MPINT - Documentation
Version 1.0

Author:
Grégory Essertel

28th January 2016

Contents

1 Introduction

MPINT is a multi-precision integer library supported for the Xinu operating system. Tested so far only for the Galileo version.

This library provides the basic arithmetic operation such as addition, subtraction, multiplication, division etc. The initial design is not efficiency.

2 Background

The reader is expecting to have a knowledge about Xinu, more precisely the low level memory management with *getmem* and *freemem*. Also the constant $SYSERR$ is used in this document.

3 Setup

Some constants have to be set in the *mpint.h* file.

The operations require to define an HALF_WORD. The HALF_WORD should be half of the size of the registers, this trick is used to handle carries easily. On the Galileo board for example we define:

```
typedef uint16  HALF_WORD;
typedef uint32  WORD;
typedef int32   SWORD;
```

Some other constant are used for the computation:

```
#define HW_BITS (16)        // 8 * sizeof(HALF_WORD)
#define HW_MASK (0xFFFF)    // 1 << HW_BITS - 1
```

NOTE: the HALF_WORD and WORD have to be unsigned.

4 Data Structure

```
typedef struct mpint {
  HALF_WORD*  val;
  uint32      size;
  uint32      order;
  int32       sign;
} MPINT;
```

Code 1: MPINT structure

The basic structure used in the complete project is the $MPINT$. The structure stores a pointer to an array of $HALF\_WORD$ as well as the size of the corresponding allocated memory. The little endian convention is used here, meaning that the value of a number of two $HALF\_WORD$ is "$val[1] << HW\_BITS + val[0]$". The size of the array has to be a multiple of the variable $MPINT\_HWINIT$ plus one $HALF\_WORD$. When more space is required, the size is increase by $MPINT\_HWINIT$.

```
#define MPINT_INIT   1024
#define MPINT_HWINIT   64 // MPINT_INIT / (8 * sizeof(HALF_WORD))
```

With these values, the smaller $MPINT$ is a 1024 bits integer and 2048, 3062... are possible values.

The order of a number is defined as the index of the Most Significant Halfword +1. Therefore a number is equal to zero if and only if the order is equal to 0.

The sign is either +1 or -1. When constructed 0 has the sign +1, but there is no grantees on the sign of 0 if it is the output of an operation.

5 Utils

The library offers some utility function in order to manipulate the $MPINT$.

5.1 new_mpint

Prototype

```
MPINT* new_mpint();
```

Description Allocate and initialize a new $MPINT$ on the heap.

Error Return $SYSERR$ if an error occurs (Out Of Memory)

5.2 delete_mpint

Prototype

```
void delete_mpint(MPINT*);
```

Description Release memory used by an $MPINT$ previously allocated on the heap using *new_mpint*.

5.3 init_mpint

Prototype

```
void init_mpint(MPINT*);
```

Description Initialize a $MPINT$.

Note Initializing an already initialized $MPINT$ can lead to memory leak.

5.4 clear_mpint

Prototype

```
void clear_mpint(MPINT*);
```

Description Releases the memory used by a $MPINT$, but not the memory used by the structure $MPINT$.

Note Generally used for a $MPINT$ declared on the stack.

5.5 extend_mpint

Prototype

```
bool8 extend_mpint(MPINT* n, int32 new_size);
```

Description Extends the size of n to be at least *new_size*.

Note This function is destined to be used by developers, a user should not have to ue it.

Return the function returns $TRUE$ if it has been successful, $FALSE$ otherwise.

5.6 write_bigendian_mpint

Prototype

```
void write_bigendian_mpint(byte* buff, MPINT* src, int32 len);
```

Description Write the *len* less significant bytes of *src* into the buffer *buff*

5.7 hton_mpint

Prototype

```
int32 hton_mpint(byte* buff, MPINT* src, int32 len);
```

Description Formats *src* into a machine independant representation using maximum *len* bytes. The format is: —length (4 bytes)—sign (0 or 1 byte)— value using big endian representation.

Return The function returns the number of bytes used or $SYSERR$ if an error occured. (Buffer to small).

5.8 read_bigendian_mpint

Prototype

```
bool8 read_bigendian_mpint(MPINT* dst, byte* buff, int32 length);
```

Description Sets the value of the *dst* $MPINT$ to the value in the buffer using the *length* first byte in a big endian order.

Return The function returns $TRUE$ if it has been successful, $FALSE$ otherwise.

5.9 ntoh_mpint

Prototype

```
bool8 ntoh_mpint(MPINT* dst, byte* buff);
```

Description Reads a $MPINT$ formated with the *hton_mpint* function.

Return The function returns the number of bytes read and $SYSERR$ if an error occured.

5.10 copy_mpint

Prototype

```
bool8 copy_mpint(MPINT* dst, MPINT* src);
```

Description Sets the value of *dst* to the value of *src*.

Return The function returns $TRUE$ if it has been successful, $FALSE$ otherwise.

5.11 subval_mpint

Prototype

```
bool8 subval_mpint(MPINT* dst, int32 offset, MPINT* src, int32 start,
int32 length);
```

Description Sets the value of *dst* between the *HALF_WORD* *offset* and *offset* + *length* - 1 to the value of *src* between the *HALF_WORD* *start* and *start* + *length* - 1.

Return The function returns *TRUE* if it has been successful, *FALSE* otherwise.

5.12 subvalghost_mpint

Prototype

```
bool8 subvalghost_mpint(MPINT* dst, MPINT* src, int32 start, int32 length);
```

Description Sets the value of *dst* to the value of *src* between the *HALF_WORD* *start* and *start* + *length* - 1.

Note WARNING: This function does not copy the value, therefore it should only be used in a read only mode.

Return The function returns *TRUE* if it has been successful, *FALSE* otherwise.

5.13 uporder_mpint

Prototype

```
bool8 uporder_mpint(MPINT* n);
```

Description Updates the order of the *MPINT*

Note This function is for developer only, user should not have to us it.

Return The function returns *FALSE* if the order is greater than the size of the *MPINT*.

5.14 swap_mpint

Prototype

```
void swap_mpint(MPINT* v1, MPINT* v2);
```

Description Swaps two *MPINT*.

5.15 parse_mpint

Prototype

```
bool8 parse_mpint(MPINT* n, char* val, int32 length);
```

Description Parses an *MPINT* from the *length* first character of the string *val* (lower and upper case for a,b,c,d, e and f are accepted). Negative value can be parsed as well.

Note If a non hexadecimal digit is found in the string of character, the behavior is undefined (other than the '-' at the beginning).

Return The function returns *TRUE* if it has been successful, *FALSE* otherwise.

5.16 print_mpint

Prototype

```
void print_mpint(MPINT* n);
```

Description Prints a $MPINT$ in its hexadecimal representation.

6 Comparaison

6.1 cmp_mpint

Prototype

```
int32 cmp_mpint(MPINT* v1, MPINT* v2);
```

Description Compares two $MPINT$.

Return 1 if $v1 > v2$, 0 if $v1 = v2$ and -1 if $v1 < v2$

6.2 unsigncmp_mpint

Prototype

```
int32 unsigncmp_mpint(MPINT* v1, MPINT* v2);
```

Description Compares the absolute value of two $MPINT$.

Return 1 if $|v1| > |v2|$, 0 if $|v1| = |v2|$ and -1 if $|v1| < |v2|$

7 Summation

7.1 add_mpint / sub_mpint

Prototype

```
bool8 add_mpint(MPINT* v1, MPINT* v2);
bool8 sub_mpint(MPINT* v1, MPINT* v2);
```

Description Computes the operation $v1 + / - v2$ and stores the result in *v1*.

Return The function returns $TRUE$ if it has been successful, $FALSE$ otherwise.

7.2 low order functions

The lower order function aim to provide programmers with more efficient routines. However they are more difficult to use and can lead to unexpected bugs.

7.2.1 addcst_mpint

Prototype

```
bool8 addcst_mpint(MPINT* v, HALF_WORD d, int32 index);
```

Description Computes $v + d \times 2^{index \times HW\_BITS}$ and stores the result in *v*.

Note This function does not extends the $MPINT$ if necessary. The result is truncated to the size of the original $MPINT$.

Return The function returns $TRUE$ if it has been successful, $DIRTY$ if the result has been truncated and $FALSE$ if an error occurred.

7.2.2 unsignadd_mpint

Prototype

```
bool8 unsignadd_mpint(MPINT* v1, MPINT* v2);
```

Description Computes $|v1| + |v2|$ and stores the result in *v1*.

Note The sign of *v1* is undefined afterward.

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

7.2.3 diff_mpint

Prototype

```
bool8 diff_mpint(MPINT* v1, MPINT* v2, int32 biggest);
```

Description Computes $|v1 - v2|$ and stores the result in *v1*. If the value of *biggest* is 1, it indicates that the absolute value of *v1* id grater than the one of *v2*, -1 is the opposite and 0 means that the user doesn't know.

Note The sign of *v1* is undefined afterward.

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

8 Product

8.1 mul_mpint

Prototype

```
bool8 mul_mpint(MPINT* dst, MPINT* v1, MPINT* v2, bool8 trunc);
```

Description Computes the operation $dst + v1 \times v2$ and stores the result in *dst*. If *trunc* is $TRUE$, the size of *dst* is extend to *v1* size and the result is then truncated.

Note

- If *dst* is not empty and its size bigger than *v1*, the result with *trunc* set to $TRUE$ will be undefined. (The carry won't be propagated further than *v1* size).
- If *dst* is not empty and $sizeof(dst + v1 \times v2) > sizeof(v1) + sizeof(v2)$, then the extra $HALF\_WORD$ will be used and the $DIRTY$ value will be returned. The user has to ensure that this state is temporary.

  Example: $9999 + 99 \times 99 = 19800$, the number requires 5 digits but has been prepared for only 4.

Return The function returns $TRUE$ if it has been successful, $DIRTY$ (cf. NOTE) and $FALSE$ otherwise.

8.2 square_mpint

Prototype

```
bool8 square_mpint(MPINT* dst, MPINT* v);
```

Description Computes v^2 and store the result in *dst*.

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

8.3 mulcst_mpint

Prototype

```
bool8 mulcst_mpint(MPINT* v1, HALF_WORD n);
```

Description Computes $v1 \times n$ in place.

Note *v1* is not extended, the $DIRTY$ value is returned if the extra $HALF\_WORD$ is used. The user has to ensure that this state is temporary.

Example: $99 \times 9 = 891$. 3 digits are used but the original size of *v1* was 2.

Return The function returns $TRUE$ if it has been successful, $DIRTY$ (cf. NOTE) and $FALSE$ if an error occurred.

9 Division

9.1 unsigndiv_mpint / unsignmod_mpint

Prototype

```
bool8 unsigndiv_mpint(MPINT* a, MPINT* b, MPINT* q, MPINT* r);
bool8 unsignmod_mpint(MPINT* r, MPINT* a, MPINT* b);
```

Description Computes $||\lfloor \frac{a}{b} \rfloor||$ in q and $a \mod b$ in r.

Note Theses functions are using two auxiliary functions called recursively.

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

9.2 invmod_mpint

Prototype

```
bool8 invmod_mpint(MPINT* x, MPINT* p);
```

Description Computes y such that $x \times y = 1 \mod p$ and put the result in x.

Note If such a value doesn't exist the function simply returns $FALSE$.

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

10 Exponentiation

Currently, this library only support modular exponentiation with the modulus being an ODD number. This is due to the fact that the implementation uses the Montgomery transformation: Wikipedia page.

10.1 MPPRIME

10.1.1 Data structure

Like the $MPINT$, each $MPPRIME$ is considered to be a 1024 bits prime or 2048 bits.... Therefore we define R to be the upper bound of the $MPINT$ with the particular size. For example, if a number is a 1024 bits integer, then the corresponding R is 2^{1024}.

```
typedef struct prime {
  MPINT num;
  MPINT inv;
  MPINT r_square;
} MPPRIME;
```

Code 2: MPPRIME structure

- num is the value of the prime number.
- inv is the number such that $num \times inv \mod R = -1$, with R defined as above.
- r_square is the number $R^2 \mod num$, with R defined as above.

NOTE: this data structure could be used with any odd integer.

10.1.2 init_mprime

Prototype

```
void init_mpprime(MPPRIME* mp);
```

Description Initialize a $MPPRIME$.

10.1.3 clear_mprime

Prototype

```
void clear_mpprime(MPPRIME* mp);
```

Description Releases the memory used by a $MPPRIME$, but not the memory used by the structure $MPPRIME$.

10.1.4 precompprime_mpint

Prototype

```
bool8 precompprime_mpint(MPPRIME* prime);
```

Description Computes y such that $x \times y = 1 \mod p$ and put the result in x.

Note If such a value doesn't exist the function simply returns $FALSE$.

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

10.2 Montgomery operations

We first define some routines that we describe here. A user should use the macro defined in the next section 10.2.3.

10.2.1 montgomery_mpint

Prototype

```
bool8 montgomery_mpint(MPINT *v1, MPINT *v2, MPINT* n, MPINT* inv_n);
bool8 montgomery_square_mpint(MPINT *v, MPINT* n, MPINT* inv_n);
```

Description Computes the Montgomery reduction of *v1* by *v2* modulo n and puts the result in *v1*.

Note The value *v1* and *v2* are the representation of some numbers in the Montgomery space (cf Examples 10.5).

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

10.2.2 montgomery_inv_mpint

Prototype

```
bool8 montgomery_inv_mpint(MPINT *v1, MPINT* n, MPINT* inv_n);
```

Description Computes the real value of the value *v1* from the Montgomery space modulo n and puts the result in *v1*.

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

10.2.3 Macros

```
#define montgomery_mul_mpint(v1, v2, p) \
montgomery_mpint(v1, v2, &((p)->num), &((p)->inv))

#define montgomery_square_mpint(v1, p) \
montgomery_square_mpint(v1, &((p)->num), &((p)->inv))

#define to_montgomery_mpint(v1, p) \
montgomery_mpint(v1, &((p)->r_square), &((p)->num), &((p)->inv))

#define from_montgomery_mpint(v1, p) \
montgomery_inv_mpint(v1, &((p)->num), &((p)->fginv))
```

10.3 powmod_mpint

Prototype

```
#define REAL        0
#define MONTGOMERY  1
bool8 powmod_mpint(MPINT *b, MPINT *n, MPPRIME *N, int32 output_space);
```

Description Computes the value $b^n \mod N$ and store it in b. If *output_space* is equal to $MONTGOMERY$, the result is stored in its Montgomery form. Otherwise it is in its real form.

Note Having the result in its Montomery form can be useful for computing $(a \times b^c) \mod N$. See Example 10.6

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

10.4 Examples

10.5 Exampel 1

Compute $34 \times 43 \mod 97$. (This is not efficient for only one operation, it is better to use *mul_mpint* and *unsignmod_mpint*).

```
MPPRIME p97;
MPINT i34, i43;

// Initialize
init_mpint(&i34);
init_mpint(&i43);
init_mpprime(&p97);

// Set value
word_mpint(&i34, 34, 1);
word_mpint(&i43, 43, 1);
word_mpint(&(p97.num), 97, 1);
precompprime_mpint(&p97);

// To Montgomery space
to_montgomery_mpint(&i34, &p97);
to_montgomery_mpint(&i43, &p97);

// Multiplication
montgomery_mul_mpint(&i34, &i43, &p97);

// Back to real value
from_montgomery_mpint(&i34, &p97);

// print result
print_mpint(&i34);

// clear memory
clear_mpint(&i34);
clear_mpint(&i43);
clear_mpprime(&p97);
```

10.6 Exampel 2

Compute $(34 \times 43^{55}) \mod 97$.

```
MPPRIME p97;
MPINT i34, i43, i55;

// Initialize
init_mpint(&i34);
init_mpint(&i43);
init_mpint(&i55);
init_mpprime(&p97);

// Set value
word_mpint(&i34, 34, 1);
word_mpint(&i43, 43, 1);
word_mpint(&i55, 55, 1);
word_mpint(&(p97.num), 97, 1);
precompprime_mpint(&p97);

```

```
// To Montgomery space
to_montgomery_mpint(&i34, &p97);

// Exponenciation
powmod_mpint(&i43, &i55, &p97, MONTGOMERY);

// print intermediate result
print_mpint(&i43);

// Multiplication
montgomery_mul_mpint(&i34, &i43, &p97);

// Back to real value
from_montgomery_mpint(&i34, &p97);

// print result
print_mpint(&i34);

// clear memory
clear_mpint(&i34);
clear_mpint(&i43);
clear_mpint(&i55);
clear_mpprime(&p97);
```

11 bits operations

11.1 rshift_mpint

Prototype

```
bool8 rshift_mpint(MPINT* x, int32 shift);
```

Description Shift *shift* bits on the right.

Note For negative number, the absolute value is shifted and the sign stays negative.

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

11.2 rshiftw_mpint

Prototype

```
bool8 rshiftw_mpint(MPINT* x, int32 shift);
```

Description Shift *shift* $HALF\_WORD$ on the right.

Note This function is not hardware independant.

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

11.3 lshift_mpint

Prototype

```
bool8 lshift_mpint(MPINT* x, int32 shift);
```

Description Shift *shift* bits on the left.

Note For negative number, the absolute value is shifted and the sign stays negative.

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

11.4 safelshift_mpint

Prototype

```
bool8 safelshift_mpint(MPINT* x, int32 shift);
```

Description Shift *shift* bits on the left **without** extending the $MPINT$.

Note The maximum value for *shift* is HW_BITS. It used for example in the *unsigndiv_mpint* function to ensure that the high order bits of the high order $HALF\_WORD$ is 1.

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

12 Random

The $MPINT$ library is using the pseudo random generator ISAAC.

A random context is used has parameter of each function, see the following example on how to create a context:

```
#include <random.h>

randctx ctx;
int i;
byte* wt;

// seed array of size RANDIZL * sizeof(uint32)
wt = (byte *)ctx.randrsl;

// set a seed
for (i = 0 ; i < RANDSIZL * sizeof(uint32) ; i++) {
  *wt++ = i & 0xFF;
}

// Init the context TRUE is set to use the seed provided.
// FALSE create a default context
randinit(&ctx, TRUE);
```

The function *init_context(randctx*)* can also be used. It seeds the context with a random value taken from a randompool inside the Xinu kernel[1].

12.1 rand_mpint

Prototype

```
bool8 rand_mpint(MPINT *x, int32 length, randctx* ctx);
```

[1] It is still experimental, the quality of the randomness may not be perfect

Description Create a random $MPINT$ with *length* $HALF\_WORD$ with a uniform probability law.

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

12.2 randmod_mpint

Prototype

```
bool8 randmod_mpint(MPINT* k, MPINT* q, randctx* ctx);
```

Description Create a random $MPINT$ in the range 0 to *q-1* with a uniform probability law.

Return The function returns $TRUE$ if it has been successful, $FALSE$ if an error occurred.

13 Performance evaluation