FIWE

1.0

Generated by Doxygen 1.8.18

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 Aff_Point_s Struct Reference

Structure to represent an affine point.

`#include <montgomery.h>`

Public Attributes

- ui x
- ui y

3.1.1 Detailed Description

Structure to represent an affine point.

3.1.2 Member Data Documentation

3.1.2.1 x

`ui Aff_Point_s::x`

x coordinate of the point

3.1.2.2 y

`ui Aff_Point_s::y`

y coordinate of the point

The documentation for this struct was generated from the following file:

- montgomery.h

3.2 Montg_Curve_s Struct Reference

Structure to represent a Montgomery curve.

`#include <montgomery.h>`

Public Attributes

- ui A
- ui B
- ui n

3.2.1 Detailed Description

Structure to represent a Montgomery curve.

Curve Equations:

- Affine Coordinates: B∗y^2 = x^3 + A∗x^2 + x
- Projective Coordinates: B∗y^2∗Z = x^3 + A∗x^2∗Z + x∗Z^2

3.2.2 Member Data Documentation

3.2.2.1 A

`ui Montg_Curve_s::A`

Coefficient A in the equation

3.2.2.2 B

`ui Montg_Curve_s::B`

Coefficient B in the equation

3.2.2.3 n

`ui Montg_Curve_s::n`

Modular base of the curve

The documentation for this struct was generated from the following file:

- montgomery.h

3.3 Pro_Point_s Struct Reference

Structure to represent a projective point.

`#include <montgomery.h>`

Public Attributes

- ui X
- ui Y
- ui Z

3.3.1 Detailed Description

Structure to represent a projective point.

3.3.2 Member Data Documentation

3.3.2.1 X

`ui Pro_Point_s::X`

X coordinate of the point

3.3.2.2 Y

`ui Pro_Point_s::Y`

Y coordinate of the point

3.3.2.3 Z

`ui Pro_Point_s::Z`

Z coordinate of the point

The documentation for this struct was generated from the following file:

- montgomery.h

Chapter 4

File Documentation

4.1 ecm.c File Reference

Implementation of ecm.h library.

```
#include <gmp.h>
#include <stdlib.h>
#include <time.h>
#include "montgomery.h"
#include "mplib.h"
#include "ecm.h"
```

Functions

- int ecm (ui d, ui n, ui_t nl)

  Factorizes the given composite using Elliptic Curve Method.

4.1.1 Detailed Description

Implementation of ecm.h library.

4.1.2 Function Documentation

4.1.2.1 ecm()

```
int ecm (
            ui d,
            ui n,
            ui_t nl )
```

Factorizes the given composite using Elliptic Curve Method.

Parameters

out	*d*	factor of n
in	*n*	number to be factorized
in	*nl*	number of digits of n in base 2^W

Returns

an integer, positive when ECM succeeds, 0 otw

4.2 ecm.h File Reference

A library to implement Elliptic Curve Method to factorize a composite number.

```
#include "mplib.h"
```

Macros

- #define B_THRESHOLD 1000

  Number of different B values to try during the factorization.
- #define CRV_THRESHOLD 20

  Number of different curves to try during the factorization.

Functions

- int ecm (ui d, ui n, ui_t nl)

  Factorizes the given composite using Elliptic Curve Method.

4.2.1 Detailed Description

A library to implement Elliptic Curve Method to factorize a composite number.

4.2.2 Macro Definition Documentation

4.2.2.1 B_THRESHOLD

```
#define B_THRESHOLD 1000
```

Number of different B values to try during the factorization.

If a B value does not produce a factor, another one is tried until the number of trials exceed B_THRESHOLD.

4.2.2.2 CRV_THRESHOLD

```
#define CRV_THRESHOLD 20
```

Number of different curves to try during the factorization.

If a curve does not produce a factor, another one is tried until the number of trials exceed CRV_THRESHOLD.

4.2.3 Function Documentation

4.2.3.1 ecm()

```
int ecm (
            ui d,
            ui n,
            ui_t nl )
```

Factorizes the given composite using Elliptic Curve Method.

Parameters

out	*d*	factor of n
in	*n*	number to be factorized
in	*nl*	number of digits of n in base 2^W

Returns

an integer, positive when ECM succeeds, 0 otw

4.3 main.c File Reference

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "test.h"
```

Functions

- int main ()

4.3.1 Function Documentation

4.3.1.1 main()

```
int main ( )
```

4.4 montgomery.c File Reference

Implementation of montgomery.h library.

```
#include <gmp.h>
#include <stdlib.h>
#include <time.h>
#include "montgomery.h"
#include "mplib.h"
```

Functions

- void pro_curve_point (ui d, MONTG_CURVE c, PRO_POINT p, ui n, ui_t nl, ui mu, ui_t mul, int *flag)

  Initializes a randomly generated Montgomery curve and a projective point on the curve.
- void aff_curve_point (ui d, MONTG_CURVE c, AFF_POINT p, ui n, ui_t nl, ui mu, ui_t mul, int *flag)

  Initializes a randomly generated Montgomery curve and a projective point on the curve.
- void pro_add (PRO_POINT p, PRO_POINT p1, PRO_POINT p2, PRO_POINT pd, ui n, ui_t nl, ui mu, ui_t mul)

  Adds two projective points on a curve.
- void aff_add (AFF_POINT p, AFF_POINT p1, AFF_POINT p2, ui A, ui B, ui n, ui_t nl, ui mu, ui_t mul)

  Adds two affine points on a curve.
- void pro_dbl (PRO_POINT p, PRO_POINT p1, ui A24, ui n, ui_t nl, ui mu, ui_t mul)

  Doubles a projective point on a curve.
- void aff_dbl (ui x, ui z, ui x1, ui y1, ui A, ui B, ui n)
- void pro_ladder (PRO_POINT p, PRO_POINT p1, ui A24, ui k, ui_t kl, ui n, ui_t nl, ui mu, ui_t mul)

  Multiplies a projective point with a constant.
- void aff_ladder (ui x, ui y, ui x1, ui y1, ui k, ui n)
- int pro_is_on_curve (ui A, ui B, ui X, ui Y, ui Z, ui n)
- int aff_is_on_curve (ui A, ui B, ui x, ui y, ui n)

4.4.1 Detailed Description

Implementation of montgomery.h library.

4.4.2 Function Documentation

4.4.2.1 aff_add()

```
void aff_add (
              AFF_POINT p,
              AFF_POINT p1,
              AFF_POINT p2,
              ui A,
              ui B,
              ui n,
              ui_t nl,
              ui mu,
              ui_t mul )
```

Adds two affine points on a curve.

Parameters

out	*p*	resulted projective point
in	*p1*	first operand of the addition
in	*p2*	second operand of the addition
in	*A*	coefficient A of the curve
in	*B*	coefficient B of the curve
in	*n*	modular base of the curve that is going to be generated
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.4.2.2 aff_curve_point()

```
void aff_curve_point (
              ui d,
              MONTG_CURVE c,
              AFF_POINT p,
              ui n,
              ui_t nl,
              ui mu,
              ui_t mul,
              int * flag )
```

Initializes a randomly generated Montgomery curve and a projective point on the curve.

Parameters

out	*d*	factor of n, that may be found while generating the curve
out	*c*	curve to be initialized
out	*p*	affine point to be initialized
in	*n*	modular base of the curve that is going to be generated
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W
in	*flag*	0 when factor found, 1 when curve and point generated, -1 when function failed due to singular curve generation

4.4.2.3 aff_dbl()

```
void aff_dbl (
             ui x,
             ui z,
             ui x1,
             ui y1,
             ui A,
             ui B,
             ui n )
```

4.4.2.4 aff_is_on_curve()

```
int aff_is_on_curve (
             ui A,
             ui B,
             ui x,
             ui y,
             ui n )
```

4.4.2.5 aff_ladder()

```
void aff_ladder (
             ui x,
             ui y,
             ui x1,
             ui y1,
             ui k,
             ui n )
```

4.4.2.6 pro_add()

```
void pro_add (
             PRO_POINT p,
             PRO_POINT p1,
             PRO_POINT p2,
             PRO_POINT pd,
             ui n,
             ui_t nl,
             ui mu,
             ui_t mul )
```

Adds two projective points on a curve.

Parameters

out	*p*	resulted projective point
in	*p1*	first operand of the addition
in	*p2*	second operand of the addition
in	*pd*	differences of the first and second operands
in	*n*	modular base of the curve that is going to be generated
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.4.2.7 pro_curve_point()

```
void pro_curve_point (
            ui d,
            MONTG_CURVE c,
            PRO_POINT p,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul,
            int * flag )
```

Initializes a randomly generated Montgomery curve and a projective point on the curve.

Parameters

out	*d*	factor of n, that may be found while generating the curve
out	*c*	curve to be initialized
out	*p*	projective point to be initialized
in	*n*	modular base of the curve that is going to be generated
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W
in	*flag*	0 when factor found, 1 when curve and point generated, -1 when function failed due to singular curve generation

4.4.2.8 pro_dbl()

```
void pro_dbl (
            PRO_POINT p,
            PRO_POINT p1,
            ui A24,
            ui n,
```

```
            ui_t nl,
            ui mu,
            ui_t mul )
```

Doubles a projective point on a curve.

Parameters

out	*p*	resulted projective point
in	*p1*	point to be doubled
in	*A24*	$(A+2)/4$ where A is the coefficient of the curve
in	*n*	modular base of the curve that is going to be generated
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.4.2.9 pro_is_on_curve()

```
int pro_is_on_curve (
            ui A,
            ui B,
            ui X,
            ui Y,
            ui Z,
            ui n )
```

4.4.2.10 pro_ladder()

```
void pro_ladder (
            PRO_POINT p,
            PRO_POINT p1,
            ui A24,
            ui k,
            ui_t kl,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul )
```

Multiplies a projective point with a constant.

Parameters

out	*p*	resulted projective point
in	*p1*	point to be multiplied
in	*A24*	$(A+2)/4$ where A is the coefficient of the curve
in	*k*	constant to multiply with p1

Parameters

in	*kl*	number of digits of k in base 2^W
in	*n*	modular base of the curve that is going to be generated
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.5 montgomery.h File Reference

A library to represent Montgomery curves and points on them, and to do arithmetic with that points.

```
#include "mplib.h"
```

Classes

- struct Montg_Curve_s

  Structure to represent a Montgomery curve.
- struct Pro_Point_s

  Structure to represent a projective point.
- struct Aff_Point_s

  Structure to represent an affine point.

Typedefs

- typedef struct Montg_Curve_s MONTG_CURVE_t[1]

  Structure to represent a Montgomery curve.
- typedef struct Montg_Curve_s * MONTG_CURVE[1]
- typedef struct Pro_Point_s PRO_POINT_t[1]

  Structure to represent a projective point.
- typedef struct Pro_Point_s * PRO_POINT[1]
- typedef struct Aff_Point_s AFF_POINT_t[1]

  Structure to represent an affine point.
- typedef struct Aff_Point_s * AFF_POINT[1]

Functions

- void pro_curve_point (ui d, MONTG_CURVE c, PRO_POINT p, ui n, ui_t nl, ui mu, ui_t mul, int *flag)

  Initializes a randomly generated Montgomery curve and a projective point on the curve.
- void aff_curve_point (ui d, MONTG_CURVE c, AFF_POINT p, ui n, ui_t nl, ui mu, ui_t mul, int *flag)

  Initializes a randomly generated Montgomery curve and a projective point on the curve.
- void pro_add (PRO_POINT p, PRO_POINT p1, PRO_POINT p2, PRO_POINT pd, ui n, ui_t nl, ui mu, ui_t mul)

  Adds two projective points on a curve.
- void aff_add (AFF_POINT p, AFF_POINT p1, AFF_POINT p2, ui A, ui B, ui n, ui_t nl, ui mu, ui_t mul)

Adds two affine points on a curve.

- void pro_dbl (PRO_POINT p, PRO_POINT p1, ui A24, ui n, ui_t nl, ui mu, ui_t mul)

  Doubles a projective point on a curve.
- void aff_dbl (ui x, ui z, ui x1, ui y1, ui A, ui B, ui n)
- void pro_ladder (PRO_POINT p, PRO_POINT p1, ui A24, ui k, ui_t kl, ui n, ui_t nl, ui mu, ui_t mul)

  Multiplies a projective point with a constant.
- void aff_ladder (ui x, ui y, ui x1, ui y1, ui k, ui n)
- int pro_is_on_curve (ui A, ui B, ui X, ui Y, ui Z, ui n)
- int aff_is_on_curve (ui A, ui B, ui x, ui y, ui n)

4.5.1 Detailed Description

A library to represent Montgomery curves and points on them, and to do arithmetic with that points.

4.5.2 Typedef Documentation

4.5.2.1 AFF_POINT

```
typedef struct Aff_Point_s * AFF_POINT[1]
```

4.5.2.2 AFF_POINT_t

```
typedef struct Aff_Point_s AFF_POINT_t[1]
```

Structure to represent an affine point.

4.5.2.3 MONTG_CURVE

```
typedef struct Montg_Curve_s * MONTG_CURVE[1]
```

4.5.2.4 MONTG_CURVE_t

```
typedef struct Montg_Curve_s MONTG_CURVE_t[1]
```

Structure to represent a Montgomery curve.

Curve Equations:

- Affine Coordinates: B∗y^2 = x^3 + A∗x^2 + x
- Projective Coordinates: B∗y^2∗Z = x^3 + A∗x^2∗Z + x∗Z^2

4.5.2.5 PRO_POINT

```
typedef struct Pro_Point_s * PRO_POINT[1]
```

4.5.2.6 PRO_POINT_t

```
typedef struct Pro_Point_s PRO_POINT_t[1]
```

Structure to represent a projective point.

4.5.3 Function Documentation

4.5.3.1 aff_add()

```
void aff_add (
            AFF_POINT p,
            AFF_POINT p1,
            AFF_POINT p2,
            ui A,
            ui B,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul )
```

Adds two affine points on a curve.

Parameters

out	*p*	resulted projective point
in	*p1*	first operand of the addition
in	*p2*	second operand of the addition
in	*A*	coefficient A of the curve
in	*B*	coefficient B of the curve
in	*n*	modular base of the curve that is going to be generated
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.5.3.2 aff_curve_point()

```
void aff_curve_point (
```

```
            ui d,
            MONTG_CURVE c,
            AFF_POINT p,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul,
            int * flag )
```

Initializes a randomly generated Montgomery curve and a projective point on the curve.

Parameters

out	*d*	factor of n, that may be found while generating the curve
out	*c*	curve to be initialized
out	*p*	affine point to be initialized
in	*n*	modular base of the curve that is going to be generated
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W
in	*flag*	0 when factor found, 1 when curve and point generated, -1 when function failed due to singular curve generation

4.5.3.3 aff_dbl()

```
void aff_dbl (
            ui x,
            ui z,
            ui x1,
            ui y1,
            ui A,
            ui B,
            ui n )
```

4.5.3.4 aff_is_on_curve()

```
int aff_is_on_curve (
            ui A,
            ui B,
            ui x,
            ui y,
            ui n )
```

4.5.3.5 aff_ladder()

```
void aff_ladder (
            ui x,
            ui y,
            ui x1,
            ui y1,
            ui k,
            ui n )
```

4.5.3.6 pro_add()

```
void pro_add (
            PRO_POINT p,
            PRO_POINT p1,
            PRO_POINT p2,
            PRO_POINT pd,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul )
```

Adds two projective points on a curve.

Parameters

out	*p*	resulted projective point
in	*p1*	first operand of the addition
in	*p2*	second operand of the addition
in	*pd*	differences of the first and second operands
in	*n*	modular base of the curve that is going to be generated
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.5.3.7 pro_curve_point()

```
void pro_curve_point (
            ui d,
            MONTG_CURVE c,
            PRO_POINT p,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul,
            int * flag )
```

Initializes a randomly generated Montgomery curve and a projective point on the curve.

Parameters

out	*d*	factor of n, that may be found while generating the curve
out	*c*	curve to be initialized
out	*p*	projective point to be initialized
in	*n*	modular base of the curve that is going to be generated
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W
in	*flag*	0 when factor found, 1 when curve and point generated, -1 when function failed due to singular curve generation

4.5.3.8 pro_dbl()

```
void pro_dbl (
            PRO_POINT p,
            PRO_POINT p1,
            ui A24,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul )
```

Doubles a projective point on a curve.

Parameters

out	*p*	resulted projective point
in	*p1*	point to be doubled
in	*A24*	$(A+2)/4$ where A is the coefficient of the curve
in	*n*	modular base of the curve that is going to be generated
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.5.3.9 pro_is_on_curve()

```
int pro_is_on_curve (
            ui A,
            ui B,
            ui X,
            ui Y,
            ui Z,
            ui n )
```

4.5.3.10 pro_ladder()

```
void pro_ladder (
            PRO_POINT p,
            PRO_POINT p1,
            ui A24,
            ui k,
            ui_t kl,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul )
```

Multiplies a projective point with a constant.

Parameters

out	*p*	resulted projective point
in	*p1*	point to be multiplied
in	*A24*	$(A+2)/4$ where A is the coefficient of the curve
in	*k*	constant to multiply with p1
in	*kl*	number of digits of k in base 2^W
in	*n*	modular base of the curve that is going to be generated
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.6 mplib.c File Reference

Implementation of mplib.h library.

```
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include "mplib.h"
```

Functions

- void big_rand (ui z, ui_t l)

  Initializes z with a random multi-precision number.
- void big_mod_rand (ui z, ui_t l, ui n, ui_t nl, ui mu, ui_t mul)

  Initializes z with a random multi-precision number mod n.
- void big_print (FILE *fp, ui a, ui_t al, char *s, char *R)

  Prints the given multi-precision number in Magma assignment format.
- void big_is_equal (int *z, ui a, ui b, ui_t l)

  Checks if two multi-precision numbers are equal.
- void big_is_equal_ui (int *z, ui a, ui_t al, ui_t b)

Checks if a multi-precision number is equal to given unsigned integer.

- void big_add (ui z, ui a, ui_t al, ui b, ui_t bl)

  Adds two multi-precision numbers.
- void big_mod_add (ui z, ui a, ui_t al, ui b, ui_t bl, ui n, ui_t nl, ui mu, ui_t mul)

  Adds two multi-precision numbers in mod n.
- void big_sub (ui z, int *d, ui a, ui_t al, ui b, ui_t bl)

  Subtracts two multi-precision numbers.
- void big_mod_sub (ui z, ui a, ui_t al, ui b, ui_t bl, ui n, ui_t nl)

  Subtracts two multi-precision numbers in mod n.
- void big_mul (ui z, ui a, ui_t al, ui b, ui_t bl)

  Multiplies two multi-precision numbers.
- void big_mod_mul (ui z, ui a, ui_t al, ui b, ui_t bl, ui n, ui_t nl, ui mu, ui_t mul)

  Multiplies two multi-precision numbers in mod n.
- void big_get_mu (ui mu, ui n, ui_t nl)

  Calculates $(2^W)^{2*nl}/n$.
- void big_get_A24 (ui z, ui A, ui n, ui_t nl, ui mu, ui_t mul, int *flag)

  Calculates $(A+2)/4$.
- uni_t barret_reduction_UL (uni_t p, uni_t b, uni_t k, uni_t z, uni_t m, uni_t L)
- void barret_reduction (ui z, ui m, ui_t ml, ui n, ui_t nl, ui mu, ui_t mul)

  Calculates m mod n.
- void big_gcd (ui d, ui_t dl, ui a, ui_t al, ui b, ui_t bl)
- int big_invert (ui z, ui a, ui_t al, ui b, ui_t bl)

4.6.1 Detailed Description

Implementation of mplib.h library.

4.6.2 Function Documentation

4.6.2.1 barret_reduction()

```
void barret_reduction (
            ui z,
            ui m,
            ui_t ml,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul )
```

Calculates m mod n.

Parameters

out	*z*	result of the reduction
in	*m*	number to be reduced
in	*ml*	number of digits of m in base 2^W
in	*n*	modular base for the reduction
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.6.2.2 barret_reduction_UL()

```
uni_t barret_reduction_UL (
            uni_t p,
            uni_t b,
            uni_t k,
            uni_t z,
            uni_t m,
            uni_t L )
```

4.6.2.3 big_add()

```
void big_add (
            ui z,
            ui a,
            ui_t al,
            ui b,
            ui_t bl )
```

Adds two multi-precision numbers.

Parameters

out	*z*	result of the addition
in	*a*	first number
in	*al*	number of digits of a in base 2^W
in	*b*	second number
in	*bl*	number of digits of b in base 2^W

4.6.2.4 big_gcd()

```
void big_gcd (
            ui d,
            ui_t dl,
            ui a,
            ui_t al,
            ui b,
            ui_t bl )
```

4.6.2.5 big_get_A24()

```
void big_get_A24 (
            ui A24,
            ui A,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul,
            int * flag )
```

Calculates $(A+2)/4$.

Parameters

out	*A24*	result of $(A+2)/4$ or a factor of n
in	*A*	A in the equation
in	*n*	n modular base for the calculation
in	*number*	of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W
in	*flag*	1 when calculation succeeds, 0 when factor gets found

4.6.2.6 big_get_mu()

```
void big_get_mu (
            ui z,
            ui n,
            ui_t nl )
```

Calculates $(2^W)^{2*nl}/n$.

Parameters

out	*z*	result of the calculation
in	*n*	n in the equation
in	*nl*	number of digits of n in base 2^W

4.6.2.7 big_invert()

```
int big_invert (
            ui z,
            ui a,
            ui_t al,
```

```
            ui b,
            ui_t bl )
```

4.6.2.8 big_is_equal()

```
void big_is_equal (
            int * z,
            ui a,
            ui b,
            ui_t l )
```

Checks if two multi-precision numbers are equal.

Parameters

out	z	1 if equal, 0 otw
in	a	first number
in	b	second number
in	l	number of digits of a and b in base 2^W

4.6.2.9 big_is_equal_ui()

```
void big_is_equal_ui (
            int * z,
            ui a,
            ui_t al,
            ui_t b )
```

Checks if a multi-precision number is equal to given unsigned integer.

Parameters

out	z	1 if equal, 0 otw
in	a	first number
in	al	number of digits of a in base 2^W
in	b	unsigned int to be compared

4.6.2.10 big_mod_add()

```
void big_mod_add (
            ui z,
            ui a,
```

```
            ui_t al,
            ui b,
            ui_t bl,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul )
```

Adds two multi-precision numbers in mod n.

Parameters

out	*z*	result of the addition
in	*a*	first number
in	*al*	number of digits of a in base 2^W
in	*b*	second number
in	*bl*	number of digits of b in base 2^W
in	*n*	modular base for the addition
in	*nl*	number of digits of n in base 2^W

4.6.2.11 big_mod_mul()

```
void big_mod_mul (
            ui z,
            ui a,
            ui_t al,
            ui b,
            ui_t bl,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul )
```

Multiplies two multi-precision numbers in mod n.

Parameters

out	*z*	result of the multiplication
in	*a*	first number
in	*al*	number of digits of a in base 2^W
in	*b*	second number
in	*bl*	number of digits of b in base 2^W
in	*n*	modular base for the multiplication
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.6.2.12 big_mod_rand()

```
void big_mod_rand (
            ui z,
            ui_t l,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul )
```

Initializes z with a random multi-precision number mod n.

Parameters

out	*z*	multi-precision number to be initialized
in	*l*	number of digits of z in base 2^W
in	*n*	modular base for z
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.6.2.13 big_mod_sub()

```
void big_mod_sub (
            ui z,
            ui a,
            ui_t al,
            ui b,
            ui_t bl,
            ui n,
            ui_t nl )
```

Subtracts two multi-precision numbers in mod n.

Parameters

out	*z*	result of the subtraction
in	*a*	first number
in	*al*	number of digits of a in base 2^W
in	*b*	second number
in	*bl*	number of digits of b in base 2^W
in	*n*	modular base for the subtraction
in	*nl*	number of digits of n in base 2^W

4.6.2.14 big_mul()

```
void big_mul (
            ui z,
            ui a,
            ui_t al,
            ui b,
            ui_t bl )
```

Multiplies two multi-precision numbers.

Parameters

out	*z*	result of the multiplication
in	*a*	first number
in	*al*	number of digits of a in base 2^W
in	*b*	second number
in	*bl*	number of digits of b in base 2^W

4.6.2.15 big_print()

```
void big_print (
            FILE * fp,
            ui a,
            ui_t al,
            char * s,
            char * R )
```

Prints the given multi-precision number in Magma assignment format.

Parameters

in	*fp*	pointer to the file to print
in	*a*	multi-precision number to be printed
in	*al*	number of digits of a in base 2^W
in	*s*	name of the variable going to be assigned to a
in	*R*	name of the ring that a is going to be defined in (optional)

4.6.2.16 big_rand()

```
void big_rand (
            ui z,
            ui_t l )
```

Initializes z with a random multi-precision number.

Parameters

out	*z*	multi-precision number to be initialized
in	*l*	number of digits of z in base 2^W

4.6.2.17 big_sub()

```
void big_sub (
            ui z,
            int * d,
            ui a,
            ui_t al,
            ui b,
            ui_t bl )
```

Subtracts two multi-precision numbers.

Parameters

out	*z*	result of the subtraction
in	*a*	first number
in	*al*	number of digits of a in base 2^W
in	*b*	second number
in	*bl*	number of digits of b in base 2^W

4.7 mplib.h File Reference

A library to represent multi-precision integers and do arithmetic on them.

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

Macros

- #define W 32

  Size of a digit of a multi-precision integer.

- #define big_cpy(z, a, start, end)

  Copies $end - start$ elements from a to z.

Typedefs

- typedef unsigned long ∗ uni

  Type definition for unsigned long pointer.
- typedef unsigned long uni_t

  Type definition for unsigned long.
- typedef unsigned int ∗ ui

  Type definition for unsigned integer pointer.
- typedef unsigned int ui_t

  Type definition for unsigned integer.

Functions

- void big_rand (ui z, ui_t l)

  Initializes z with a random multi-precision number.
- void big_mod_rand (ui z, ui_t l, ui n, ui_t nl, ui mu, ui_t mul)

  Initializes z with a random multi-precision number mod n.
- void big_print (FILE ∗fp, ui a, ui_t al, char ∗s, char ∗R)

  Prints the given multi-precision number in Magma assignment format.
- void big_is_equal (int ∗z, ui a, ui b, ui_t l)

  Checks if two multi-precision numbers are equal.
- void big_is_equal_ui (int ∗z, ui a, ui_t al, ui_t b)

  Checks if a multi-precision number is equal to given unsigned integer.
- void big_add (ui z, ui a, ui_t al, ui b, ui_t bl)

  Adds two multi-precision numbers.
- void big_mod_add (ui z, ui a, ui_t al, ui b, ui_t bl, ui n, ui_t nl, ui mu, ui_t mul)

  Adds two multi-precision numbers in mod n.
- void big_sub (ui z, int ∗d, ui a, ui_t al, ui b, ui_t bl)

  Subtracts two multi-precision numbers.
- void big_mod_sub (ui z, ui a, ui_t al, ui b, ui_t bl, ui n, ui_t nl)

  Subtracts two multi-precision numbers in mod n.
- void big_mul (ui z, ui a, ui_t al, ui b, ui_t bl)

  Multiplies two multi-precision numbers.
- void big_mod_mul (ui z, ui a, ui_t al, ui b, ui_t bl, ui n, ui_t nl, ui mu, ui_t mul)

  Multiplies two multi-precision numbers in mod n.
- void big_get_mu (ui z, ui n, ui_t nl)

  Calculates $(2^W)^{2*nl}/n$.
- void big_get_A24 (ui A24, ui A, ui n, ui_t nl, ui mu, ui_t mul, int ∗flag)

  Calculates $(A+2)/4$.
- uni_t barret_reduction_UL (uni_t p, uni_t b, uni_t k, uni_t z, uni_t m, uni_t L)
- void barret_reduction (ui z, ui m, ui_t ml, ui n, ui_t nl, ui mu, ui_t mul)

  Calculates m mod n.
- void big_gcd (ui d, ui_t dl, ui a, ui_t al, ui b, ui_t bl)
- int big_invert (ui z, ui a, ui_t al, ui b, ui_t bl)

4.7.1 Detailed Description

A library to represent multi-precision integers and do arithmetic on them.

4.7.2 Macro Definition Documentation

4.7.2.1 big_cpy

```
#define big_cpy(
                z,
                a,
                start,
                end )
```

Value:

```
    if(1) { \
    int i, j; \
    for(i = 0, j = (start); i < (end); i++, j++) { \
        z[i] = a[j]; \
    } \
};
```

Copies $end - start$ elements from a to z.

Parameters

out	*z*	destination of the copy operation
in	*a*	source of the copy operation
in	*start*	starting index for the copy operation
in	*end*	ending index for the copy operation

4.7.2.2 W

```
#define W 32
```

Size of a digit of a multi-precision integer.

The numbers are represented in base 2^W such that a digit of the number cannot exceed 2^W.

4.7.3 Typedef Documentation

4.7.3.1 ui

```
typedef unsigned int* ui
```

Type definition for unsigned integer pointer.

4.7.3.2 ui_t

```
typedef unsigned int ui_t
```

Type definition for unsigned integer.

4.7.3.3 uni

```
typedef unsigned long* uni
```

Type definition for unsigned long pointer.

4.7.3.4 uni_t

```
typedef unsigned long uni_t
```

Type definition for unsigned long.

4.7.4 Function Documentation

4.7.4.1 barret_reduction()

```
void barret_reduction (
            ui z,
            ui m,
            ui_t ml,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul )
```

Calculates m mod n.

Parameters

out	*z*	result of the reduction
in	*m*	number to be reduced
in	*ml*	number of digits of m in base 2^W
in	*n*	modular base for the reduction
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.7.4.2 barret_reduction_UL()

```
uni_t barret_reduction_UL (
            uni_t p,
            uni_t b,
            uni_t k,
            uni_t z,
            uni_t m,
            uni_t L )
```

4.7.4.3 big_add()

```
void big_add (
            ui z,
            ui a,
            ui_t al,
            ui b,
            ui_t bl )
```

Adds two multi-precision numbers.

Parameters

out	*z*	result of the addition
in	*a*	first number
in	*al*	number of digits of a in base 2^W
in	*b*	second number
in	*bl*	number of digits of b in base 2^W

4.7.4.4 big_gcd()

```
void big_gcd (
            ui d,
            ui_t dl,
            ui a,
            ui_t al,
            ui b,
            ui_t bl )
```

4.7.4.5 big_get_A24()

```
void big_get_A24 (
            ui A24,
            ui A,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul,
            int * flag )
```

Calculates $(A+2)/4$.

Parameters

out	*A24*	result of $(A+2)/4$ or a factor of n
in	*A*	A in the equation
in	*n*	n modular base for the calculation
in	*number*	of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W
in	*flag*	1 when calculation succeeds, 0 when factor gets found

4.7.4.6 big_get_mu()

```
void big_get_mu (
            ui z,
            ui n,
            ui_t nl )
```

Calculates $(2^W)^{2*nl}/n$.

Parameters

out	*z*	result of the calculation
in	*n*	n in the equation
in	*nl*	number of digits of n in base 2^W

4.7.4.7 big_invert()

```
int big_invert (
            ui z,
            ui a,
            ui_t al,
```

```
            ui b,
            ui_t bl )
```

4.7.4.8 big_is_equal()

```
void big_is_equal (
            int * z,
            ui a,
            ui b,
            ui_t l )
```

Checks if two multi-precision numbers are equal.

Parameters

out	z	1 if equal, 0 otw
in	a	first number
in	b	second number
in	l	number of digits of a and b in base 2^W

4.7.4.9 big_is_equal_ui()

```
void big_is_equal_ui (
            int * z,
            ui a,
            ui_t al,
            ui_t b )
```

Checks if a multi-precision number is equal to given unsigned integer.

Parameters

out	z	1 if equal, 0 otw
in	a	first number
in	al	number of digits of a in base 2^W
in	b	unsigned int to be compared

4.7.4.10 big_mod_add()

```
void big_mod_add (
            ui z,
            ui a,
```

```
            ui_t al,
            ui b,
            ui_t bl,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul )
```

Adds two multi-precision numbers in mod n.

Parameters

out	*z*	result of the addition
in	*a*	first number
in	*al*	number of digits of a in base 2^W
in	*b*	second number
in	*bl*	number of digits of b in base 2^W
in	*n*	modular base for the addition
in	*nl*	number of digits of n in base 2^W

4.7.4.11 big_mod_mul()

```
void big_mod_mul (
            ui z,
            ui a,
            ui_t al,
            ui b,
            ui_t bl,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul )
```

Multiplies two multi-precision numbers in mod n.

Parameters

out	*z*	result of the multiplication
in	*a*	first number
in	*al*	number of digits of a in base 2^W
in	*b*	second number
in	*bl*	number of digits of b in base 2^W
in	*n*	modular base for the multiplication
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.7.4.12 big_mod_rand()

```
void big_mod_rand (
            ui z,
            ui_t l,
            ui n,
            ui_t nl,
            ui mu,
            ui_t mul )
```

Initializes z with a random multi-precision number mod n.

Parameters

out	*z*	multi-precision number to be initialized
in	*l*	number of digits of z in base 2^W
in	*n*	modular base for z
in	*nl*	number of digits of n in base 2^W
in	*mu*	precalculated value of $(2^W)^{2*nl}/n$
in	*mul*	number of digits of mu in base 2^W

4.7.4.13 big_mod_sub()

```
void big_mod_sub (
            ui z,
            ui a,
            ui_t al,
            ui b,
            ui_t bl,
            ui n,
            ui_t nl )
```

Subtracts two multi-precision numbers in mod n.

Parameters

out	*z*	result of the subtraction
in	*a*	first number
in	*al*	number of digits of a in base 2^W
in	*b*	second number
in	*bl*	number of digits of b in base 2^W
in	*n*	modular base for the subtraction
in	*nl*	number of digits of n in base 2^W

4.7.4.14 big_mul()

```
void big_mul (
            ui z,
            ui a,
            ui_t al,
            ui b,
            ui_t bl )
```

Multiplies two multi-precision numbers.

Parameters

out	*z*	result of the multiplication
in	*a*	first number
in	*al*	number of digits of a in base 2^W
in	*b*	second number
in	*bl*	number of digits of b in base 2^W

4.7.4.15 big_print()

```
void big_print (
            FILE * fp,
            ui a,
            ui_t al,
            char * s,
            char * R )
```

Prints the given multi-precision number in Magma assignment format.

Parameters

in	*fp*	pointer to the file to print
in	*a*	multi-precision number to be printed
in	*al*	number of digits of a in base 2^W
in	*s*	name of the variable going to be assigned to a
in	*R*	name of the ring that a is going to be defined in (optional)

4.7.4.16 big_rand()

```
void big_rand (
            ui z,
            ui_t l )
```

Initializes z with a random multi-precision number.

Parameters

out	*z*	multi-precision number to be initialized
in	*l*	number of digits of z in base 2^W

4.7.4.17 big_sub()

```
void big_sub (
            ui z,
            int * d,
            ui a,
            ui_t al,
            ui b,
            ui_t bl )
```

Subtracts two multi-precision numbers.

Parameters

out	*z*	result of the subtraction
in	*a*	first number
in	*al*	number of digits of a in base 2^W
in	*b*	second number
in	*bl*	number of digits of b in base 2^W

4.8 test.c File Reference

Implementation of test.h library.

```
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include "mplib.h"
#include "montgomery.h"
#include "ecm.h"
#include "test.h"
```

Functions

- void pro_curve_point_gmp_test ()

  Tests pro_curve_point function using GMP.
- void aff_curve_point_gmp_test ()

  Tests aff_curve_point function using GMP.
- void pro_add_gmp_test ()

Tests pro_add function using GMP.

- void pro_add_magma_test ()

  Tests pro_add function using Magma.

- void pro_dbl_magma_test ()

  Tests pro_dbl function using Magma.

- void pro_ladder_gmp_test ()
- void pro_ladder_magma_test ()
- void ecm_test ()

  Tests ecm function.

4.8.1 Detailed Description

Implementation of test.h library.

4.8.2 Function Documentation

4.8.2.1 aff_curve_point_gmp_test()

```
void aff_curve_point_gmp_test ( )
```

Tests aff_curve_point function using GMP.

1. Generates a random curve and a affine point using the function
2. Checks if the coefficients of the curve and the point satisfies the curve equation using GMP

4.8.2.2 ecm_test()

```
void ecm_test ( )
```

Tests ecm function.

1. Generates a random composite number
2. Calculates a factor of the number using the function
3. Checks whether the factor found actually divides the composite number

4.8.2.3 pro_add_gmp_test()

```
void pro_add_gmp_test ( )
```

Tests pro_add function using GMP.

1. Generates two random points
2. Computes the addition by implementing the algebraic calculations using GMP
3. Compares the result of the function with the GMP result

4.8.2.4 pro_add_magma_test()

```
void pro_add_magma_test ( )
```

Tests pro_add function using Magma.

1. Generates two random points
2. Computes the addition by implementing the algebraic calculations using Magma
3. Compares the result of the function with the Magma result

4.8.2.5 pro_curve_point_gmp_test()

```
void pro_curve_point_gmp_test ( )
```

Tests pro_curve_point function using GMP.

1. Generates a random curve and a projective point using the function
2. Checks if the coefficients of the curve and the point satisfies the curve equation using GMP

4.8.2.6 pro_dbl_magma_test()

```
void pro_dbl_magma_test ( )
```

Tests pro_dbl function using Magma.

1. Generates a random point
2. Computes the double by implementing the algebraic calculations using Magma
3. Compares the result of the function with the Magma result

4.8.2.7 pro_ladder_gmp_test()

```
void pro_ladder_gmp_test ( )
```

4.8.2.8 pro_ladder_magma_test()

```
void pro_ladder_magma_test ( )
```

4.9 test.h File Reference

A library to test ecm.

Functions

- void pro_curve_point_gmp_test ()

  Tests pro_curve_point function using GMP.
- void aff_curve_point_gmp_test ()

  Tests aff_curve_point function using GMP.
- void pro_add_gmp_test ()

  Tests pro_add function using GMP.
- void pro_add_magma_test ()

  Tests pro_add function using Magma.
- void pro_dbl_magma_test ()

  Tests pro_dbl function using Magma.
- void pro_ladder_gmp_test ()
- void pro_ladder_magma_test ()
- void ecm_test ()

  Tests ecm function.

4.9.1 Detailed Description

A library to test ecm.

4.9.2 Function Documentation

4.9.2.1 aff_curve_point_gmp_test()

```
void aff_curve_point_gmp_test ( )
```

Tests aff_curve_point function using GMP.

1. Generates a random curve and a affine point using the function
2. Checks if the coefficients of the curve and the point satisfies the curve equation using GMP

4.9.2.2 ecm_test()

```
void ecm_test ( )
```

Tests ecm function.

1. Generates a random composite number
2. Calculates a factor of the number using the function
3. Checks whether the factor found actually divides the composite number

4.9.2.3 pro_add_gmp_test()

```
void pro_add_gmp_test ( )
```

Tests pro_add function using GMP.

1. Generates two random points
2. Computes the addition by implementing the algebraic calculations using GMP
3. Compares the result of the function with the GMP result

4.9.2.4 pro_add_magma_test()

```
void pro_add_magma_test ( )
```

Tests pro_add function using Magma.

1. Generates two random points
2. Computes the addition by implementing the algebraic calculations using Magma
3. Compares the result of the function with the Magma result

4.9.2.5 pro_curve_point_gmp_test()

```
void pro_curve_point_gmp_test ( )
```

Tests pro_curve_point function using GMP.

1. Generates a random curve and a projective point using the function
2. Checks if the coefficients of the curve and the point satisfies the curve equation using GMP

4.9.2.6 pro_dbl_magma_test()

```
void pro_dbl_magma_test ( )
```

Tests pro_dbl function using Magma.

1. Generates a random point
2. Computes the double by implementing the algebraic calculations using Magma
3. Compares the result of the function with the Magma result

4.9.2.7 pro_ladder_gmp_test()

```
void pro_ladder_gmp_test ( )
```

4.9.2.8 pro_ladder_magma_test()

```
void pro_ladder_magma_test ( )
```

Index