

LibTomFloat User Manual

v0.02

Tom St Denis
tomstdenis@iahu.ca

June 21, 2004

This text and the library are hereby placed in the public domain. This book has been formatted for B5 [176x250] paper using the L^AT_EX *book* macro package.

Open Source. Open Academia. Open Minds.

Tom St Denis,
Ontario, Canada

Contents

1	Introduction	1
1.1	What is LibTomFloat?	1
1.2	License	1
1.3	Building LibTomFloat	2
1.4	Purpose of LibTomFloat	2
1.5	How the types work	2
2	Getting Started with LibTomFloat	5
2.1	Building Programs	5
2.2	Return Codes	5
2.3	Data Types	6
2.4	Function Organization	6
2.5	Initialization	7
2.5.1	Single Initializers	7
2.5.2	Multiple Initializers	8
2.5.3	Initialization of Copies	8
2.6	Data Movement	9
2.6.1	Copying	9
2.6.2	Exchange	10
3	Basic Operations	11
3.1	Normalization	11
3.1.1	Simple Normalization	11
3.1.2	Normalize to New Radix	11
3.2	Constants	11
3.2.1	Quick Constants	11
3.3	Sign Manipulation	13

4	Basic Algebra	15
4.1	Algebraic Operators	15
4.1.1	Additional Interfaces	15
4.1.2	Additional Operators	16
4.2	Comparisons	16
5	Advanced Algebra	17
5.1	Powers	17
5.1.1	Exponential	17
5.1.2	Power Operator	17
5.1.3	Natural Logarithm	17
5.2	Inversion and Roots	18
5.2.1	Inverse Square Root	18
5.2.2	Inverse	18
5.2.3	Square Root	18
5.3	Trigonometry Functions	18

List of Figures

- 2.1 Return Codes 5
- 3.1 LibTomFloat Constants. 12

Chapter 1

Introduction

1.1 What is LibTomFloat?

LibTomFloat is a library of source code that provides multiple precision floating point arithmetic. It allows developers to manipulate floating point numbers of variable precision. The library was written in portable ISO C source code and depends upon the public domain LibTomMath package.

Along with providing the core mathematical operations such as addition and subtraction LibTomFloat also provides various complicated algorithms such as trigonometry's sine, cosine and tangent operators as well as Calculus's square root, inverse square root, exponential and logarithm operators.

LibTomFloat has been written for portability and numerical stability and is not particularly optimized for any given platform. It uses optimal algorithms for manipulating the mantissa by using LibTomMath and uses numerically stable series for the various trig and calculus functions.

1.2 License

LibTomFloat is public domain.

1.3 Building LibTomFloat

LibTomFloat requires version 0.30 or higher of LibTomMath to be installed in order to build. Once LibTomMath is installed building LibTomFloat is as simple as:

```
make
```

Which will build “libtomfloat.a” and along with “tomfloat.h” complete an installation of LibTomFloat. You can also use the make target “install” to automatically build and copy the files (into *NIX specific) locations.

```
make install
```

Note: LibTomFloat does not use ISO C’s native floating point types which means that the standard math library does not have to be linked in. This also means that LibTomFloat will work decently on platforms that do not have a floating point unit.

1.4 Purpose of LibTomFloat

LibTomFloat is as much as an exercise in hardcore math for myself as it is a service to any programmer who needs high precision float point data types. ISO C provides for fairly reasonable precision floating point data types but is limited. A proper analogy is LibTomFloat solves ISO C’s floating point problems in the same way LibTomMath solves ISO C’s integer data type problems.

A classic example of a good use for large precision floats is long simulations where the numbers are not perfectly stable. A 128-bit mantissa (for example) can provide for exceptional precision.

That and knowing the value of e to 512 bits is fun.

1.5 How the types work

The floating point types are emulated with three components. The **mantissa**, the **exponent** and the **radix**. The mantissa forms the digits of number being represented. The exponent scales the number to give it a larger range. The radix controls how many bits there are in the mantissa. The larger the radix the more precise the types become.

The representation of a number is given by the simple product $m \cdot 2^e$ where m is the mantissa and e the exponent. Numbers are always normalized such that there are *radix* bits per mantissa. For example, with *radix* = 16 the number 2 is represented by $32768 \cdot 2^{-14}$. A zero is represented by a mantissa of zero and an exponent of one and is a special case.

The sign flag is a standard ISO C “long” which gives it the range $2^{-31} \leq e < 2^{31}$ which is considerably large.

Technically, LibTomFloat does not implement IEEE standard floating point types. The exponent is not normalized and the sign flag does not count as a bit in the radix. There is also no “implied” bit in this system. The mantissa explicitly dictates the digits.

Chapter 2

Getting Started with LibTomFloat

2.1 Building Programs

In order to use libTomFloat you must include “tomfloat.h” and link against the appropriate library file (typically libtomfloat.a). There is no library initialization required and the entire library is thread safe.

2.2 Return Codes

There are three possible return codes a function may return.

Code	Meaning
MP_OKAY	The function succeeded.
MP_VAL	The function input was invalid.
MP_MEM	Heap memory exhausted.
MP_YES	Response is yes.
MP_NO	Response is no.

Figure 2.1: Return Codes

The last two codes listed are not actually “return’ed” by a function. They

are placed in an integer (the caller must provide the address of an integer it can store to) which the caller can access. To convert one of the three return codes to a string use the following function.

```
char *mp_error_to_string(int code);
```

This will return a pointer to a string which describes the given error code. It will not work for the return codes MP_YES and MP_NO.

2.3 Data Types

To better work with LibTomFloat it helps to know what makes up the primary data type within LibTomFloat.

```
typedef struct {
    mp_int mantissa;
    long  radix,
        exp;
} mp_float;
```

The `mp_float` data type is what all LibTomFloat functions will operate with and upon. The members of the structure are as follows:

1. The **mantissa** variable is a LibTomMath `mp_int` that represents the digits of the float. Since it's a `mp_int` it can accommodate any practical range of numbers.
2. The **radix** variable is the precision desired for the `mp_float` in bits. The higher the value the more precise (and slow) the calculations are. This value must be larger than two and ideally shouldn't be lower than what a "double" provides (55-bits of mantissa).
3. The **exp** variable is the exponent associated with the number.

2.4 Function Organization

Many of the functions operate as their LibTomMath counterparts. That is the source operands are on the left and the destination is on the right. For instance:

```
mpf_add(&a, &b, &c);      /* c = a + b */
mpf_mul(&a, &a, &c);      /* c = a * a */
mpf_div(&a, &b, &c);      /* c = a / b */
```

One major difference (and similar to LibTomPoly) is that the radix of the destination operation controls the radix of the internal computation and the final result. For instance, if a and b have a 24-bit mantissa and c has a 96-bit mantissa then all three operations are performed with 96-bits of precision.

This is non-issue for algorithms such as addition or multiplication but more important for the series calculations such as division, inversion, square roots, etc.

All functions normalize the result before returning.

2.5 Initialization

2.5.1 Single Initializers

To initialize or clear a single `mp_float` use the following two functions.

```
int mpf_init(mp_float *a, long radix);
void mpf_clear(mp_float *a);
```

`mpf_init` will initialize a with the given radix to the default value of zero. `mpf_clear` will free the memory used by the `mp_float`.

```
int main(void)
{
    mp_float a;
    int err;

    /* initialize a mp_float with a 96-bit mantissa */
    if ((err = mpf_init(&a, 96)) != MP_OKAY) {
        // error handle
    }

    /* we now have a 96-bit mp_float ready ... do work */

    /* done */
    mpf_clear(&a);
}
```

```

    return EXIT_SUCCESS;
}

```

2.5.2 Multiple Initializers

To initialize or clear multiple `mp_floats` simultaneously use the following two functions.

```

int mpf_init_multi(long radix, mp_float *a, ...);
void mpf_clear_multi(mp_float *a, ...);

```

`mpf_init_multi` will initialize a **NULL** terminated list of `mp_floats` with the same given radix. `mpf_clear_multi` will free up a **NULL** terminated list of `mp_floats`.

```

int main(void)
{
    mp_float a, b;
    int err;

    /* initialize two mp_floats with a 96-bit mantissa */
    if ((err = mpf_init_multi(96, &a, &b, NULL)) != MP_OKAY) {
        // error handle
    }

    /* we now have two 96-bit mp_floats ready ... do work */

    /* done */
    mpf_clear_multi(&a, &b, NULL);

    return EXIT_SUCCESS;
}

```

2.5.3 Initialization of Copies

In order to initialize an `mp_float` and make a copy of a source `mp_float` the following function has been provided.

```

int mpf_init_copy(mp_float *a, mp_float *b);

```

This will initialize *b* and make it a copy of *a*.

```
int main(void)
{
    mp_float a, b;
    int err;

    /* initialize a mp_float with a 96-bit mantissa */
    if ((err = mpf_init(&a, 96)) != MP_OKAY) {
        // error handle
    }

    /* we now have a 96-bit mp_float ready ... do work */

    /* now make our copy */
    if ((err = mpf_init_copy(&a, &b)) != MP_OKAY) {
        // error handle
    }

    /* now b is a copy of a */

    /* done */
    mpf_clear_multi(&a, &b, NULL);

    return EXIT_SUCCESS;
}
```

2.6 Data Movement

2.6.1 Copying

In order to copy one `mp_float` into another `mp_float` the following function has been provided.

```
int mpf_copy(mp_float *src, mp_float *dest);
```

This will copy the `mp_float` from *src* into *dest*. Note that the final radix of *dest* will be that of *src*.

```
int main(void)
{
    mp_float a, b;
    int err;

    /* initialize two mp_floats with a 96-bit mantissa */
    if ((err = mpf_init_multi(96, &a, &b, NULL)) != MP_OKAY) {
        // error handle
    }

    /* we now have two 96-bit mp_floats ready ... do work */

    /* put a into b */
    if ((err = mpf_copy(&a, &b)) != MP_OKAY) {
        // error handle
    }

    /* done */
    mpf_clear_multi(&a, &b, NULL);

    return EXIT_SUCCESS;
}
```

2.6.2 Exchange

To exchange the contents of two `mp_float` data types use this `f00`.

```
void mpf_exch(mp_float *a, mp_float *b);
```

This will swap the contents of *a* and *b*.

Chapter 3

Basic Operations

3.1 Normalization

3.1.1 Simple Normalization

Normalization is not required by the user unless they fiddle with the mantissa on their own. If that's the case you can use this function.

```
int mpf_normalize(mp_float *a);
```

This will fix up the mantissa of a such that the leading bit is one (if the number is non-zero).

3.1.2 Normalize to New Radix

In order to change the radix of a non-zero number you must call this function.

```
int mpf_normalize_to(mp_float *a, long radix);
```

This will change the radix of a then normalize it accordingly.

3.2 Constants

3.2.1 Quick Constants

The following are helpers for various numbers.

```
int mpf_const_0(mp_float *a);
int mpf_const_d(mp_float *a, long d);
int mpf_const_ln_d(mp_float *a, long b);
int mpf_const_sqrt_d(mp_float *a, long b);
```

`mpf_const_0` will set a to a valid representation of zero. `mpf_const_d` will set a to a valid signed representation of d . `mpf_const_ln_d` will set a to the natural logarithm of b . `mpf_const_sqrt_d` will set a to the square root of b .

The next set of constants (fig. 3.1) compute the standard constants as defined in “math.h”.

Function Name	Value
<code>mpf_const_e</code>	e
<code>mpf_const_l2e</code>	$\log_2(e)$
<code>mpf_const_l10e</code>	$\log_{10}(e)$
<code>mpf_const_le2</code>	$\ln(2)$
<code>mpf_const_pi</code>	π
<code>mpf_const_pi2</code>	$\pi/2$
<code>mpf_const_pi4</code>	$\pi/4$
<code>mpf_const_1pi</code>	$1/\pi$
<code>mpf_const_2pi</code>	$2/\pi$
<code>mpf_const_2rpi</code>	$2/\sqrt{\pi}$
<code>mpf_const_r2</code>	$\sqrt{2}$
<code>mpf_const_lr2</code>	$1/\sqrt{2}$

Figure 3.1: LibTomFloat Constants.

All of these functions accept a single input argument. They calculate the constant at run-time using the precision specified in the input argument.

```
int main(void)
{
    mp_float a;
    int err;

    /* initialize a mp_float with a 96-bit mantissa */
    if ((err = mpf_init(&a, 96)) != MP_OKAY) {
        // error handle
    }
}
```

```

/* let's find out what the square root of 2 is (approximately ;-)) */
if ((err = mpf_const_r2(&a)) != MP_OKAY) {
    // error handle
}

/* now a has sqrt(2) to 96-bits of precision */

/* done */
mpf_clear(&a);

return EXIT_SUCCESS;
}

```

3.3 Sign Manipulation

To manipulate the sign of a `mp_float` use the following two functions.

```

int mpf_abs(mp_float *a, mp_float *b);
int mpf_neg(mp_float *a, mp_float *b);

```

`mpf_abs` computes the absolute of a and stores it in b . `mpf_neg` computes the negative of a and stores it in b . Note that the numbers are normalized to the radix of b before being returned.

```

int main(void)
{
    mp_float a;
    int err;

    /* initialize a mp_float with a 96-bit mantissa */
    if ((err = mpf_init(&a, 96)) != MP_OKAY) {
        // error handle
    }

    /* let's find out what the square root of 2 is (approximately ;-)) */
    if ((err = mpf_const_r2(&a)) != MP_OKAY) {
        // error handle
    }
}

```

```
/* now make it negative */
if ((err = mpf_neg(&a, &a)) != MP_OKAY) {
    // error handle
}

/* done */
mpf_clear(&a);

return EXIT_SUCCESS;
}
```

Chapter 4

Basic Algebra

4.1 Algebraic Operators

The following four functions provide for basic addition, subtraction, multiplication and division of `mp_float` numbers.

```
int mpf_add(mp_float *a, mp_float *b, mp_float *c);
int mpf_sub(mp_float *a, mp_float *b, mp_float *c);
int mpf_mul(mp_float *a, mp_float *b, mp_float *c);
int mpf_div(mp_float *a, mp_float *b, mp_float *c);
```

These functions perform their respective operations on a and b and store the result in c .

4.1.1 Additional Interfaces

In order to make programming easier with the library the following four functions have been provided as well.

```
int mpf_add_d(mp_float *a, long b, mp_float *c);
int mpf_sub_d(mp_float *a, long b, mp_float *c);
int mpf_mul_d(mp_float *a, long b, mp_float *c);
int mpf_div_d(mp_float *a, long b, mp_float *c);
```

These work like the previous four functions except the second argument is a “long” type. This allow operations with mixed `mp_float` and integer types (specifically constants) to be performed relatively easy.

I will put an example of all op/op_d functions here...

4.1.2 Additional Operators

The next three functions round out the simple algebraic operators.

```
int mpf_mul_2(mpf_float *a, mpf_float *b);
int mpf_div_2(mpf_float *a, mpf_float *b);
int mpf_sqr(mpf_float *a, mpf_float *b);
```

`mpf_mul_2` and `mpf_div_2` multiply (or divide) a by two and store it in b . `mpf_sqr` squares a and stores it in b . `mpf_sqr` is faster than using `mpf_mul` for squaring `mp_floats`.

4.2 Comparisons

To compare two `mp_floats` the following function can be used.

```
int mpf_cmp(mpf_float *a, mp_float *b);
```

This will compare a to b and return one of the LibTomMath comparison flags. Simply put, if a is larger than b it returns `MP_GT`. If a is smaller than b it returns `MP_LT`, otherwise it returns `MP_EQ`. The comparison is signed.

To quickly compare an `mp_float` to a “long” the following is provided.

```
int mpf_cmp_d(mpf_float *a, long b, int *res);
```

Which compares a to b and stores the result in res . This function can fail which is unlike the digit compare from LibTomMath.

Chapter 5

Advanced Algebra

5.1 Powers

5.1.1 Exponential

The following function computes $\exp(x)$ otherwise known as e^x .

```
int mpf_exp(mp_float *a, mp_float *b);
```

This computes e^a and stores it into b .

5.1.2 Power Operator

The following function computes the generic a^b operation.

```
int mpf_pow(mp_float *a, mp_float *b, mp_float *c);
```

This computes a^b and stores the result in c .

5.1.3 Natural Logarithm

The following function computes the natural logarithm.

```
int mpf_ln(mp_float *a, mp_float *b);
```

This computes $\ln(a)$ and stores the result in b .

5.2 Inversion and Roots

5.2.1 Inverse Square Root

The following function computes $1/\sqrt{x}$.

```
int mpf_invsqrt(mp_float *a, mp_float *b);
```

This computes $1/\sqrt{a}$ and stores the result in b .

5.2.2 Inverse

The following function computes $1/x$.

```
int mpf_inv(mp_float *a, mp_float *b);
```

This computes $1/a$ and stores the result in b .

5.2.3 Square Root

The following function computes \sqrt{x} .

```
int mpf_sqrt(mp_float *a, mp_float *b);
```

This computes \sqrt{a} and stores the result in b .

5.3 Trigonometry Functions

The following functions compute various trigonometric functions. All inputs are assumed to be in radians.

```
int mpf_cos(mp_float *a, mp_float *b);  
int mpf_sin(mp_float *a, mp_float *b);  
int mpf_tan(mp_float *a, mp_float *b);  
int mpf_acos(mp_float *a, mp_float *b);  
int mpf_asin(mp_float *a, mp_float *b);  
int mpf_atan(mp_float *a, mp_float *b);
```

These all compute their respective trigonometric function on a and store the result in b . The “a” prefix stands for “arc” or more commonly known as inverse.

Index

exponent, 2

mantissa, 2

mp_cmp, 16

mp_error_to_string, 6

MP_MEM, 5

MP_NO, 5

MP_OKAY, 5

MP_VAL, 5

MP_YES, 5

mpf_abs, 13

mpf_acos, 18

mpf_add, 15

mpf_add_d, 15

mpf_asin, 18

mpf_atan, 18

mpf_clear, 7

mpf_clear_multi, 8

mpf_cmp_d, 16

mpf_const_0, 11

mpf_const_d, 11

mpf_const_ln_d, 11

mpf_const_sqrt_d, 11

mpf_copy, 9

mpf_cos, 18

mpf_div, 15

mpf_div_2, 16

mpf_div_d, 15

mpf_exch, 10

mpf_exp, 17

mpf_init, 7

mpf_init_copy, 8

mpf_init_multi, 8

mpf_inv, 18

mpf_invsqrt, 18

mpf_ln, 17

mpf_mul, 15

mpf_mul_2, 16

mpf_mul_d, 15

mpf_neg, 13

mpf_normalize, 11

mpf_normalize_to, 11

mpf_pow, 17

mpf_sin, 18

mpf_sqr, 16

mpf_sqrt, 18

mpf_sub, 15

mpf_sub_d, 15

mpf_tan, 18