## NAME

DSASign, DSAVerify, randBytesNeededForDSAParams, randBytesNeededForDSAKeySet, randBytesNeededForDSASign

## SYNOPSIS

**#include <libcrypt.h>**

**int randBytesNeededForDSAParams(int primebits, int facbits)**

**int randBytesNeededForDSAKeySet(int facbits)**

**int randBytesNeededForDSASign(int facbits)**

**DSAParams *genDSAParams(int primebits, int facbits, BigInt randomStart)**

**DSAKeySet *genDSAKeySet(DSAParams *params, int pbits, int qbits, BigInt randomStart)**

**DSASignature *DSASign(BigInt m, DSAPrivateKey *key, BigInt randomStart)**

**DSASignature *quantized_DSASign(BigInt m, DSAPrivateKey *key, BigInt randomStart)**

**Boolean DSAVerify(BigInt m, DSASignature *sig, DSAPublicKey *key)**

## DESCRIPTION

For all of these functions *BigInt* arguments are assumed to be initialized *BigInts*. *bigInit*(0) should be used for initialization of empty *BigInts*. If more space is needed than is provided the functions themselves take care of the space allocation.

In all of the function which have an argument, *randomStart*, if this argument is chosen NULL, the default, DES-based pseudo-RNG seeded with truerand() will be used to generate necessary random material. If *randomStart* is not NULL, the accompanying functions, *randBytesNeededForEG...* are used to find out how many random bytes are expected.

In all above functions, *primebits* is the number of bits in the modulus, and *facbits* is the number of bits in $p-1$. Typically these are 1024 and 160 respectively.

*genDSAParams* generates a prime p, a prime factor of p-1, and a primitive root mod p. It is sometimes the case that large groups will use the same set of parameters. This is the reason for decoupling this from the genKeySet routine which actually generates the secret exponent and public and private keys.

These functions create digital signatures along the guidelines proposed by the National Institute of Standards and Technology (NIST). The cryptosystem is a variant of El Gamal systems. The keys are the same as those used by our implementation of the El Gamal system (actually the El Gamal implementation uses the key scheme suggested by NIST and Schnorr).

*DSASign* generates a digital signature for message *m* using a private key. The quantized version use STD_QUANTUM milliseconds as the quantum for execution to defend against Kocher-style timing attacks. STD_QUANTUM is defined in libcrypt.h and should be tuned to your environment.

*DSAVerify* verifies that a signature received with a message *m* is valid using the public key which corresponds to the private key used to generate the signature.

Usage example:

randbytes = randBytesNeededForDSAParams(1024, 160);

BigInt randomStart = bigInit(0);

bigRand(randbytes, randomStart, PSEUDO);

params = genDSAParams(1024, 160, randomStart);

randbytes = randBytesNeededForDSAKeySet(160);

bigRand(randbytes, randomStart, PSEUDO);

DSAKeys = genDSAKeySet(params, 1024, 160, randomStart);

To sign the SHS digest of file "foo":

randbytes = randBytesNeededForDSASign(160);

bigRand(randbytes, randomStart, PSEUDO);

BigInt mdigest = bigInit(0);

BigMessageDigest("foo", mdigest, SHS);

DSAsig = DSASign(mdigest, DSAKeys->DSAPrivateKey, randomStart);

To verify:

BigInt mdigest = bigInit(0);

BigMessageDigest("foo", mdigest, SHS);

Boolean verify = DSAVerify(mdigest, DSAsig, DSAKeys->DSAPublicKey);

## SEE ALSO

**bigInit**(3BM), **bigTest**(3BM), **bigBits**(3BM), **bigToBuf**(3BM), **bufToBig**(3BM), **fBigPrint**(3BM), **atobig**(3BM), **bigPow**(3BM), **EGSign**(3BM), **EGVerify**(3BM), **genStrongPrimeSet**(3BM) **start_quantize**(3BM) **end_quantize**(3BM)

## NAME

genRSAKeySet, getRSAPublicKey, getRSAPrivateKey, RSAEncrypt, RSADecrypt, RSASign, RSAVerify, randBytesNeededForRSA, freeRSAKeys, freeRSASig, RSAPrivateKeyDesEncrypt, RSAPrivateKeyDesDecrypt

## SYNOPSIS

**#include <libcrypt.h>**

**int randBytesNeededForRSA(int modbits, int publicExpBits)**

**RSAKeySet *genRSAKeySet(modulusBits, publicExpBits, randomStart)**
**int modulusBits, publicExpBits;**
**BigInt randomStart;**

**RSAPublicKey *getRSAPublicKey(RSAKeySet)**
**RSAKeySet *RSAKeySet;**

**RSAPrivateKey *getRSAPrivateKey(RSAKeySet)**
**RSAKeySet *RSAKeySet;**

**BigInt RSAEncrypt(message, RSAPubKey)**
**BigInt message;**
**RSAPublicKey *RSAPubKey;**

**BigInt RSADecrypt(message, RSAPrivKey)**
**BigInt message;**
**RSAPrivateKey *RSAPrivKey;**

**BigInt quantized_RSADecrypt(message, RSAPrivKey)**
**BigInt message;**
**RSAPrivateKey *RSAPrivKey;**

**RSASignature *RSASign(message, RSAPrivKey)**
**BigInt message;**
**RSAPRivateKey *RSAPrivKey;**

**RSASignature *quantized_RSASign(message, RSAPrivKey)**
**BigInt message;**
**RSAPRivateKey *RSAPrivKey;**

**Boolean RSAVerify(message, signature, RSAPubKey)**
**BigInt message;**
**RSASignature *signature;**
**RSAPublicKey *RSAPubKey;**

**freeRSAKeys(RSAKeySet)**
**RSAKeySet *RSAKeySet;**

**freeRSASig(RSASig)**
**RSASignature *RSASig;**

**void RSAPrivateKeyDesEncrypt(rsakey, deskey)**
**RSAPrivateKey *rsakey;**
**unsigned char deskey[8];**

**void RSAPrivateKeyDesDecrypt(rsakey, deskey)**
**RSAPrivateKey *rsakey;**
**unsigned char deskey[8];**

## DESCRIPTION

For all of these functions *BigInt* arguments are assumed to be initialized *BigInts*. *bigInit*(0) should be used for initialization of empty *BigInts*. If more space is needed than is provided the functions themselves take care of the space allocation.

These functions assume the general RSA cryptosystem in which a public exponent *e*, a private exponent *d*, and a modulus *n* are needed to do encryption and decryption. For message *m*, encryption is done by calculating $E(m) = m^e \, mod \;\; n$ and decryption by calculating $D(E(m)) = m = E(m)^d \, mod \;\; n$. Decryption may be sped up using the Chinese Remainder Theorem if the factorization of *n* is known.

*randBytesNeededForRSA* returns the number of bytes needed for a non-NULL *randomStart* in the key generation routine following.

*genRSAKeySet* takes as input the length needed for the modulus and the length (in bits) needed for the public exponent and generates *n*, *e* and *d*. It also generates the necessary pieces for Chinese Remainder Theorem modular exponentiation. These pieces are included as part of the private key. The structure which *genRSAKeySet* returns contains both the public and private keys. These can be retrieved by using *getRSAPublicKey* and *getRSAPrivateKey* with the key set as argument. If *randomStart* is NULL, the default, pseudo-random number generator (desRandom()) is used. If it is not NULL, fill it with the appropriate number of bytes retrieved using *randBytesNeededForRSA* using the RNG of your choice.

*RSAEncrypt* does the modular exponentiation $m^e \, mod \;\; n$. and returns the *BigInt* result.

*RSADecrypt* uses the private key and the Chinese Remainder Theorem to do the same calculation much more quickly. The quantized version use STD_QUANTUM milliseconds as the quantum for execution to defend against Kocher-style timing attacks. (The same is true of quantized_RSASign). STD_QUANTUM is defined in libcrypt.h and should be tuned to your environment.

*RSASign* computes an RSA signature using an RSA private key (actually just an RSA Decryption).

*RSAVerify* verifies that an RSA signature was indeed generated using the given message (actually an RSA encryption of the signature using an RSA public key is compared to the message).

*freeRSAKeySet* frees the key set structure and all of its associated BigInts.

*freeRSASig* frees the RSA signature (actually just a BigInt).

*RSAPrivateKeyDes[En/De]crypt* should be used to use DES to [en/de]crypt an RSA private key. *f[put/get]RSAPrivateKey* can then be used to store/retrieve the private key.

## SEE ALSO

**bigInit**(3BM), **bigTest**(3BM), **bigBits**(3BM), **bigToBuf**(3BM), **bufToBig**(3BM), **fBigPrint**(3BM), **atobig**(3BM), **bigPow**(3BM), **randomBytes**(3BM), **fputRSAPrivateKey**(3BM), **start_quantize**(3BM), **end_quantize**(3BM)

## NAME

bigAdd, bigSubtract, bigDivide, bigLeftShift, bigRightShift, bigCompare, bigCopy, bigMultiply, bigMod, bigPow, negate, crtCombine

## SYNOPSIS

**#include <libcrypt.h>**

```
typedef struct Bignum {
        Sign sign;
        size_t length; /* length of number (in Ulong units)    */
        size_t space; /* storage space (in Ulong units) */
        Ulong *num;
} Bignum;

typedef Bignum * BigInt;
```

**void bigAdd(BigInt a, BigInt b, BigInt result)**

**void bigSubtract(BigInt a, BigInt b, BigInt result)**

**void bigDivide(BigInt a, BigInt divisor, BigInt quotient, BigInt remainder)**

**void bigLeftShift(BigInt a, int numbits, BigInt result)**

**void bigRightShift(BigInt a, int numbits, BigInt result)**

**void bigAnd(BigInt a, BigInt b, BigInt result)**

**void bigOr(BigInt a, BigInt b, BigInt result)**

**void bigXor(BigInt a, BigInt b, BigInt result)**

**int bigCompare(BigInt a, BigInt b)**

**void bigCopy(BigInt a, BigInt result)**

**void bigMultiply(BigInt a, BigInt b, BigInt result)**

**void bigMod(BigInt a, BigInt modulus, BigInt result)**

**void bigPow(BigInt a, BigInt exponent, BigInt modulus, BigInt result)**

**void double_bigPow(BigInt a1, BigInt a2, BigInt exp1, BigInt exp2, BigInt modulus, BigInt result)**

**Table *g16_bigpow(BigInt a, BigInt modulus, int len_of_exp_in_4bit_nibs)**

**void brickell_bigpow(Table *table, BigInt exponent, BigInt modulus, BigInt result)**

**void void negate (BigInt a, BigInt modulus, BigInt result)**

**void void crtCombine (BigInt a, BigInt b, BigInt p, BigInt q, BigInt c12, BigInt result)**

## DESCRIPTION

For all of these functions *BigInt* arguments are assumed to be initialized *BigInts*. *bigInit*(0) should be used for initialization of empty *BigInts*. If more space is needed than is provided the functions themselves take care of the space allocation. For example:

```
result = bigInit(0);
bigAdd(a, b, result);
```

*bigAdd* adds two *BigInts* together and stores the result in *result*.

*bigSubtract* subtracts *BigInt b* from *BigInt a* and stores the result in *result*. If the second argument is larger than the first, the sign of the result is negative. *bigTest*(*b*) is used to check the sign of a *BigInt*.

*bigDivide* divides *a* by *divisor* and stores the quotient in *quotient* and the remainder in *remainder*.

```
quotient = bigInit(0);
remainder = bigInit(0);
bigDivide(a, divisor, quotient, remainder);
```

*bigLeftShift* and *bigRightShift* shift their first argument by *numbits* to the left and right respectively, and place the result in *result*.

*bigAnd*, *bigOr* and *bigXor* do the obvious.

*bigCompare* compares two *BigInts*. The result of the comparison is an integer which is positive if $a > b$, negative if $a < b$ or zero if $a = b$.

*bigCopy* copies a *BigInt* into another.

*bigMultiply* multiplies two *BigInts* using a combination of $O(n^{log 3})$ and $O(n^2)$ algorithms and places the result in *result*.

*bigMod* returns $a\ mod\ modulus$ in *result*.

*bigPow* computes $result = a^b mod\ n$ using *addition chaining tables* and *Montgomery reduction*.

*double_bigPow* computes $result = a^b c^d mod\ p$

If one expects to use modular exponentiation using the same base and modulus (as in El Gamal or Diffie-Hellman systems) one can precompute a table to be used in Brickell's speedup for exponentiation. If the calculation is $r = \alpha^e mod\ p$, we calculate $table = g16\_bigpow(\alpha, p, length\_of\_e)$ and $brickell\_bigpow(table, e, p, r)$. Here, length_of_e is the number of 4 bit nibbles in the exponent.

*negate* returns $-a\ mod\ p$ by calculating $a*(p-1)\ modp$.

*crtCombine* does a Chinese Remainder combination of a and b with respect to primes p and q with coefficient, $c12 = p\ mod\ q$.

## SEE ALSO

**bigInit**(3BM), **bigTest**(3BM), **bigBits**(3BM), **bigToBuf**(3BM), **bufToBig**(3BM), **fBigPrint**(3BM), **atobig**(3BM)

## NAME

bigInit, bigTest, bigBits, bigBytes, freeBigInt, fBigPrint, bufToBig, bigToBuf, atobig, bigsprint, handle_exception

## SYNOPSIS

**#include <libcrypt.h>**

**BigInt bigInit(unsigned long i)**

**Sign bigTest(BigInt big)**

**int bigBits(BigInt big)**

**int bigBytes(BigInt big)**

**void freeBignum(BigInt big)**

**void fBigPrint(BigInt big, FILE *fp)**

**void bufToBig(unsigned char *buf, int len, BigInt big)**

**void bigToBuf(BigInt big, int bufsize, unsigned char *buf)**

**BigInt atobig(char *a)**

**int bigsprint(BigInt, unsigned char *buf)**

**void handle_exception(ExceptionType type, char *exception_msg)**

## DESCRIPTION

For all of these functions *BigInt* arguments are assumed to be initialized *BigInts*. *bigInit*(0) should be used for initialization of empty *BigInts*. If more space is needed than is provided the functions themselves take care of the space allocation.

*bigInit* takes an unsigned long integer and returns an equivalent *BigInt* structure. *bigInit*(0) is the best way to initialize a *BigInt*. For example:

```
result = bigInit(0);
bigAdd(a, b, result);
```

*bigTest* returns the sign of a *BigInt*. It will be either *POS* or *NEG*.

*bigBits* returns the number of bits in a *BigInt*.

*bigBytes* returns the number of bytes in a *BigInt*.

*freeBigInt* frees a *BigInt* struct.

*fBigPrint* prints a *BigInt* in hexadecimal format to a stream. The number is printed most significant byte first.

*bufToBig* converts a buffer of characters to a *BigInt*. The least significant char in the buffer is the least significant in the big.

*bigToBuf* converts a *BigInt* to a buffer of characters. The least significant byte n the big goes into the least significant byte in the buffer.

If the buffer size is larger than that needed for the bignum, the extra bytes are zeroed. *bufToBig* will trim these leading zeroes when used to reconvert. Obviously, *bufsize* needs to be at least the same as the bignum (4*LENGTH(big)).

*atobig* converts a string to a *BigInt*. It is assumed that the string is a hexadecimal representation of a *BigInt*.

If one writes a *BigInt* to a file using *fBigPrint* and reads it back into a string using *fscanf*, *atobig* is used to convert the string to a *BigInt*.

*bigsprint* fills a buffer with the hex representation of a big.

*handle_exception* handles all known error conditions. There are two types: CRITICAL and WARNING. A CRITICAL error prints a message contained in exception_msg and exits. A WARNING just prints the message.

## NAME

key_crunch, setupDESState, setupTripleDESState, blockEncrypt, blockDecrypt, block3Encrypt, block3Decrypt, bufferEncrypt, bufferDecrypt, buffer3Encrypt, buffer3Decrypt

## SYNOPSIS

**#include "libcrypt.h"**

```
typedef struct {
        ModeType mode;
        unsigned char icv[8];
        unsigned char int_key[128];
        int count;
        Ulong setup;
} DESState;

typedef struct {
        ModeType mode;
        unsigned char icv[8];
        unsigned char int_key[3][128];
        int count;
        Ulong setup;      /* Set to 0xdeadbeef when setupTripleDESState is done */
} TripleDESState;

typedef int ModeType;

#define ECB 10
#define CBC 20
#define OFM 30
#define ECB3 13
#define CBC3 23
#define OFM3 33
```

**void setupDESState (DESState *s, unsigned char *key, unsigned char *icv, ModeType mode)**

**void setupTripleDESState (TripleDESState *s, unsigned char *key[3], unsigned char *icv, ModeType mode)**

**void key_crunch (unsigned char buffer[], int buflen, unsigned char key[8])**

**void blockEncrypt (unsigned char block[8], DESState *state)**

**void blockDecrypt (unsigned char block[8], DESState *state)**

**void bufferEncrypt (unsigned char *buf, int buf_len, DESState *state)**

**void bufferDecrypt (unsigned char *buf, int buf_len, DESState *state)**

**void block3Encrypt (unsigned char block[8], TripleDESState *state)**

**void block3Decrypt (unsigned char block[8], TripleDESState *state)**

**void buffer3Encrypt (unsigned char *buf, int buf_len, TripleDESState *state)**

**void buffer3Decrypt (unsigned char *buf, int buf_len, TripleDESState *state)**

**void bignumDesEncrypt (BigInt, unsigned char key[8])**

**void bignumDesDecrypt (BigInt, unsigned char key[8])**

**void bignumCBCDesEncrypt (BigInt, unsigned char key[8])**

**void bignumCBCDesDecrypt (BigInt, unsigned char key[8])**

**void blockEightBitCFMEncrypt (unsigned char in[8], int len, unsigned char sreg[8], unsigned char key[8])**

**void blockEightBitCFMDecrypt (unsigned char in[8], int len, unsigned char sreg[8], unsigned char key[8])**

**void blockSixtyFourBitCFMEncrypt (unsigned char in[8], int len, unsigned char sreg[8], unsigned char key[8])**

**void blockSixtyFourBitCFMDecrypt (unsigned char in[8], int len, unsigned char sreg[8], unsigned char key[8])**

## DESCRIPTION

Single DES modes use DESState and *setupDESState* to initialize state. Triple DES modes used TripleDESState and *setupTripleDESState*. Each of these setup functions transform a key into a 128 byte expanded key, set the mode in the state and do the appropriate thing with initialization vectors (icv's).

*key_crunch*R expects input which is *numbytes* long. This is converted to an 8 byte key to be used in the block encryption functions.

*blockEncrypt* and *blockDecrypt* are the basic DES block [en/de]cryption functions. The mode in the state causes the appropriate mode implementations to function.

The buffer encryption functions operate on buffers 8 bytes at a time. These functions handle buffers which are not an even multiple of 8 bytes in a way which is interoperable with the corresponding buffer decryption functions. The ECB buffer encryption functions encrypt an array containing the first 8 digits [0-7] and XOR the result with the remaining fractured block. The CBC and OFM functions just continue in their normal mode but only XOR their icv or mask with the appropriate number of bytes left in the fractured block.

The functions which contain a 3 in their name, are triple DES [en/de]cryption functions which correspond to the single encryption functions described. Each of these assumes three 8 byte DES keys. The most commmon way of using triple DES is to make key 1 and key 3 the same. All of these functions do encryption as, encrypt[key1], decrypt[key2], encrypt[key3] and decryption as decrypt[key1], encrypt[key2], decrypt[key3].

The bignum DES [en/de]cryption functions use eith CBC or ECB modes and single DES encryption. Note that the length of an encrypted bignum is 8 bytes longer than the original. The decrypt functions take this into account.

The cipher feedback mode (CFM) functions are currently under revision.

## NAME

genDiffieHellmanSet, DiffieHellmanInit, DiffieHellmanGneKey, freeDiffieHellmanSet, randBytesNeededForDHSet, randBytesNeededForDHInit

## SYNOPSIS

**#include <libcrypt.h>**

```
typedef struct {
        int qbits;
        BigInt alpha, p, q;
        Table *alphatab;
} DiffieHellmanSet;
```

**int randByteNeededForDHSet (int pbits, int qbits);**

**DiffieHellmanSet *GenDiffieHellmanSet (int pbits, int qbits, BigInt randomStart);**

**int randBytesNeededForDHInit (int qbits);**

**void DiffieHellmanInit (DiffieHellmanSet *myDHset, BigInt my_exponent, BigInt my_msg1, BigInt randomStart);**

**void quantized_DiffieHellmanInit (DiffieHellmanSet *myDHset, BigInt my_exponent, BigInt my_msg1);**

**void DiffieHellmanGenKey (DiffieHellmanSet *myDHset, BigInt recd_msg1,**
**BigInt my_exponent, BigInt DH_key));**

**void quantized_DiffieHellmanGenKey (DiffieHellmanSet *myDHset, BigInt recd_msg1,**
**BigInt my_exponent, BigInt DH_key));**

**void freeDiffieHellmanSet (DiffieHellmanSet *DHset);**

## DESCRIPTION

An implementation of Diffie-Hellman key exchange. If *randomStart* is not to be NULL in *genDiffieHellmanSet* use *randBytesNeededForDHSet* with the length of p and q in bits to determine the amount of random data needed for parameter generation. Generate a DiffieHellmanSet which contains a base, a prime p, a prime factor of p-1, and a table for using Brickell's speedup for exponentiation.

This Set must be agreed upon by the participants in the exchange. my_exponent should usually be initialized to 0 unless you want to use a prespecified value.

*DiffieHellmanInit* is then called in which the caller's exponent is used to calculate his half of the exchange. If *randomStart* is not NULL, use *randBytesNeededForDHInit* with the legnth of the sub-prime, q, in bits as its argument, to find the number of random bytes needed to initialize the search for a proper exponent. After this function call, my_exponent should be protected (say using BignumDESEncrypt()) and my_msg1 sent to the other party in the exchange.

After receiving the other party's message, recd_msg1, uncover your exponent and proceed. DH_key will contain the full pbits worth of Diffie Hellman key material for you to process as necessary.

The quantized version use STD_QUANTUM milliseconds as the quantum for execution to defend against Kocher-style timing attacks. STD_QUANTUM is defined in libcrypt.h and should be tuned to your environment.

## SEE ALSO

**bigInit**(3BM), **bigTest**(3BM), **bigBits**(3BM), **bigBytes**(3BM), **bigToBuf**(3BM), **bufToBig**(3BM), **fBigPrint**(3BM), **atobig**(3BM), **bigPow**(3BM), **bignumDES[En/De]crypt**(3BM) **start_quantize**(3BM) **end_quantize**(3BM)

## NAME

genEGParams, genEGKeySet, EGEncrypt, EGDecrypt, EGSign, EGVerify freeEGKeys, freeEGSig, freeEGParams, freeEGPublicKey, freeEGPrivateKey randBytesNeededForEGParams, randBytesNeededForEGKeySet randBytesNeededForEGEncrypt, randBytesNeededForEGSign

## SYNOPSIS

**#include <libcrypt.h>**

**int randBytesNeededForEGParams(int pbits, int qbits);**

**int randBytesNeededForEGKeySet(int qbits);**

**int randBytesNeededForEGEncrypt(int qbits);**

**int randBytesNeededForEGSign(int qbits);**

**EGParams *genEGParams(int numbits_in_p, int numbits_in_q, BigInt randomStart)**

**EGKeySet *genEGKeySet(EGParams *params, int pbits, int qbits, BigInt randomStart)**

**BigInt EGEncrypt(BigInt m, EGPublicKey *key, BigInt randomStart)**

**BigInt EGDecrypt(BigInt encryptedMessage, EGPrivateKey *key)**

**BigInt quantized_EGDecrypt(BigInt encryptedMessage, EGPrivateKey *key)**

**EGSignature *EGSign(BigInt m, EGPrivateKey *key, BigInt randomStart)**

**EGSignature *quantized_EGSign(BigInt m, EGPrivateKey *key)**

**Boolean EGVerify(BigInt m, EGSignature *sig, EGPublicKey *key)**

**void freeEGKeys(EGKeySet *keyset)**

**void freeEGPublicKey(EGPublicKey *key)**

**void freeEGPrivateKey(EGPrivateKey *key)**

**void freeEGSig(EGSignature *sig)**

## DESCRIPTION

For all of these functions *BigInt* arguments are assumed to be initialized *BigInts*. *bigInit*(0) should be used for initialization of empty *BigInts*. If more space is needed than is provided the functions themselves take care of the space allocation.

In all of the function which have an argument, *randomStart*, if this argument is chosen NULL, the default, DES-based pseudo-RNG seeded with truerand() will be used to generate necessary random material. If *randomStart* is not NULL, the accompanying functions, *randBytesNeededForEG...* are used to find out how many random bytes are expected.

El Gamal cryptosystems may use a different random prime *p*, a primitive root $\alpha (mod\ p)$ and a secret $X_a$ for each user. If genEGParams() is called before creating keys the Params may be used to ensure that all (or a group of) users share the same parameters. If *genEGKeySet*( ) is called with params = NULL, a new set of parameters will be created for each key set. A user's public key is $Y_a = \alpha^{X_a}\ mod\ p$. To encrypt a message *m*, one creates a key $K = Y_b^{X_a}\ mod\ p$. The encrypted message is then represented as the pair $(Y_a, K*m\ mod\ p)$ A recipient of the pair can then regenerate the key as $K = Y_a^{X_b}\ mod\ p$ and then divide the second part of the received pair by *K* to get *m*.

*genEGParams* generates the prime *p* and the primitive root α and returns them in an *EGParams* structure. It takes as argument the desired length of *p* in bits.

*genEGKeySet* generates the public and private keys for El Gamal systems. The private key differs from the public key in that the secret $X_a$ is included. This function takes the an *EGParams* structure pointer as generated by *genEGParams* as argument.

*getEGPublicKey* and *getEGPrivateKey* take an *EGKeySet* structure pointer as an argument and return the public or private key.

If the thing being signed or encrypted is smaller than q, the large prime factor of p-1, the El Gamal functions will use Brickell's modular exponentiation enhancement whereever possible.

*EGEncrypt* encrypts a *BigIntmessage* using a *publickey* and returns an *EGEncryptedMessage* structure which is the pair of *BigInts* discussed above.

*EGDecrypt* takes the pair generated by *EGEncrypt* and and the private key associated with the public key used for encryption and returns the original *BigIntmessage*. The quantized version use STD_QUANTUM milliseconds as the quantum for execution to defend against Kocher-style timing attacks. (The same is true of quantized_EGSign). STD_QUANTUM is defined in libcrypt.h and should be tuned to your environment.

*EGSign* generates a digital signature for message *m* using a private key.

*EGVerify* verifies that a signature received with a message *m* is valid using the public key which corresponds to the private key used to generate the signature.

*DSSSign* and *DSSVerify* are much more efficient and use the same keys as the El Gamal counterparts. The signature is also shorter.

*free** free their respective structures.

**SEE ALSO**

**bigInit**(3BM), **bigTest**(3BM), **bigBits**(3BM), **bigBytes**(3BM), **bigToBuf**(3BM), **bufToBig**(3BM), **fBigPrint**(3BM), **atobig**(3BM), **bigPow**(3BM) **start_quantize**(3BM) **end_quantize**(3BM)

## NAME

getInverse, extendedGcd, gcd

## SYNOPSIS

**#include <libcrypt.h>**

**void getInverse(p, q, result)**
**BigInt p, q, result;**

**void extendedGcd(u, v, u′, v′, gcd)**
**BigInt u, v, u′, v′, gcd;**

**BigInt gcd(u, v)**
**BigInt u, v;**

## DESCRIPTION

For all of these functions *BigInt* arguments are assumed to be initialized *BigInts*. *bigInit*(0) should be used for initialization of empty *BigInts*. If more space is needed than is provided the functions themselves take care of the space allocation.

*getInverse* returns $p^{-1} \, mod \; q$ in *result*.

*extendedGcd* is Euclid's extended Greatest Common Divisor algorithm. It is based on the equation:

$$u*u' - v*v' = gcd(u, v).$$

*getInverse* is just a call to *extendedGcd* when it is known that $gcd(p, q) = 1$.

## SEE ALSO

**bigInit**(3BM), **bigTest**(3BM), **bigBits**(3BM), **bigToBuf**(3BM), **bufToBig**(3BM), **fBigPrint**(3BM), **atobig**(3BM), **bigMod**(3BM), **bigDivide**(3BM), **bigSubtract**(3BM), **bigMultiply**(3BM)

## NAME

MD5Init, MD5Update, MD5Final

## SYNOPSIS

**#include <libcrypt.h>**

**MD5Init(MD5_CTX *mdContext)**

**MD5Update(MD5_CTX *mdContext, unsigned char *data, int numbytes)**

**MD5Final(unsigned char digest[16], MD5_CTX *mdContext)**

## DESCRIPTION

For all of these functions *BigInt* arguments are assumed to be initialized *BigInts*. *bigInit*(0) should be used for initialization of empty *BigInts*. If more space is needed than is provided the functions themselves take care of the space allocation.

*MD5Init* takes a *MD5_CTX* structure as input initialized simply as:

```
mdContext = (MD5_CTX *)malloc(sizeof(MD5_CTX));
```

This function must be called to start the digest process.

*MD5Update* is the digest work routine. It is fed 16 byte chunks of data and updates the digest.

*MDFinal* is called to terminate the digest.

*bigMessageDigest* uses these three functions to create a digest for a buffer of characters. The digest is returned in a *BigInt*.

*MD5Print* prints the digest to stdout.

MD2 and MD4 are also supported and have exactly the same interface except that MD4Final leaves the digest in the context structure: *MD4Final*(*MD4_CTX** *context*).

## SEE ALSO

**bigInit**(3BM), **bigTest**(3BM), **bigBits**(3BM), **bigToBuf**(3BM), **bufToBig**(3BM), **fBigPrint**(3BM), **atobig**(3BM)

## NAME

fBigMessageDigest, bigMessageDigest

## SYNOPSIS

**#include <libcrypt.h>**

**void fBigMessageDigest(char *filename, BigInt digest, DigestType digestType)**

**void bigMessageDigest(unsigned char *buf, int buflen, BigInt digest, DigestType digestType)**

This is a simple interface to SHS, MD2, MD4 and MD5.
*digest* must be initialized (digest = bigInit(0)).
The digest types are defined as follows:

```
#define SHS 1
#define MD5 5
#define MD4 4
#define MD2 3
typedef int DigestType;
```

## SEE ALS

**bigInit**(3BM), **bigTest**(3BM), **bigBits**(3BM), **bigToBuf**(3BM), **bufToBig**(3BM), **fBigPrint**(3BM), **atobig**(3BM),

**NAME**

**SYNOPSIS**

**#include <libcrypt.h>**

Stdio-based functions:

**long fgetLength P((FILE *stream);**

**void fputLength P((long length, register FILE *stream);**

**int fgetBoolean P((FILE *stream, int form);**

**void fputBoolean P((int value, register FILE *stream, int form);**

**long fgetInteger P((FILE *stream, int form);**

**void fputInteger P((long value, register FILE *stream, int form);**

**long fgetString P((unsigned char *string, int limi,**
register FILE *stream, int form));

**void fputString P((unsigned char *cp, long lengt,**
register FILE *stream, int form, int stringtype));

**unsigned char *fgetBitString P((unsigned char *bstring, int limi,**
register FILE *stream, int form));

**void fputBitString P((unsigned char *bstring, int blengt,**
register FILE *stream,int form));

**int featDataUnit P((FILE *stream, int form, register tag);**

**void fgetEOC P((FILE *stream, int form);**

**void fputEOC P((FILE *stream);**

**void fputBigInt P((BigInt big, FILE *stream);**

**BigInt fgetBigInt P((FILE *stream);**

**void fputTable P((Table *table, FILE *stream);**

**Table * fgetTable P((FILE *stream);**

**void fputRSAPublicKey P((RSAPublicKey *key, FILE *stream);**

**RSAPublicKey * fgetRSAPublicKey P((FILE *stream);**

**void fputRSAPrivateKey P((RSAPrivateKey *key, FILE *stream);**

**RSAPrivateKey * fgetRSAPrivateKey P((FILE *stream);**

**void fputRSASignature P((RSASignature *sig, FILE *stream);**

**RSASignature * fgetRSASignature P((FILE *stream);**

**RSAPublicKey * nfgetRSAPublicKey P((char *);**

**RSAPrivateKey * nfgetRSAPrivateKey P((char *);**

**void fputEGParams P((EGParams *params, FILE *stream);**

**EGParams * fgetEGParams P((FILE *stream);**

**void fputEGPublicKey P((EGPublicKey *key, FILE *stream);**

**EGPublicKey * fgetEGPublicKey P((FILE *stream);**

**void fputEGPrivateKey P((EGPrivateKey *key, FILE *stream);**

**EGPrivateKey * fgetEGPrivateKey P((FILE *stream);**

**void fputEGSignature P((EGSignature *sig, FILE *stream);**

**EGSignature * fgetEGSignature P((FILE *stream);**

**void fputDSASignature P((DSASignature *sig, FILE *stream);**

**DSASignature * fgetDSASignature P((FILE *stream);**

Buffer-based functions:

**long bufGetLength P((unsigned char **buffer);**

**void bufPutLength P((long length, register unsigned char **buffer);**

**int bufGetBoolean P((unsigned char **buffer, int form);**

**void bufPutBoolean P((int value, register unsigned char **buffer, int form);**

**long bufGetInteger P((unsigned char **buffer, int form);**

**void bufPutInteger P((long value, register unsigned char **buffer, int form);**

**long bufGetString P((unsigned char *string, int limit,**
**register unsigned char **buffer, int form));**

**void bufPutString P((unsigned char *cp, long length,**
**register unsigned char **buffer, int form, int stringtype));**

**unsigned char *bufGetBitString P((unsigned char *bstring, int limit,**
**register unsigned char **buffer, int form));**

**void bufPutBitString P((unsigned char *bstring, int blength,**
**register unsigned char **buffer,int form));**

**int bufEatDataUnit P((unsigned char **buffer, int form, register tag);**

**void bufGetEOC P((unsigned char **buffer, int form);**

**void bufPutEOC P((unsigned char **buffer);**

**void bufPutBigInt P((BigInt big, unsigned char **buffer);**

**BigInt bufGetBigInt P((unsigned char **buffer);**

**void bufPutTable P((Table *table, unsigned char **buffer);**

**Table * bufGetTable P((unsigned char **buffer);**

**void bufPutRSAPublicKey P((RSAPublicKey *key, unsigned char **buffer);**

**RSAPublicKey * bufGetRSAPublicKey P((unsigned char **buffer);**

**void bufPutRSAPrivateKey P((RSAPrivateKey *key, unsigned char **buffer);**

**RSAPrivateKey * bufGetRSAPrivateKey P((unsigned char **buffer);**

**void bufPutRSASignature P((RSASignature *sig, unsigned char **buffer);**

**RSASignature * bufGetRSASignature P((unsigned char **buffer);**

**void bufPutEGParams P((EGParams *params, unsigned char **buffer);**

**EGParams * bufGetEGParams P((unsigned char **buffer);**

**void bufPutEGPublicKey P((EGPublicKey *key, unsigned char **buffer);**

**EGPublicKey * bufGetEGPublicKey P((unsigned char **buffer);**

**void bufPutEGPrivateKey P((EGPrivateKey *key, unsigned char **buffer);**

**EGPrivateKey * bufGetEGPrivateKey P((unsigned char **buffer);**

**void bufPutEGSignature P((EGSignature *sig, unsigned char **buffer);**

**EGSignature * bufGetEGSignature P((unsigned char **buffer);**

**void bufPutDSASignature P((DSASignature *sig, unsigned char **buffer);**

**DSASignature * bufGetDSASignature P((unsigned char **buffer);**

## DESCRIPTION

These functions may be used to move various cryptolib structures in and out of buffers or stdio streams. Functions of the form bufPut or bufGet use usigned char ** buffers and are useful for creating messages out of various structures. Stdio-based functions are useful for moving objects in and out of files. ASN.1 (abstract syntax notation 1) is used to push these structures around. This is not BER or DER (Basic and Distinguished Encoding rules).

*This part of CryptoLib is probably most subject to change and should not be relied upon too strictly.*

## NAME

setPrimeAttempts, primeTest, getPrime, genStrongPrimeSet, genStrongPrime, randBytesNeededForPrime, randBytesNeededForRoot

## SYNOPSIS

**#include <libcrypt.h>**

**void setPrimeAttempts(int i)**

**Boolean primeTest(BigInt big)**

**int randBytesNeededForPrime(int pbits, int qbits, PrimeType type)**

**void getPrime(int numbits, BigInt result, BigInt randomStart)**

**void genStrongPrimeSet(int pbits, BigInt p, int qlen, BigInt q, PrimeType type, BigInt randomStart)**
**int type (either NIST or GORDON)**

**void genStrongPrime(int pbits, BigInt p, BigInt randomStart)**

**int randBytesNeededForRoot(int pbits)**

**void getPrimitiveElement(BigInt el, BigInt p, BigInt q, BigInt randomStart)**

## DESCRIPTION

For all of these functions *BigInt* arguments are assumed to be initialized *BigInts*. *bigInit*(0) should be used for initialization of empty *BigInts*. If more space is needed than is provided the functions themselves take care of the space allocation.

*primeTest* has two phases. The first is a test for divisibility of the candidate by one of the first 54 primes (2 - 251). After this test is passed, the second phase begins in which the candidate is passed to a probabilistic phase for a number of trials. The probability that a number which passes this test is not prime is $1/25^n$ where n is the number of passes. *primeTest* returns *TRUE* or *FALSE*.

*setPrimeAttempts* is used to set the number of passes *primeTest* should make through its probabilistic phase. The default value of *primeTestAttempts* is 5.

*randBytesNeededForPrime* returns the number of bytes needed for *randomStart* in any of the prime gerneration routines. If type is GORDON, qbits is 0, otherwise, qbits if the number of bits desired in the large prime factor of $p-1$.

*getPrime* generates a random prime of length *numbits*. If *randomStart* is NULL, the pseudo-random number generator is called. This is true for all of the functions containing a randomStart variable.

*genStrongPrimeSet* is called with the length in bits of the prime and of the prime factor of p-1 and two empty *BigInt* pointers as its arguments. The first pointer will contain the *strong prime* and the second pointer will contain a prime factor of $strongPrime - 1$. If $type \equiv GORDON$ the prime will be a Gordon strong prime. If $type \equiv NIST$ it will be strong in the sense discussed in the NIST Digital Signature Standard. In either case, $strongPrime - 1$ has a large, prime factor, $factorOfPminus1$.

*genStrongPrime* is just $genStrongPrimeSet(numbits, prime, bigInit(0)\, NIST)$ That is, this function is used when the factor and the extra properties of Gordon strong primes are not needed. Note here the use of *bigInit*(0). This is not recommended as it will be difficult to free a BigInt allocated in this way.

*randBytesNeededForRoot* returns the number of bytes needed if *randomStart* is not to be NULL in the following function.

*getPrimitiveElement* returns a primitive element mod p. The factor argument is just the large factor of p-1 gotten from one pf the strong prime generation functions.

## SEE ALSO

**bigInit**(3BM), **bigTest**(3BM), **bigBits**(3BM), **bigToBuf**(3BM), **bufToBig**(3BM), **fBigPrint**(3BM), **atobig**(3BM), **bigPseudoRand**(3BM), **bigReallyRand**(3BM)

## NAME

quadResidue, compositeQuadResidue, squareRoot, compositeSquareRoot, bigIsQR, bigJacobi

## SYNOPSIS

**#include <libcrypt.h>**

**Boolean quadResidue (BigInt a, BigInt p);**

**Boolean compositeQuadResidue (BigInt a, BigInt p, BigInt q);**

**void squareRoot (BigInt a, BigInt p, BigInt root);**

**void compositeSquareRoot (BigInt a, BigInt p, BigInt q, BigInt root1, BIgInt root2);**

**Boolean bigIsQR(BigInt a, BigInt p, BigInt q)**

**int bigJacobi(BigInt a, BigInt b)**

## DESCRIPTION

For all of these functions *BigInt* arguments are assumed to be initialized *BigInts*. *bigInit*(0) should be used for initialization of empty *BigInts*. If more space is needed than is provided the functions themselves take care of the space allocation.

$a$ is a quadratic residue if $x^2 \, mod \; n = a$ has a solution $x$. If $n$ is prime, $a^{(n-1)/2} \, mod \; n = 1$. If $n$ is composite, $n = p * q$ for $p$ and $q$ prime, $a$ is a quadratic residue if it is a quadratic residue modulo each prime. In each square root function, it is assumed that the primes are of the form, $p mod 4 = 3$. For the composite modulus, the square roots are formed using the Chinese remainder theorem to combine the square roots modulo each prime. For the composite modulus, there are 4 roots. *compositeSquareRoot* returns two of them, $root1$ and $root2$. The others are $n - root1$ and $n - root2$.

*bigJacobi* calculates the Jacobi symbol (a/b).

*bigIsQR* determines whether a is a quadratic residue mode p∗q where p and q are prime. It does this by calculating the Jacobi symbols (a/p) and (a/q). If *both* are 1, then a is a quadratic residue modulo p∗q.

## SEE ALSO

**bigInit**(3BM), **bigTest**(3BM), **bigBits**(3BM), **bigToBuf**(3BM), **bufToBig**(3BM), **fBigPrint**(3BM), **atobig**(3BM), **crtCombine**(3BM).

## NAME

start_quantize, end_quantize - start/end CPU time quantization

## SYNOPSIS

**int** start_quantize(**int quantum**);

**int** end_quantize();

## DESCRIPTION

The quantize functions provide a simple method to encapsulate code blocks that should run in a multiple of a coarse-grained "quantized" amount of CPU time. They are mainly useful in building certain on-line cryptographic protocols in which an attacker could learn key information by observing the time the target takes to perform calculations that use the secret (c.f., Paul Kocher's attacks against various public-key cryptosystems).

start_quantize(quantum) begins a quantized execution environment that will run in multiples of **quantum** milliseconds. end_quantize() ends the environment, busy-waiting until (approximately) the next quantum multiple.

The quantum should be at least the expected range of normal CPU runtime. 100 ms is a good starting point for exponentiation-based protocols on typical processors, but one should measure each implementation for safest results. Quanta that are too small leak timing information; quanta that are too large are wasteful of CPU usage. The quantized versions of public key operations making use of secrets use STD_QUANTUM millisecs. STD_QUANTUM is defined in libcrypt.h and should be tuned to your environment.

## DIAGNOSTICS

start_quantize() and end_quantize() return 0 on normal exit. If an error occurred (quantization could not be scheduled or a non-positive quantum with start_quantize, or no previous active start_quantum() on end_quantum()) -1 is returned.

## EXAMPLE

start_quantize(100); RSA_Decrypt(...); end_quantize();

## SEE ALSO

setitimer(2), signal(2)

## BUGS

Quantized environments do not nest.

The quantization is only approximate, based on the setitimer(2) function. In particular, the actual quanta will vary with each call (but are independent of the encapsulated code runtime / quantum). Do not rely on fine-grained quanta (less than about 20 milliseconds).

There is no mechanism to adjust the quantum for MIPS

## AUTHORS

Matt Blaze and Jack Lacy, AT&T Bell Laboratories.

## NAME

genRabinKeySet, getRabinPublicKey, getRabinPrivateKey, RabinEncrypt, RabinDecrypt, RabinSign, RabinVerify, freeRabinKeys, freeRabinSig randBytesNeededForRabinSet, randBytesNeededForRabinEncrypt, randBytesNeededForRabinSign

## SYNOPSIS

**#include <libcrypt.h>**

**int randBytesNeededForRabinSet(int modbits)**

**int randBytesNeededForRabinEncrypt(int modbits)**

**int randBytesNeededForRabinSign(int modbits)**

**RabinKeySet *genRabinKeySet(int modulusdBits, BitInt randomStart)**

**BigInt RabinEncrypt(message, RabinPubKey, randomStart)**
**BigInt message;**
**RabinPublicKey *RabinPubKey;**
**BigInt randomStart;**

**BigInt RabinDecrypt(message, RabinPrivKey)**
**BigInt message;**
**RabinPrivateKey *RabinPrivKey;**

**BigInt quantized_RabinDecrypt(message, RabinPrivKey)**
**BigInt message;**
**RabinPrivateKey *RabinPrivKey;**

**RabinSignature *RabinSign(message, RabinPrivKey, randomStart)**
**BigInt message;**
**RabinPRivateKey *RabinPrivKey;**
**BigInt randomStart;**

**RabinSignature *quantized_RabinSign(message, RabinPrivKey, randomStart)**
**BigInt message;**
**RabinPRivateKey *RabinPrivKey;**
**BigInt randomStart;**

**Boolean RabinVerify(message, signature, RabinPubKey)**
**BigInt message;**
**RabinSignature *signature;**
**RabinPublicKey *RabinPubKey;**

**freeRabinKeys(RabinKeySet)**
**RabinKeySet *RabinKeySet;**

**freeRabinSig(RabinSig)**
**RabinSignature *RabinSig;**

## DESCRIPTION

For all of these functions *BigInt* arguments are assumed to be initialized *BigInts*. *bigInit*(0) should be used for initialization of empty *BigInts*. If more space is needed than is provided the functions themselves take care of the space allocation.

In all of the function which have an argument, *randomStart*, if this argument is chosen NULL, the default, DES-based pseudo-RNG seeded with truerand() will be used to generate necessary random material. If *randomStart* is not NULL, the accompanying functions, *randBytesNeededForEG*... are used to find out how many random bytes are expected.

Rabin's scheme is essentially RSA with 2 as the public exponent. Care has been taken to avoid the small exponent attack (See Simmons' Contemporary Cryptology). Encryption, Decryption, Signature and Verification work exactly as with RSA. However, since squaring is the encryption process, to decrypt the square root is taken. This yields 4 square roots. Thus some way to detect which if these is correct is needed. The message to be encrypted is thus given "structure". From most-sig-byte to least, the format is: [random pad | message | msg_len (4 bytes) | digest (pad | msg | msg_len) ] The digest functions supported are SHS, MD{2, 4, 5}. The functions below take as arguments the digest type and digest length in bytes.

In order to sign messages the square root must be calculated. Thus the message to be signed must be put into a form which is a quadratic residue modulo N. It is for this reason that Rabin's signature is usually less efficient than doing RSASign.

The quantized version use STD_QUANTUM milliseconds as the quantum for execution to defend against Kocher-style timing attacks. STD_QUANTUM is defined in libcrypt.h and should be tuned to your environment.

*freeRabinKeySet* frees the key set structure and all of its associated BigInts.

*freeRabinSig* frees the Rabin signature (actually just a BigInt).

## SEE ALSO

**bigInit**(3BM), **bigTest**(3BM), **bigBits**(3BM), **bigToBuf**(3BM), **bufToBig**(3BM), **fBigPrint**(3BM), **atobig**(3BM), **bigPow**(3BM), **fputRabinPrivateKey**(3BM) **start_quantize**(3BM) **end_quantize**(3BM)

## NAME

randomBytes, bigRand, getRandBetween

## SYNOPSIS

**#include <libcrypt.h>**

**void randomBytes(unsigned char *buf, int numbytes, int type)**

**bigRand(int numbytes, BigInt result, int type)**

**getRandBetween(BigInt big1, BigInt big2, BigInt result, int type, BigInt randomStart)**

**seed_rng(unsigned char *seed, int seedlen)**

## DESCRIPTION

For all of these functions *BigInt* arguments are assumed to be initialized *BigInts*. *bigInit*(0) should be used for initialization of empty *BigInts*. If more space is needed than is provided the functions themselves take care of the space allocation.

*randomBytes* generates truly random bytes and puts them into a buffer if *type* = *REALLY*. It is based on the randomness in system interrupts. It is very slow relative to pseudo random number generators. It is primarily useful for initiating a search for primes or initializing pseudo random number generators or when a truly random number is necessary.

If *type* = *PSEUDO randomBytes* generates random bytes using a generator based on 3DES in counter mode using a random key. The 64 bit output of 3DES is xor'd with SHA(prefix, counter, suffix). he 64 bit counter is incrememnted for every call to desRandom(). The 3DES keys, prefix and suffix are initialized using truerand() by default or using seed_rng(seed, seedlen). If the latter is used, the seed must be 64 bytes and seedlen = 64. If truerand() is used, the seeding happens only once.

*bigRand* creates a *BigInt* from random bytes generated in the above ways. *type* can be *REALLY* or *PSEUDO* as above.

*getRandBetween* returns a random *BigInt* which is between *big*1 and *big*2. *type* is either *REALLY* or *PSEUDO*. *randomStart* is a random starting point for the new value. If it is NULL, then either truerand (if type = REALLY) or desRandom() (type = PSEUDO) are used for a starting value. If *randomStart* is not NULL, it should contain as many bytes as the *larger* of *a* and *b*.

*seed_rng* seeds the pseudo random number generator to allow for repeatable random bytes. Seeding the generator with one's password can be used to avoid storing private El Gamal exponents. They can just be regenerated at the time of use.

## SEE ALSO

**bigInit**(3BM), **bigBits**(3BM), **bigToBuf**(3BM), **bufToBig**(3BM), **fBigPrint**(3BM), **atobig**(3BM)

## NAME

shs, bigMessageDigest, shsInit, shsUpdate, shsFinal

## SYNOPSIS

**#include <libcrypt.h>**

**(unsigned long *)shs(unsigned char *message, int messageLength)**

**void shsInit (SHS_CTX *context)**

**void shsUpdate (SHS_CTX *context, unsigned char *buffer, int length)**

**void shsFinal (SHS_CTX *context)**

**bigMessageDigest(unsigned char *message, int messageLength, BigInt digest, int type)**

## DESCRIPTION

For all of these functions *BigInt* arguments are assumed to be initialized *BigInts*. *bigInit*(0) should be used for initialization of empty *BigInts*. If more space is needed than is provided the functions themselves take care of the space allocation.

*shs* is an implementation of the NIST proposed Secure Hash Standard. *shs* returns a pointer to 5 unsigned longs which make up a hash value for the message.

```
W struct SHS_CTX {
      long totalLength;
      unsigned long h[5];
      unsigned long w[80];
} SHS_CTX;
```

The normal way to use *shs* is to do the following:

```
SHS_CTX context;
char buffer[BUF_SIZE];
long length;
FILE *fp;

shsInit(&context);
while (length = fread(buffer, 1, BUF_SIZE, fp)
      shsUpdate(&context, buffer, length);
shsFinal(&context);
```

Using this method allows for digest seeding. There are times when *shsUpdate* will be called with a password in the buffer, before entering the read loop. After *shsFinal*() is called, the digest is in context.h[i], with i = 0 to 4. *bigMessageDigest* calls shs to create a digest for a buffer of characters. The digest is returned in a *BigInt*. Ing *bigMessageDigest*, the type argument can be SHS, MD2, MD4 or MD5. MD* syntax is identical to that for SHS substituting "MD2, MD4 or MD5" for "SHS" in structures and commands.

## SEE ALSO

**MD5**(3BM), **bigInit**(3BM), **bigTest**(3BM), **bigBits**(3BM), **bigToBuf**(3BM), **bufToBig**(3BM), **fBigPrint**(3BM), **atobig**(3BM)