/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

File name: SM3.c

Version: SM3\_V1.1

Date: Sep 18,2016

Description: to calculate a hash message from a given message Function List:

1.SM3\_256 //calls SM3\_init, SM3\_process and SM3\_done to calculate hash value

2.SM3\_init //init the SM3 state

3.SM3\_process //compress the the first len/64 blocks of the message

4.SM3\_done //compress the rest message and output the hash value

5.SM3\_compress //called by SM3\_process and SM3\_done, compress a single block of message

6.BiToW //called by SM3\_compress,to calculate W from Bi

7.WToW1 //called by SM3\_compress, calculate W1 from W

8.CF //called by SM3\_compress, to calculate CF function.

9.BigEndian //called by SM3\_compress and SM3\_done.GM/T 0004-2012 requires to use

big-endian.

//if CPU uses little-endian, BigEndian function is a necessary call to

change the

//little-endian format into big-endian format.

10.SM3\_SelfTest //test whether the SM3 calculation is correct by comparing the hash result

with the standard result History:

1. Date: Sep 18,2016

Author: Mao Yingying, Huo Lili

Modification: 1)add notes to all the functions 2)add SM3\_SelfTest function

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "SM3.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function: BiToW

Description: calculate W from Bi

Calls:

Called By: SM3\_compress

Input: Bi[16] //a block of a message

Output: W[64]

Return: null

Others:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ void BiToW(unsigned int Bi[], unsigned int W[])

{

int i;

unsigned int tmp;

for (i = 0; i <= 15; i++) {

W[i] = Bi[i];

}

for (i = 16; i <= 67; i++) {

tmp = W[i - 16]

^ W[i - 9]

^ SM3\_rotl32(W[i - 3], 15); W[i] = SM3\_p1(tmp)

^ (SM3\_rotl32(W[i - 13], 7)) ^ W[i - 6];

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function: WToW1

Description: calculate W1 from W

Calls:

Called By: SM3\_compress

Input: W[64]

Output: W1[64]

Return: null

Others:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ void WToW1(unsigned int W[], unsigned int W1[])

{

int i;

for (i = 0; i <= 63; i++) {

W1[i] = W[i] ^ W[i + 4];

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function: CF

Description: calculate the CF compress function and update V

Calls:

Called By: SM3\_compress

Input: W[64]

W1[64] V[8]

Output: V[8]

Return: null

Others:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ void CF(unsigned int W[], unsigned int W1[], unsigned int V[])

{

unsigned int SS1;

unsigned int SS2;

unsigned int TT1;

unsigned int TT2;

unsigned int A, B, C, D, E, F, G, H;

unsigned int T = SM3\_T1;

unsigned int FF;

unsigned int GG;

int j;

//reg init,set ABCDEFGH=V0

A=V[0];B = V[1]; C = V[2]; D = V[3]; E = V[4]; F = V[5]; G = V[6]; H = V[7];

for (j = 0; j <= 63; j++) {

//SS1

if (j == 0) {

T = SM3\_T1;

}

else if (j == 16) {

T = SM3\_rotl32(SM3\_T2, 16);

}

else {

T = SM3\_rotl32(T, 1);

}

SS1 = SM3\_rotl32((SM3\_rotl32(A, 12) + E + T), 7);

//SS2

SS2 = SS1 ^ SM3\_rotl32(A, 12);

//TT1

if (j <= 15) {

FF = SM3\_ff0(A, B, C);

}

else {

FF = SM3\_ff1(A, B, C);

}

TT1 = FF + D + SS2 + \*W1; W1++;

//TT2

if (j <= 15) {

GG = SM3\_gg0(E, F, G);

}

else {

GG = SM3\_gg1(E, F, G);

}

TT2 = GG + H + SS1 + \*W; W++;

//D D=C;

//C

C = SM3\_rotl32(B, 9);

//B B=A;

//A

A = TT1;

//H H=G;

//G

G = SM3\_rotl32(F, 19);

//F F=E;

//E

E = SM3\_p0(TT2);

}

//update V

V[0] = A ^ V[0]; V[1] = B ^ V[1]; V[2] = C ^ V[2]; V[3] = D ^ V[3]; V[4] = E ^ V[4]; V[5] = F ^ V[5]; V[6] = G ^ V[6]; V[7] = H ^ V[7];

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function: BigEndian

Description: U32 endian converse.GM/T 0004-2012 requires to use big-endian.

if CPU uses little-endian, BigEndian function is a necessary

call to change the little-endian format into big-endian format.

Calls:

Called By: SM3\_compress, SM3\_done

Input: src[bytelen]

bytelen

Output: des[bytelen]

Return: null

Others: src and des could implies the same address

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ void BigEndian(unsigned char src[], unsigned int bytelen, unsigned char des[])

{

unsigned char tmp = 0; unsigned int i = 0;

for (i = 0; i < bytelen / 4; i++) {

tmp = des[4 \* i];

des[4 \* i] = src[4 \* i + 3]; src[4 \* i + 3] = tmp;

tmp = des[4 \* i + 1];

des[4 \* i + 1] = src[4 \* i + 2]; des[4 \* i + 2] = tmp;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function: SM3\_init

Description: initiate SM3 state

Calls:

Called By: SM3\_256

Input: SM3\_STATE \*md

Output: SM3\_STATE \*md

Return: null

Others:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ void SM3\_init(SM3\_STATE\* md)

{

md->curlen = md->length = 0; md->state[0] = SM3\_IVA;

md->state[1] = SM3\_IVB; md->state[2] = SM3\_IVC; md->state[3] = SM3\_IVD; md->state[4] = SM3\_IVE; md->state[5] = SM3\_IVF; md->state[6] = SM3\_IVG; md->state[7] = SM3\_IVH;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function: SM3\_compress

Description: compress a single block of message

Calls: BigEndian

BiToW WToW1 CF

Called By: SM3\_256

Input: SM3\_STATE \*md

Output: SM3\_STATE \*md

Return: null

Others:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ void SM3\_compress(SM3\_STATE\* md)

{

unsigned int W[68]; unsigned int W1[64];

//if CPU uses little-endian, BigEndian function is a necessary call BigEndian(md->buf, 64, md->buf);

BiToW((unsigned int\*)md->buf, W); WToW1(W, W1);

CF(W, W1, md->state);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function: SM3\_process

Description: compress the first (len/64) blocks of message

Calls: SM3\_compress

Called By: SM3\_256

Input: SM3\_STATE \*md

unsigned char buf[len] int len

//the input message //bytelen of message

Output: SM3\_STATE \*md

Return: null

Others:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ void SM3\_process(SM3\_STATE\* md, unsigned char\* buf, int len)

{

while (len--) {

/\* copy byte \*/

md->buf[md->curlen] = \*buf++; md->curlen++;

/\* is 64 bytes full? \*/ if (md->curlen == 64)

{

SM3\_compress(md); md->length += 512; md->curlen = 0;

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function: SM3\_done

Description: compress the rest message that the SM3\_process has left behind

Calls: SM3\_compress

Called By: SM3\_256

Input: SM3\_STATE \*md

Output: unsigned char \*hash

Return: null

Others:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ void SM3\_done(SM3\_STATE\* md, unsigned char hash[])

{

int i;

unsigned char tmp = 0;

/\* increase the bit length of the message \*/ md->length += md->curlen << 3;

/\* append the '1' bit \*/

md->buf[md->curlen] = 0x80; md->curlen++;

/\* if the length is currently above 56 bytes, appends zeros till it reaches 64 bytes, compress the current block, creat a new block by appending zeros and length,and then compress it

\*/

if (md->curlen > 56) {

for (; md->curlen < 64;) {

md->buf[md->curlen] = 0; md->curlen++;

}

SM3\_compress(md); md->curlen = 0;

}

/\* if the length is less than 56 bytes, pad upto 56 bytes of zeroes \*/ for (; md->curlen < 56;)

{

md->buf[md->curlen] = 0; md->curlen++;

}

/\* since all messages are under 2^32 bits we mark the top bits zero \*/ for (i = 56; i < 60; i++)

{

md->buf[i] = 0;

}

/\* append length \*/

md->buf[63] = md->length & 0xff;

md->buf[62] = (md->length >> 8) & 0xff; md->buf[61] = (md->length >> 16) & 0xff; md->buf[60] = (md->length >> 24) & 0xff;

SM3\_compress(md);

/\* copy output \*/

memcpy(hash, md->state, SM3\_len / 8);

BigEndian(hash, SM3\_len / 8, hash);//if CPU uses little-endian, BigEndian function is a necessary call

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function: SM3\_256

Description: calculate a hash value from a given message

Calls: SM3\_init

SM3\_process SM3\_done

Called By:

Input: unsigned char buf[len] //the input message

int len //bytelen of the message

Output: unsigned char hash[32]

Return: null

Others:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ void SM3\_256(unsigned char buf[], int len, unsigned char hash[])

{

SM3\_STATE md; SM3\_init(&md);

SM3\_process(&md, buf, len);

SM3\_done(&md, hash);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Function: SM3\_SelfTest

Description: test whether the SM3 calculation is correct by comparing

the hash result with the standard result

Calls: SM3\_256

Called By:

Input: null

Output: null

Return: 0

1 //the SM3 operation is correct //the sm3 operation is wrong

Others:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ int SM3\_SelfTest()

{

unsigned int i = 0, a = 1, b = 1;

unsigned char Msg1[3] = { 0x61,0x62,0x63 }; int MsgLen1 = 3;

unsigned char MsgHash1[32] = { 0 }; unsigned char

StdHash1[32] = { 0x66,0xC7,0xF0,0xF4,0x62,0xEE,0xED,0xD9,0xD1,0xF2,0xD4,0x6B,0xDC,0x10,0xE4,0xE2,

0x41,0x67,0xC4,0x87,0x5C,0xF2,0xF7,0xA2,0x29,0x7D,0xA0,0x2B,0x8F,0x4B,0xA8,0xE0 };

unsigned char

Msg2[64] = { 0x61,0x62,0x63,0x64,0x61,0x62,0x63,0x64,0x61,0x62,0x63,0x64,0x61,0x62,0x63,0x64,

0x61,0x62,0x63,0x64,0x61,0x62,0x63,0x64,0x61,0x62,0x63,0x64,0x61,0x62,0x63,0x64,

0x61,0x62,0x63,0x64,0x61,0x62,0x63,0x64,0x61,0x62,0x63,0x64,0x61,0x62,0x63,0x64,

0x61,0x62,0x63,0x64,0x61,0x62,0x63,0x64,0x61,0x62,0x63,0x64,0x61,0x62,0x63,0x64 }; int MsgLen2 = 64;

unsigned char MsgHash2[32] = { 0 }; unsigned char

StdHash2[32] = { 0xde,0xbe,0x9f,0xf9,0x22,0x75,0xb8,0xa1,0x38,0x60,0x48,0x89,0xc1,0x8e,0x5a,0x4d,

0x6f,0xdb,0x70,0xe5,0x38,0x7e,0x57,0x65,0x29,0x3d,0xcb,0xa3,0x9c,0x0c,0x57,0x32 };

SM3\_256(Msg1, MsgLen1, MsgHash1); SM3\_256(Msg2, MsgLen2, MsgHash2);

a = memcmp(MsgHash1, StdHash1, SM3\_len / 8); b = memcmp(MsgHash2, StdHash2, SM3\_len / 8);

if ((a == 0) && (b == 0)) {

return 0;

}

else {

return 1;

}

}