interface to MPFR, MPC and MPIR by Chris Saunders.

Fortran

- Omni F77 `http://phase.hpcc.jp/Omni/home.html`
  Arbitrary precision floats.

Haskell

- Glasgow Haskell Compiler `http://www.haskell.org/ghc/`

Java

- Kaffe `http://www.kaffe.org/`

Lisp

- Embeddable Common Lisp `http://ecls.sourceforge.net/download.html`
- GNU Common Lisp `http://www.gnu.org/software/gcl/gcl.html`
- Librep `http://librep.sourceforge.net/`
- XEmacs (21.5.18 beta and up) `http://www.xemacs.org`
  Optional big integers, rationals and floats using MPIR.

M4

- GNU m4 betas `http://www.seindal.dk/rene/gnu/`
  Optionally provides an arbitrary precision `mpeval`.

ML

- MLton compiler `http://mlton.org/`

Objective Caml

- Numerix `http://pauillac.inria.fr/~quercia/`
  Optionally using GMP.

Oz

- Mozart `http://www.mozart-oz.org/`

Pascal

- GNU Pascal Compiler `http://www.gnu-pascal.de/`
  GMP unit.
- Numerix `http://pauillac.inria.fr/~quercia/`
  For Free Pascal, optionally using GMP.

Perl

- GMP module, see `demos/perl` on the MPIR website.
- Math::GMP `http://www.cpan.org/`
  Compatible with Math::BigInt, but not as many functions as the GMP module above.
- Math::BigInt::GMP `http://www.cpan.org/`
  Plug Math::GMP into normal Math::BigInt operations.

PHP

- mpz module in the main distribution, `http://php.net/`

Pike

- mpz module in the standard distribution, `http://pike.ida.liu.se/`

Prolog

- SWI Prolog `http://www.swi-prolog.org/`
  Arbitrary precision floats.

Python

- mpz module in the standard distribution, `http://www.python.org/`
- GMPY `http://gmpy.sourceforge.net/`

Scheme

- GNU Guile (upcoming 1.8) `http://www.gnu.org/software/guile/guile.html`
- RScheme `http://www.rscheme.org/`

Smalltalk

- GNU Smalltalk `http://www.smalltalk.org/versions/GNUSmalltalk.html`

Other

- ALGLIB `http://www.alglib.net/`
  Numerical analysis and data processing library.
- Axiom `http://savannah.nongnu.org/projects/axiom`
  Computer algebra using GCL.
- GiNaC `http://www.ginac.de/`
  C++ computer algebra using CLN.
- GOO `http://www.googoogaga.org/`
  Dynamic object oriented language.
- Maxima `http://www.ma.utexas.edu/users/wfs/maxima.html`
  Macsyma computer algebra using GCL.
- Q `http://q-lang.sourceforge.net/`
  Equational programming system.
- Regina `http://regina.sourceforge.net/`
  Topological calculator.
- Sage `http://www.sagemath.org/`
  Computer Algebra System written in Python and Cython.

- Yacas `http://yacas.sourceforge.net/homepage.html`
  Yet another computer algebra system.

16 Algorithms

This chapter is an introduction to some of the algorithms used for various MPIR operations. The code is likely to be hard to understand without knowing something about the algorithms.

Some MPIR internals are mentioned, but applications that expect to be compatible with future MPIR releases should take care to use only the documented functions.

16.1 Multiplication

N×N limb multiplications and squares are done using one of six algorithms, as the size N increases.

Algorithm	Mul Threshold
Basecase	(none)
Karatsuba	`MUL_KARATSUBA_THRESHOLD`
Toom-3	`MUL_TOOM3_THRESHOLD`
Toom-4	`MUL_TOOM4_THRESHOLD`
Toom-8(.5)	`MUL_TOOM8H_THRESHOLD`
FFT	`MUL_FFT_FULL_THRESHOLD`

Algorithm	Sqr Threshold
Basecase	(none)
Karatsuba	`SQR_KARATSUBA_THRESHOLD`
Toom-3	`SQR_TOOM3_THRESHOLD`
Toom-4	`SQR_TOOM4_THRESHOLD`
Toom-8	`SQR_TOOM8_THRESHOLD`
FFT	`SQR_FFT_FULL_THRESHOLD`

N×M multiplications of operands with different sizes above `MUL_KARATSUBA_THRESHOLD` are done using unbalanced Toom algorithms or with the FFT. See (see Section 16.1.7 [Unbalanced Multiplication], page 117).

16.1.1 Basecase Multiplication

Basecase N×M multiplication is a straightforward rectangular set of cross-products, the same as long multiplication done by hand and for that reason sometimes known as the schoolbook or grammar school method. This is an $O(NM)$ algorithm. See Knuth section 4.3.1 algorithm M (see Appendix B [References], page 145), and the `mpn/generic/mul_basecase.c` code.

Assembler implementations of `mpn_mul_basecase` are essentially the same as the generic C code, but have all the usual assembler tricks and obscurities introduced for speed.

A square can be done in roughly half the time of a multiply, by using the fact that the cross products above and below the diagonal are the same. A triangle of products below the diagonal is formed, doubled (left shift by one bit), and then the products on the diagonal added. This can be seen in `mpn/generic/sqr_basecase.c`. Again the assembler implementations take essentially the same approach.

	u0	u1	u2	u3	u4
u0	d				
u1		d			
u2			d		
u3				d	
u4					d

In practice squaring isn't a full 2× faster than multiplying, it's usually around 1.5×. Less than 1.5× probably indicates `mpn_sqr_basecase` wants improving on that CPU.

On some CPUs `mpn_mul_basecase` can be faster than the generic C `mpn_sqr_basecase` on some small sizes. `SQR_BASECASE_THRESHOLD` is the size at which to use `mpn_sqr_basecase`, this will be zero if that routine should be used always.

16.1.2 Karatsuba Multiplication

The Karatsuba multiplication algorithm is described in Knuth section 4.3.3 part A, and various other textbooks. A brief description is given here.

The inputs x and y are treated as each split into two parts of equal length (or the most significant part one limb shorter if N is odd).

high	low
x_1	x_0
y_1	y_0

Let b be the power of 2 where the split occurs, ie. if x_0 is k limbs (y_0 the same) then $b = 2^{k*\texttt{mp\_bits\_per\_limb}}$. With that $x = x_1 b + x_0$ and $y = y_1 b + y_0$, and the following holds,

$$xy \;=\; (b^2+b)x_1y_1 \;-\; b(x_1-x_0)(y_1-y_0) \;+\; (b+1)x_0y_0$$

This formula means doing only three multiplies of (N/2)×(N/2) limbs, whereas a basecase multiply of N×N limbs is equivalent to four multiplies of (N/2)×(N/2). The factors (b^2+b) etc represent the positions where the three products must be added.

high		low
x_1y_1		x_0y_0
+	x_1y_1	
+	x_0y_0	
−	$(x_1-x_0)(y_1-y_0)$	

The term $(x_1-x_0)(y_1-y_0)$ is best calculated as an absolute value, and the sign used to choose to add or subtract. Notice the sum $\text{high}(x_0y_0) + \text{low}(x_1y_1)$ occurs twice, so it's possible to do $5k$ limb additions, rather than $6k$, but in MPIR extra function call overheads outweigh the saving.

Squaring is similar to multiplying, but with $x = y$ the formula reduces to an equivalent with three squares,

$$x^2 \;=\; (b^2+b)x_1^2 \;-\; b(x_1-x_0)^2 \;+\; (b+1)x_0^2$$

The final result is accumulated from those three squares the same way as for the three multiplies above. The middle term $(x_1-x_0)^2$ is now always positive.

A similar formula for both multiplying and squaring can be constructed with a middle term $(x_1+x_0)(y_1+y_0)$. But those sums can exceed k limbs, leading to more carry handling and additions than the form above.

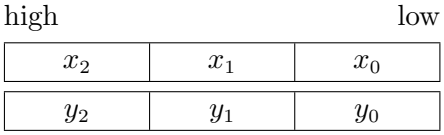
Karatsuba multiplication is asymptotically an $O(N^{1.585})$ algorithm, the exponent being $\log 3/\log 2$, representing 3 multiplies each 1/2 the size of the inputs. This is a big improvement over the basecase multiply at $O(N^2)$ and the advantage soon overcomes the extra additions Karatsuba performs. `MUL_KARATSUBA_THRESHOLD` can be as little as 10 limbs. The `SQR` threshold is usually about twice the `MUL`.

The basecase algorithm will take a time of the form $M(N) = aN^2 + bN + c$ and the Karatsuba algorithm $K(N) = 3M(N/2)+dN+e$, which expands to $K(N) = \frac{3}{4}aN^2+\frac{3}{2}bN+3c+dN+e$. The factor $\frac{3}{4}$ for a means per-crossproduct speedups in the basecase code will increase the threshold since they benefit $M(N)$ more than $K(N)$. And conversely the $\frac{3}{2}$ for b means linear style speedups of b will increase the threshold since they benefit $K(N)$ more than $M(N)$. The latter can be seen for instance when adding an optimized `mpn_sqr_diagonal` to `mpn_sqr_basecase`. Of course all speedups reduce total time, and in that sense the algorithm thresholds are merely of academic interest.

16.1.3 Toom 3-Way Multiplication

The Karatsuba formula is the simplest case of a general approach to splitting inputs that leads to both Toom and FFT algorithms. A description of Toom can be found in Knuth section 4.3.3, with an example 3-way calculation after Theorem A. The 3-way form used in MPIR is described here.

The operands are each considered split into 3 pieces of equal length (or the most significant part 1 or 2 limbs shorter than the other two).



These parts are treated as the coefficients of two polynomials

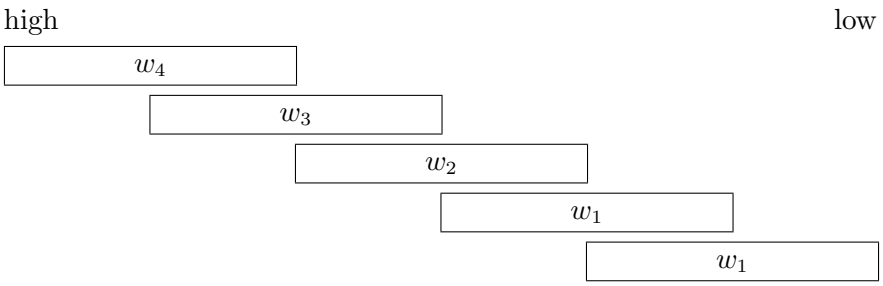
$$X(t) = x_2t^2 + x_1t + x_0$$
$$Y(t) = y_2t^2 + y_1t + y_0$$

Let b equal the power of 2 which is the size of the x_0, x_1, y_0 and y_1 pieces, ie. if they're k limbs each then $b = 2^{k*\texttt{mp\_bits\_per\_limb}}$. With this $x = X(b)$ and $y = Y(b)$.

Let a polynomial $W(t) = X(t)Y(t)$ and suppose its coefficients are

$$W(t) = w_4t^4 + w_3t^3 + w_2t^2 + w_1t + w_0$$

The w_i are going to be determined, and when they are they'll give the final result using $w = W(b)$, since $xy = X(b)Y(b)$. The coefficients will be roughly b^2 each, and the final $W(b)$ will be an addition like,



The w_i coefficients could be formed by a simple set of cross products, like $w_4 = x_2y_2$, $w_3 = x_2y_1 + x_1y_2$, $w_2 = x_2y_0 + x_1y_1 + x_0y_2$ etc, but this would need all nine x_iy_j for $i, j = 0, 1, 2$, and would be equivalent merely to a basecase multiply. Instead the following approach is used.

$X(t)$ and $Y(t)$ are evaluated and multiplied at 5 points, giving values of $W(t)$ at those points. In MPIR the following points are used,

Point	Value
$t = 0$	x_0y_0, which gives w_0 immediately
$t = 1$	$(x_2 + x_1 + x_0)(y_2 + y_1 + y_0)$
$t = -1$	$(x_2 - x_1 + x_0)(y_2 - y_1 + y_0)$
$t = 2$	$(4x_2 + 2x_1 + x_0)(4y_2 + 2y_1 + y_0)$
$t = \infty$	x_2y_2, which gives w_4 immediately

At $t = -1$ the values can be negative and that's handled using the absolute values and tracking the sign separately. At $t = \infty$ the value is actually $\lim_{t\to\infty} \frac{X(t)Y(t)}{t^4}$, but it's much easier to think of as simply x_2y_2 giving w_4 immediately (much like x_0y_0 at $t = 0$ gives w_0 immediately).

Each of the points substituted into $W(t) = w_4t^4 + \cdots + w_0$ gives a linear combination of the w_i coefficients, and the value of those combinations has just been calculated.

$$\begin{array}{rcccccccccc}
W(0) &=& & & & & & & & & w_0 \\
W(1) &=& w_4 &+& w_3 &+& w_2 &+& w_1 &+& w_0 \\
W(-1) &=& w_4 &-& w_3 &+& w_2 &-& w_1 &+& w_0 \\
W(2) &=& 16w_4 &+& 8w_3 &+& 4w_2 &+& 2w_1 &+& w_0 \\
W(\infty) &=& w_4 & & & & & & & &
\end{array}$$

This is a set of five equations in five unknowns, and some elementary linear algebra quickly isolates each w_i. This involves adding or subtracting one $W(t)$ value from another, and a couple of divisions by powers of 2 and one division by 3, the latter using the special `mpn_divexact_by3` (see Section 16.2.4 [Exact Division], page 119).

The conversion of $W(t)$ values to the coefficients is interpolation. A polynomial of degree 4 like $W(t)$ is uniquely determined by values known at 5 different points. The points are arbitrary and can be chosen to make the linear equations come out with a convenient set of steps for quickly isolating the w_i.

Squaring follows the same procedure as multiplication, but there's only one $X(t)$ and it's evaluated at the 5 points, and those values squared to give values of $W(t)$. The interpolation is then identical, and in fact the same `toom3_interpolate` subroutine is used for both squaring and multiplying.

Toom-3 is asymptotically $O(N^{1.465})$, the exponent being $\log 5/\log 3$, representing 5 recursive multiplies of $1/3$ the original size each. This is an improvement over Karatsuba at $O(N^{1.585})$, though Toom does more work in the evaluation and interpolation and so it only realizes its advantage above a certain size.

Near the crossover between Toom-3 and Karatsuba there's generally a range of sizes where the difference between the two is small. `MUL_TOOM3_THRESHOLD` is a somewhat arbitrary point in that range and successive runs of the tune program can give different values due to small variations in measuring. A graph of time versus size for the two shows the effect, see `tune/README`.

At the fairly small sizes where the Toom-3 thresholds occur it's worth remembering that the asymptotic behaviour for Karatsuba and Toom-3 can't be expected to make accurate predictions, due of course to the big influence of all sorts of overheads, and the fact that only a few recursions

of each are being performed. Even at large sizes there's a good chance machine dependent effects like cache architecture will mean actual performance deviates from what might be predicted.

The formula given for the Karatsuba algorithm (see Section 16.1.2 [Karatsuba Multiplication], page 112) has an equivalent for Toom-3 involving only five multiplies, but this would be complicated and unenlightening.

An alternate view of Toom-3 can be found in Zuras (see Appendix B [References], page 145), using a vector to represent the x and y splits and a matrix multiplication for the evaluation and interpolation stages. The matrix inverses are not meant to be actually used, and they have elements with values much greater than in fact arise in the interpolation steps. The diagram shown for the 3-way is attractive, but again doesn't have to be implemented that way and for example with a bit of rearrangement just one division by 6 can be done.

16.1.4 Toom 4-Way Multiplication

Karatsuba and Toom-3 split the operands into 2 and 3 coefficients, respectively. Toom-4 analogously splits the operands into 4 coefficients. Using the notation from the section on Toom-3 multiplication, we form two polynomials:

$$X(t) = x_3t^3 + x_2t^2 + x_1t + x_0$$
$$Y(t) = y_3t^3 + y_2t^2 + y_1t + y_0$$

$X(t)$ and $Y(t)$ are evaluated and multiplied at 7 points, giving values of $W(t)$ at those points. In MPIR the following points are used,

Point	Value
$t = 0$	x_0y_0, which gives w_0 immediately
$t = 1/2$	$(x_3 + 2x_2 + 4x_1 + 8x_0)(y_3 + 2y_2 + 4y_1 + 8y_0)$
$t = -1/2$	$(-x_3 + 2x_2 - 4x_1 + 8x_0)(-y_3 + 2y_2 - 4y_1 + 8y_0)$
$t = 1$	$(x_3 + x_2 + x_1 + x_0)(y_3 + y_2 + y_1 + y_0)$
$t = -1$	$(-x_3 + x_2 - x_1 + x_0)(-y_3 + y_2 - y_1 + y_0)$
$t = 2$	$(8x_3 + 4x_2 + 2x_1 + x_0)(8y_3 + 4y_2 + 2y_1 + y_0)$
$t = \infty$	x_3y_3, which gives w_6 immediately

The number of additions and subtractions for Toom-4 is much larger than for Toom-3. But several subexpressions occur multiple times, for example $x_2 + x_0$, occurs for both $t = 1$ and $t = -1$.

Toom-4 is asymptotically $O(N^{1.404})$, the exponent being $\log 7 / \log 4$, representing 7 recursive multiplies of $1/4$ the original size each.

16.1.5 FFT Multiplication

This section is out-of-date and will be updated when the new FFT is added.

At large to very large sizes a Fermat style FFT multiplication is used, following Schönhage and Strassen (see Appendix B [References], page 145). Descriptions of FFTs in various forms can be found in many textbooks, for instance Knuth section 4.3.3 part C or Lipson chapter IX. A brief description of the form used in MPIR is given here.

The multiplication done is $xy \bmod 2^N + 1$, for a given N. A full product xy is obtained by choosing $N \geq \text{bits}(x) + \text{bits}(y)$ and padding x and y with high zero limbs. The modular product is the native form for the algorithm, so padding to get a full product is unavoidable.

The algorithm follows a split, evaluate, pointwise multiply, interpolate and combine similar to that described above for Karatsuba and Toom-3. A k parameter controls the split, with an FFT-

k splitting into 2^k pieces of $M = N/2^k$ bits each. N must be a multiple of $2^k \times$ `mp_bits_per_limb` so the split falls on limb boundaries, avoiding bit shifts in the split and combine stages.

The evaluations, pointwise multiplications, and interpolation, are all done modulo $2^{N'}+1$ where N' is $2M + k + 3$ rounded up to a multiple of 2^k and of `mp_bits_per_limb`. The results of interpolation will be the following negacyclic convolution of the input pieces, and the choice of N' ensures these sums aren't truncated.

$$w_n = \sum_{\substack{i+j=b2^k+n \\ b=0,1}} (-1)^b x_i y_j$$

The points used for the evaluation are g^i for $i = 0$ to $2^k - 1$ where $g = 2^{2N'/2^k}$. g is a 2^kth root of unity mod $2^{N'}+1$, which produces necessary cancellations at the interpolation stage, and it's also a power of 2 so the fast fourier transforms used for the evaluation and interpolation do only shifts, adds and negations.

The pointwise multiplications are done modulo $2^{N'} + 1$ and either recurse into a further FFT or use a plain multiplication (Toom-3, Karatsuba or basecase), whichever is optimal at the size N'. The interpolation is an inverse fast fourier transform. The resulting set of sums of $x_i y_j$ are added at appropriate offsets to give the final result.

Squaring is the same, but x is the only input so it's one transform at the evaluate stage and the pointwise multiplies are squares. The interpolation is the same.

For a mod $2^N + 1$ product, an FFT-k is an $O(N^{k/(k-1)})$ algorithm, the exponent representing 2^k recursed modular multiplies each $1/2^{k-1}$ the size of the original. Each successive k is an asymptotic improvement, but overheads mean each is only faster at bigger and bigger sizes. In the code, `MUL_FFT_TABLE` and `SQR_FFT_TABLE` are the thresholds where each k is used. Each new k effectively swaps some multiplying for some shifts, adds and overheads.

A mod 2^N+1 product can be formed with a normal $N \times N \to 2N$ bit multiply plus a subtraction, so an FFT and Toom-3 etc can be compared directly. A $k = 4$ FFT at $O(N^{1.333})$ can be expected to be the first faster than Toom-3 at $O(N^{1.465})$. In practice this is what's found, with `MUL_FFT_MODF_THRESHOLD` and `SQR_FFT_MODF_THRESHOLD` being between 300 and 1000 limbs, depending on the CPU. So far it's been found that only very large FFTs recurse into pointwise multiplies above these sizes.

When an FFT is to give a full product, the change of N to $2N$ doesn't alter the theoretical complexity for a given k, but for the purposes of considering where an FFT might be first used it can be assumed that the FFT is recursing into a normal multiply and that on that basis it's doing 2^k recursed multiplies each $1/2^{k-2}$ the size of the inputs, making it $O(N^{k/(k-2)})$. This would mean $k = 7$ at $O(N^{1.4})$ would be the first FFT faster than Toom-3. In practice `MUL_FFT_FULL_THRESHOLD` and `SQR_FFT_FULL_THRESHOLD` have been found to be in the $k = 8$ range, somewhere between 3000 and 10000 limbs.

The way N is split into 2^k pieces and then $2M + k + 3$ is rounded up to a multiple of 2^k and `mp_bits_per_limb` means that when $2^k \geq$ `mp_bits_per_limb` the effective N is a multiple of 2^{2k-1} bits. The $+k + 3$ means some values of N just under such a multiple will be rounded to the next. The complexity calculations above assume that a favourable size is used, meaning one which isn't padded through rounding, and it's also assumed that the extra $+k + 3$ bits are negligible at typical FFT sizes.

The practical effect of the 2^{2k-1} constraint is to introduce a step-effect into measured speeds. For example $k = 8$ will round N up to a multiple of 32768 bits, so for a 32-bit limb there'll be 512 limb groups of sizes for which `mpn_mul_n` runs at the same speed. Or for $k = 9$ groups of 2048 limbs, $k = 10$ groups of 8192 limbs, etc. In practice it's been found each k is used at quite

small multiples of its size constraint and so the step effect is quite noticeable in a time versus size graph.

The threshold determinations currently measure at the mid-points of size steps, but this is sub-optimal since at the start of a new step it can happen that it's better to go back to the previous k for a while. Something more sophisticated for `MUL_FFT_TABLE` and `SQR_FFT_TABLE` will be needed.

16.1.6 Other Multiplication

The Toom algorithms described above (see Section 16.1.3 [Toom 3-Way Multiplication], page 113), see Section 16.1.4 [Toom 4-Way Multiplication], page 115) generalize to split into an arbitrary number of pieces, as per Knuth section 4.3.3 algorithm C. MPIR currently implements Toom 8 routines.

These are generated automatically via a technique due to Bodrato (see Appendix B [References], page 145) which mixes evaluation, pointwise multiplication and interpolation phases. The routine used is called Toom 8.5. See Bodrato's paper.

For general Toom-n a split into $r + 1$ pieces is made, and evaluations and pointwise multiplications done at $2r + 1$ points. A 4-way split does 7 pointwise multiplies, 5-way does 9, etc. Asymptotically an $(r + 1)$-way algorithm is $O(N^{log(2r+1)/log(r+1)})$. Only the pointwise multiplications count towards big-O complexity, but the time spent in the evaluate and interpolate stages grows with r and has a significant practical impact, with the asymptotic advantage of each r realized only at bigger and bigger sizes. The overheads grow as $O(Nr)$, whereas in an $r = 2^k$ FFT they grow only as $O(N \log r)$.

Knuth algorithm C evaluates at points 0,1,2,. . .,$2r$, but exercise 4 uses $-r$,. . .,0,. . .,r and the latter saves some small multiplies in the evaluate stage (or rather trades them for additions), and has a further saving of nearly half the interpolate steps. The idea is to separate odd and even final coefficients and then perform algorithm C steps C7 and C8 on them separately. The divisors at step C7 become j^2 and the multipliers at C8 become $2tj - j^2$.

Splitting odd and even parts through positive and negative points can be thought of as using -1 as a square root of unity. If a 4th root of unity was available then a further split and speedup would be possible, but no such root exists for plain integers. Going to complex integers with $i = \sqrt{-1}$ doesn't help, essentially because in cartesian form it takes three real multiplies to do a complex multiply. The existence of 2^kth roots of unity in a suitable ring or field lets the fast fourier transform keep splitting and get to $O(N \log r)$.

Floating point FFTs use complex numbers approximating Nth roots of unity. Some processors have special support for such FFTs. But these are not used in MPIR since it's very difficult to guarantee an exact result (to some number of bits). An occasional difference of 1 in the last bit might not matter to a typical signal processing algorithm, but is of course of vital importance to MPIR.

16.1.7 Unbalanced Multiplication

Multiplication of operands with different sizes, both below `MUL_KARATSUBA_THRESHOLD` are done with plain schoolbook multiplication (see Section 16.1.1 [Basecase Multiplication], page 111).

For really large operands, we invoke the FFT directly.

For operands between these sizes, we use Toom inspired algorithms suggested by Alberto Zanoni and Marco Bodrato. The idea is to split the operands into polynomials of different degree. These algorithms are denoted ToomMN where the first input is broken into M components and the second operand is broken into N components. MPIR currently implements Toom32, Toom33,

Toom44, Toom53 and Toom8h which deals with a variety of sizes where the product polynomial will have length 15 or 16.

16.2 Division Algorithms

16.2.1 Single Limb Division

N×1 division is implemented using repeated 2×1 divisions from high to low, either with a hardware divide instruction or a multiplication by inverse, whichever is best on a given CPU.

The multiply by inverse follows section 8 of "Division by Invariant Integers using Multiplication" by Granlund and Montgomery (see Appendix B [References], page 145) and is implemented as `udiv_qrnnd_preinv` in `gmp-impl.h`. The idea is to have a fixed-point approximation to $1/d$ (see `invert_limb`) and then multiply by the high limb (plus one bit) of the dividend to get a quotient q. With d normalized (high bit set), q is no more than 1 too small. Subtracting qd from the dividend gives a remainder, and reveals whether q or $q-1$ is correct.

The result is a division done with two multiplications and four or five arithmetic operations. On CPUs with low latency multipliers this can be much faster than a hardware divide, though the cost of calculating the inverse at the start may mean it's only better on inputs bigger than say 4 or 5 limbs.

When a divisor must be normalized, either for the generic C `__udiv_qrnnd_c` or the multiply by inverse, the division performed is actually $a2^k$ by $d2^k$ where a is the dividend and k is the power necessary to have the high bit of $d2^k$ set. The bit shifts for the dividend are usually accomplished "on the fly" meaning by extracting the appropriate bits at each step. Done this way the quotient limbs come out aligned ready to store. When only the remainder is wanted, an alternative is to take the dividend limbs unshifted and calculate $r = a \bmod d2^k$ followed by an extra final step $r2^k \bmod d2^k$. This can help on CPUs with poor bit shifts or few registers.

The multiply by inverse can be done two limbs at a time. The calculation is basically the same, but the inverse is two limbs and the divisor treated as if padded with a low zero limb. This means more work, since the inverse will need a 2×2 multiply, but the four 1×1s to do that are independent and can therefore be done partly or wholly in parallel. Likewise for a 2×1 calculating qd. The net effect is to process two limbs with roughly the same two multiplies worth of latency that one limb at a time gives. This extends to 3 or 4 limbs at a time, though the extra work to apply the inverse will almost certainly soon reach the limits of multiplier throughput.

A similar approach in reverse can be taken to process just half a limb at a time if the divisor is only a half limb. In this case the 1×1 multiply for the inverse effectively becomes two $\frac{1}{2\times1}$ for each limb, which can be a saving on CPUs with a fast half limb multiply, or in fact if the only multiply is a half limb, and especially if it's not pipelined.

16.2.2 Basecase Division

This section is out-of-date.

Basecase N×M division is like long division done by hand, but in base $2^{\texttt{mp\_bits\_per\_limb}}$. See Knuth section 4.3.1 algorithm D.

Briefly stated, while the dividend remains larger than the divisor, a high quotient limb is formed and the N×1 product qd subtracted at the top end of the dividend. With a normalized divisor (most significant bit set), each quotient limb can be formed with a 2×1 division and a 1×1 multiplication plus some subtractions. The 2×1 division is by the high limb of the divisor and is done either with a hardware divide or a multiply by inverse (the same as in Section 16.2.1

[Single Limb Division], page 118) whichever is faster. Such a quotient is sometimes one too big, requiring an addback of the divisor, but that happens rarely.

With Q=N−M being the number of quotient limbs, this is an $O(QM)$ algorithm and will run at a speed similar to a basecase Q×M multiplication, differing in fact only in the extra multiply and divide for each of the Q quotient limbs.

16.2.3 Divide and Conquer Division

This section is out-of-date

For divisors larger than `DIV_DC_THRESHOLD`, division is done by dividing. Or to be precise by a recursive divide and conquer algorithm based on work by Moenck and Borodin, Jebelean, and Burnikel and Ziegler (see Appendix B [References], page 145).

The algorithm consists essentially of recognising that a 2N×N division can be done with the basecase division algorithm (see Section 16.2.2 [Basecase Division], page 118), but using N/2 limbs as a base, not just a single limb. This way the multiplications that arise are (N/2)×(N/2) and can take advantage of Karatsuba and higher multiplication algorithms (see Section 16.1 [Multiplication Algorithms], page 111). The two "digits" of the quotient are formed by recursive N×(N/2) divisions.

If the (N/2)×(N/2) multiplies are done with a basecase multiplication then the work is about the same as a basecase division, but with more function call overheads and with some subtractions separated from the multiplies. These overheads mean that it's only when N/2 is above `MUL_KARATSUBA_THRESHOLD` that divide and conquer is of use.

`DIV_DC_THRESHOLD` is based on the divisor size N, so it will be somewhere above twice `MUL_KARATSUBA_THRESHOLD`, but how much above depends on the CPU. An optimized `mpn_mul_basecase` can lower `DIV_DC_THRESHOLD` a little by offering a ready-made advantage over repeated `mpn_submul_1` calls.

Divide and conquer is asymptotically $O(M(N)\log N)$ where $M(N)$ is the time for an N×N multiplication done with FFTs. The actual time is a sum over multiplications of the recursed sizes, as can be seen near the end of section 2.2 of Burnikel and Ziegler. For example, within the Toom-3 range, divide and conquer is $2.63M(N)$. With higher algorithms the $M(N)$ term improves and the multiplier tends to $\log N$. In practice, at moderate to large sizes, a 2N×N division is about 2 to 4 times slower than an N×N multiplication.

Newton's method used for division is asymptotically $O(M(N))$ and should therefore be superior to divide and conquer, but it's believed this would only be for large to very large N.

16.2.4 Exact Division

This section is out-of-date

A so-called exact division is when the dividend is known to be an exact multiple of the divisor. Jebelean's exact division algorithm uses this knowledge to make some significant optimizations (see Appendix B [References], page 145).

The idea can be illustrated in decimal for example with 368154 divided by 543. Because the low digit of the dividend is 4, the low digit of the quotient must be 8. This is arrived at from $4\times7 \bmod 10$, using the fact 7 is the modular inverse of 3 (the low digit of the divisor), since $3\times7 \equiv 1 \bmod 10$. So $8\times543 = 4344$ can be subtracted from the dividend leaving 363810. Notice the low digit has become zero.

The procedure is repeated at the second digit, with the next quotient digit 7 (1×7 mod 10), subtracting 7×543 = 3801, leaving 325800. And finally at the third digit with quotient digit 6 (8×7 mod 10), subtracting 6×543 = 3258 leaving 0. So the quotient is 678.

Notice however that the multiplies and subtractions don't need to extend past the low three digits of the dividend, since that's enough to determine the three quotient digits. For the last quotient digit no subtraction is needed at all. On a 2N×N division like this one, only about half the work of a normal basecase division is necessary.

For an N×M exact division producing Q=N−M quotient limbs, the saving over a normal basecase division is in two parts. Firstly, each of the Q quotient limbs needs only one multiply, not a 2×1 divide and multiply. Secondly, the crossproducts are reduced when $Q > M$ to $QM-M(M+1)/2$, or when $Q \leq M$ to $Q(Q-1)/2$. Notice the savings are complementary. If Q is big then many divisions are saved, or if Q is small then the crossproducts reduce to a small number.

The modular inverse used is calculated efficiently by `modlimb_invert` in `gmp-impl.h`. This does four multiplies for a 32-bit limb, or six for a 64-bit limb. `tune/modlinv.c` has some alternate implementations that might suit processors better at bit twiddling than multiplying.

The sub-quadratic exact division described by Jebelean in "Exact Division with Karatsuba Complexity" is not currently implemented. It uses a rearrangement similar to the divide and conquer for normal division (see Section 16.2.3 [Divide and Conquer Division], page 119), but operating from low to high. A further possibility not currently implemented is "Bidirectional Exact Integer Division" by Krandick and Jebelean which forms quotient limbs from both the high and low ends of the dividend, and can halve once more the number of crossproducts needed in a 2N×N division.

A special case exact division by 3 exists in `mpn_divexact_by3`, supporting Toom-3 multiplication and `mpq` canonicalizations. It forms quotient digits with a multiply by the modular inverse of 3 (which is `0xAA..AAB`) and uses two comparisons to determine a borrow for the next limb. The multiplications don't need to be on the dependent chain, as long as the effect of the borrows is applied, which can help chips with pipelined multipliers.

16.2.5 Exact Remainder

If the exact division algorithm is done with a full subtraction at each stage and the dividend isn't a multiple of the divisor, then low zero limbs are produced but with a remainder in the high limbs. For dividend a, divisor d, quotient q, and $b = 2^{\texttt{mp\_bits\_per\_limb}}$, this remainder r is of the form

$$a = qd + rb^n$$

n represents the number of zero limbs produced by the subtractions, that being the number of limbs produced for q. r will be in the range $0 \leq r < d$ and can be viewed as a remainder, but one shifted up by a factor of b^n.

Carrying out full subtractions at each stage means the same number of cross products must be done as a normal division, but there's still some single limb divisions saved. When d is a single limb some simplifications arise, providing good speedups on a number of processors.

`mpn_bdivmod`, `mpn_divexact_by3`, `mpn_modexact_1_odd` and the `redc` function in `mpz_powm` differ subtly in how they return r, leading to some negations in the above formula, but all are essentially the same.

Clearly r is zero when a is a multiple of d, and this leads to divisibility or congruence tests which are potentially more efficient than a normal division.

The factor of b^n on r can be ignored in a GCD when d is odd, hence the use of `mpn_bdivmod` in `mpn_gcd`, and the use of `mpn_modexact_1_odd` by `mpn_gcd_1` and `mpz_kronecker_ui` etc (see Section 16.3 [Greatest Common Divisor Algorithms], page 121).

Montgomery's REDC method for modular multiplications uses operands of the form of xb^{-n} and yb^{-n} and on calculating $(xb^{-n})(yb^{-n})$ uses the factor of b^n in the exact remainder to reach a product in the same form $(xy)b^{-n}$ (see Section 16.4.2 [Modular Powering Algorithm], page 124).

Notice that r generally gives no useful information about the ordinary remainder $a \bmod d$ since $b^n \bmod d$ could be anything. If however $b^n \equiv 1 \bmod d$, then r is the negative of the ordinary remainder. This occurs whenever d is a factor of $b^n - 1$, as for example with 3 in `mpn_divexact_by3`. For a 32 or 64 bit limb other such factors include 5, 17 and 257, but no particular use has been found for this.

16.2.6 Small Quotient Division

An N×M division where the number of quotient limbs Q=N−M is small can be optimized somewhat.

An ordinary basecase division normalizes the divisor by shifting it to make the high bit set, shifting the dividend accordingly, and shifting the remainder back down at the end of the calculation. This is wasteful if only a few quotient limbs are to be formed. Instead a division of just the top 2Q limbs of the dividend by the top Q limbs of the divisor can be used to form a trial quotient. This requires only those limbs normalized, not the whole of the divisor and dividend.

A multiply and subtract then applies the trial quotient to the M−Q unused limbs of the divisor and N−Q dividend limbs (which includes Q limbs remaining from the trial quotient division). The starting trial quotient can be 1 or 2 too big, but all cases of 2 too big and most cases of 1 too big are detected by first comparing the most significant limbs that will arise from the subtraction. An addback is done if the quotient still turns out to be 1 too big.

This whole procedure is essentially the same as one step of the basecase algorithm done in a Q limb base, though with the trial quotient test done only with the high limbs, not an entire Q limb "digit" product. The correctness of this weaker test can be established by following the argument of Knuth section 4.3.1 exercise 20 but with the $v_2\hat{q} > b\hat{r} + u_2$ condition appropriately relaxed.

16.3 Greatest Common Divisor

16.3.1 Binary GCD

At small sizes MPIR uses an $O(N^2)$ binary style GCD. This is described in many textbooks, for example Knuth section 4.5.2 algorithm B. It simply consists of successively reducing odd operands a and b using

$a, b = \text{abs}\,(a - b), \min\,(a, b)$
strip factors of 2 from a

The Euclidean GCD algorithm, as per Knuth algorithms E and A, reduces using $a \bmod b$ but this has so far been found to be slower everywhere. One reason the binary method does well is that the implied quotient at each step is usually small, so often only one or two subtractions are needed to get the same effect as a division. Quotients 1, 2 and 3 for example occur 67.7% of the time, see Knuth section 4.5.3 Theorem E.

When the implied quotient is large, meaning b is much smaller than a, then a division is worthwhile. This is the basis for the initial $a \bmod b$ reductions in `mpn_gcd` and `mpn_gcd_1` (the latter

for both N×1 and 1×1 cases). But after that initial reduction, big quotients occur too rarely to make it worth checking for them.

The final 1×1 GCD in `mpn_gcd_1` is done in the generic C code as described above. For two N-bit operands, the algorithm takes about 0.68 iterations per bit. For optimum performance some attention needs to be paid to the way the factors of 2 are stripped from a.

Firstly it may be noted that in twos complement the number of low zero bits on $a - b$ is the same as $b - a$, so counting or testing can begin on $a - b$ without waiting for $\text{abs}\,(a - b)$ to be determined.

A loop stripping low zero bits tends not to branch predict well, since the condition is data dependent. But on average there's only a few low zeros, so an option is to strip one or two bits arithmetically then loop for more (as done for AMD K6). Or use a lookup table to get a count for several bits then loop for more (as done for AMD K7). An alternative approach is to keep just one of a or b odd and iterate

$a, b = \text{abs}\,(a - b), \min\,(a, b)$
$a = a/2$ if even
$b = b/2$ if even

This requires about 1.25 iterations per bit, but stripping of a single bit at each step avoids any branching. Repeating the bit strip reduces to about 0.9 iterations per bit, which may be a worthwhile tradeoff.

Generally with the above approaches a speed of perhaps 6 cycles per bit can be achieved, which is still not terribly fast with for instance a 64-bit GCD taking nearly 400 cycles. It's this sort of time which means it's not usually advantageous to combine a set of divisibility tests into a GCD.

16.3.2 Lehmer's GCD

Lehmer's improvement of the Euclidean algorithms is based on the observation that the initial part of the quotient sequence depends only on the most significant parts of the inputs. The variant of Lehmer's algorithm used in MPIR splits off the most significant two limbs, as suggested, e.g., in "A Double-Digit Lehmer-Euclid Algorithm" by Jebelean (see Appendix B [References], page 145). The quotients of two double-limb inputs are collected as a 2 by 2 matrix with single-limb elements. This is done by the function `mpn_hgcd2`. The resulting matrix is applied to the inputs using `mpn_mul_1` and `mpn_submul_1`. Each iteration usually reduces the inputs by almost one limb. In the rare case of a large quotient, no progress can be made by examining just the most significant two limbs, and the quotient is computing using plain division.

The resulting algorithm is asymptotically $O(N^2)$, just as the Euclidean algorithm and the binary algorithm. The quadratic part of the work are the calls to `mpn_mul_1` and `mpn_submul_1`. For small sizes, the linear work is also significant. There are roughly N calls to the `mpn_hgcd2` function. This function uses a couple of important optimizations:

- It uses the same relaxed notion of correctness as `mpn_hgcd` (see next section). This means that when called with the most significant two limbs of two large numbers, the returned matrix does not always correspond exactly to the initial quotient sequence for the two large numbers; the final quotient may sometimes be one off.
- It takes advantage of the fact the quotients are usually small. The division operator is not used, since the corresponding assembler instruction is very slow on most architectures. (This code could probably be improved further, it uses many branches that are unfriendly to prediction).

- It switches from double-limb calculations to single-limb calculations half-way through, when the input numbers have been reduced in size from two limbs to one and a half.

16.3.3 Subquadratic GCD

For inputs larger than `GCD_DC_THRESHOLD`, GCD is computed via the HGCD (Half GCD) function, as a generalization to Lehmer's algorithm.

Let the inputs a, b be of size N limbs each. Put $S = \lfloor N/2 \rfloor + 1$. Then HGCD(a,b) returns a transformation matrix T with non-negative elements, and reduced numbers $(c; d) = T^{-1}(a; b)$. The reduced numbers c, d must be larger than S limbs, while their difference $abs(c - d)$ must fit in S limbs. The matrix elements will also be of size roughly $N/2$.

The HGCD base case uses Lehmer's algorithm, but with the above stop condition that returns reduced numbers and the corresponding transformation matrix half-way through. For inputs larger than `HGCD_THRESHOLD`, HGCD is computed recursively, using the divide and conquer algorithm in "On Schönhage's algorithm and subquadratic integer GCD computation" by Möller (see Appendix B [References], page 145). The recursive algorithm consists of these main steps.

- Call HGCD recursively, on the most significant $N/2$ limbs. Apply the resulting matrix T_1 to the full numbers, reducing them to a size just above $3N/2$.
- Perform a small number of division or subtraction steps to reduce the numbers to size below $3N/2$. This is essential mainly for the unlikely case of large quotients.
- Call HGCD recursively, on the most significant $N/2$ limbs of the reduced numbers. Apply the resulting matrix T_2 to the full numbers, reducing them to a size just above $N/2$.
- Compute $T = T_1 T_2$.
- Perform a small number of division and subtraction steps to satisfy the requirements, and return.

GCD is then implemented as a loop around HGCD, similarly to Lehmer's algorithm. Where Lehmer repeatedly chops off the top two limbs, calls `mpn_hgcd2`, and applies the resulting matrix to the full numbers, the subquadratic GCD chops off the most significant third of the limbs (the proportion is a tuning parameter, and 1/3 seems to be more efficient than, e.g, 1/2), calls `mpn_hgcd`, and applies the resulting matrix. Once the input numbers are reduced to size below `GCD_DC_THRESHOLD`, Lehmer's algorithm is used for the rest of the work.

The asymptotic running time of both HGCD and GCD is $O(M(N) \log N)$, where $M(N)$ is the time for multiplying two N-limb numbers.

16.3.4 Extended GCD

The extended GCD function, or gcdext, calculates $\gcd(a, b)$ and also one of the cofactors x and y satisfying $ax + by = \gcd(a, b)$. The algorithms used for plain GCD are extended to handle this case.

Lehmer's algorithm is used for sizes up to `GCDEXT_DC_THRESHOLD`. Above this threshold, GCDEXT is implemented as a loop around HGCD, but with more book-keeping to keep track of the cofactors.

16.3.5 Jacobi Symbol

`mpz_jacobi` and `mpz_kronecker` are currently implemented with a simple binary algorithm similar to that described for the GCDs (see Section 16.3.1 [Binary GCD], page 121). They're not very fast when both inputs are large. Lehmer's multi-step improvement or a binary based multi-step algorithm is likely to be better.

When one operand fits a single limb, and that includes `mpz_kronecker_ui` and friends, an initial reduction is done with either `mpn_mod_1` or `mpn_modexact_1_odd`, followed by the binary algorithm on a single limb. The binary algorithm is well suited to a single limb, and the whole calculation in this case is quite efficient.

In all the routines sign changes for the result are accumulated using some bit twiddling, avoiding table lookups or conditional jumps.

16.4 Powering Algorithms

16.4.1 Normal Powering

Normal `mpz` or `mpf` powering uses a simple binary algorithm, successively squaring and then multiplying by the base when a 1 bit is seen in the exponent, as per Knuth section 4.6.3. The "left to right" variant described there is used rather than algorithm A, since it's just as easy and can be done with somewhat less temporary memory.

16.4.2 Modular Powering

Modular powering is implemented using a 2^k-ary sliding window algorithm, as per "Handbook of Applied Cryptography" algorithm 14.85 (see Appendix B [References], page 145). k is chosen according to the size of the exponent. Larger exponents use larger values of k, the choice being made to minimize the average number of multiplications that must supplement the squaring.

The modular multiplies and squares use either a simple division or the REDC method by Montgomery (see Appendix B [References], page 145). REDC is a little faster, essentially saving N single limb divisions in a fashion similar to an exact remainder (see Section 16.2.5 [Exact Remainder], page 120). The current REDC has some limitations. It's only $O(N^2)$ so above `POWM_THRESHOLD` division becomes faster and is used. It doesn't attempt to detect small bases, but rather always uses a REDC form, which is usually a full size operand. And lastly it's only applied to odd moduli.

16.5 Root Extraction Algorithms

16.5.1 Square Root

Square roots are taken using the "Karatsuba Square Root" algorithm by Paul Zimmermann (see Appendix B [References], page 145).

An input n is split into four parts of k bits each, so with $b = 2^k$ we have $n = a_3b^3+a_2b^2+a_1b+a_0$. Part a_3 must be "normalized" so that either the high or second highest bit is set. In MPIR, k is kept on a limb boundary and the input is left shifted (by an even number of bits) to normalize.

The square root of the high two parts is taken, by recursive application of the algorithm (bottoming out in a one-limb Newton's method),

$$s', r' = \text{sqrtrem}\,(a_3b + a_2)$$

This is an approximation to the desired root and is extended by a division to give s,r,

$$\begin{aligned} q, u &= \text{divrem}\,(r'b + a_1, 2s') \\ s &= s'b + q \\ r &= ub + a_0 - q^2 \end{aligned}$$

The normalization requirement on a_3 means at this point s is either correct or 1 too big. r is negative in the latter case, so

$$\begin{aligned} &\text{if } r < 0 \text{ then} \\ &\quad r \leftarrow r + 2s - 1 \\ &\quad s \leftarrow s - 1 \end{aligned}$$

The algorithm is expressed in a divide and conquer form, but as noted in the paper it can also be viewed as a discrete variant of Newton's method, or as a variation on the schoolboy method (no longer taught) for square roots two digits at a time.

If the remainder r is not required then usually only a few high limbs of r and u need to be calculated to determine whether an adjustment to s is required. This optimization is not currently implemented.

In the Karatsuba multiplication range this algorithm is $O(\frac{3}{2}M(N/2))$, where $M(n)$ is the time to multiply two numbers of n limbs. In the FFT multiplication range this grows to a bound of $O(6M(N/2))$. In practice a factor of about 1.5 to 1.8 is found in the Karatsuba and Toom-3 ranges, growing to 2 or 3 in the FFT range.

The algorithm does all its calculations in integers and the resulting `mpn_sqrtrem` is used for both `mpz_sqrt` and `mpf_sqrt`. The extended precision given by `mpf_sqrt_ui` is obtained by padding with zero limbs.

16.5.2 Nth Root

Integer Nth roots are taken using Newton's method with the following iteration, where A is the input and n is the root to be taken.

$$a_{i+1} = \frac{1}{n}\left(\frac{A}{a_i^{n-1}} + (n-1)a_i\right)$$

The initial approximation a_1 is generated bitwise by successively powering a trial root with or without new 1 bits, aiming to be just above the true root. The iteration converges quadratically when started from a good approximation. When n is large more initial bits are needed to get good convergence. The current implementation is not particularly well optimized.

16.5.3 Perfect Square

A significant fraction of non-squares can be quickly identified by checking whether the input is a quadratic residue modulo small integers.

`mpz_perfect_square_p` first tests the input mod 256, which means just examining the low byte. Only 44 different values occur for squares mod 256, so 82.8% of inputs can be immediately identified as non-squares.

On a 32-bit system similar tests are done mod 9, 5, 7, 13 and 17, for a total 99.25% of inputs identified as non-squares. On a 64-bit system 97 is tested too, for a total 99.62%.

These moduli are chosen because they're factors of $2^{24}-1$ (or $2^{48}-1$ for 64-bits), and such a remainder can be quickly taken just using additions (see `mpn_mod_34lsub1`).

When nails are in use moduli are instead selected by the `gen-psqr.c` program and applied with an `mpn_mod_1`. The same $2^{24}-1$ or $2^{48}-1$ could be done with nails using some extra bit shifts, but this is not currently implemented.

In any case each modulus is applied to the `mpn_mod_34lsub1` or `mpn_mod_1` remainder and a table lookup identifies non-squares. By using a "modexact" style calculation, and suitably permuted tables, just one multiply each is required, see the code for details. Moduli are also combined to save operations, so long as the lookup tables don't become too big. `gen-psqr.c` does all the pre-calculations.

A square root must still be taken for any value that passes these tests, to verify it's really a square and not one of the small fraction of non-squares that get through (ie. a pseudo-square to all the tested bases).

Clearly more residue tests could be done, `mpz_perfect_square_p` only uses a compact and efficient set. Big inputs would probably benefit from more residue testing, small inputs might be better off with less. The assumed distribution of squares versus non-squares in the input would affect such considerations.

16.5.4 Perfect Power

Detecting perfect powers is required by some factorization algorithms. Currently `mpz_perfect_power_p` is implemented using repeated Nth root extractions, though naturally only prime roots need to be considered. (See Section 16.5.2 [Nth Root Algorithm], page 125.)

If a prime divisor p with multiplicity e can be found, then only roots which are divisors of e need to be considered, much reducing the work necessary. To this end divisibility by a set of small primes is checked.

16.6 Radix Conversion

Radix conversions are less important than other algorithms. A program dominated by conversions should probably use a different data representation.

16.6.1 Binary to Radix

Conversions from binary to a power-of-2 radix use a simple and fast $O(N)$ bit extraction algorithm.

Conversions from binary to other radices use one of two algorithms. Sizes below `GET_STR_PRECOMPUTE_THRESHOLD` use a basic $O(N^2)$ method. Repeated divisions by b^n are made, where b is the radix and n is the biggest power that fits in a limb. But instead of simply using the remainder r from such divisions, an extra divide step is done to give a fractional limb representing r/b^n. The digits of r can then be extracted using multiplications by b rather than divisions. Special case code is provided for decimal, allowing multiplications by 10 to optimize to shifts and adds.

Above `GET_STR_PRECOMPUTE_THRESHOLD` a sub-quadratic algorithm is used. For an input t, powers b^{n2^i} of the radix are calculated, until a power between t and $\sqrt{t}$ is reached. t is then divided by that largest power, giving a quotient which is the digits above that power, and a remainder which is those below. These two parts are in turn divided by the second highest power, and so on recursively. When a piece has been divided down to less than `GET_STR_DC_THRESHOLD` limbs, the basecase algorithm described above is used.

The advantage of this algorithm is that big divisions can make use of the sub-quadratic divide and conquer division (see Section 16.2.3 [Divide and Conquer Division], page 119), and big divisions tend to have less overheads than lots of separate single limb divisions anyway. But in any case the cost of calculating the powers b^{n2^i} must first be overcome.

`GET_STR_PRECOMPUTE_THRESHOLD` and `GET_STR_DC_THRESHOLD` represent the same basic thing, the point where it becomes worth doing a big division to cut the input in half. `GET_STR_PRECOMPUTE_THRESHOLD` includes the cost of calculating the radix power required, whereas `GET_STR_DC_THRESHOLD` assumes that's already available, which is the case when recursing.

Since the base case produces digits from least to most significant but they want to be stored from most to least, it's necessary to calculate in advance how many digits there will be, or at least be sure not to underestimate that. For MPIR the number of input bits is multiplied by `chars_per_bit_exactly` from `mp_bases`, rounding up. The result is either correct or one too big.

Examining some of the high bits of the input could increase the chance of getting the exact number of digits, but an exact result every time would not be practical, since in general the difference between numbers 100... and 99... is only in the last few bits and the work to identify 99... might well be almost as much as a full conversion.

`mpf_get_str` doesn't currently use the algorithm described here, it multiplies or divides by a power of b to move the radix point to the just above the highest non-zero digit (or at worst one above that location), then multiplies by b^n to bring out digits. This is $O(N^2)$ and is certainly not optimal.

The r/b^n scheme described above for using multiplications to bring out digits might be useful for more than a single limb. Some brief experiments with it on the base case when recursing didn't give a noticeable improvement, but perhaps that was only due to the implementation. Something similar would work for the sub-quadratic divisions too, though there would be the cost of calculating a bigger radix power.

Another possible improvement for the sub-quadratic part would be to arrange for radix powers that balanced the sizes of quotient and remainder produced, ie. the highest power would be an b^{nk} approximately equal to $\sqrt{t}$, not restricted to a 2^i factor. That ought to smooth out a graph of times against sizes, but may or may not be a net speedup.

16.6.2 Radix to Binary

This section is out-of-date.

Conversions from a power-of-2 radix into binary use a simple and fast $O(N)$ bitwise concatenation algorithm.

Conversions from other radices use one of two algorithms. Sizes below `SET_STR_THRESHOLD` use a basic $O(N^2)$ method. Groups of n digits are converted to limbs, where n is the biggest power of the base b which will fit in a limb, then those groups are accumulated into the result by multiplying by b^n and adding. This saves multi-precision operations, as per Knuth section 4.4 part E (see Appendix B [References], page 145). Some special case code is provided for decimal, giving the compiler a chance to optimize multiplications by 10.

Above `SET_STR_THRESHOLD` a sub-quadratic algorithm is used. First groups of n digits are converted into limbs. Then adjacent limbs are combined into limb pairs with $xb^n + y$, where x and y are the limbs. Adjacent limb pairs are combined into quads similarly with $xb^{2n} + y$. This continues until a single block remains, that being the result.

The advantage of this method is that the multiplications for each x are big blocks, allowing Karatsuba and higher algorithms to be used. But the cost of calculating the powers b^{n2^i} must be overcome. `SET_STR_THRESHOLD` usually ends up quite big, around 5000 digits, and on some processors much bigger still.

`SET_STR_THRESHOLD` is based on the input digits (and tuned for decimal), though it might be better based on a limb count, so as to be independent of the base. But that sort of count isn't used by the base case and so would need some sort of initial calculation or estimate.

The main reason `SET_STR_THRESHOLD` is so much bigger than the corresponding `GET_STR_PRECOMPUTE_THRESHOLD` is that `mpn_mul_1` is much faster than `mpn_divrem_1` (often by a factor of 10, or more).

16.7 Other Algorithms

16.7.1 Prime Testing

This section is somewhat out-of-date.

The primality testing in `mpz_probab_prime_p` (see Section 5.9 [Number Theoretic Functions], page 36) first does some trial division by small factors and then uses the Miller-Rabin probabilistic primality testing algorithm, as described in Knuth section 4.5.4 algorithm P (see Appendix B [References], page 145).

For an odd input n, and with $n = q2^k + 1$ where q is odd, this algorithm selects a random base x and tests whether $x^q \bmod n$ is 1 or -1, or an $x^{q2^j} \bmod n$ is 1, for $1 \leq j \leq k$. If so then n is probably prime, if not then n is definitely composite.

Any prime n will pass the test, but some composites do too. Such composites are known as strong pseudoprimes to base x. No n is a strong pseudoprime to more than $1/4$ of all bases (see Knuth exercise 22), hence with x chosen at random there's no more than a $1/4$ chance a "probable prime" will in fact be composite.

In fact strong pseudoprimes are quite rare, making the test much more powerful than this analysis would suggest, but $1/4$ is all that's proven for an arbitrary n.

16.7.2 Factorial

This section is out-of-date.

Factorials are calculated by a combination of removal of twos, powering, and binary splitting. The procedure can be best illustrated with an example,

$$23! = 1.2.3.4.5.6.7.8.9.10.11.12.13.14.15.16.17.18.19.20.21.22.23$$

has factors of two removed,

$$23! = 2^{19}.1.1.3.1.5.3.7.1.9.5.11.3.13.7.15.1.17.9.19.5.21.11.23$$

and the resulting terms collected up according to their multiplicity,

$$23! = 2^{19}.(3.5)^3.(7.9.11)^2.(13.15.17.19.21.23)$$

Each sequence such as 13.15.17.19.21.23 is evaluated by splitting into every second term, as for instance (13.17.21).(15.19.23), and the same recursively on each half. This is implemented iteratively using some bit twiddling.

Such splitting is more efficient than repeated N×1 multiplies since it forms big multiplies, allowing Karatsuba and higher algorithms to be used. And even below the Karatsuba threshold a big block of work can be more efficient for the basecase algorithm.

Splitting into subsequences of every second term keeps the resulting products more nearly equal in size than would the simpler approach of say taking the first half and second half of the sequence. Nearly equal products are more efficient for the current multiply implementation.

16.7.3 Binomial Coefficients

Binomial coefficients $\binom{n}{k}$ are calculated by first arranging $k \leq n/2$ using $\binom{n}{k} = \binom{n}{n-k}$ if necessary, and then evaluating the following product simply from $i = 2$ to $i = k$.

$$\binom{n}{k} = (n-k+1) \prod_{i=2}^{k} \frac{n-k+i}{i}$$

It's easy to show that each denominator i will divide the product so far, so the exact division algorithm is used (see Section 16.2.4 [Exact Division], page 119).

The numerators $n-k+i$ and denominators i are first accumulated into as many fit a limb, to save multi-precision operations, though for `mpz_bin_ui` this applies only to the divisors, since n is an `mpz_t` and $n-k+i$ in general won't fit in a limb at all.

16.7.4 Fibonacci Numbers

The Fibonacci functions `mpz_fib_ui` and `mpz_fib2_ui` are designed for calculating isolated F_n or F_n,F_{n-1} values efficiently.

For small n, a table of single limb values in `__gmp_fib_table` is used. On a 32-bit limb this goes up to F_{47}, or on a 64-bit limb up to F_{93}. For convenience the table starts at F_{-1}.

Beyond the table, values are generated with a binary powering algorithm, calculating a pair F_n and F_{n-1} working from high to low across the bits of n. The formulas used are

$$\begin{aligned} F_{2k+1} &= 4F_k^2 - F_{k-1}^2 + 2(-1)^k \\ F_{2k-1} &= F_k^2 + F_{k-1}^2 \\ F_{2k} &= F_{2k+1} - F_{2k-1} \end{aligned}$$

At each step, k is the high b bits of n. If the next bit of n is 0 then F_{2k},F_{2k-1} is used, or if it's a 1 then F_{2k+1},F_{2k} is used, and the process repeated until all bits of n are incorporated. Notice these formulas require just two squares per bit of n.

It'd be possible to handle the first few n above the single limb table with simple additions, using the defining Fibonacci recurrence $F_{k+1} = F_k + F_{k-1}$, but this is not done since it usually turns out to be faster for only about 10 or 20 values of n, and including a block of code for just those doesn't seem worthwhile. If they really mattered it'd be better to extend the data table.

Using a table avoids lots of calculations on small numbers, and makes small n go fast. A bigger table would make more small n go fast, it's just a question of balancing size against desired speed. For MPIR the code is kept compact, with the emphasis primarily on a good powering algorithm.

`mpz_fib2_ui` returns both F_n and F_{n-1}, but `mpz_fib_ui` is only interested in F_n. In this case the last step of the algorithm can become one multiply instead of two squares. One of the following two formulas is used, according as n is odd or even.

$$\begin{aligned} F_{2k} &= F_k(F_k + 2F_{k-1}) \\ F_{2k+1} &= (2F_k + F_{k-1})(2F_k - F_{k-1}) + 2(-1)^k \end{aligned}$$

F_{2k+1} here is the same as above, just rearranged to be a multiply. For interest, the $2(-1)^k$ term both here and above can be applied just to the low limb of the calculation, without a carry or borrow into further limbs, which saves some code size. See comments with `mpz_fib_ui` and the internal `mpn_fib2_ui` for how this is done.

16.7.5 Lucas Numbers

`mpz_lucnum2_ui` derives a pair of Lucas numbers from a pair of Fibonacci numbers with the following simple formulas.

$$\begin{aligned} L_k &= F_k + 2F_{k-1} \\ L_{k-1} &= 2F_k - F_{k-1} \end{aligned}$$

`mpz_lucnum_ui` is only interested in L_n, and some work can be saved. Trailing zero bits on n can be handled with a single square each.

$$L_{2k} = L_k^2 - 2(-1)^k$$

And the lowest 1 bit can be handled with one multiply of a pair of Fibonacci numbers, similar to what `mpz_fib_ui` does.

$$L_{2k+1} = 5F_{k-1}(2F_k + F_{k-1}) - 4(-1)^k$$

16.7.6 Random Numbers

For the `urandomb` functions, random numbers are generated simply by concatenating bits produced by the generator. As long as the generator has good randomness properties this will produce well-distributed N bit numbers.

For the `urandomm` functions, random numbers in a range $0 \le R < N$ are generated by taking values R of $\lceil \log_2 N \rceil$ bits each until one satisfies $R < N$. This will normally require only one or two attempts, but the attempts are limited in case the generator is somehow degenerate and produces only 1 bits or similar.

The Mersenne Twister generator is by Matsumoto and Nishimura (see Appendix B [References], page 145). It has a non-repeating period of $2^{19937}-1$, which is a Mersenne prime, hence the name of the generator. The state is 624 words of 32-bits each, which is iterated with one XOR and shift for each 32-bit word generated, making the algorithm very fast. Randomness properties are also very good and this is the default algorithm used by MPIR.

Linear congruential generators are described in many text books, for instance Knuth volume 2 (see Appendix B [References], page 145). With a modulus M and parameters A and C, a integer state S is iterated by the formula $S \leftarrow AS + C \bmod M$. At each step the new state is a linear function of the previous, mod M, hence the name of the generator.

In MPIR only moduli of the form 2^N are supported, and the current implementation is not as well optimized as it could be. Overheads are significant when N is small, and when N is large clearly the multiply at each step will become slow. This is not a big concern, since the Mersenne Twister generator is better in every respect and is therefore recommended for all normal applications.

For both generators the current state can be deduced by observing enough output and applying some linear algebra (over GF(2) in the case of the Mersenne Twister). This generally means raw output is unsuitable for cryptographic applications without further hashing or the like.

16.8 Assembler Coding

The assembler subroutines in MPIR are the most significant source of speed at small to moderate sizes. At larger sizes algorithm selection becomes more important, but of course speedups in low level routines will still speed up everything proportionally.

Carry handling and widening multiplies that are important for MPIR can't be easily expressed in C. GCC `asm` blocks help a lot and are provided in `longlong.h`, but hand coding low level routines invariably offers a speedup over generic C by a factor of anything from 2 to 10.

16.8.1 Code Organisation

The various `mpn` subdirectories contain machine-dependent code, written in C or assembler. The `mpn/generic` subdirectory contains default code, used when there's no machine-specific version of a particular file.

Each `mpn` subdirectory is for an ISA family. Generally 32-bit and 64-bit variants in a family cannot share code and have separate directories. Within a family further subdirectories may exist for CPU variants.

In each directory a `nails` subdirectory may exist, holding code with nails support for that CPU variant. A `NAILS_SUPPORT` directive in each file indicates the nails values the code handles.

Nails code only exists where it's faster, or promises to be faster, than plain code. There's no effort put into nails if they're not going to enhance a given CPU.

16.8.2 Assembler Basics

`mpn_addmul_1` and `mpn_submul_1` are the most important routines for overall MPIR performance. All multiplications and divisions come down to repeated calls to these. `mpn_add_n`, `mpn_sub_n`, `mpn_lshift` and `mpn_rshift` are next most important.

On some CPUs assembler versions of the internal functions `mpn_mul_basecase` and `mpn_sqr_basecase` give significant speedups, mainly through avoiding function call overheads. They can also potentially make better use of a wide superscalar processor, as can bigger primitives like `mpn_addmul_2` or `mpn_addmul_4`.

The restrictions on overlaps between sources and destinations (see Chapter 8 [Low-level Functions], page 58) are designed to facilitate a variety of implementations. For example, knowing `mpn_add_n` won't have partly overlapping sources and destination means reading can be done far ahead of writing on superscalar processors, and loops can be vectorized on a vector processor, depending on the carry handling.

16.8.3 Carry Propagation

The problem that presents most challenges in MPIR is propagating carries from one limb to the next. In functions like `mpn_addmul_1` and `mpn_add_n`, carries are the only dependencies between limb operations.

On processors with carry flags, a straightforward CISC style `adc` is generally best. AMD K6 `mpn_addmul_1` however is an example of an unusual set of circumstances where a branch works out better.

On RISC processors generally an add and compare for overflow is used. This sort of thing can be seen in `mpn/generic/aors_n.c`. Some carry propagation schemes require 4 instructions, meaning at least 4 cycles per limb, but other schemes may use just 1 or 2. On wide superscalar processors performance may be completely determined by the number of dependent instructions between carry-in and carry-out for each limb.

On vector processors good use can be made of the fact that a carry bit only very rarely propagates more than one limb. When adding a single bit to a limb, there's only a carry out if that limb was `0xFF...FF` which on random data will be only 1 in $2^{\texttt{mp\_bits\_per\_limb}}$. `mpn/cray/add_n.c` is an example of this, it adds all limbs in parallel, adds one set of carry bits in parallel and then only rarely needs to fall through to a loop propagating further carries.

On the x86s, GCC (as of version 2.95.2) doesn't generate particularly good code for the RISC style idioms that are necessary to handle carry bits in C. Often conditional jumps are generated where `adc` or `sbb` forms would be better. And so unfortunately almost any loop involving carry bits needs to be coded in assembler for best results.

16.8.4 Cache Handling

MPIR aims to perform well both on operands that fit entirely in L1 cache and those which don't.

Basic routines like `mpn_add_n` or `mpn_lshift` are often used on large operands, so L2 and main memory performance is important for them. `mpn_mul_1` and `mpn_addmul_1` are mostly used for multiply and square basecases, so L1 performance matters most for them, unless assembler versions of `mpn_mul_basecase` and `mpn_sqr_basecase` exist, in which case the remaining uses are mostly for larger operands.

For L2 or main memory operands, memory access times will almost certainly be more than the calculation time. The aim therefore is to maximize memory throughput, by starting a load of the next cache line while processing the contents of the previous one. Clearly this is only possible if the chip has a lock-up free cache or some sort of prefetch instruction. Most current chips have both these features.

Prefetching sources combines well with loop unrolling, since a prefetch can be initiated once per unrolled loop (or more than once if the loop covers more than one cache line).

On CPUs without write-allocate caches, prefetching destinations will ensure individual stores don't go further down the cache hierarchy, limiting bandwidth. Of course for calculations which are slow anyway, like `mpn_divrem_1`, write-throughs might be fine.

The distance ahead to prefetch will be determined by memory latency versus throughput. The aim of course is to have data arriving continuously, at peak throughput. Some CPUs have limits on the number of fetches or prefetches in progress.

If a special prefetch instruction doesn't exist then a plain load can be used, but in that case care must be taken not to attempt to read past the end of an operand, since that might produce a segmentation violation.

Some CPUs or systems have hardware that detects sequential memory accesses and initiates suitable cache movements automatically, making life easy.

16.8.5 Functional Units

When choosing an approach for an assembler loop, consideration is given to what operations can execute simultaneously and what throughput can thereby be achieved. In some cases an algorithm can be tweaked to accommodate available resources.

Loop control will generally require a counter and pointer updates, costing as much as 5 instructions, plus any delays a branch introduces. CPU addressing modes might reduce pointer updates, perhaps by allowing just one updating pointer and others expressed as offsets from it, or on CISC chips with all addressing done with the loop counter as a scaled index.

The final loop control cost can be amortised by processing several limbs in each iteration (see Section 16.8.9 [Assembler Loop Unrolling], page 134). This at least ensures loop control isn't a big fraction the work done.

Memory throughput is always a limit. If perhaps only one load or one store can be done per cycle then 3 cycles/limb will the top speed for "binary" operations like `mpn_add_n`, and any code achieving that is optimal.

Integer resources can be freed up by having the loop counter in a float register, or by pressing the float units into use for some multiplying, perhaps doing every second limb on the float side (see Section 16.8.6 [Assembler Floating Point], page 132).

Float resources can be freed up by doing carry propagation on the integer side, or even by doing integer to float conversions in integers using bit twiddling.

16.8.6 Floating Point

Floating point arithmetic is used in MPIR for multiplications on CPUs with poor integer multipliers. It's mostly useful for `mpn_mul_1`, `mpn_addmul_1` and `mpn_submul_1` on 64-bit machines, and `mpn_mul_basecase` on both 32-bit and 64-bit machines.

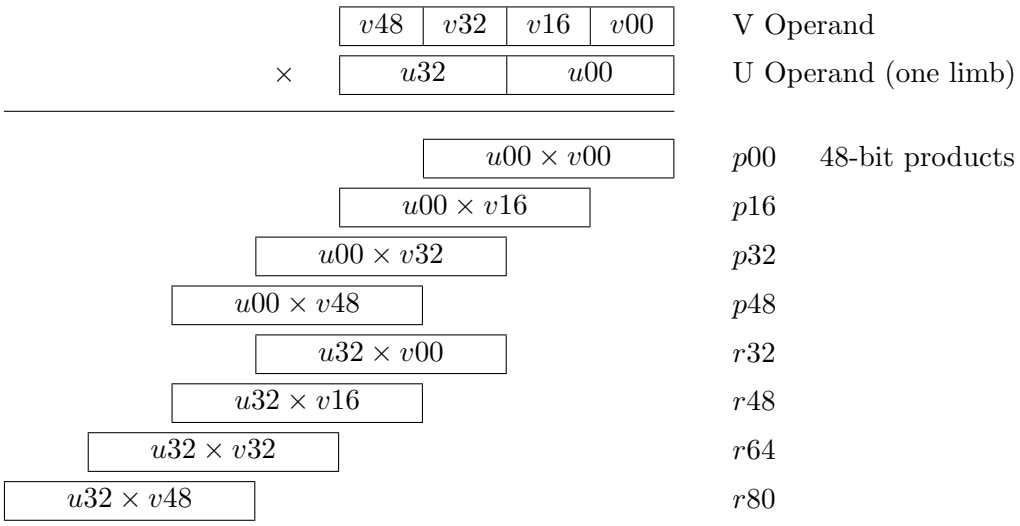
With IEEE 53-bit double precision floats, integer multiplications producing up to 53 bits will give exact results. Breaking a 64×64 multiplication into eight $16\times32 \rightarrow 48$ bit pieces is convenient.

With some care though six $21\times32 \rightarrow 53$ bit products can be used, if one of the lower two 21-bit pieces also uses the sign bit.

For the `mpn_mul_1` family of functions on a 64-bit machine, the invariant single limb is split at the start, into 3 or 4 pieces. Inside the loop, the bignum operand is split into 32-bit pieces. Fast conversion of these unsigned 32-bit pieces to floating point is highly machine-dependent. In some cases, reading the data into the integer unit, zero-extending to 64-bits, then transferring to the floating point unit back via memory is the only option.

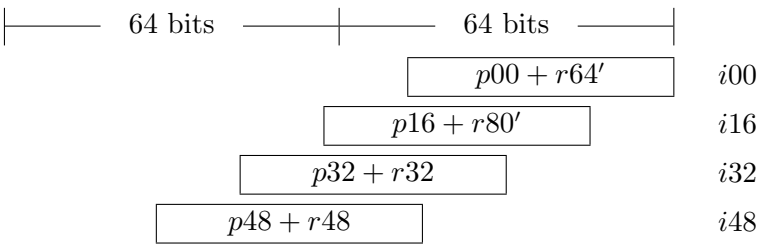
Converting partial products back to 64-bit limbs is usually best done as a signed conversion. Since all values are smaller than 2^{53}, signed and unsigned are the same, but most processors lack unsigned conversions.

Here is a diagram showing 16×32 bit products for an `mpn_mul_1` or `mpn_addmul_1` with a 64-bit limb. The single limb operand V is split into four 16-bit parts. The multi-limb operand U is split in the loop into two 32-bit parts.



$p32$ and $r32$ can be summed using floating-point addition, and likewise $p48$ and $r48$. $p00$ and $p16$ can be summed with $r64$ and $r80$ from the previous iteration.

For each loop then, four 49-bit quantities are transfered to the integer unit, aligned as follows,



The challenge then is to sum these efficiently and add in a carry limb, generating a low 64-bit result limb and a high 33-bit carry limb ($i48$ extends 33 bits into the high half).

16.8.7 SIMD Instructions

The single-instruction multiple-data support in current microprocessors is aimed at signal processing algorithms where each data point can be treated more or less independently. There's generally not much support for propagating the sort of carries that arise in MPIR.

SIMD multiplications of say four 16×16 bit multiplies only do as much work as one 32×32 from MPIR's point of view, and need some shifts and adds besides. But of course if say the SIMD form is fully pipelined and uses less instruction decoding then it may still be worthwhile.

On the x86 chips, MMX has so far found a use in `mpn_rshift` and `mpn_lshift`, and is used in a special case for 16-bit multipliers in the P55 `mpn_mul_1`. SSE2 is used for Pentium 4 `mpn_mul_1`, `mpn_addmul_1`, and `mpn_submul_1`.

16.8.8 Software Pipelining

Software pipelining consists of scheduling instructions around the branch point in a loop. For example a loop might issue a load not for use in the present iteration but the next, thereby allowing extra cycles for the data to arrive from memory.

Naturally this is wanted only when doing things like loads or multiplies that take several cycles to complete, and only where a CPU has multiple functional units so that other work can be done in the meantime.

A pipeline with several stages will have a data value in progress at each stage and each loop iteration moves them along one stage. This is like juggling.

If the latency of some instruction is greater than the loop time then it will be necessary to unroll, so one register has a result ready to use while another (or multiple others) are still in progress. (see Section 16.8.9 [Assembler Loop Unrolling], page 134).

16.8.9 Loop Unrolling

Loop unrolling consists of replicating code so that several limbs are processed in each loop. At a minimum this reduces loop overheads by a corresponding factor, but it can also allow better register usage, for example alternately using one register combination and then another. Judicious use of `m4` macros can help avoid lots of duplication in the source code.

Any amount of unrolling can be handled with a loop counter that's decremented by N each time, stopping when the remaining count is less than the further N the loop will process. Or by subtracting N at the start, the termination condition becomes when the counter C is less than 0 (and the count of remaining limbs is $C + N$).

Alternately for a power of 2 unroll the loop count and remainder can be established with a shift and mask. This is convenient if also making a computed jump into the middle of a large loop.

The limbs not a multiple of the unrolling can be handled in various ways, for example

- A simple loop at the end (or the start) to process the excess. Care will be wanted that it isn't too much slower than the unrolled part.
- A set of binary tests, for example after an 8-limb unrolling, test for 4 more limbs to process, then a further 2 more or not, and finally 1 more or not. This will probably take more code space than a simple loop.
- A `switch` statement, providing separate code for each possible excess, for example an 8-limb unrolling would have separate code for 0 remaining, 1 remaining, etc, up to 7 remaining. This might take a lot of code, but may be the best way to optimize all cases in combination with a deep pipelined loop.
- A computed jump into the middle of the loop, thus making the first iteration handle the excess. This should make times smoothly increase with size, which is attractive, but setups for the jump and adjustments for pointers can be tricky and could become quite difficult in combination with deep pipelining.

16.8.10 Writing Guide

This is a guide to writing software pipelined loops for processing limb vectors in assembler.

First determine the algorithm and which instructions are needed. Code it without unrolling or scheduling, to make sure it works. On a 3-operand CPU try to write each new value to a new register, this will greatly simplify later steps.

Then note for each instruction the functional unit and/or issue port requirements. If an instruction can use either of two units, like U0 or U1 then make a category "U0/U1". Count the total using each unit (or combined unit), and count all instructions.

Figure out from those counts the best possible loop time. The goal will be to find a perfect schedule where instruction latencies are completely hidden. The total instruction count might be the limiting factor, or perhaps a particular functional unit. It might be possible to tweak the instructions to help the limiting factor.

Suppose the loop time is N, then make N issue buckets, with the final loop branch at the end of the last. Now fill the buckets with dummy instructions using the functional units desired. Run this to make sure the intended speed is reached.

Now replace the dummy instructions with the real instructions from the slow but correct loop you started with. The first will typically be a load instruction. Then the instruction using that value is placed in a bucket an appropriate distance down. Run the loop again, to check it still runs at target speed.

Keep placing instructions, frequently measuring the loop. After a few you will need to wrap around from the last bucket back to the top of the loop. If you used the new-register for new-value strategy above then there will be no register conflicts. If not then take care not to clobber something already in use. Changing registers at this time is very error prone.

The loop will overlap two or more of the original loop iterations, and the computation of one vector element result will be started in one iteration of the new loop, and completed one or several iterations later.

The final step is to create feed-in and wind-down code for the loop. A good way to do this is to make a copy (or copies) of the loop at the start and delete those instructions which don't have valid antecedents, and at the end replicate and delete those whose results are unwanted (including any further loads).

The loop will have a minimum number of limbs loaded and processed, so the feed-in code must test if the request size is smaller and skip either to a suitable part of the wind-down or to special code for small sizes.

17 Internals

This chapter is provided only for informational purposes and the various internals described here may change in future MPIR releases. Applications expecting to be compatible with future releases should use only the documented interfaces described in previous chapters.

17.1 Integer Internals

`mpz_t` variables represent integers using sign and magnitude, in space dynamically allocated and reallocated. The fields are as follows.

`_mp_size` The number of limbs, or the negative of that when representing a negative integer. Zero is represented by `_mp_size` set to zero, in which case the `_mp_d` data is unused.

`_mp_d` A pointer to an array of limbs which is the magnitude. These are stored "little endian" as per the `mpn` functions, so `_mp_d[0]` is the least significant limb and `_mp_d[ABS(_mp_size)-1]` is the most significant. Whenever `_mp_size` is non-zero, the most significant limb is non-zero.

Currently there's always at least one limb allocated, so for instance `mpz_set_ui` never needs to reallocate, and `mpz_get_ui` can fetch `_mp_d[0]` unconditionally (though its value is then only wanted if `_mp_size` is non-zero).

`_mp_alloc`

`_mp_alloc` is the number of limbs currently allocated at `_mp_d`, and naturally `_mp_alloc >= ABS(_mp_size)`. When an `mpz` routine is about to (or might be about to) increase `_mp_size`, it checks `_mp_alloc` to see whether there's enough space, and reallocates if not. `MPZ_REALLOC` is generally used for this.

The various bitwise logical functions like `mpz_and` behave as if negative values were twos complement. But sign and magnitude is always used internally, and necessary adjustments are made during the calculations. Sometimes this isn't pretty, but sign and magnitude are best for other routines.

Some internal temporary variables are setup with `MPZ_TMP_INIT` and these have `_mp_d` space obtained from `TMP_ALLOC` rather than the memory allocation functions. Care is taken to ensure that these are big enough that no reallocation is necessary (since it would have unpredictable consequences).

`_mp_size` and `_mp_alloc` are `int`, although `mp_size_t` is usually a `long`. This is done to make the fields just 32 bits on some 64 bits systems, thereby saving a few bytes of data space but still providing plenty of range.

17.2 Rational Internals

`mpq_t` variables represent rationals using an `mpz_t` numerator and denominator (see Section 17.1 [Integer Internals], page 136).

The canonical form adopted is denominator positive (and non-zero), no common factors between numerator and denominator, and zero uniquely represented as 0/1.

It's believed that casting out common factors at each stage of a calculation is best in general. A GCD is an $O(N^2)$ operation so it's better to do a few small ones immediately than to delay and have to do a big one later. Knowing the numerator and denominator have no common factors can be used for example in `mpq_mul` to make only two cross GCDs necessary, not four.

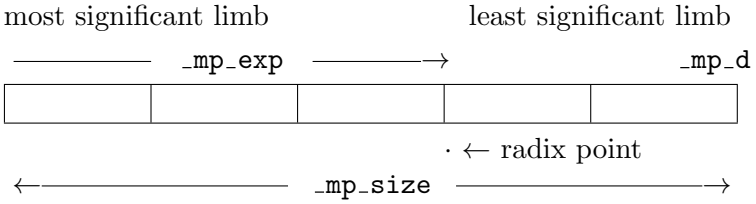
This general approach to common factors is badly sub-optimal in the presence of simple factorizations or little prospect for cancellation, but MPIR has no way to know when this will occur.

As per Section 3.11 [Efficiency], page 21, that's left to applications. The `mpq_t` framework might still suit, with `mpq_numref` and `mpq_denref` for direct access to the numerator and denominator, or of course `mpz_t` variables can be used directly.

17.3 Float Internals

Efficient calculation is the primary aim of MPIR floats and the use of whole limbs and simple rounding facilitates this.

`mpf_t` floats have a variable precision mantissa and a single machine word signed exponent. The mantissa is represented using sign and magnitude.



The fields are as follows.

`_mp_size` The number of limbs currently in use, or the negative of that when representing a negative value. Zero is represented by `_mp_size` and `_mp_exp` both set to zero, and in that case the `_mp_d` data is unused. (In the future `_mp_exp` might be undefined when representing zero.)

`_mp_prec` The precision of the mantissa, in limbs. In any calculation the aim is to produce `_mp_prec` limbs of result (the most significant being non-zero).

`_mp_d` A pointer to the array of limbs which is the absolute value of the mantissa. These are stored "little endian" as per the `mpn` functions, so `_mp_d[0]` is the least significant limb and `_mp_d[ABS(_mp_size)-1]` the most significant.

The most significant limb is always non-zero, but there are no other restrictions on its value, in particular the highest 1 bit can be anywhere within the limb.

`_mp_prec+1` limbs are allocated to `_mp_d`, the extra limb being for convenience (see below). There are no reallocations during a calculation, only in a change of precision with `mpf_set_prec`.

`_mp_exp` The exponent, in limbs, determining the location of the implied radix point. Zero means the radix point is just above the most significant limb. Positive values mean a radix point offset towards the lower limbs and hence a value ≥ 1, as for example in the diagram above. Negative exponents mean a radix point further above the highest limb.

Naturally the exponent can be any value, it doesn't have to fall within the limbs as the diagram shows, it can be a long way above or a long way below. Limbs other than those included in the `{_mp_d,_mp_size}` data are treated as zero.

`_mp_size` and `_mp_prec` are `int`, although `mp_size_t` is usually a `long`. This is done to make the fields just 32 bits on some 64 bits systems, thereby saving a few bytes of data space but still providing plenty of range.

The following various points should be noted.

Low Zeros The least significant limbs `_mp_d[0]` etc can be zero, though such low zeros can always be ignored. Routines likely to produce low zeros check and avoid them to

save time in subsequent calculations, but for most routines they're quite unlikely and aren't checked.

Mantissa Size Range
: The `_mp_size` count of limbs in use can be less than `_mp_prec` if the value can be represented in less. This means low precision values or small integers stored in a high precision `mpf_t` can still be operated on efficiently.

  `_mp_size` can also be greater than `_mp_prec`. Firstly a value is allowed to use all of the `_mp_prec+1` limbs available at `_mp_d`, and secondly when `mpf_set_prec_raw` lowers `_mp_prec` it leaves `_mp_size` unchanged and so the size can be arbitrarily bigger than `_mp_prec`.

Rounding
: All rounding is done on limb boundaries. Calculating `_mp_prec` limbs with the high non-zero will ensure the application requested minimum precision is obtained.

  The use of simple "trunc" rounding towards zero is efficient, since there's no need to examine extra limbs and increment or decrement.

Bit Shifts
: Since the exponent is in limbs, there are no bit shifts in basic operations like `mpf_add` and `mpf_mul`. When differing exponents are encountered all that's needed is to adjust pointers to line up the relevant limbs.

  Of course `mpf_mul_2exp` and `mpf_div_2exp` will require bit shifts, but the choice is between an exponent in limbs which requires shifts there, or one in bits which requires them almost everywhere else.

Use of `_mp_prec+1` Limbs
: The extra limb on `_mp_d` (`_mp_prec+1` rather than just `_mp_prec`) helps when an `mpf` routine might get a carry from its operation. `mpf_add` for instance will do an `mpn_add` of `_mp_prec` limbs. If there's no carry then that's the result, but if there is a carry then it's stored in the extra limb of space and `_mp_size` becomes `_mp_prec+1`.

  Whenever `_mp_prec+1` limbs are held in a variable, the low limb is not needed for the intended precision, only the `_mp_prec` high limbs. But zeroing it out or moving the rest down is unnecessary. Subsequent routines reading the value will simply take the high limbs they need, and this will be `_mp_prec` if their target has that same precision. This is no more than a pointer adjustment, and must be checked anyway since the destination precision can be different from the sources.

  Copy functions like `mpf_set` will retain a full `_mp_prec+1` limbs if available. This ensures that a variable which has `_mp_size` equal to `_mp_prec+1` will get its full exact value copied. Strictly speaking this is unnecessary since only `_mp_prec` limbs are needed for the application's requested precision, but it's considered that an `mpf_set` from one variable into another of the same precision ought to produce an exact copy.

Application Precisions
: `__GMPF_BITS_TO_PREC` converts an application requested precision to an `_mp_prec`. The value in bits is rounded up to a whole limb then an extra limb is added since the most significant limb of `_mp_d` is only non-zero and therefore might contain only one bit.

  `__GMPF_PREC_TO_BITS` does the reverse conversion, and removes the extra limb from `_mp_prec` before converting to bits. The net effect of reading back with `mpf_get_prec` is simply the precision rounded up to a multiple of `mp_bits_per_limb`.

  Note that the extra limb added here for the high only being non-zero is in addition to the extra limb allocated to `_mp_d`. For example with a 32-bit limb, an application request for 250 bits will be rounded up to 8 limbs, then an extra added for the high being only non-zero, giving an `_mp_prec` of 9. `_mp_d` then gets 10 limbs allocated.

Reading back with `mpf_get_prec` will take `_mp_prec` subtract 1 limb and multiply by 32, giving 256 bits.

Strictly speaking, the fact the high limb has at least one bit means that a float with, say, 3 limbs of 32-bits each will be holding at least 65 bits, but for the purposes of `mpf_t` it's considered simply to be 64 bits, a nice multiple of the limb size.

17.4 Raw Output Internals

`mpz_out_raw` uses the following format.

size	data bytes

The size is 4 bytes written most significant byte first, being the number of subsequent data bytes, or the twos complement negative of that when a negative integer is represented. The data bytes are the absolute value of the integer, written most significant byte first.

The most significant data byte is always non-zero, so the output is the same on all systems, irrespective of limb size.

In GMP 1, leading zero bytes were written to pad the data bytes to a multiple of the limb size. `mpz_inp_raw` will still accept this, for compatibility.

The use of "big endian" for both the size and data fields is deliberate, it makes the data easy to read in a hex dump of a file. Unfortunately it also means that the limb data must be reversed when reading or writing, so neither a big endian nor little endian system can just read and write `_mp_d`.

17.5 C++ Interface Internals

A system of expression templates is used to ensure something like `a=b+c` turns into a simple call to `mpz_add` etc. For `mpf_class` the scheme also ensures the precision of the final destination is used for any temporaries within a statement like `f=w*x+y*z`. These are important features which a naive implementation cannot provide.

A simplified description of the scheme follows. The true scheme is complicated by the fact that expressions have different return types. For detailed information, refer to the source code.

To perform an operation, say, addition, we first define a "function object" evaluating it,

```
struct __gmp_binary_plus
{
  static void eval(mpf_t f, mpf_t g, mpf_t h) { mpf_add(f, g, h); }
};
```

And an "additive expression" object,

```
__gmp_expr<__gmp_binary_expr<mpf_class, mpf_class, __gmp_binary_plus> >
operator+(const mpf_class &f, const mpf_class &g)
{
  return __gmp_expr
    <__gmp_binary_expr<mpf_class, mpf_class, __gmp_binary_plus> >(f, g);
}
```

The seemingly redundant `__gmp_expr<__gmp_binary_expr<...>>` is used to encapsulate any possible kind of expression into a single template type. In fact even `mpf_class` etc are `typedef` specializations of `__gmp_expr`.

Next we define assignment of __gmp_expr to mpf_class.

```
template <class T>
mpf_class & mpf_class::operator=(const __gmp_expr<T> &expr)
{
  expr.eval(this->get_mpf_t(), this->precision());
  return *this;
}

template <class Op>
void __gmp_expr<__gmp_binary_expr<mpf_class, mpf_class, Op> >::eval
(mpf_t f, mp_bitcnt_t precision)
{
  Op::eval(f, expr.val1.get_mpf_t(), expr.val2.get_mpf_t());
}
```

where expr.val1 and expr.val2 are references to the expression's operands (here expr is the __gmp_binary_expr stored within the __gmp_expr).

This way, the expression is actually evaluated only at the time of assignment, when the required precision (that of f) is known. Furthermore the target mpf_t is now available, thus we can call mpf_add directly with f as the output argument.

Compound expressions are handled by defining operators taking subexpressions as their arguments, like this:

```
template <class T, class U>
__gmp_expr
<__gmp_binary_expr<__gmp_expr<T>, __gmp_expr<U>, __gmp_binary_plus> >
operator+(const __gmp_expr<T> &expr1, const __gmp_expr<U> &expr2)
{
  return __gmp_expr
    <__gmp_binary_expr<__gmp_expr<T>, __gmp_expr<U>, __gmp_binary_plus> >
    (expr1, expr2);
}
```

And the corresponding specializations of __gmp_expr::eval:

```
template <class T, class U, class Op>
void __gmp_expr
<__gmp_binary_expr<__gmp_expr<T>, __gmp_expr<U>, Op> >::eval
(mpf_t f, mp_bitcnt_t precision)
{
  // declare two temporaries
  mpf_class temp1(expr.val1, precision), temp2(expr.val2, precision);
  Op::eval(f, temp1.get_mpf_t(), temp2.get_mpf_t());
}
```

The expression is thus recursively evaluated to any level of complexity and all subexpressions are evaluated to the precision of f.

Appendix A 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 initial versions of `mpz_jacobi` and `mpz_legendre`.

Andreas Schwab contributed the files `mpn/m68k/lshift.S` and `mpn/m68k/rshift.S` (now in `.asm` form).

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, 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 Divide and Conquer division code, the REDC code, the REDC-based mpz_powm code, the FFT multiply code, and the Karatsuba square root code. He also rewrote the Toom3 code for GMP 4.2. The ECMNET project Paul is organizing was a driving force behind many of the optimizations in GMP 3.

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

Kent Boortz made the Mac OS 9 port.

Kevin Ryde worked on a number of things: optimized x86 code, m4 asm macros, parameter tuning, speed measuring, the configure system, function inlining, divisibility tests, bit scanning, Jacobi symbols, Fibonacci and Lucas number functions, printf and scanf functions, perl interface, demo expression parser, the algorithms chapter in the manual, `gmpasm-mode.el`, and various miscellaneous improvements elsewhere.

Steve Root helped write the optimized alpha 21264 assembly code.

Gerardo Ballabio wrote the `gmpxx.h` C++ class interface and the C++ `istream` input routines.

GNU MP 4 was finished and released by Torbjorn Granlund and Kevin Ryde. Torbjorn's work was partially funded by the IDA Center for Computing Sciences, USA.

Jason Moxham rewrote `mpz_fac_ui`.

Pedro Gimeno implemented the Mersenne Twister and made other random number improvements.

(This list is chronological, not ordered after significance. If you have contributed to GMP/MPIR but are not listed above, please tell `http://groups.google.com/group/mpir-devel` about the omission!)

Thanks go to Hans Thorsen for donating an SGI system for the GMP test system environment.

In 2008 GMP was forked and gave rise to the MPIR (Multiple Precision Integers and Rationals) project. In 2010 version 2.0.0 of MPIR switched to LGPL v3+ and much code from GMP was again incorporated into MPIR.

The MPIR project has largely been a collaboration of William Hart, Brian Gladman and Jason Moxham. MPIR code not obtained from GMP and not specifically mentioned elsewhere below is likely written by one of these three.

William Hart did much of the early MPIR coding including build system fixes. His contributions also include Toom 4 and 7 code and variants, extended GCD based on Niels Mollers ngcd work, asymptotically fast division code. He does much of the release management work.

Brian Gladman wrote and maintains MSVC project files. He has also done much of the conversion of assembly code to yasm format. He rewrote the benchmark program and developed MSVC ports of tune, speed, try and the benchmark code. He helped with many aspects of the merging of GMP code into MPIR after the switch to LGPL v3+.

Jason Moxham has contributed a great deal of x86 assembly code. He has also contributed improved root code and mulhi and mullo routines and implemented Peter Montgomery's single limb remainder algorithm. He has also contributed a command line build system for Windows and numerous build system fixes.

The following people have either contributed directly to the MPIR project, made code available on their websites or contributed code to the official GNU project which has been used in MPIR.

Jason Martin wrote some fast assembly patches for Core 2 and converted them to intel format. He also did the initial merge of Niels Moller's fast GCD patches. He wrote fast addmul functions for Itanium.

Gonzalo Tornaria helped patch config.guess and associated files to distinguish modern processors. He also patched mpirbench.

Michael Abshoff helped resolve some build issues on various platforms. He served for a while as release manager for the MPIR project.

Mariah Lennox contributed patches to mpirbench and various build failure reports. She has also reported gcc bugs found during MPIR development.

Niels Moller wrote the fast ngcd code for computing integer GCD, the quadratic Hensel division code and precomputed inverse code for Euclidean division, along with fast jacobi symbols code. He also made contributions to the Toom multiply code, especially helper functions to simplify Toom evaluations.

Burcin Erocal helped with build testing on Pentium-D

Pierrick Gaudry provided initial AMD 64 assembly support and revised the FFT code.

Paul Zimmermann provided an mpz implementation of Toom 4, wrote much of the FFT code, wrote some of the rootrem code and contributed invert.c for computing precomputed inverses.

Alexander Kruppa revised the FFT code and helped write and superoptimise assembly code for Skylake, Haswell and Bulldozer and helped write a superoptimiser.

Torbjorn Granlund revised the FFT code and wrote a lot of division code, including the quadratic Euclidean division code, many parts of the divide and conquer division code, both Hensel and Euclidean, and his code was also reused for parts of the asymptotically fast division code. He also helped write the root code and wrote much of the Itanium assembly code and a couple of Core 2 assembly functions and part of the basecase middle product assembly code for x86 64 bit. He also wrote the improved string input and output code and made improvements to the GCD and extended GCD code. He also contributed the nextprime code and coauthored the bin_uiui code. He also wrote or maintained the binvert, mullow_n_basecase, powlo, redc_n code and the powm and powm_ui improvements. Torbjorn is also responsible for numerous other bits and pieces that have been used from the GNU project.

Marco Bodrato and Alberto Zanoni suggested the unbalanced multiply strategy and found optimal Toom multiplication sequences.

Marco Bodrato wrote an mpz implementation of the Toom 7 code and wrote most of the Toom 8.5 multiply and squaring code. He also helped write the divide and conquer Euclidean division code. He also contributed many improved number theoretical functions including factorial, multi-factorial, primorial, n-choose-k.

Marc Glisse improved gmpxx.h

Robert Gerbicz contributed fast factorial code.

Martin Boij made assorted contributions to the nextprime code.

David Harvey wrote fast middle product code and divide and conquer approximate quotient code for both Euclidean and Hensel division and contributed to the quadratic Hensel code.

T. R. Nicely wrote primality tests used in the benchmark code.

Jeff Gilchrist assisted with the porting of T. R. Nicely's primality code to MPIR and helped with tuning.

David Kirkby helped with build testing on Sun servers

Peter Shrimpton wrote the BPSW primality test used up to GMP_LIMB_BITS.

Thanks to Microsoft for supporting Jason Moxham to work on a command line build system for Windows and some assembly improvements for Windows.

Thanks to William Stein for giving us access to his sage.math machines for testing and for hosting the MPIR website, and for supporting us in inumerably many other ways.

Minh Van Nguyen served as release manager for MPIR 2.1.0.

Case Vanhorsen helped with release testing.

David Cleaver filed a bug report.

Julien Puydt provided tuning values.

Leif Lionhardy supplied build patches and provided tuning values.

Jean-Pierre Flori ported the powm, powm_ui improvements from GMP, supplied many build system patches and improvements and provided tuning values.

Thanks to an anonymous Japanese contributor for assembly improvements

Marshall Hampton reported an issue on apple machines

Jens Nurmann contributed significant quantities of Skylake assembly code and contributed assembly improvements that have been used elsewhere.

Alex Best wrote an assembly superoptimiser.

Vincent Delecroix ported mpq_cmp_z from GMP.

Sisyphus (Rob) submitted tuning values.

sav-ix (Alexander) provided a patch for t-locale on Windows.

Isurus Fernando provided tuning values, numerous build system patches, did release testing and helped with continuous integration.

Alex Dyachenko wrote mpir.net for interfacing MPIR to .net languages.

Tommy Hoffman supplied a sed patch.

Averkhaturau fixed a C++ compilation problem.

Marcell Keller fixed a sign conversion bug.

Sergey Taymanov fixed some Windows build file issues.

jengelh reported a bug and helped with build testing

Appendix B References

B.1 Books

- Jonathan M. Borwein and Peter B. Borwein, "Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity", Wiley, 1998.
- Henri Cohen, "A Course in Computational Algebraic Number Theory", Graduate Texts in Mathematics number 138, Springer-Verlag, 1993.
  `http://www.math.u-bordeaux.fr/~cohen/`
- Richard Crandall, Carl Pomerance, "Prime Numbers: A Computational Perspective" 2nd edition, Springer, 2005.
- Donald E. Knuth, "The Art of Computer Programming", volume 2, "Seminumerical Algorithms", 3rd edition, Addison-Wesley, 1998.
  `http://www-cs-faculty.stanford.edu/~knuth/taocp.html`
- John D. Lipson, "Elements of Algebra and Algebraic Computing", The Benjamin Cummings Publishing Company Inc, 1981.
- Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone, "Handbook of Applied Cryptography", `http://www.cacr.math.uwaterloo.ca/hac/`
- Richard M. Stallman, "Using and Porting GCC", Free Software Foundation, 1999, available online `http://gcc.gnu.org/onlinedocs/`, and in the GCC package `ftp://ftp.gnu.org/gnu/gcc/`

B.2 Papers

- Dan Bernstein, "Detecting perfect powers in essentially linear time", Math. Comp. (67) pp. 1253-1283, 1998.
- Yves Bertot, Nicolas Magaud and Paul Zimmermann, "A Proof of GMP Square Root", Journal of Automated Reasoning, volume 29, 2002, pp. 225-252. Also available online as INRIA Research Report 4475, June 2001, `http://www.inria.fr/rrrt/rr-4475.html`
- Marco Bodrato, Alberto Zanoni, "Integer and Polynomial Multiplication: Towards optimal Toom-Cook Matrices", ISAAC 2007 Proceedings, Ontario, Canada, July 29 - August 1, 2007, ACM Press. Available online at `http://ln.bodrato.it/issac2007_pdf`
- Marco Bodrato, "High degree Toom'n'half for balanced and unbalanced multiplication", E. Antelo, D. Hough and P. Ienne, editors, Proceedings of the 20th IEEE Symposium on Computer Arithmetic, IEEE, Tubingen, Germany, July 25-27, 2011, pp. 15–222. See `http://bodrato.it/papers`
- Richard Brent and Paul Zimmermann, "Modern Computer Arithmetic", version 0.4, November 2009, `http://www.loria.fr/~zimmerma/mca/mca-0.4.pdf`
- Christoph Burnikel and Joachim Ziegler, "Fast Recursive Division", Max-Planck-Institut fuer Informatik Research Report MPI-I-98-1-022,
  `http://data.mpi-sb.mpg.de/internet/reports.nsf/NumberView/1998-1-022`
- Agner Fog, "Software optimization resources", online at `http://www.agner.org/optimize/`
- Pierrick Gaudry, Alexander Kruppa, Paul Zimmermann, "A GMP-based implementation of Schoenhage-Strassen's large integer multiplication algorithm", ISAAC 2007 Proceedings, Ontario, Canada, July 29 - August 1, 2007, pp. 167-174, ACM Press. Full text available at `http://hal.inria.fr/docs/00/14/86/20/PDF/fft.final.pdf`
- Torbjorn Granlund and Peter L. Montgomery, "Division by Invariant Integers using Multiplication", in Proceedings of the SIGPLAN PLDI'94 Conference, June 1994. Also available `ftp://ftp.cwi.nl/pub/pmontgom/divcnst.psa4.gz` (and .psl.gz).

- Niels Möller and Torbjörn Granlund, "Improved division by invariant integers", to appear.
- Torbjörn Granlund and Niels Möller, "Division of integers large and small", to appear.
- David Harvey, "The Karatsuba middle product for integers", (preprint), 2009. Available at `http://www.cims.nyu.edu/~harvey/mulmid/mulmid.pdf`
- Tudor Jebelean, "An algorithm for exact division", Journal of Symbolic Computation, volume 15, 1993, pp. 169-180. Research report version available `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1992/92-35.ps.gz`
- Tudor Jebelean, "Exact Division with Karatsuba Complexity - Extended Abstract", RISC-Linz technical report 96-31, `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1996/96-31.ps.gz`
- Tudor Jebelean, "Practical Integer Division with Karatsuba Complexity", ISSAC 97, pp. 339-341. Technical report available `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1996/96-29.ps.gz`
- Tudor Jebelean, "A Generalization of the Binary GCD Algorithm", ISSAC 93, pp. 111-116. Technical report version available `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1993/93-01.ps.gz`
- Tudor Jebelean, "A Double-Digit Lehmer-Euclid Algorithm for Finding the GCD of Long Integers", Journal of Symbolic Computation, volume 19, 1995, pp. 145-157. Technical report version also available `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1992/92-69.ps.gz`
- Werner Krandick, Jeremy R. Johnson, "Efficient Multiprecision Floating Point Multiplication with Exact Rounding", Technical Report, RISC Linz, 1993, available at `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1993/93-76.ps.gz`
- Werner Krandick and Tudor Jebelean, "Bidirectional Exact Integer Division", Journal of Symbolic Computation, volume 21, 1996, pp. 441-455. Early technical report version also available `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1994/94-50.ps.gz`
- Makoto Matsumoto and Takuji Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Transactions on Modelling and Computer Simulation, volume 8, January 1998, pp. 3-30. Available online `http://www.math.keio.ac.jp/~nisimura/random/doc/mt.ps.gz` (or .pdf)
- R. Moenck and A. Borodin, "Fast Modular Transforms via Division", Proceedings of the 13th Annual IEEE Symposium on Switching and Automata Theory, October 1972, pp. 90-96. Reprinted as "Fast Modular Transforms", Journal of Computer and System Sciences, volume 8, number 3, June 1974, pp. 366-386.
- Niels Möller, "On Schoenhage's algorithm and subquadratic integer GCD computation", Math. Comp. 2007. Available online at `http://www.lysator.liu.se/~nisse/archive/S0025-5718-07-02017-0.pdf`
- Peter L. Montgomery, "Modular Multiplication Without Trial Division", in Mathematics of Computation, volume 44, number 170, April 1985.
- Thom Mulders, "On short multiplications and divisions", Appl. Algebra Engrg. Comm. Comput. 11 (2000), no. 1, pp. 69-88. Tech. report No. 276, Dept. of Comp. Sci., ETH Zurich, Nov 1997, available online at `ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/2xx/276.pdf`
- Arnold Schönhage and Volker Strassen, "Schnelle Multiplikation grosser Zahlen", Computing 7, 1971, pp. 281-292.
- A. Schönhage, A. F. W. Grotefeld and E. Vetter, "Fast Algorithms, A Multitape Turing Machine Implementation" BI Wissenschafts-Verlag, Mannheim, 1994.
- Kenneth Weber, "The accelerated integer GCD algorithm", ACM Transactions on Mathematical Software, volume 21, number 1, March 1995, pp. 111-122.

- Paul Zimmermann, "Karatsuba Square Root", INRIA Research Report 3805, November 1999, `http://www.inria.fr/rrrt/rr-3805.html`
- Paul Zimmermann, "A Proof of GMP Fast Division and Square Root Implementations", `http://www.loria.fr/~zimmerma/papers/proof-div-sqrt.ps.gz`
- Dan Zuras, "On Squaring and Multiplying Large Integers", ARITH-11: IEEE Symposium on Computer Arithmetic, 1993, pp. 260 to 271. Reprinted as "More on Multiplying and Squaring Large Integers", IEEE Transactions on Computers, volume 43, number 8, August 1994, pp. 899-908.

Appendix C GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

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

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

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

1. APPLICABILITY AND DEFINITIONS

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

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

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

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

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

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

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

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

The "publisher" means any person or entity that distributes copies of the Document to the public.

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

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

2. VERBATIM COPYING

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

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

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both

covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

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

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

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

4. MODIFICATIONS

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

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

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

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

D. Preserve all the copyright notices of the Document.

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

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

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

H. Include an unaltered copy of this License.

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

Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

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

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

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

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

O. Preserve any Warranty Disclaimers.

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

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

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

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

5. COMBINING DOCUMENTS

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

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

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled

"Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

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

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

7. AGGREGATION WITH INDEPENDENT WORKS

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

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

8. TRANSLATION

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

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

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been

terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

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

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

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

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

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

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

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

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

Concept Index

#

—

.

2

8

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Function and Type Index

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W