

LibTomMath User Manual

v0.28

Tom St Denis
tomstdenis@iahu.ca

December 24, 2003

This text, the library and the accompanying textbook are all 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 LibTomMath? | 1 |
| 1.2 | License | 1 |
| 1.3 | Building LibTomMath | 2 |
| 1.3.1 | Testing | 2 |
| 1.4 | Purpose of LibTomMath | 3 |
| 2 | Getting Started with LibTomMath | 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 Initialization | 7 |
| 2.5.2 | Single Free | 7 |
| 2.5.3 | Multiple Initializations | 8 |
| 2.5.4 | Other Initializers | 9 |
| 2.6 | Maintenance Functions | 11 |
| 2.6.1 | Reducing Memory Usage | 11 |
| 2.6.2 | Adding additional digits | 12 |
| 3 | Basic Operations | 15 |
| 3.1 | Small Constants | 15 |
| 3.1.1 | Single Digit | 15 |
| 3.1.2 | Long Constant | 16 |
| 3.2 | Comparisons | 17 |
| 3.2.1 | Unsigned comparison | 17 |

| | | |
|----------|--|-----------|
| 3.2.2 | Signed comparison | 18 |
| 3.2.3 | Single Digit | 20 |
| 3.3 | Logical Operations | 21 |
| 3.3.1 | Multiplication by two | 21 |
| 3.3.2 | Polynomial Basis Operations | 23 |
| 3.3.3 | AND, OR and XOR Operations | 23 |
| 3.4 | Addition and Subtraction | 23 |
| 3.5 | Sign Manipulation | 24 |
| 3.5.1 | Negation | 24 |
| 3.5.2 | Absolute | 24 |
| 4 | Multiplication and Squaring | 25 |
| 4.1 | Multiplication | 25 |
| 4.2 | Squaring | 25 |
| 4.3 | Tuning Polynomial Basis Routines | 26 |
| 5 | Modular Reduction | 29 |
| 5.1 | Integer Division and Remainder | 29 |
| 5.2 | Barrett Reduction | 29 |
| 5.3 | Montgomery Reduction | 29 |
| 5.4 | Restricted Diminished Radix | 29 |
| 5.5 | Unrestricted Diminished Radix | 29 |
| 6 | Exponentiation | 31 |
| 6.1 | Single Digit Exponentiation | 31 |
| 6.2 | Modular Exponentiation | 31 |
| 6.3 | Root Finding | 31 |
| 7 | Prime Numbers | 33 |
| 7.1 | Trial Division | 33 |
| 7.2 | Fermat Test | 33 |
| 7.3 | Miller-Rabin Test | 33 |
| 7.3.1 | Required Number of Tests | 34 |
| 7.4 | Primality Testing | 34 |
| 7.5 | Next Prime | 34 |
| 7.6 | Random Primes | 35 |

| | | |
|----------|-----------------------------------|-----------|
| 8 | Input and Output | 37 |
| 8.1 | ASCII Conversions | 37 |
| 8.2 | Binary Conversions | 37 |
| 8.3 | Stream Functions | 37 |
| 9 | Algebraic Functions | 39 |
| 9.1 | Greatest Common Divisor | 39 |
| 9.2 | Least Common Multiple | 39 |
| 9.3 | Jacobi Symbol | 39 |
| 9.4 | Modular Inverse | 40 |
| 9.5 | Single Digit Functions | 40 |

List of Figures

| | | |
|-----|---|----|
| 1.1 | LibTomMath Valuation | 4 |
| 2.1 | Return Codes | 5 |
| 3.1 | Comparison Codes for a, b | 17 |
| 4.1 | Build Names for Tuning Programs | 26 |

Chapter 1

Introduction

1.1 What is LibTomMath?

LibTomMath is a library of source code which provides a series of efficient and carefully written functions for manipulating large integer numbers. It was written in portable ISO C source code so that it will build on any platform with a conforming C compiler.

In a nutshell the library was written from scratch with verbose comments to help instruct computer science students how to implement “bignum” math. However, the resulting code has proven to be very useful. It has been used by numerous universities, commercial and open source software developers. It has been used on a variety of platforms ranging from Linux and Windows based x86 to ARM based Gameboys and PPC based MacOS machines.

1.2 License

As of the v0.25 the library source code has been placed in the public domain with every new release. As of the v0.28 release the textbook “Implementing Multiple Precision Arithmetic” has been placed in the public domain with every new release as well. This textbook is meant to compliment the project by providing a more solid walkthrough of the development algorithms used in the library.

Since both¹ are in the public domain everyone is entitled to do with them as they see fit.

¹Note that the MPI files under mtest/ are copyrighted by Michael Fromberger.

1.3 Building LibTomMath

LibTomMath is meant to be very “GCC friendly” as it comes with a makefile well suited for GCC. However, the library will also build in MSVC, Borland C out of the box. For any other ISO C compiler a makefile will have to be made by the end developer.

To build the library for GCC simply issue the

```
make
```

command. This will build the library and archive the object files in “libtommath.a”. Now you simply link against that and include “tommath.h” within your programs.

Alternatively to build with MSVC type

```
nmake -f makefile.msvc
```

This will build the library and archive the object files in “tommath.lib”. This has been tested with MSVC version 6.00 with service pack 5.

There is limited support for making a “DLL” in windows via the “makefile.cygwin_dll” makefile. It requires Cygwin to work with since it requires the auto-export/import functionality. The resulting DLL and imprt library “libtomcrypt.dll.a” can be used to link LibTomMath dynamically to any Windows program using Cygwin.

1.3.1 Testing

To build the library and the test harness type

```
make test
```

This will build the library, “test” and “mtest/mtest”. The “test” program will accept test vectors and verify the results. “mtest/mtest” will generate test vectors using the MPI library by Michael Fromberger². Simply pipe mtest into test using

```
mtest/mtest | test
```

²A copy of MPI is included in the package

If you do not have a “/dev/urandom” style RNG source you will have to write your own PRNG and simply pipe that into `mtest`. For example, if your PRNG program is called “myprng” simply invoke

```
myprng | mtest/mtest | test
```

This will output a row of numbers that are increasing. Each column is a different test (such as addition, multiplication, etc) that is being performed. The numbers represent how many times the test was invoked. If an error is detected the program will exit with a dump of the relevant numbers it was working with.

1.4 Purpose of LibTomMath

Unlike GNU MP (GMP) Library, LIP, OpenSSL or various other commercial kits (Miracl), LibTomMath was not written with bleeding edge performance in mind. First and foremost LibTomMath was written to be entirely open. Not only is the source code public domain (unlike various other GPL/etc licensed code), not only is the code freely downloadable but the source code is also accessible for computer science students attempting to learn “BigNum” or multiple precision arithmetic techniques.

LibTomMath was written to be an instructive collection of source code. This is why there are many comments, only one function per source file and often I use a “middle-road” approach where I don’t cut corners for an extra 2% speed increase.

Source code alone cannot really teach how the algorithms work which is why I also wrote a textbook that accompanies the library (beat that!).

So you may be thinking “should I use LibTomMath?” and the answer is a definite maybe. Let me tabulate what I think are the pros and cons of LibTomMath by comparing it to the math routines from GnuPG³.

³GnuPG v1.2.3 versus LibTomMath v0.28

| Criteria | Pro | Con | Notes |
|-----------------------------------|-----|-----|--|
| Few lines of code per file | X | | GnuPG = 300.9, LibTomMath = 76.04 |
| Commented function prototypes | X | | GnuPG function names are cryptic. |
| Speed | | X | LibTomMath is slower. |
| Totally free | X | | GPL has unfavourable restrictions. |
| Large function base | X | | GnuPG is barebones. |
| Four modular reduction algorithms | X | | Faster modular exponentiation. |
| Portable | X | | GnuPG requires configuration to build. |

Figure 1.1: LibTomMath Valuation

It may seem odd to compare LibTomMath to GnuPG since the math in GnuPG is only a small portion of the entire application. However, LibTomMath was written with cryptography in mind. It provides essentially all of the functions a cryptosystem would require when working with large integers.

So it may feel tempting to just rip the math code out of GnuPG (or GnuMP where it was taken from originally) in your own application but I think there are reasons not to. While LibTomMath is slower than libraries such as GnuMP it is not normally significantly slower. On x86 machines the difference is normally a factor of two when performing modular exponentiations.

Essentially the only time you wouldn't use LibTomMath is when blazing speed is the primary concern.

Chapter 2

Getting Started with LibTomMath

2.1 Building Programs

In order to use LibTomMath you must include “tommath.h” and link against the appropriate library file (typically libtommath.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

The basic “multiple precision integer” type is known as the “mp_int” within LibTomMath. This data type is used to organize all of the data required to manipulate the integer it represents. Within LibTomMath it has been prototyped as the following.

```
typedef struct {
    int used, alloc, sign;
    mp_digit *dp;
} mp_int;
```

Where “mp_digit” is a data type that represents individual digits of the integer. By default, an mp_digit is the ISO C “unsigned long” data type and each digit is 28–bits long. The mp_digit type can be configured to suit other platforms by defining the appropriate macros.

All LTM functions that use the mp_int type will expect a pointer to mp_int structure. You must allocate memory to hold the structure itself by yourself (whether off stack or heap it doesn’t matter). The very first thing that must be done to use an mp_int is that it must be initialized.

2.4 Function Organization

The arithmetic functions of the library are all organized to have the same style prototype. That is source operands are passed on the left and the destination is on the right. For instance,

```
mp_add(&a, &b, &c);      /* c = a + b */
mp_mul(&a, &a, &c);      /* c = a * a */
mp_div(&a, &b, &c, &d);  /* c = [a/b], d = a mod b */
```

Another feature of the way the functions have been implemented is that source operands can be destination operands as well. For instance,

```
mp_add(&a, &b, &b);      /* b = a + b */
mp_div(&a, &b, &a, &c); /* a = [a/b], c = a mod b */
```

This allows operands to be re-used which can make programming simpler.

2.5 Initialization

2.5.1 Single Initialization

A single mp_int can be initialized with the “mp_init” function.

```
int mp_init (mp_int * a);
```

This function expects a pointer to an mp_int structure and will initialize the members of the structure so the mp_int represents the default integer which is zero. If the function returns MP_OKAY then the mp_int is ready to be used by the other LibTomMath functions.

```
int main(void)
{
    mp_int number;
    int result;

    if ((result = mp_init(&number)) != MP_OKAY) {
        printf("Error initializing the number. %s",
            mp_error_to_string(result));
        return EXIT_FAILURE;
    }

    /* use the number */

    return EXIT_SUCCESS;
}
```

2.5.2 Single Free

When you are finished with an mp_int it is ideal to return the heap it used back to the system. The following function provides this functionality.

```
void mp_clear (mp_int * a);
```

The function expects a pointer to a previously initialized `mp_int` structure and frees the heap it uses. It sets the pointer¹ within the `mp_int` to **NULL** which is used to prevent double free situations. It is legal to call `mp_clear()` twice on the same `mp_int` in a row.

```
int main(void)
{
    mp_int number;
    int result;

    if ((result = mp_init(&number)) != MP_OKAY) {
        printf("Error initializing the number.  %s",
              mp_error_to_string(result));
        return EXIT_FAILURE;
    }

    /* use the number */

    /* We're done with it. */
    mp_clear(&number);

    return EXIT_SUCCESS;
}
```

2.5.3 Multiple Initializations

Certain algorithms require more than one large integer. In these instances it is ideal to initialize all of the `mp_int` variables in an “all or nothing” fashion. That is, they are either all initialized successfully or they are all not initialized.

The `mp_init_multi()` function provides this functionality.

```
int mp_init_multi(mp_int *mp, ...);
```

It accepts a **NULL** terminated list of pointers to `mp_int` structures. It will attempt to initialize them all at once. If the function returns `MP_OKAY` then all of the `mp_int` variables are ready to use, otherwise none of them are available for use. A complementary `mp_clear_multi()` function allows multiple `mp_int` variables to be free'd from the heap at the same time.

¹The “dp” member.


```

int main(void)
{
    mp_int num1, num2, num3;
    int result;

    if ((result = mp_init_multi(&num1,
                               &num2,
                               &num3, NULL)) != MP_OKAY) {
        printf("Error initializing the numbers.  %s",
              mp_error_to_string(result));
        return EXIT_FAILURE;
    }

    /* use the numbers */

    /* We're done with them. */
    mp_clear_multi(&num1, &num2, &num3, NULL);

    return EXIT_SUCCESS;
}

```

2.5.4 Other Initializers

To initialize and make a copy of an mp_int the mp_init_copy() function has been provided.

```
int mp_init_copy (mp_int * a, mp_int * b);
```

This function will initialize “a” and make it a copy of “b” if all goes well.

```

int main(void)
{
    mp_int num1, num2;
    int result;

    /* initialize and do work on num1 ... */

    /* We want a copy of num1 in num2 now */
    if ((result = mp_init_copy(&num2, &num1)) != MP_OKAY) {
        printf("Error initializing the copy.  %s",
              mp_error_to_string(result));
        return EXIT_FAILURE;
    }
}

```

```

}

/* now num2 is ready and contains a copy of num1 */

/* We're done with them. */
mp_clear_multi(&num1, &num2, NULL);

return EXIT_SUCCESS;
}

```

Another less common initializer is `mp_init_size()` which allows the user to initialize an `mp_int` with a given default number of digits. By default, all initializers allocate **MP_PREC** digits. This function lets you override this behaviour.

```
int mp_init_size (mp_int * a, int size);
```

The “size” parameter must be greater than zero. If the function succeeds the `mp_int` “a” will be initialized to have “size” digits (which are all initially zero).

```
int main(void)
{
    mp_int number;
    int result;

    /* we need a 60-digit number */
    if ((result = mp_init_size(&number, 60)) != MP_OKAY) {
        printf("Error initializing the number.  %s",
            mp_error_to_string(result));
        return EXIT_FAILURE;
    }

    /* use the number */

    return EXIT_SUCCESS;
}

```

2.6 Maintenance Functions

2.6.1 Reducing Memory Usage

When an `mp_int` is in a state where it won't be changed again² excess digits can be removed to return memory to the heap with the `mp_shrink()` function.

```
int mp_shrink (mp_int * a);
```

This will remove excess digits of the `mp_int` “a”. If the operation fails the `mp_int` should be intact without the excess digits being removed. Note that you can use a shrunk `mp_int` in further computations, however, such operations will require heap operations which can be slow. It is not ideal to shrink `mp_int` variables that you will further modify in the system (unless you are seriously low on memory).

```
int main(void)
{
    mp_int number;
    int result;

    if ((result = mp_init(&number)) != MP_OKAY) {
        printf("Error initializing the number. %s",
            mp_error_to_string(result));
        return EXIT_FAILURE;
    }

    /* use the number [e.g. pre-computation] */

    /* We're done with it for now. */
    if ((result = mp_shrink(&number)) != MP_OKAY) {
        printf("Error shrinking the number. %s",
            mp_error_to_string(result));
        return EXIT_FAILURE;
    }

    /* use it .... */

    /* we're done with it. */
```

²A Diffie-Hellman modulus for instance.

```

    mp_clear(&number);

    return EXIT_SUCCESS;
}

```

2.6.2 Adding additional digits

Within the `mp_int` structure are two parameters which control the limitations of the array of digits that represent the integer the `mp_int` is meant to equal. The *used* parameter dictates how many digits are significant, that is, contribute to the value of the `mp_int`. The *alloc* parameter dictates how many digits are currently available in the array. If you need to perform an operation that requires more digits you will have to `mp_grow()` the `mp_int` to your desired size.

```
int mp_grow (mp_int * a, int size);
```

This will grow the array of digits of “a” to “size”. If the *alloc* parameter is already bigger than “size” the function will not do anything.

```

int main(void)
{
    mp_int number;
    int result;

    if ((result = mp_init(&number)) != MP_OKAY) {
        printf("Error initializing the number. %s",
              mp_error_to_string(result));
        return EXIT_FAILURE;
    }

    /* use the number */

    /* We need to add 20 digits to the number */
    if ((result = mp_grow(&number, number.alloc + 20)) != MP_OKAY) {
        printf("Error growing the number. %s",
              mp_error_to_string(result));
        return EXIT_FAILURE;
    }

    /* use the number */

```

```
/* we're done with it. */  
mp_clear(&number);  
  
return EXIT_SUCCESS;  
}
```


Chapter 3

Basic Operations

3.1 Small Constants

Setting `mp_ints` to small constants is a relatively common operation. To accommodate these instances there are two small constant assignment functions. The first function is used to set a single digit constant while the second sets an ISO C style “unsigned long” constant. The reason for both functions is efficiency. Setting a single digit is quick but the domain of a digit can change (it’s always at least $0 \dots 127$).

3.1.1 Single Digit

Setting a single digit can be accomplished with the following function.

```
void mp_set (mp_int * a, mp_digit b);
```

This will zero the contents of “a” and make it represent an integer equal to the value of “b”. Note that this function has a return type of **void**. It cannot cause an error so it is safe to assume the function succeeded.

```
int main(void)
{
    mp_int number;
    int result;

    if ((result = mp_init(&number)) != MP_OKAY) {
```

```

    printf("Error initializing the number. %s",
           mp_error_to_string(result));
    return EXIT_FAILURE;
}

/* set the number to 5 */
mp_set(&number, 5);

/* we're done with it. */
mp_clear(&number);

return EXIT_SUCCESS;
}

```

3.1.2 Long Constant

When you want to set a constant that is the size of an ISO C “unsigned long” and larger than a single digit the following function is provided.

```
int mp_set_int (mp_int * a, unsigned long b);
```

This will assign the value of the 32-bit variable “b” to the mp_int “a”. Unlike mp_set() this function will always accept a 32-bit input regardless of the size of a single digit. However, since the value may span several digits this function can fail if it runs out of heap memory.

```

int main(void)
{
    mp_int number;
    int result;

    if ((result = mp_init(&number)) != MP_OKAY) {
        printf("Error initializing the number. %s",
               mp_error_to_string(result));
        return EXIT_FAILURE;
    }

    /* set the number to 654321 (note this is bigger than 127) */
    if ((result = mp_set_int(&number, 654321)) != MP_OKAY) {
        printf("Error setting the value of the number. %s",
               mp_error_to_string(result));
        return EXIT_FAILURE;
    }
}

```



```

    }
    /* we're done with it. */
    mp_clear(&number);

    return EXIT_SUCCESS;
}

```

3.2 Comparisons

Comparisons in LibTomMath are always performed in a “left to right” fashion. There are three possible return codes for any comparison.

| Result Code | Meaning |
|-------------|---------|
| MP_GT | $a > b$ |
| MP_EQ | $a = b$ |
| MP_LT | $a < b$ |

Figure 3.1: Comparison Codes for a, b

In figure 3.1 two integers a and b are being compared. In this case a is said to be “to the left” of b .

3.2.1 Unsigned comparison

An unsigned comparison considers only the digits themselves and not the associated *sign* flag of the `mp_int` structures. This is analogous to an absolute comparison. The function `mp_cmp_mag()` will compare two `mp_int` variables based on their digits only.

```
int mp_cmp(mp_int * a, mp_int * b);
```

This will compare “a” to “b” placing “a” to the left of “b”. This function cannot fail and will return one of the three compare codes listed in figure 3.1.

```
int main(void)
{
    mp_int number1, number2;
    int result;

```

```

if ((result = mp_init_multi(&number1, &number2, NULL)) != MP_OKAY) {
    printf("Error initializing the numbers. %s",
        mp_error_to_string(result));
    return EXIT_FAILURE;
}

/* set the number1 to 5 */
mp_set(&number1, 5);

/* set the number2 to -6 */
mp_set(&number2, 6);
if ((result = mp_neg(&number2, &number2)) != MP_OKAY) {
    printf("Error negating number2. %s",
        mp_error_to_string(result));
    return EXIT_FAILURE;
}

switch(mp_cmp_mag(&number1, &number2)) {
    case MP_GT: printf("|number1| > |number2|"); break;
    case MP_EQ: printf("|number1| = |number2|"); break;
    case MP_LT: printf("|number1| < |number2|"); break;
}

/* we're done with it. */
mp_clear_multi(&number1, &number2, NULL);

return EXIT_SUCCESS;
}

```

If this program¹ completes successfully it should print the following.

```
|number1| < |number2|
```

This is because $|-6| = 6$ and obviously $5 < 6$.

3.2.2 Signed comparison

To compare two `mp_int` variables based on their signed value the `mp_cmp()` function is provided.

```
int mp_cmp(mp_int * a, mp_int * b);
```

¹This function uses the `mp_neg()` function which is discussed in section 3.5.1.

This will compare “a” to the left of “b”. It will first compare the signs of the two mp_int variables. If they differ it will return immediately based on their signs. If the signs are equal then it will compare the digits individually. This function will return one of the compare conditions codes listed in figure 3.1.

```
int main(void)
{
    mp_int number1, number2;
    int result;

    if ((result = mp_init_multi(&number1, &number2, NULL)) != MP_OKAY) {
        printf("Error initializing the numbers. %s",
            mp_error_to_string(result));
        return EXIT_FAILURE;
    }

    /* set the number1 to 5 */
    mp_set(&number1, 5);

    /* set the number2 to -6 */
    mp_set(&number2, 6);
    if ((result = mp_neg(&number2, &number2)) != MP_OKAY) {
        printf("Error negating number2. %s",
            mp_error_to_string(result));
        return EXIT_FAILURE;
    }

    switch(mp_cmp(&number1, &number2)) {
        case MP_GT: printf("number1 > number2"); break;
        case MP_EQ: printf("number1 = number2"); break;
        case MP_LT: printf("number1 < number2"); break;
    }

    /* we're done with it. */
    mp_clear_multi(&number1, &number2, NULL);

    return EXIT_SUCCESS;
}
```

If this program² completes successfully it should print the following.

```
number1 > number2
```

²This function uses the mp_neg() function which is discussed in section 3.5.1.

3.2.3 Single Digit

To compare a single digit against an `mp_int` the following function has been provided.

```
int mp_cmp_d(mp_int * a, mp_digit b);
```

This will compare “a” to the left of “b” using a signed comparison. Note that it will always treat “b” as positive. This function is rather handy when you have to compare against small values such as 1 (which often comes up in cryptography). The function cannot fail and will return one of the tree compare condition codes listed in figure 3.1.

```
int main(void)
{
    mp_int number;
    int result;

    if ((result = mp_init(&number)) != MP_OKAY) {
        printf("Error initializing the number. %s",
            mp_error_to_string(result));
        return EXIT_FAILURE;
    }

    /* set the number to 5 */
    mp_set(&number, 5);

    switch(mp_cmp_d(&number, 7)) {
        case MP_GT:  printf("number > 7"); break;
        case MP_EQ:  printf("number = 7"); break;
        case MP_LT:  printf("number < 7"); break;
    }

    /* we're done with it. */
    mp_clear(&number);

    return EXIT_SUCCESS;
}
```

If this program functions properly it will print out the following.

```
number < 7
```

3.3 Logical Operations

Logical operations are operations that can be performed either with simple shifts or boolean operators such as AND, XOR and OR directly. These operations are very quick.

3.3.1 Multiplication by two

Multiplications and divisions by any power of two can be performed with quick logical shifts either left or right depending on the operation.

When multiplying or dividing by two a special case routine can be used which are as follows.

```
int mp_mul_2(mp_int * a, mp_int * b);
int mp_div_2(mp_int * a, mp_int * b);
```

The former will assign twice “a” to “b” while the latter will assign half “a” to “b”. These functions are fast since the shift counts and masks are hardcoded into the routines.

```
int main(void)
{
    mp_int number;
    int result;

    if ((result = mp_init(&number)) != MP_OKAY) {
        printf("Error initializing the number. %s",
            mp_error_to_string(result));
        return EXIT_FAILURE;
    }

    /* set the number to 5 */
    mp_set(&number, 5);

    /* multiply by two */
    if ((result = mp_mul_2(&number, &number)) != MP_OKAY) {
        printf("Error multiplying the number. %s",
            mp_error_to_string(result));
        return EXIT_FAILURE;
    }
    switch(mp_cmp_d(&number, 7)) {
        case MP_GT: printf("2*number > 7"); break;
    }
}
```

```

        case MP_EQ: printf("2*number = 7"); break;
        case MP_LT: printf("2*number < 7"); break;
    }

    /* now divide by two */
    if ((result = mp_div_2(&number, &number)) != MP_OKAY) {
        printf("Error dividing the number. %s",
            mp_error_to_string(result));
        return EXIT_FAILURE;
    }
    switch(mp_cmp_d(&number, 7)) {
        case MP_GT: printf("2*number/2 > 7"); break;
        case MP_EQ: printf("2*number/2 = 7"); break;
        case MP_LT: printf("2*number/2 < 7"); break;
    }

    /* we're done with it. */
    mp_clear(&number);

    return EXIT_SUCCESS;
}

```

If this program is successful it will print out the following text.

```

2*number > 7
2*number/2 < 7

```

Since $10 > 7$ and $5 < 7$. To multiply by a power of two the following function can be used.

```
int mp_mul_2d(mp_int * a, int b, mp_int * c);
```

This will multiply “a” by 2^b and store the result in “c”. If the value of b is less than or equal to zero the function will copy “a” to “c” without performing any further actions.

To divide by a power of two use the following.

```
int mp_div_2d (mp_int * a, int b, mp_int * c, mp_int * d);
```

Which will divide “a” by 2^b , store the quotient in “c” and the remainder in “d”. If $b \leq 0$ then the function simply copies “a” over to “c” and zeroes “d”. The variable “d” may be passed as a **NULL** value to signal that the remainder is not desired.

3.3.2 Polynomial Basis Operations

Strictly speaking the organization of the integers within the `mp_int` structures is what is known as a “polynomial basis”. This simply means a field element is stored by divisions of a radix. For example, if $f(x) = \sum_{i=0}^k y_i x^i$ for any vector \vec{y} then the array of digits in \vec{y} are said to be the polynomial basis representation of z if $f(\beta) = z$ for a given radix β .

To multiply by the polynomial $g(x) = x$ all you have to do is shift the digits of the basis left one place. The following function provides this operation.

```
int mp_lshd (mp_int * a, int b);
```

This will multiply “a” in place by x^b which is equivalent to shifting the digits left b places and inserting zeroes in the least significant digits. Similarly to divide by a power of x the following function is provided.

```
void mp_rshd (mp_int * a, int b)
```

This will divide “a” in place by x^b and discard the remainder. This function cannot fail as it performs the operations in place and no new digits are required to complete it.

3.3.3 AND, OR and XOR Operations

While AND, OR and XOR operations are not typical “bignum functions” they can be useful in several instances. The three functions are prototyped as follows.

```
int mp_or   (mp_int * a, mp_int * b, mp_int * c);
int mp_and  (mp_int * a, mp_int * b, mp_int * c);
int mp_xor  (mp_int * a, mp_int * b, mp_int * c);
```

Which compute $c = a \odot b$ where \odot is one of OR, AND or XOR.

3.4 Addition and Subtraction

To compute an addition or subtraction the following two functions can be used.

```
int mp_add  (mp_int * a, mp_int * b, mp_int * c);
int mp_sub  (mp_int * a, mp_int * b, mp_int * c)
```

Which perform $c = a \odot b$ where \odot is one of signed addition or subtraction. The operations are fully sign aware.

3.5 Sign Manipulation

3.5.1 Negation

Simple integer negation can be performed with the following.

```
int mp_neg (mp_int * a, mp_int * b);
```

Which assigns $-b$ to a .

3.5.2 Absolute

Simple integer absolute can be performed with the following.

```
int mp_abs (mp_int * a, mp_int * b);
```

Which assigns $|b|$ to a .

Chapter 4

Multiplication and Squaring

4.1 Multiplication

A full signed integer multiplication can be performed with the following.

```
int mp_mul (mp_int * a, mp_int * b, mp_int * c);
```

Which assigns the full signed product ab to c . This function actually breaks into one of four cases which are specific multiplication routines optimized for given parameters. First there are the Toom-Cook multiplications which should only be used with very large inputs. This is followed by the Karatsuba multiplications which are for moderate sized inputs. Then followed by the Comba and baseline multipliers.

Fortunately for the developer you don't really need to know this unless you really want to fine tune the system. `mp_mul()` will determine on its own¹ what routine to use automatically when it is called.

4.2 Squaring

Since squaring can be performed faster than multiplication it is performed it's own function instead of just using `mp_mul()`.

```
int mp_sqr (mp_int * a, mp_int * b);
```

¹Some tweaking may be required.

Will square “a” and store it in “b”. Like the case of multiplication there are four different squaring algorithms all which can be called from `mp_sqr()`.

4.3 Tuning Polynomial Basis Routines

Both Toom-Cook and Karatsuba multiplication algorithms are faster than the traditional $O(n^2)$ approach that the Comba and baseline algorithms use. At $O(n^{1.46})$ and $O(n^{1.58})$ running times respectfully they require considerably less work. For example, a 10000-digit multiplication would take roughly 692,000 single precision multiplications with Toom-Cook or 100,000,000 single precision multiplications with the standard Comba (a factor of 144).

So why not always use Karatsuba or Toom-Cook? The simple answer is that they have so much overhead that they’re not actually faster than Comba until you hit distinct “cutoff” points. For Karatsuba with the default configuration, GCC 3.3.1 and an Athlon XP processor the cutoff point is roughly 110 digits (about 70 for the Intel P4). That is, at 110 digits Karatsuba and Comba multiplications just about break even and for 110+ digits Karatsuba is faster.

Toom-Cook has incredible overhead and is probably only useful for very large inputs. So far no known cutoff points exist and for the most part I just set the cutoff points very high to make sure they’re not called.

A demo program in the “etc/” directory of the project called “tune.c” can be used to find the cutoff points. This can be built with GCC as follows

```
make XXX
```

Where “XXX” is one of the following entries from the table 4.1.

| Value of XXX | Meaning |
|--------------|--|
| tune | Builds portable tuning application |
| tune86 | Builds x86 (pentium and up) program for COFF |
| tune86c | Builds x86 program for Cygwin |
| tune86l | Builds x86 program for Linux (ELF format) |

Figure 4.1: Build Names for Tuning Programs

When the program is running it will output a series of measurements for different cutoff points. It will first find good Karatsuba squaring and multiplication points. Then it proceeds to find Toom-Cook points. Note that the

Toom-Cook tuning takes a very long time as the cutoff points are likely to be very high.

Chapter 5

Modular Reduction

5.1 Integer Division and Remainder

To perform a complete and general integer division with remainder use the following function.

```
int mp_div (mp_int * a, mp_int * b, mp_int * c, mp_int * d);
```

This divides “a” by “b” and stores the quotient in “c” and “d”. The signed quotient is computed such that $bc + d = a$. Note that either of “c” or “d” can be set to **NULL** if their value is not required.

5.2 Barrett Reduction

5.3 Montgomery Reduction

5.4 Restricted Diminished Radix

5.5 Unrestricted Diminished Radix

Chapter 6

Exponentiation

6.1 Single Digit Exponentiation

```
int mp_expt_d (mp_int * a, mp_digit b, mp_int * c)
```

This computes $c = a^b$ using a simple binary left-to-write algorithm. It is faster than repeated multiplications for all values of b greater than three.

6.2 Modular Exponentiation

```
int mp_exptmod (mp_int * G, mp_int * X, mp_int * P, mp_int * Y)
```

This computes $Y \equiv G^X \pmod{P}$ using a variable width sliding window algorithm. This function will automatically detect the fastest modular reduction technique to use during the operation. For negative values of X the operation is performed as $Y \equiv (G^{-1} \pmod{P})^{|X|} \pmod{P}$ provided that $\gcd(G, P) = 1$.

6.3 Root Finding

```
int mp_n_root (mp_int * a, mp_digit b, mp_int * c)
```

This computes $c = a^{1/b}$ such that $c^b \leq a$ and $(c+1)^b > a$. The implementation of this function is not ideal for values of b greater than three. It will work but become very slow. So unless you are working with very small numbers (less than

1000 bits) I'd avoid $b > 3$ situations. Will return a positive root only for even roots and return a root with the sign of the input for odd roots. For example, performing $4^{1/2}$ will return 2 whereas $(-8)^{1/3}$ will return -2 .

Chapter 7

Prime Numbers

7.1 Trial Division

```
int mp_prime_is_divisible (mp_int * a, int *result)
```

This will attempt to evenly divide a by a list of primes¹ and store the outcome in “result”. That is if $result = 0$ then a is not divisible by the primes, otherwise it is. Note that if the function does not return **MP_OKAY** the value in “result” should be considered undefined².

7.2 Fermat Test

```
int mp_prime_fermat (mp_int * a, mp_int * b, int *result)
```

Performs a Fermat primality test to the base b . That is it computes $b^a \bmod a$ and tests whether the value is equal to b or not. If the values are equal then a is probably prime and $result$ is set to one. Otherwise $result$ is set to zero.

7.3 Miller-Rabin Test

```
int mp_prime_miller_rabin (mp_int * a, mp_int * b, int *result)
```

¹Default is the first 256 primes.

²Currently the default is to set it to zero first.

Performs a Miller-Rabin test to the base b of a . This test is much stronger than the Fermat test and is very hard to fool (besides with Carmichael numbers). If a passes the test (therefore is probably prime) *result* is set to one. Otherwise *result* is set to zero.

Note that is suggested that you use the Miller-Rabin test instead of the Fermat test since all of the failures of Miller-Rabin are a subset of the failures of the Fermat test.

7.3.1 Required Number of Tests

Generally to ensure a number is very likely to be prime you have to perform the Miller-Rabin with at least a half-dozen or so unique bases. However, it has been proven that the probability of failure goes down as the size of the input goes up. This is why a simple function has been provided to help out.

```
int mp_prime_rabin_miller_trials(int size)
```

This returns the number of trials required for a 2^{-96} (or lower) probability of failure for a given “size” expressed in bits. This comes in handy specially since larger numbers are slower to test. For example, a 512-bit number would require ten tests whereas a 1024-bit number would only require four tests.

You should always still perform a trial division before a Miller-Rabin test though.

7.4 Primality Testing

```
int mp_prime_is_prime (mp_int * a, int t, int *result)
```

This will perform a trial division followed by t rounds of Miller-Rabin tests on a and store the result in *result*. If a passes all of the tests *result* is set to one, otherwise it is set to zero. Note that t is bounded by $1 \leq t < PRIME_SIZE$ where $PRIME_SIZE$ is the number of primes in the prime number table (by default this is 256).

7.5 Next Prime

```
int mp_prime_next_prime(mp_int *a, int t, int bbs_style)
```

This finds the next prime after a that passes `mp_prime_is_prime()` with t tests. Set `bbs_style` to one if you want only the next prime congruent to 3 mod 4, otherwise set it to zero to find any next prime.

7.6 Random Primes

```
int mp_prime_random(mp_int *a, int t, int size, int bbs,  
                  ltm_prime_callback cb, void *dat)
```

This will find a prime greater than 256^{size} which can be “bbs_style” or not depending on `bbs` and must pass t rounds of tests. The “`ltm_prime_callback`” is a typedef for

```
typedef int ltm_prime_callback(unsigned char *dst, int len, void *dat);
```

Which is a function that must read len bytes (and return the amount stored) into `dst`. The `dat` variable is simply copied from the original input. It can be used to pass RNG context data to the callback.

The function `mp_prime_random()` is more suitable for generating primes which must be secret (as in the case of RSA) since there is no skew on the least significant bits.

Chapter 8

Input and Output

8.1 ASCII Conversions

8.2 Binary Conversions

8.3 Stream Functions

Chapter 9

Algebraic Functions

9.1 Greatest Common Divisor

```
int mp_gcd (mp_int * a, mp_int * b, mp_int * c)
```

This will compute the greatest common divisor of a and b and store it in c .

9.2 Least Common Multiple

```
int mp_lcm (mp_int * a, mp_int * b, mp_int * c)
```

This will compute the least common multiple of a and b and store it in c .

9.3 Jacobi Symbol

```
int mp_jacobi (mp_int * a, mp_int * p, int *c)
```

This will compute the Jacobi symbol for a with respect to p . If p is prime this essentially computes the Legendre symbol. The result is stored in c and can take on one of three values $\{-1, 0, 1\}$. If p is prime then the result will be -1 when a is not a quadratic residue modulo p . The result will be 0 if a divides p and the result will be 1 if a is a quadratic residue modulo p .

9.4 Modular Inverse

```
int mp_invmod (mp_int * a, mp_int * b, mp_int * c)
```

Computes the multiplicative inverse of a modulo b and stores the result in c such that $ac \equiv 1 \pmod{b}$.

9.5 Single Digit Functions

Index

mp_add, 23
mp_and, 23
mp_clear, 7
mp_clear_multi, 8
mp_cmp, 18
mp_cmp_d, 20
mp_cmp_mag, 17
mp_div, 29
mp_div_2, 21
mp_div_2d, 22
MP_EQ, 17
mp_error_to_string, 6
mp_expt_d, 31
mp_exptmod, 31
mp_gcd, 39
mp_grow, 12
MP_GT, 17
mp_init, 7
mp_init_copy, 9
mp_init_multi, 8
mp_init_size, 10
mp_int, 6
mp_invmod, 40
mp_jacobi, 39
mp_lcm, 39
mp_lshd, 23
MP_LT, 17
MP_MEM, 5
mp_mul, 25
mp_mul_2, 21
mp_mul_2d, 22
mp_n_root, 31
mp_neg, 24
MP_NO, 5
MP_OKAY, 5
mp_or, 23
mp_prime_fermat, 33
mp_prime_is_divisible, 33
mp_prime_is_prime, 34
mp_prime_miller_rabin, 33
mp_prime_next_prime, 34
mp_prime_rabin_miller_trials, 34
mp_prime_random, 35
mp_rshd, 23
mp_set, 15
mp_set_int, 16
mp_shrink, 11
mp_sqr, 25
mp_sub, 23
MP_VAL, 5
mp_xor, 23
MP_YES, 5