IS8U256A/B 密码算法函数库

用户手册

（版本：1.1.2）

北京华大信安科技有限公司
Beijing Huada Infosec Technology Co.,Ltd

2014 年 11 月

声　明

修　订

版本（RC）	修订内容	日期	修订者	备注
1.0	初稿	2013.11.25	裴超 刘景景	
1.0.1	对 RSA 签名和验证增加了部分说明。	2013.12.12	裴超	
1.0.2	对 RSA 密钥生成函数进行修改调整。	2014.01.24	裴超	
1.0.3	a) 功耗寄存器的保护； b) 部分 IP 互斥开启； c) 增加对 SM2 的使用介绍。	2014.02.12	裴超	
1.0.4	增加各函数中的 RAM 校验的操作。	2014.03.07	裴超	
1.0.5	修正 HASH 函数对 55 字节长度处理错误的问题。	2014.05.23	裴超	
1.1.0	a) 增加新功能，支持通用 ECC 算法； b) 增加文档中的部门功能说明。	2014.05.16	刘景景 裴超	
1.1.1	修正 SM2 解密函数中 R2 寄存器错误保护的问题。	2014.6.25	裴超	
1.1.2	a) 调整 RSA 签名函数的定义方式； b) 补充算法库使用配置功能说明。	2014-11-21	裴超	

目　录

第 1 章　概 述

1.1　简介

IS8U256A/B 密码算法软件函数库开发包（Crypto Software Development Kits，以下简称“CSDK”）为针对北京华大信安科技有限公司 IS8U256 系列芯片定制设计，提供所支持的密码算法的 C 语言和 51 汇编语言的函数调用接口，支持符合国际标准和国密标准的对称、非对称和哈希等密码算法。

1.2　功能说明

算法库支持 DES、Triple DES、RSA、ECC、SHA 等国际算法，同时支持 SM1、SM2、SM3、SM4 等国密算法，可辅助进行 CSP、PKCS#11 的开发，帮助快速实现 COS 的基本安全功能。

CSDK 的目录组织结构图如下：

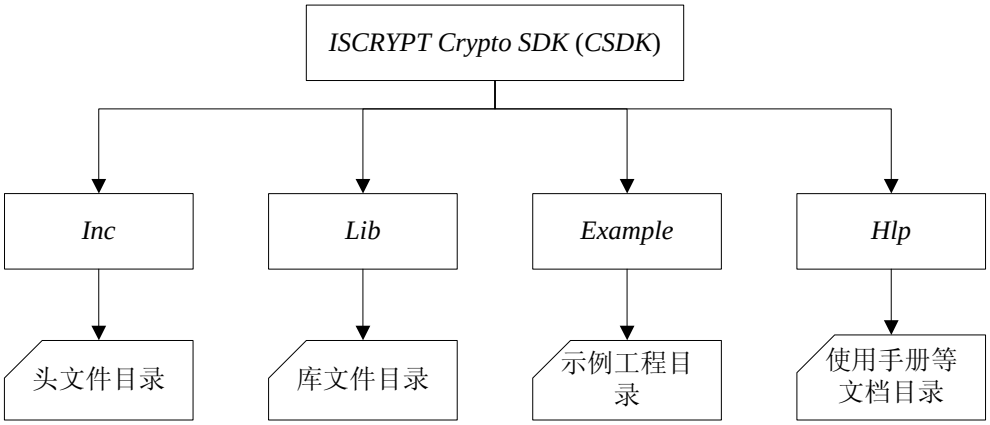


图 1　CSDK 目录组织结构图

..\Inc 中包含了各算法相应的头文件，提供相应的密码算法函数接口的声明。

..\Lib 中提供了编译后的各密码算法的函数库文件，可根据相应的文件名确认对应的密码算法。

..\Hlp 中提供用户使用手册。

..\Example 中提供有各密码算法函数库的使用参考示例，同时提供有至少一组的标准参考数据，可供 CSDK 的使用者调试或测试使用。

具体 Example 中提供的示例包含如下内容：

..\Example\iscrypt_des_tdes_lib_test

..\Example\iscrypt_hash_bkpt_rst_lib_test

..\Example\iscrypt_random_lib_test

..\Example\iscrypt_rsa_lib_test

..\Example\iscrypt_ecc_lib_test

..\Example\iscrypt_sha_lib_test

..\Example\iscrypt_sm1_lib_test

..\Example\iscrypt_sm2_lib_test

..\Example\iscrypt_sm3_lib_test

..\Example\iscrypt_sm4_lib_test

CSDK 支持的密码算法功能图如下：

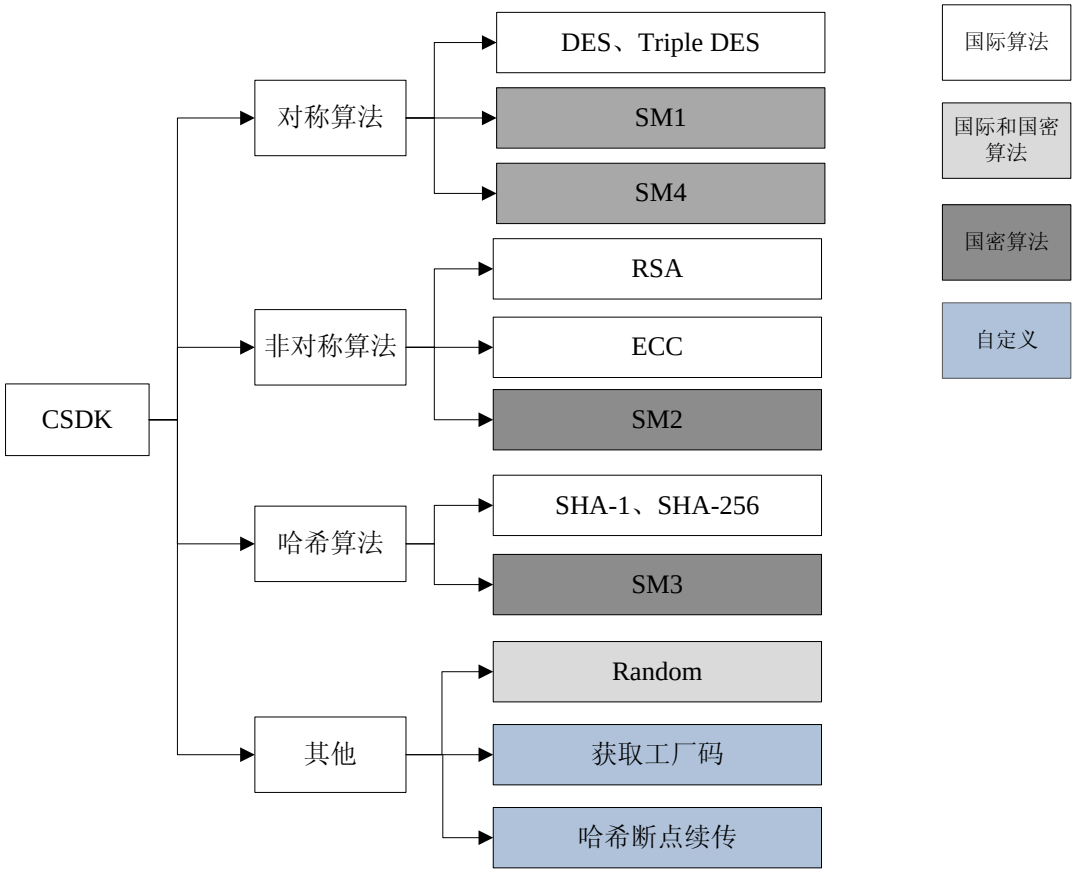


图 2 CSDK 支持的算法功能结构图

1.3 文件说明

CSDK 中提供的算法库文件和头文件，及所支持的功能函数对应关系如下表 1 所示。除 iscrypt.h 头文件为必须包含以外，其他算法头文件配合相应的算法库文件使用，其中 iscrypt_symmetric.h 为使用对称算法时必须包含的头文件，上述两个头文件不在下表中赘述。

表 1 CSDK 文件与功能函数表

库文件	头文件	函数	支持功能
iscrypt_des_tdes.lib	iscrypt_des_tdes.h	DESCrypt TDESCrypt	DES、TDES 加解密；
iscrypt_rsa_kir.lib[1] iscrypt_rsa_gen_key_kir.lib iscrypt_rsa_kif.lib iscrypt_rsa_gen_key_kif.lib	iscrypt_rsa.h	RSAGenKeyPair RSAEncrypt RSADecrypt RSASign RSAVerify	RSA 密钥生成、加解密、签名验证；
iscrypt_ecc.lib	iscrypt_ecc.h	ECC_Init ECC_PointMult ECC_ECDSA_Sign ECC_ECDSA_Verify ECC_ECES_Encrypt ECC_ECES_Decrypt ECC_ECDH	通用 ECC 算法的签名验证和加解密；
iscrypt_sha.lib	iscrypt_sha.h	SHA1Init SHA1Update SHA1Final SHA256Init SHA256Update SHA256Final	SHA-1、SHA-256 哈希算法；

[1] 根据芯片特性，选择使用不同的 RSA 算法库。其中，iscrypt_rsa_xxx_kir.lib 只适用于 IS8U256A 型的芯片；iscrypt_rsa_xxx_kif.lib 只适用于 IS8U256B 型的芯片。

iscrypt_sm1.lib	iscrypt_sm1.h	SM1Crypt	SM1 加解密；
iscrypt_sm2.lib	iscrypt_sm2.h	SM2GenKeyPair SM2Encrypt SM2Decrypt SM2Sign SM2Verify SM2KeyExchange SM2GetZ	SM2 密钥生成、加解密、签名验证、密钥协商、中间参数 Z 值计算；
iscrypt_sm2_ext.lib	iscrypt_sm2_ext.h	SM2EncryptInit SM2EncryptUpdate SM2EncryptFinal SM2DecryptInit SM2DecryptUpdate SM2DecryptFinal	扩展的 SM2 加解密功能；
iscrypt_sm3.lib	iscrypt_sm3.h	SM3Init SM3Update SM3Final	SM3 哈希算法；
iscrypt_sm4.lib	iscrypt_sm4.h	SM4Crypt	SM1 加解密；
iscrypt_random.lib	iscrypt_random.h	GenRandom GetFactoryCode	随机数生成； 获取工厂码；
iscrypt_hash_bkpt_rst.lib	iscrypt_hash_bkpt_rst.h	HASHSetBkpt HASHRestore	哈希算法的断点续传；

第 2 章 CSDK 使用介绍

2.1 CSDK 的使用

建立工程后，根据使用需要添加 CSDK 中 Lib 目录下的算法库文件，并同时包含 Inc 目录下对应的头文件，编译工程即可。

以 CSDK 中提供的 DES 和 RSA 为例，首先添加 CSDK 中的 iscrypt_des_tdes.lib 和 RSA 相关的库文件文件，见图 3。

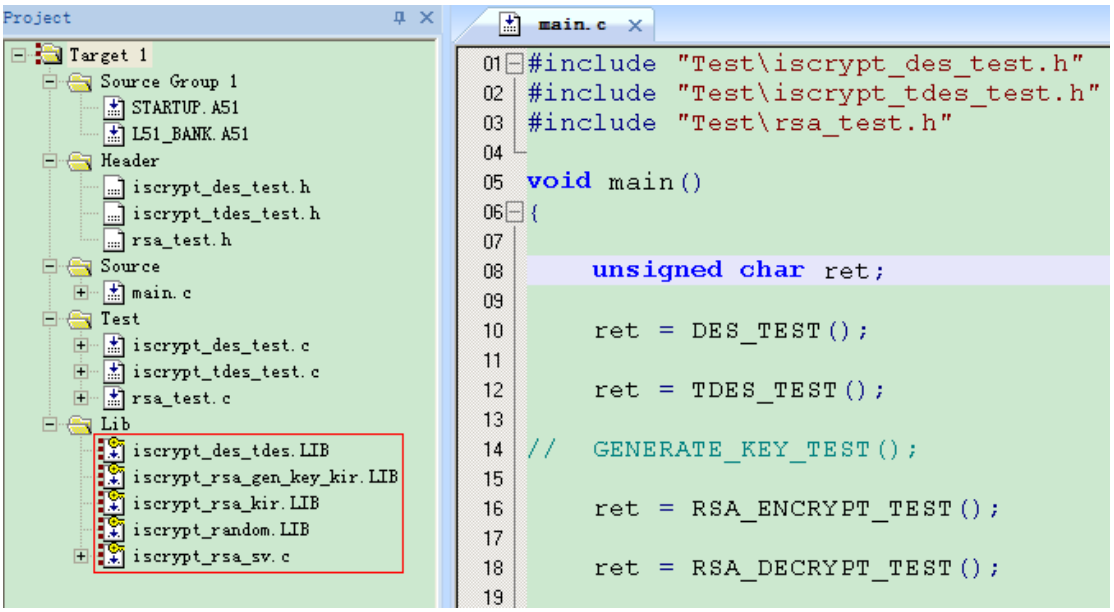


图 3 添加库文件至工程

其次设置头文件路径。一种方法是直接将 CSDK 中 Inc 目录下的相应算法头文件，如 iscrypt_des_tdes.h 复制至当前工程目录下，然后添加至工程中；另一种方法是鼠标右键“Project”栏中的“Target”，在弹出“Option for Target”对话框后，选择“C51”属性页，在“Include Paths”栏中设置头文件路径，见图 4。

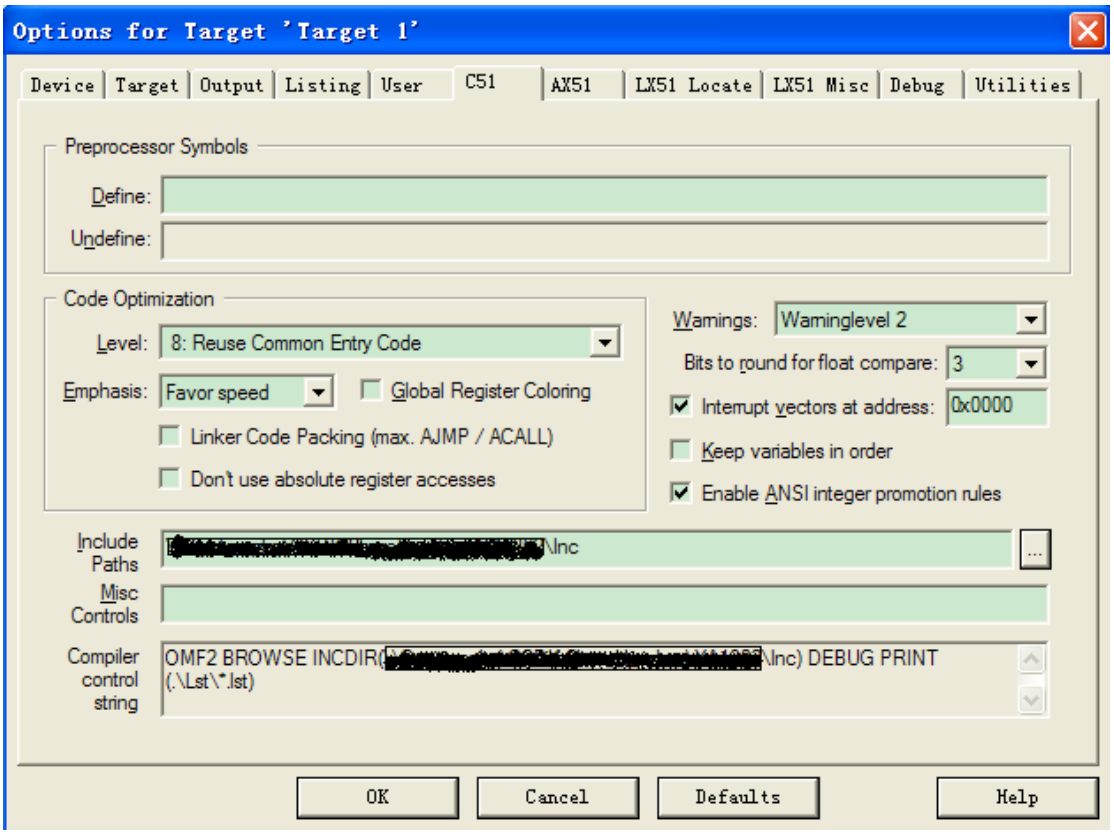


图 4　设置工程中头文件路径

说明：CSDK 中封装的函数均使用以字节为单位的自然顺序。

假设自然顺序时该数据（16 进制）为：

1234567890ABCDEF13579BDF……

则在交付 CSDK 所提供的函数进行运算处理前，只需对自然序的数据进行定义为：

0x12,0x34,0x56,0x78,0x90,0xAB,0xCD,0xEF,0x13,0x57,0x9B,0xDF,0x……

CSDK 中函数库处理后的结果同样为自然序。

2.2　CSDK 的 Multi-Bank 配置

芯片提供 Multi-Bank 应用机制，使用者建立自己的工程后，添加进库文件和头文件，然后添加“L51_BANK.A51”文件，该文件可在 KEIL 的安装目录的\C51\LIB 文件夹下找到，具体操作流程如下图所示。

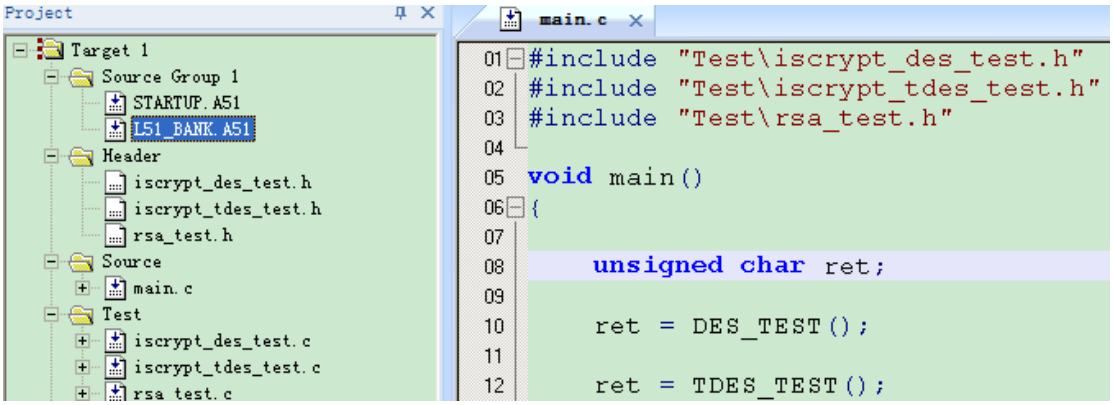


图 5　添加 L51_BANK.A51 文件

同时更改工程中“STARTUP.A51”中相应的配置，只需勾选“Code Banking”即可。

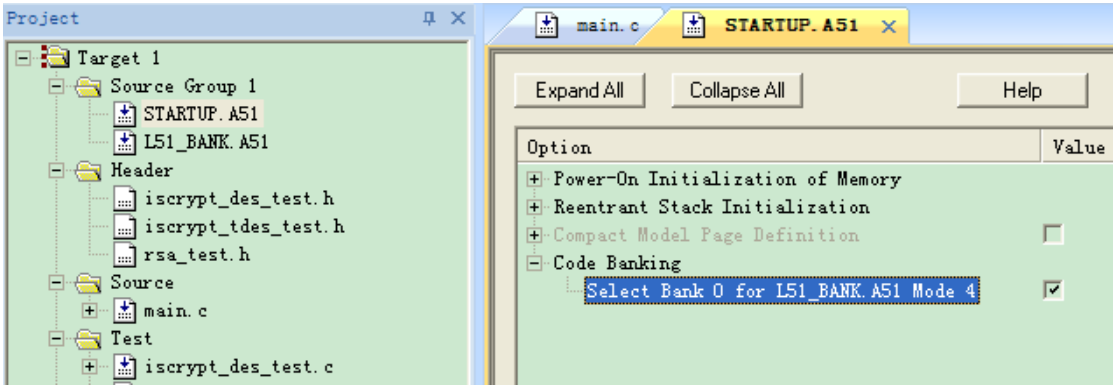


图 6　修改 STARTUP.A51 文件

上述完成后，鼠标右键点击库文件，如“iscrypt_des_tdes.lib”，选择“Options for Files ‘iscrypt_des_tdes.lib’”，弹出 对话框后，进行“Code Bank”选择 ，在“Select Modules to Always Include”中，勾选算法模块，点击“OK”完成操作。

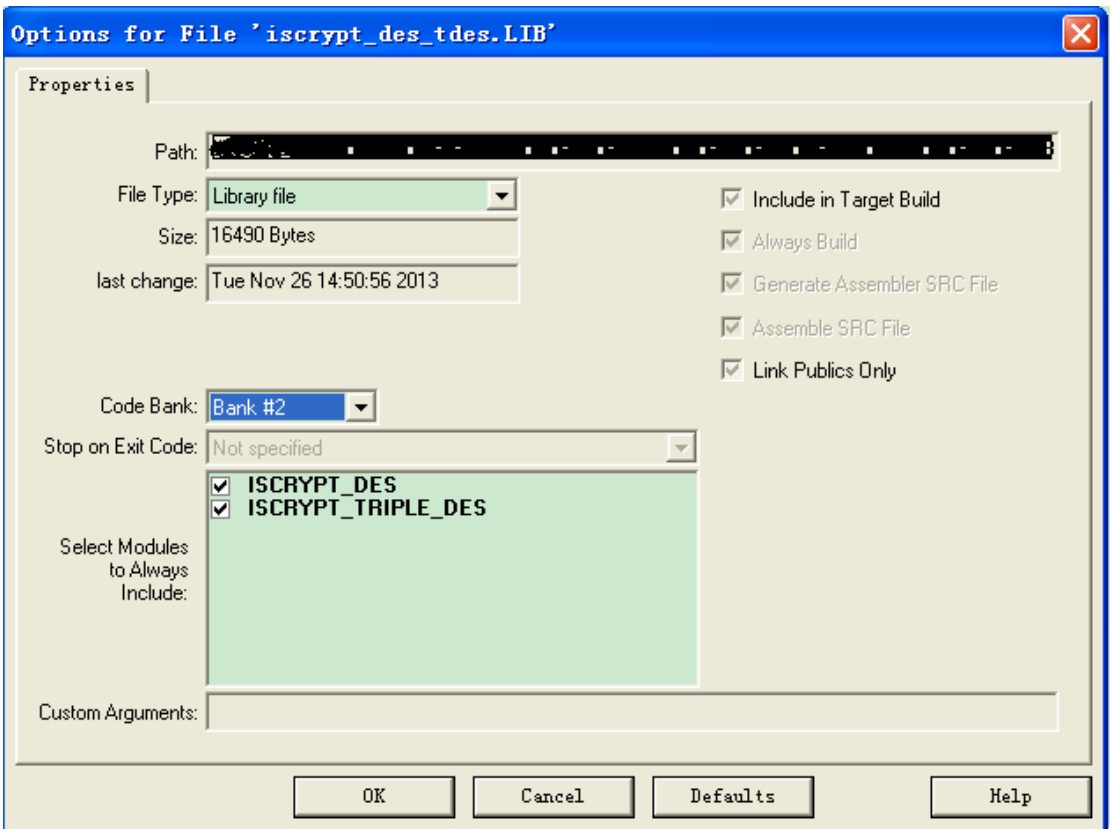


图 7　设置算法库文件存放的 BANK

在“Select Modules to Always Include”中，可根据需要选择算法模块，而不必全部选择。比如图 7 中，当只选择 ISCRYPT_TRIPLE_DES 而不勾选 ISCRYPT_DES 时，未被选用的模块儿，在同时保证相关的函数未被调用的情形下，该模块儿不会被编译，从而减少工程代码量。

注意事项：

1. **对于使用 IS8U256A 型的芯片，由于使用的 RSA 密钥生成的函数库为 iscrypt_rsa_gen_key_pair_kir.lib，Multi-BANK 配置时，必须将该库文件中的算法模块配置在 COMMON BANK，如下图，否则密钥生成函数调用时将返回错误。**

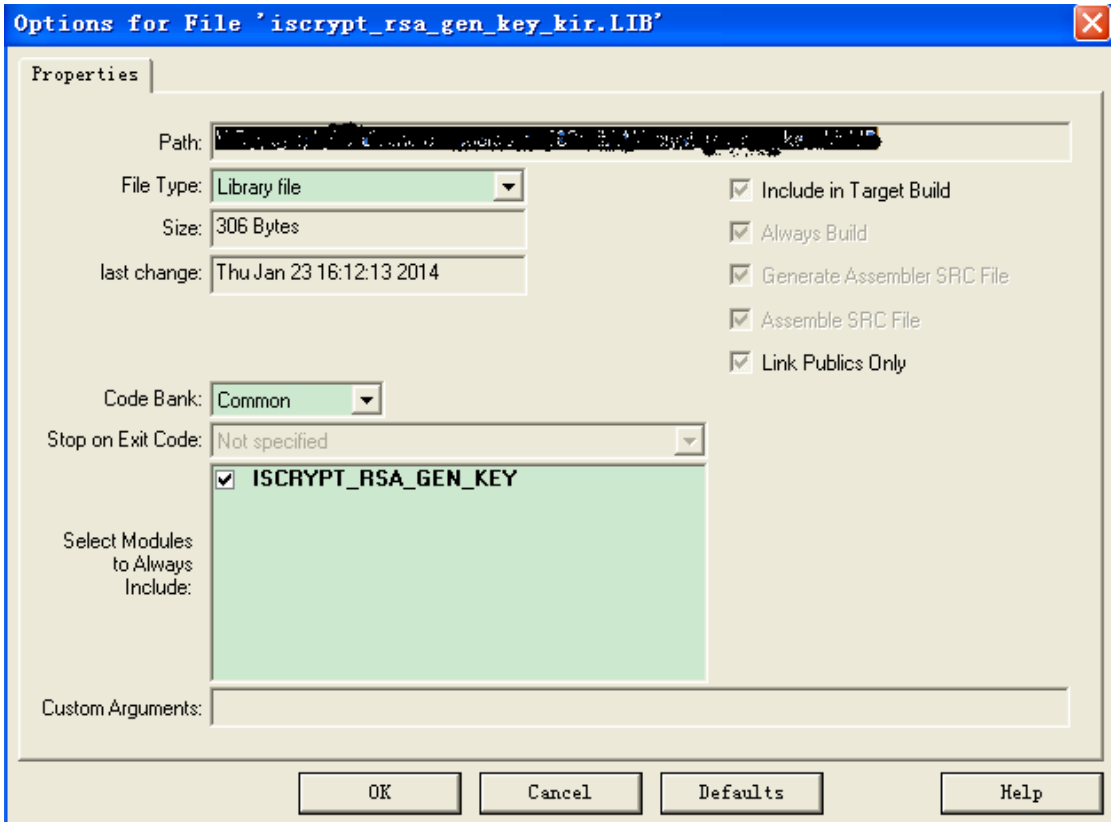


图 8　设置 iscrypt_rsa_gen_key_pair_kir 算法库文件存放的 BANK

2. **IS8U256B 型的芯片使用 iscrypt_rsa_gen_key_pair_kif.lib，不能使用 iscrypt_rsa_gen_key_pair_kir.lib，且无上述的必须设置为 COMMON BANK 的要求，因此可按照一般库的使用进行设置。**
3. **除上述描述的关于 RSA 生成密钥的函数库外，其他函数库使用时不应放在 COMMON BANK。**

第 3 章 CSDK 的函数接口说明

3.1 DES

3.1.1 DESCrypt

DESCrypt 函数用于提供基于 DES 算法的 ECB 和 CBC 模式的加解密功能操作。

```
#include “iscrypt_des_tdes.h”
unsigned char DESCrypt (
    BLOCKCIPHERPARAM xdata *pBlockCipherParam        //    in,out
);
```

参数

pBlockCipherParam

指向分组密码算法结构体 **BLOCKCIPHERPARAM**，用以传送DES密钥、操作模式、输入输出等数据信息。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，错误代码列举如下。

Error Code	Description
NOT_SUPPORT_YET_ERROR	暂不支持
MEMORY_ERROR	内存操作错误

注意

1. DES 为以 DES_BLOCK_SIZE（定义在头文件中）个字节为一分组。
2. 当前算法仅支持 ECB 和 CBC 模式，其他模式不支持。

快速信息

头文件：声明在 iscrypt_des_tdes.h。

库文件：使用 iscrypt_des_tdes.lib。

示例

```
#include <string.h>
#include "iscrypt_des_tdes.h"

extern unsigned char xdata space_for_test[1024];
extern BLOCKCIPHERPARAM xdata BlockCipherParam;

char DES_TEST()
{
    unsigned char ret;
    memcpy(space_for_test, DES_KEY, DES_BLOCK_SIZE);
    memcpy(space_for_test+DES_BLOCK_SIZE, DES_IV, DES_BLOCK_SIZE);
    ////////////////////////////////////////////////////////////////
    memcpy(space_for_test+2*DES_BLOCK_SIZE,DES_PLAINTEXT,5*DES_BLOCK_
SIZE);

    BlockCipherParam.bOpMode = SYM_ECB | SYM_ENCRYPT;
    BlockCipherParam.pbKey = space_for_test;
    BlockCipherParam.pbIV = space_for_test+DES_BLOCK_SIZE;

    BlockCipherParam.pbInput = space_for_test+2*DES_BLOCK_SIZE;
    BlockCipherParam.bTotalBlock = 5;
    BlockCipherParam.pbOutput = space_for_test+7*DES_BLOCK_SIZE;

    ret = DESCrypt(&BlockCipherParam);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }

    ret=memcmp(space_for_test+7*DES_BLOCK_SIZE,DES_CIPHERTEXT_ECB,5*DE
S_BLOCK_SIZE);
```

```
    if (0 != ret)
    {
        goto ErrorExit;
    }

    return 0;
ErrorExit:
    return 1;
}
```

参阅

BLOCKCIPHERPARAM、TDESCrypt

3.1.2 TDESCrypt

TDESCrypt 函数用于提供基于 TDES 算法的 ECB 和 CBC 模式的加解密功能操作。

```
#include “iscrypt_des_tdes.h”
unsigned char TDESCrypt (
    BLOCKCIPHERPARAM xdata *pBlockCipherParam          //in,out
);
```

参数

pBlockCipherParam

指向分组密码算法结构体 **BLOCKCIPHERPARAM**，用以传送TDES密钥、操作模式、输入输出等数据信息。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，错误代码列举如下。

Error Code	**Description**
NOT_SUPPORT_YET_ERROR	暂不支持
MEMORY_ERROR	内存操作错误

注意

1. TDES 为以 TDES_BLOCK_SIZE（定义在头文件中）个字节为一分组。
2. 当前算法仅支持 ECB 和 CBC 模式，其他模式不支持。

快速信息

头文件：声明在 iscrypt_des_tdes.h。

库文件：使用 iscrypt_des_tdes.lib。

示例

```
#include <string.h>
```

```
#include "iscrypt_des_tdes.h"

extern unsigned char xdata space_for_test[1024];
extern BLOCKCIPHERPARAM xdata BlockCipherParam;

char TDES_TEST()
{
    unsigned char ret;
    memcpy(space_for_test, TDES_KEY, 2*TDES_BLOCK_SIZE);
    memcpy(space_for_test+2*TDES_BLOCK_SIZE, TDES_IV, TDES_BLOCK_SIZE);
    ////////////////////////////////////////////////////////////////
    memcpy(space_for_test+3*TDES_BLOCK_SIZE,TDES_PLAINTEXT,5*TDES_BLO
CK_SIZE);

    BlockCipherParam.bOpMode = SYM_ECB | SYM_ENCRYPT;
    BlockCipherParam.pbKey = space_for_test;
    BlockCipherParam.pbIV = space_for_test+2*TDES_BLOCK_SIZE;

    BlockCipherParam.pbInput = space_for_test+3*TDES_BLOCK_SIZE;
    BlockCipherParam.bTotalBlock = 5;
    BlockCipherParam.pbOutput = space_for_test+8*TDES_BLOCK_SIZE;

    ret = TDESCrypt(&BlockCipherParam);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }

    Ret = memcmp(space_for_test+8*TDES_BLOCK_SIZE, TDES_CIPHERTEXT_ECB,
5*TDES_BLOCK_SIZE);
    if (0 != ret)
    {
```

```
            goto ErrorExit;
        }

        return 0;
    ErrorExit:
        return 1;
    }
```

参阅

BLOCKCIPHERPARAM、DESCrypt

3.2 RSA

3.2.1 RSAGenKeyPair

RSAGenKeyPair 函数用于生成设定位长的 RSA 的公私钥，其中 RSA 的公钥指数需事先指定。

```
#include “iscrypt_rsa.h”
unsigned char RSAGenKeyPair(
    unsigned char bMode,                          // in
    RSAPUBLICKEYCTX xdata *pRSAPubKey,            // in, out
    RSAPRIVATEKEYCTX xdata *pRSAPriKey            // out
);
```

参数

bMode

RSA 操作属性，当前支持方式及位标志图如下：

Bit7~Bit4 (*index*)	Bit3 (*type-flag*)	Bit2~Bit1	Bit0 (*bit-flag*)
位长索引： 值为 0 时，为选择 1024 或 2048 位长的 RSA，此时通过 Bit0 决定。 值为 1~15 时，依次对应 1024+64*i* 位长的 RSA。	类型标志	Reserve	位长标志

Value	**Description**
RSA1024	1024 位长的 RSA
RSA2048	2048 位长的 RSA
STD_TYPE	标准的 RSA 密钥对儿结构

	((*e*,*n*), (*d*,*n*))
CRT_TYPE	使用中国剩余定理的密钥对儿结构((*e*,*n*), (*p*,*q*,*dP*,*dQ*,*qInv*))

当使用 1024 和 2048 之间的任意位长的 RSA 时，*bMode* 中的索引值需要按照以下方式计算：

设选取的位长为 *nBits*，*nBits* = 64*k* 且 1024<*nBits*<2048。

bMode.index = (*nBits*-1024)/64

当使用字节表示时，设字节长为 *nBytes*，*nBytes* = 8*k* 且 128<*nBytes*<256。

bMode.index = (*nBytes*-128)/8

pRSAPubKey

指向RSA公钥结构体 **RSAPUBLICKEYCTX**，公钥指数由函数调用者传入，位长不大于 32 位。根据标准，通常推荐的公钥指数选取 65537。

pRSAPriKey

指向RSA私钥结构体 **RSAPRIVATEKEYCTX**，根据函数调用者对当前RSA的操作属性的选择生成。RSA的私钥可以有两种形式，(*d*,*n*)和(*p*,*q*,*dP*,*dQ*,*qInv*)，后者对RSA的私钥运算有提升性能的作用。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，错误代码列举如下。

Error Code	**Description**
MEMORY_ERROR[2]	函数存放地址错误

注意

1. 在 RAM 紧张的应用情况下，RSA 生成密钥对儿的操作属性选择 STD_TYPE,公钥结构体中的模数 *n* 可与私钥结构体中的模数 *n* 共享同一片存储区。

2. **函数 RSAGenKeyPair 调用时，除使用 1KB 的 PKU RAM 外，还会**

[2] 该错误只针对使用 IS8U256A 型的芯片，此时使用的是 iscrypt_rsa_gen_key_pair_kir.lib。

独占内部申请的 IS_BUFFER_RSA[1284] 作为中间处理数据使用。IS_BUFFER_RSA[1284]仅在 RSAGenKeyPair 运行时使用，其后可复用为其它用途。

3. **IS_BUFFER_RSA[1284]绝对地址定位于 0x0000 处。**

快速信息

头文件：声明在 iscrypt_rsa.h。

库文件：使用iscrypt_rsa_gen_key_pair_xxx.lib[3]。

示例

```
#include “iscrypt_rsa.h”
RSAPUBLICKEYCTX xdata RSAPubKey;
RSAPRIVATEKEYSTD xdata RSAPriKeySTD;
RSAPRIVATEKEYCRT xdata RSAPriKeyCRT;

unsigned char xdata space_for_test[1024];

void GENERATE_KEY_TEST()
{
    unsigned char xdata len;

    space_for_test[0] = 0x00;
    space_for_test[1] = 0x01;
    space_for_test[2] = 0x00;
    space_for_test[3] = 0x01;

    RSAPubKey.e = &space_for_test[0];
    RSAPubKey.n = &space_for_test[4];
```

[3]如使用的为 IS8U256A 型芯片则选用 iscrypt_rsa_gen_key_pair_kir.lib；使用的为 IS8U256B 型则选用 iscrypt_rsa_gen_key_pair_kif.lib

```
        RSAPriKeySTD.d = &space_for_test[4+256];
        RSAPriKeySTD.n = &space_for_test[4];

        RSAPriKeyCRT.p = &space_for_test[4+256];
        RSAPriKeyCRT.q = &space_for_test[260+128];
        RSAPriKeyCRT.dP = &space_for_test[388+128];
        RSAPriKeyCRT.dQ = &space_for_test[516+128];
        RSAPriKeyCRT.qInv = &space_for_test[644+128];

        /* RSA-1024 */
        RSAGenKeyPair(RSA1024 | STD_TYPE, &RSAPubKey, (RSAPRIVATEKEYCTX
*)&RSAPriKeySTD);
    // RSAGenKeyPair(RSA1024 | CRT_TYPE, &RSAPubKey, (RSAPRIVATEKEYCTX
*)&RSAPriKeyCRT);

        /* RSA-1984 */
        len = 248;
        len -= 128;
        len /= 8;
        len <<= 4;
    // RSAGenKeyPair(len | STD_TYPE, &RSAPubKey, (RSAPRIVATEKEYCTX
*)&RSAPriKeySTD);
    RSAGenKeyPair(len | CRT_TYPE, &RSAPubKey, (RSAPRIVATEKEYCTX
*)&RSAPriKeyCRT);

        return ;
    }
```

参阅

RSAPUBLICKEYCTX、RSAPRIVATEKEYCTX、RSAEncrypt、RSADecrypt、RSASign、RSAVerify

3.2.2 RSAEncrypt

RSAEncrypt 函数使用当前参数中的公钥，对指定的消息数据进行加密运算并输出运算结果至用户指定区域。

```
#include “iscrypt_rsa.h”
unsigned char RSAEncrypt(
    unsigned char bMode,                          // in
    RSAPUBLICKEYCTX xdata *pRSAPubKey,            // in
    unsigned char xdata *pbPlain                  // in, out
);
```

参数

bmode

RSA 操作属性，当前支持方式及位标志图如下：

Bit7~Bit4 (*index*)	Bit3 (*type-flag*)	Bit2~Bit1	Bit0 (*bit-flag*)
位长索引： 值为 0 时，为选择 1024 或 2048 位长的 RSA，此时通过 Bit0 决定。 值为 1~15 时，依次对应 1024+64*i* 位长的 RSA。	类型标志	Reserve	位长标志

Value	**Description**
RSA1024	1024 位长的 RSA
RSA2048	2048 位长的 RSA

当使用 1024 和 2048 之间的任意位长的 RSA 时，*bMode* 中的索引值需要按照以下方式计算：

设选取的位长为 *nBits*，*nBits* = 64*k* 且 1024<*nBits*<2048。

bMode.index = (*nBits*-1024)/64

当使用字节表示时，设字节长为 *nBytes*，*nBytes* = 8*k* 且 128<*nBytes*<256。

bMode.index = (*nBytes*-128)/8

pRSAPubKey

指向RSA公钥结构体 **RSAPUBLICKEYCTX**的指针。

pbPlain

RSA 明文，处理后的密文同时存储在此区域中。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，返回值为 MESSAGE_OUT_OF_RANGE 。

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_rsa.h。

库文件：使用 iscrypt_rsa_kif.lib 或 iscrypt_rsa_kir.lib。

示例

```
#include <string.h>
#include "iscrypt_rsa.h"

unsigned char xdata space_for_test[1024];

unsigned char RSA_ENCRYPT_TEST()
{
    unsigned char xdata ret;
```

```
    space_for_test[0] = 0x00;
    space_for_test[1] = 0x01;
    space_for_test[2] = 0x00;
    space_for_test[3] = 0x01;

    /* RSA-1024 */
    memcpy(space_for_test+4, RSA1024_PUB_KEY_N, 128);
    memcpy(space_for_test+512, RSA_PLAINTEXT, 128);

    RSAPubKey.e = &space_for_test[0];
    RSAPubKey.n = &space_for_test[4];

    ret = (unsigned char)RSAEncrypt(RSA1024, &RSAPubKey, space_for_test+512);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }

    return 0;

ErrorExit:
    return 1;
}
```

参阅

RSAPUBLICKEYCTX、RSADecrypt

3.2.3 RSADecrypt

RSADecrypt 函数使用当前参数中的私钥，对指定的密文数据进行解密运算并输出运算结果至用户指定区域。

```
#include “iscrypt_rsa.h”
unsigned char RSADecrypt(
    unsigned char bMode,                          // in
    RSAPRIVATEKEYCTX xdata *pRSAPriKey,           // in
    unsigned char xdata *pbCipher                 // in, out
);
```

参数

bmode

RSA 操作属性，当前支持方式及位标志图如下：

Bit7~Bit4 (*index*)	Bit3 (*type-flag*)	Bit2~Bit1	Bit0 (*bit-flag*)
位长索引： 值为 0 时，为选择 1024 或 2048 位长的 RSA，此时通过 Bit0 决定。 值为 1~15 时，依次对应 1024+64*i* 位长的 RSA。	类型标志	Reserve	位长标志

Value	**Description**
RSA1024	1024 位长的 RSA
RSA2048	2048 位长的 RSA
STD_TYPE	标准的 RSA 密钥对儿结构 ((*e*,*n*), (*d*,*n*))
CRT_TYPE	使用中国剩余定理的密钥对儿

结构((*e*,*n*), (*p*,*q*,*dP*,*dQ*,*qInv*))

当使用 1024 和 2048 之间的任意位长的 RSA 时，*bMode* 中的索引值需要按照以下方式计算：

设选取的位长为 *nBits*，*nBits* = 64*k* 且 1024<*nBits*<2048。

bMode.index = (*nBits*-1024)/64

当使用字节表示时，设字节长为 *nBytes*，*nBytes* = 8*k* 且 128<*nBytes*<256。

bMode.index = (*nBytes*-128)/8

pRSAPriKey

指向RSA私钥结构体 **RSAPRIVATEKEYCTX**的指针。

pbCipher

RSA 密文，处理后的明文同时存储在此区域中。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，返回值为 MESSAGE_OUT_OF_RANGE 。

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_rsa.h。

库文件：使用 iscrypt_rsa_kif.lib 或 iscrypt_rsa_kir.lib。

示例

```
#include <string.h>
#include "iscrypt_rsa.h"

unsigned char xdata space_for_test[1024];
RSAPRIVATEKEYSTD xdata RSAPriKeySTD;
```

```
unsigned char RSA_DECRYPT_TEST()
{
    unsigned char xdata ret;

    ////////////////////////////////////////////////////
    //RSA_1024_STD
    memcpy(space_for_test, RSA1024_PRI_KEY_D, 128);
    memcpy(space_for_test+256, RSA1024_PUB_KEY_N, 128);
    memcpy(space_for_test+512, RSA1024_CIPHERTEXT, 128);

    RSAPriKeySTD.d = space_for_test;
    RSAPriKeySTD.n = space_for_test+256;

   ret=RSADecrypt(RSA1024|STD_TYPE, (RSAPRIVATEKEYCTX *)&RSAPriKeySTD,
space_for_test+512);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }

    return 0;

ErrorExit:
    return 1;
}
```

参阅

RSAPRIVATEKEYCTX、RSAEncrypt

3.2.4 RSASign

RSASign 函数使用当前参数中的私钥，对输入消息进行签名运算，并生成签名结果存放在指定区域。

```
#include “iscrypt_rsa.h”
unsigned char RSASign(
    unsigned char bMode,                          // in
    RSAPRIVATEKEYCTX xdata *pRSAPriKey,           // in
    unsigned char xdata * pbDigest                // in, out
);
```

参数

bmode

RSA 操作属性，当前支持方式及位标志图如下：

Bit7~Bit4 (*index*)	Bit3 (*type-flag*)	Bit2~Bit1	Bit0 (*bit-flag*)
位长索引： 值为 0 时，为选择 1024 或 2048 位长的 RSA，此时通过 Bit0 决定。 值为 1~15 时，依次对应 1024+64*i* 位长的 RSA。	类型标志	Reserve	位长标志

Value	**Description**
RSA1024	1024 位长的 RSA
RSA2048	2048 位长的 RSA
STD_TYPE	标准的 RSA 密钥结构 ((*e*,*n*), (*d*,*n*))
CRT_TYPE	使用中国剩余定理的密钥结构

((*e*,*n*), (*p*,*q*,*dP*,*dQ*,*qInv*))

当使用 1024 和 2048 之间的任意位长的 RSA 时，*bMode* 中的索引值需要按照以下方式计算：

设选取的位长为 *nBits*，*nBits* = 64*k* 且 1024<*nBits*<2048。

bMode.index = (*nBits*-1024)/64

当使用字节表示时，设字节长为 *nBytes*，*nBytes* = 8*k* 且 128<*nBytes*<256。

bMode.index = (*nBytes*-128)/8

pRSAPriKey

指向RSA私钥结构体 **RSAPRIVATEKEYCTX**的指针。

pbDigest

待签名的消息摘要。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，返回值为 MESSAGE_OUT_OF_RANGE 。

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**
2. 该函数实现在 iscrypt_rsa_sv.c 中，使用时需要连同 iscrypt_rsa_kif.lib 或 iscrypt_rsa_kir.lib 一同加入用户工程当中。

快速信息

头文件：声明在 iscrypt_rsa.h。

库文件：使用 iscrypt_rsa_kif.lib 或 iscrypt_rsa_kir.lib。

源文件：使用 iscrypt_rsa_sv.c

示例

```
#include <string.h>
#include "iscrypt_rsa.h"
```

```
unsigned char xdata space_for_test[1500];

RSAPRIVATEKEYSTD xdata RSAPriKeySTD;
RSAPRIVATEKEYCRT xdata RSAPriKeyCRT;

unsigned char RSA_SIGN_TEST()
{
    unsigned char xdata ret;
    unsigned char xdata len;

    memcpy(space_for_test, RSA1984_PRI_KEY_D, 248);
    memcpy(space_for_test+256, RSA1984_PUB_KEY_N, 248);
    memcpy(space_for_test+512, RSA1984_CIPHERTEXT, 248);

    RSAPriKeySTD.d = space_for_test;
    RSAPriKeySTD.n = space_for_test+256;

    len = 248;
    len -= 128;
    len /= 8;
    len <<= 4;
    ret = RSASign(len|STD_TYPE, (RSAPRIVATEKEYCTX *)&RSAPriKeySTD,
space_for_test+512);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }
return 0;

ErrorExit:
    return 1;
```

}

参阅

RSAPRIVATEKEYCTX、RSAVerify

3.2.5 RSAVerify

RSAVerify 函数使用当前参数中的公钥，对输入消息和消息签名值进行验证。

```
#include “iscrypt_rsa.h”
unsigned char RSAVerify(
    unsigned char bMode,                          // in
    RSAPUBLICKEYCTX xdata *pRSAPubKey,            // in
    unsigned char xdata *pbSignature,             // in
    unsigned char *pbDigest                       // in
)
```

参数

bmode

RSA 操作属性，当前支持方式及位标志图如下：

Bit7~Bit4 (*index*)	Bit3 (*type-flag*)	Bit2~Bit1	Bit0 (*bit-flag*)
位长索引： 值为 0 时，为选择 1024 或 2048 位长的 RSA，此时通过 Bit0 决定。 值为 1~15 时，依次对应 1024+64*i* 位长的 RSA。	类型标志	Reserve	位长标志

Value	**Description**
RSA1024	1024 位长的 RSA
RSA2048	2048 位长的 RSA

当使用 1024 和 2048 之间的任意位长的 RSA 时，*bMode* 中的索引值需要按照以下方式计算：

设选取的位长为 *nBits*，*nBits* = 64*k* 且 1024<*nBits*<2048。

bMode.index = (*nBits*-1024)/64

当使用字节表示时，设字节长为 *nBytes*，*nBytes* = 8*k* 且 128<*nBytes*<256。

bMode.index = (*nBytes*-128)/8

pRSAPubKey

RSA公钥结构体 **RSAPUBLICKEYCTX**指针。

pbSignature

摘要的签名值，验证后的数据同时存储在此区域中。

pbDigest

原始摘要。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，错误代码列举如下。

Error Code	**Description**
MESSAGE_OUT_OF_RANGE	处理的消息数据超出范围
FAIL	验证失败

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**
2. 该函数实现在 iscrypt_rsa_sv.c 中，使用时需要连同 iscrypt_rsa_kif.lib 或 iscrypt_rsa_kir.lib 一同加入用户工程当中。

快速信息

头文件：声明在 iscrypt_rsa.h。

库文件：使用 iscrypt_rsa_kif.lib 或 iscrypt_rsa_kir.lib。

源文件：使用 iscrypt_rsa_sv.c

示例

```
#include <string.h>
```

```
#include "iscrypt_rsa.h"

BYTE xdata space_for_test[1500];
RSAPUBLICKEYCTX xdata RSAPubKey;

cunsigned char RSA_VERIFY_TEST()
{
    unsigned char xdata ret;
    unsigned char xdata len;

    space_for_test[0] = 0x00;
    space_for_test[1] = 0x01;
    space_for_test[2] = 0x00;
    space_for_test[3] = 0x01;

    memcpy(space_for_test+4, RSA1408_PUB_KEY_N, 176);
    memcpy(space_for_test+512, RSA_PLAINTEXT, 176);

    RSAPubKey.e = &space_for_test[0];
    RSAPubKey.n = &space_for_test[4];

    len = 176;
    len -= 128;
    len /= 8;
    len <<= 4;
    ret = RSAVerify(len, &RSAPubKey, space_for_test+512, RSA1408_CIPHERTEXT);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }

    return 0;
```

```
ErrorExit:
    return 1;
}
```

参阅

RSAPUBLICKEYCTX、RSASign

3.3 SHA

3.3.1 SHA1Init

SHA1Init 函数用于完成基于 SHA-1 算法的初始化操作。

#include “iscrypt_sha.h”

void SHA1Init();

参数

无

返回值

无

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sha.h。

库文件：使用 iscrypt_sha.lib。

示例

请参见 **SHA1Final**一节。

参阅

SHA1Update、SHA1Final

3.3.2 SHA1Update

SHA1Update 用以对一个 SHA-1 的消息分组中间分组进行处理。

```
#include “iscrypt_sha.h”
void SHA1Update(
    unsigned char xdata *pbMessage      //    in
);
```

参数

pbMessage

需要被哈希处理的消息分组，对于 SHA-1 算法来说为 64 字节为一组。

返回值

无

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sha.h。

库文件：使用 iscrypt_sha.lib。

示例

请参见 **SHA1Final**一节。

参阅

SHA1Init、SHA1Final

3.3.3 SHA1Final

SHA1Final 用以对 SHA-1 的最后消息分组进行处理。

```
#include “iscrypt_sha.h”
unsigned char SHA1Final (
    unsigned char xdata *pbMessage      //  in
    unsigned char bLength,              //  in
    unsigned char xdata *pbDigest       //  out
);
```

参数

pbMessage

需要被哈希处理的末块儿消息分组。

bLength

末块儿消息分组的字节长度，小于 64，可以为 0。

pbDigest

SHA-1 处理结束后，摘要的存储区地址。SHA-1 摘要长度为 20 字节（160 比特）。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，错误代码列举如下。

Error Code	Description
HASH_OBJECT_ERROR	哈希对象错误

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sha.h。

库文件：使用 iscrypt_sha.lib。

示例

```
#include <string.h>
#include "iscrypt_sha.h"

unsigned char xdata space_for_test[1024];
char SHA1_TEST()
{
    unsigned char xdata ret;
    memcpy(space_for_test, SHA_VECTOR, 64);

    ////////////////////////////////////////////////////////////////
    SHA1Init();
    SHA1Update(space_for_test);
    SHA1Update(space_for_test);
    ret = SHA1Final(space_for_test, 0, space_for_test+0x200);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }

    ret = memcmp(space_for_test+0x200, DIGEST_SHA1_128, 20);
    if (0 != ret)
    {
        goto ErrorExit;
    }

    return 0;
ErrorExit:
    return 1;
```

}

参阅

SHA1Init、SHA1Update

3.3.4 SHA256Init

SHA256Init 函数用于完成基于 SHA-256 算法的初始化操作。

#include “iscrypt_sha.h”

void SHA256Init();

参数

无

返回值

无

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sha.h。

库文件：使用 iscrypt_sha.lib。

示例

请参见 **SHA256Final**一节。

参阅

SHA256Update、SHA256Final

3.3.5 SHA256Update

SHA256Update 用以对一个 SHA-256 的消息分组中间块儿进行处理。

```
#include “iscrypt_sha.h”
void SHA256Update(
    unsigned char xdata *pbMessage      //    in
);
```

参数

pbMessage

需要被哈希处理的消息分组，对于 SHA-256 算法来说为 64 字节为一组。

返回值

无

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sha.h。

库文件：使用 iscrypt_sha.lib。

示例

请参见 **SHA256Final**一节。

参阅

SHA256Init、SHA256Final

3.3.6 SHA256Final

SHA256Final 用以对一个 SHA-256 的末块儿消息分组进行处理。

```
#include “iscrypt_sha.h”
unsigned char SHA256Final (
    unsigned char xdata *pbMessage      //  in
    unsigned char bLength,              //  in
    unsigned char xdata *pbDigest       //  out
);
```

参数

pbMessage

需要被哈希处理的末块儿消息分组。

bLength

末块儿消息分组的字节长度，小于 64，可以为 0。

pbDigest

SHA2561 处理结束后，摘要的存储区地址。SHA-256 摘要长度为 32 字节（256 比特）。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，错误代码列举如下。

Error Code	Description
HASH_OBJECT_ERROR	哈希对象错误

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sha.h。

库文件：使用 iscrypt_sha.lib。

示例

```
#include <string.h>
#include "iscrypt_sha.h"
unsigned char xdata space_for_test[1024];

char SHA256_TEST()
{
    unsigned char xdata ret;
    memcpy(space_for_test, SHA_VECTOR, 64);

    ////////////////////////////////////////////////////////////////
    SHA256Init();
    SHA256Update(space_for_test);
    SHA256Update(space_for_test);
    ret = SHA256Final(space_for_test, 0, space_for_test+0x200);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }

    ret = memcmp(space_for_test+0x200, DIGEST_SHA256_128, 32);
    if (0 != ret)
    {
        goto ErrorExit;
    }

    return 0;
ErrorExit:
    return 1;
```

```
}
```

参阅

SHA256Init、SHA256Update

3.4 SM1

3.4.1 SM1Crypt

SM1Crypt 函数用于提供基于 SM1 算法的 ECB 和 CBC 模式的加解密功能操作。

```
#include “iscrypt_sm1.h”
unsigned char SM1Crypt (
    BLOCKCIPHERPARAM xdata *pBlockCipherParam        //    in,out
);
```

参数

pBlockCipherParam

指向分组密码算法结构体 **BLOCKCIPHERPARAM**，用以传送SM1 密钥、操作模式、输入输出等数据信息。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，错误代码列举如下。

Error Code	Description
NOT_SUPPORT_YET_ERROR	暂不支持
MEMORY_ERROR	内存操作错误

注意

1. SM1 为以 SM1_BLOCK_SIZE（定义在头文件中）个字节为一分组。
2. 当前算法仅支持 ECB 和 CBC 模式，其他模式不支持。
3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm1.h。

库文件：使用 iscrypt_sm1.lib。

示例

```
#include <string.h>
#include "iscrypt_sm1.h"

extern unsigned char xdata space_for_test[1024];
extern BLOCKCIPHERPARAM xdata BlockCipherParam;

char SM1_TEST()
{
    unsigned char ret;

    memcpy(space_for_test, BLOCK_CRYPT_KEY, SM1_BLOCK_SIZE);
    memcpy(space_for_test+SM1_BLOCK_SIZE,BLOCK_CRYPT_IV,SM1_BLOCK_SIZ
E);

    memcpy(space_for_test+2*SM1_BLOCK_SIZE,BLOCK_CRYPT_PLAINTEXT,4*SM
1_BLOCK_SIZE);

    BlockCipherParam.bOpMode = SYM_ECB | SYM_ENCRYPT;
    BlockCipherParam.pbKey = space_for_test;
    BlockCipherParam.pbIV = space_for_test+SM1_BLOCK_SIZE;

    BlockCipherParam.pbInput = space_for_test+2*SM1_BLOCK_SIZE;
    BlockCipherParam.bTotalBlock = 4;
    BlockCipherParam.pbOutput = space_for_test+2*SM1_BLOCK_SIZE;

    ret = SM1Crypt(&BlockCipherParam);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
```

```
        }

        ret=memcmp(space_for_test+2*SM1_BLOCK_SIZE,BLOCK_CRYPT_CIPHERTEXT
_SM1_ECB, 4*SM1_BLOCK_SIZE);

        if (0 != ret)
        {
            goto ErrorExit;
        }

        return 0;
    ErrorExit:
        return 1;
    }
```

参阅

BLOCKCIPHERPARAM、DESCrypt、TDESCrypt

3.5 SM2

3.5.1 SM2GenKeyPair

SM2GenKeyPair 函数用于生成 SM2 算法的公私钥。

```
#include “iscrypt_sm2.h”
unsigned char SM2GenKeyPair(
    SM2PUBLICKEYCTX xdata *pSM2PubKey,      // out
    SM2PRIVATEKEYCTX xdata *pSM2PriKey      // out
);
```

参数

pSM2PubKey

指向SM2 公钥结构体 **SM2PUBLICKEYCTX**的指针。

pSM2PriKey

指向SM2 私钥结构体 **SM2PRIVATEKEYCTX**的指针。

返回值

该函数要求自身必须执行成功，返回 SUCCESS。

注意

1. 函数要求使用者自行开辟存储密钥的 RAM 区域。
2. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm2.h。

库文件：使用 iscrypt_sm2.lib。

示例

```
#include <string.h>
#include "iscrypt_sm2.h"
#include "iscrypt_sm2_test.h"

extern unsigned char xdata space_for_test[1024];

SM2PUBLICKEYCTX xdata PubKey_A;
SM2PRIVATEKEYCTX xdata PriKey_A;

char GENERATE_KEY_TEST()
{
    unsigned char ret;

    ////////////////////////////////////////////////////////////
    PubKey_A.x = space_for_test;
    PubKey_A.y = space_for_test+32;

    PriKey_A.d = space_for_test+64;

    ret = SM2GenKeyPair(&PubKey_A, &PriKey_A);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }

    return 0;

ErrorExit:
    return 1;
}
```

参阅

SM2PUBLICKEYCTX、SM2PRIVATEKEYCTX、 SM2Encrypt、SM2Decrypt、 SM2Sign、SM2Verify

3.5.2 SM2Encrypt

SM2Encrypt 函数使用当前参数中的公钥，对指定的消息数据进行加密运算并输出运算结果至用户指定区域。

```
#include “iscrypt_sm2.h”
unsigned char SM2Encrypt(
    SM2PUBLICKEYCTX xdata *pSM2PubKey,      // in
    SM2PLAINTEXT xdata *pPlaintext,         // in
    SM2CIPHERTEXT xdata *pCiphertext        // out
);
```

参数

pSM2PubKey

指向SM2 公钥结构体 **SM2PUBLICKEYCTX**的指针。

pPlaintext

指向SM2 明文结构体 **SM2PLAINTEXT**的指针，该结构体由明文数据指针和数据长度两部分组成。

pCiphertext

指向SM2 密文的结构体 **SM2CIPHERTEXT**的指针，由四部分组成。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，错误代码列举如下。

Error Code	**Description**
IN_DATA_LENGTH_ERROR	明文数据长度错误

注意

1. 从节省 RAM 资源的角度考虑，密文结构体中的 *pbCipher* 可以与明文结构体中的 *pbData* 共享同一片存储区。这样使用时，当原始数据需要重用

的话，则请注意备份。

2. 当前加密的数据长度限定为 1-32 字节，如果想要加密数据长度更长的数据，请使用 SM2EncryptInit、SM2EncryptUpdate、SM2EncryptFinal函数。

3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm2.h。

库文件：使用 iscrypt_sm2.lib。

示例

```
#include <string.h>
#include "iscrypt_sm2.h"
#include "iscrypt_sm2_test.h"

unsigned char xdata space_for_test[1024];

SM2POINT xdata Point;
SM2PUBLICKEYCTX xdata PubKey_A;
SM2PRIVATEKEYCTX xdata PriKey_A;

SM2PLAINTEXT xdata Plain;
SM2CIPHERTEXT xdata Cipher;

char code SM2Plaintext[] = "encryption standard";

char SM2_ENCRYPT_TEST()
{
    unsigned char ret;

    // prepare key
    memcpy(space_for_test, SM2PublicKey, 64);
    memcpy(space_for_test+64, SM2PrivateKey, 32);
```

```
    PubKey_A.x = space_for_test;
    PubKey_A.y = space_for_test+32;
    PriKey_A.d = space_for_test+64;

    // prepare plaintext
    memcpy(space_for_test+0x100, SM2Plaintext, strlen(SM2Plaintext));
    Plain.pd = space_for_test+0x100;
    Plain.len = strlen(SM2Plaintext);

    // prepare ciphertext
    Point.x = space_for_test+0x200;
    Point.y = space_for_test+0x220;

    Cipher.p = &Point;
    Cipher.c = space_for_test+0x260;
    Cipher.h = space_for_test+0x240;

    ret = SM2Encrypt(&PubKey_A, &Plain, &Cipher);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }

    return 0;

ErrorExit:
    return 1;
}
```

参阅

SM2PUBLICKEYCTX、SM2PLAINTEXT、SM2CIPHERTEXT、SM2

Decrypt、SM2EncryptInit、SM2EncryptUpdate、SM2EncryptFinal

3.5.3 SM2EncryptInit

SM2EncryptInit 函数用于 SM2 算法的加密初始化操作。

```
#include "iscrypt_sm2.h"
#include "iscrypt_sm2_ext.h"
unsigned char SM2EncryptInit(
    SM2PUBLICKEYCTX xdata *pSM2PubKey,      // in
    SM2POINT xdata *pC1                     // out
);
```

参数

pSM2PubKey

指向SM2 公钥结构体 **SM2PUBLICKEYCTX**的指针。

pC1

指向椭圆曲线上点 **SM2POINT**的指针，是SM2 密文中的C1。

返回值

返回 SUCCESS，无其它异常错误。

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm2_ext.h。

库文件：使用 iscrypt_sm2_ext.lib。

示例

略

参阅

SM2EncryptUpdate、SM2EncryptFinal

3.5.4 SM2EncryptUpdate

SM2EncryptUpdate 函数对指定的中间分组数据进行 SM2 的加密运算，并输出运算结果至用户指定区域。

```
#include "iscrypt_sm2.h"
#include "iscrypt_sm2_ext.h"
unsigned char SM2EncryptUpdate(
    unsigned char xdata *pDataPart,      // in
    unsigned char xdata *pC2Part         // out
);
```

参数

pDataPart

SM2 的中间明文分组。

pC2Part

对应中间明文分组的密文。

返回值

函数执行成功，返回值为 SUCCESS；无其它异常返回值。

注意

1. 从节省 RAM 资源的角度考虑，*pC2Part* 可以与 *pDataPart* 共享同一片存储区。这样使用时，当原始数据需要重用的话，则请注意备份。
2. 数据的分组长度以 128 字节为一单位。
3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm2_ext.h。

库文件：使用 iscrypt_sm2_ext.lib。

示例

略

参阅

SM2EncryptInit、SM2EncryptFinal

3.5.5 SM2EncryptFinal

SM2EncryptFinal 函数对指定的末块分组数据进行 SM2 的加密运算，并输出运算结果至用户指定区域。

```
#include "iscrypt_sm2.h"
#include "iscrypt_sm2_ext.h"
unsigned char SM2EncryptFinal(
    BIGINTEGER xdata *pDataLastPart,     // in
    unsigned char xdata *pC2LastPart,    // out
    unsigned char xdata *pC3             // out
);
```

参数

pDataLastPart

指向SM2 末块明文分组的结构体指针 **BIGINTEGER**。

pC2Part

对应末块明文分组的密文。

pC3

SM2 密文中的 C3 部分。

返回值

函数执行成功，返回值为 **SUCCESS**。如果失败，错误代码列举如下。

Error Code	Description
IN_DATA_LENGTH_ERROR	明文数据的长度错误。

注意

1. 从节省 RAM 资源的角度考虑，*pC2LastPart* 可以与 *pDataLastPart* 结

构体中的 *pbData* 共享同一片存储区。这样使用时，当原始数据需要重用的话，则请注意备份；

2. *pDataLastPart* 数据的长度，限制范围 0~127 个字节。

3. 函数运行时，将使用 1KB 的 PKU RAM。

快速信息

头文件：声明在 iscrypt_sm2_ext.h。

库文件：使用 iscrypt_sm2_ext.lib。

示例

略

参阅

BIGINTEGER 、SM2EncryptInit、SM2EncryptUpdate

3.5.6 SM2Decrypt

SM2Decrypt 函数使用当前参数中的私钥，对指定的密文数据进行解密运算并输出运算结果至用户指定区域。

```
#include “iscrypt_sm2.h”
unsigned char SM2Decrypt (
    SM2PRIVATEKEYCTX xdata *pSM2PriKey,      // in
    SM2CIPHERTEXT xdata *pCiphertext,        // in
    SM2PLAINTEXT xdata *pPlaintext           // out
);
```

参数

pSM2PriKey

指向SM2 私钥结构体 **SM2PRIVATEKEYCTX**的指针。

pCiphertext

指向SM2 密文结构体 **SM2CIPHERTEXT**的指针。

pPlaintext

指向SM2 明文结构体 **SM2PLAINTEXT**的指针。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，错误代码列举如下。

Error Code	Description
INVALID_PARA	密文数据的参数错误
IN_DATA_LENGTH_ERROR	密文数据的长度错误
HASH_ERROR	密文中的哈希值错误

注意

1. 从节省 RAM 资源的角度考虑，密文结构体中的 *pbCipher* 可以与明文

结构体中的 *pbData* 共享同一片存储区。

2. 当前加密的数据长度限定为 1-32 字节，如果想要加密数据长度更长的数据，请使用 SM2DecryptInit、SM2DecryptUpdate、SM2DecryptFinal函数。

3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm2.h。

库文件：使用 iscrypt_sm2.lib。

示例

```
#include <string.h>
#include "iscrypt_sm2.h"
#include "iscrypt_sm2_test.h"

unsigned char xdata space_for_test[1024];

SM2POINT xdata Point;
SM2PUBLICKEYCTX xdata PubKey_A;
SM2PRIVATEKEYCTX xdata PriKey_A;

SM2PLAINTEXT xdata Plain;
SM2CIPHERTEXT xdata Cipher;

char code SM2Plaintext[] = "encryption standard";

char SM2_ DECRYPT_TEST()
{
    unsigned char ret;

    // prepare key
    memcpy(space_for_test, SM2PublicKey, 64);
    memcpy(space_for_test+64, SM2PrivateKey, 32);
```

```
PubKey_A.x = space_for_test;
PubKey_A.y = space_for_test+32;
PriKey_A.d = space_for_test+64;

// prepare plaintext
memcpy(space_for_test+0x100, SM2Plaintext, strlen(SM2Plaintext));
Plain.pd = space_for_test+0x100;
Plain.len = strlen(SM2Plaintext);

// prepare ciphertext
Point.x = space_for_test+0x200;
Point.y = space_for_test+0x220;

Cipher.p = &Point;
Cipher.c = space_for_test+0x260;
Cipher.h = space_for_test+0x240;

//
ret = SM2Encrypt(&PubKey_A, &Plain, &Cipher);
if (SUCCESS != ret)
{
    goto ErrorExit;
}

Plain.pd = space_for_test+0x300;
Plain.len = 0x00;

ret = SM2Decrypt(&PriKey_A, &Cipher, &Plain);
if (SUCCESS != ret)
{
    goto ErrorExit;
```

```
        }

        Plain.pd[Plain.len] = '\0';
        ret = strcmp(SM2Plaintext, Plain.pd);
        if (0 != ret)
        {
            goto ErrorExit;
        }

        return 0;

    ErrorExit:
        return 1;
    }
```

参阅

SM2PRIVATEKEYCTX、SM2CIPHERTEXT、SM2PLAINTEXT、SM2Encrypt、SM2DecryptInit、SM2DecryptUpdate、SM2DecryptFinal

3.5.7 SM2DecryptInit

SM2EncryptInit 函数用于 SM2 算法的加密初始化操作。

```
#include "iscrypt_sm2.h"
#include "iscrypt_sm2_ext.h"
unsigned char SM2DecryptInit(
    SM2PRIVATEKEYCTX xdata *pSM2PriKey,      // in
    SM2POINT xdata *pC1                      // in
);
```

参数

pSM2PriKey

指向SM2 私钥结构体 **SM2PRIVATEKEYCTX**的指针。

pC1

指向椭圆曲线上点的结构体指针 **SM2POINT**，是SM2 密文中的C1。

返回值

执行成功则返回 SUCCESS；执行失败则返回错误代码如下：

Error Code	Description
INVALID_PARA	密文数据 C1 错误。

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm2_ext.h。

库文件：使用 iscrypt_sm2_ext.lib。

示例

略

参阅

SM2PRIVATEKEYCTX、SM2POINT、SM2DecryptUpdate、SM2DecryptFinal

3.5.8 SM2DecryptUpdate

SM2DecryptUpdate 函数对指定的中间分组数据进行 SM2 的解密运算，并输出运算结果至用户指定区域。

```
#include "iscrypt_sm2.h"
#include "iscrypt_sm2_ext.h"
unsigned char SM2DecryptUpdate(
    unsigned char xdata *pC2Part,        // in
    unsigned char xdata *pDataPart       // out
);
```

参数

pC2Part

SM2 的中间密文分组。

pDataPart

对应的中间密文分组的明文。

返回值

函数执行成功，返回值为 SUCCESS；无其它异常返回值。

注意

1. 从节省 RAM 资源的角度考虑，*pDataPart* 可以与 *pC2Part* 共享同一片存储区。这样使用时，当原始数据需要重用的话，则请注意备份。
2. 数据的分组长度以 128 字节为一单位。
3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm2_ext.h。

库文件：使用 iscrypt_sm2_ext.lib。

示例

无

参阅

SM2DecryptInit、SM2DecryptFinal

3.5.9 SM2DecryptFinal

SM2DecryptFinal 函数对指定的末块分组数据进行 SM2 的解密运算，并输出运算结果至用户指定区域。

```
#include "iscrypt_sm2.h"
#include "iscrypt_sm2_ext.h"
unsigned char SM2DecryptFinal(
    BIGINTEGER xdata *pC2LastPart,        // in
    unsigned char xdata *pC3,             // in
    unsigned char xdata *pDataLastPart    // out
);
```

参数

pC2LastPart

指向 **BIGINTEGER** 的结构体指针，表示SM2 的末块密文分组，由两部分构成。

pC3

SM2 密文中的 C3 部分。

pDataLastPart

对应末块密文分组的明文。

返回值

函数执行成功，返回值为 **SUCCESS**。如果失败，错误代码列举如下。

Error Code	Description
IN_DATA_LENGTH_ERROR	明文数据的长度错误。
HASH_ERROR	SM2 密文的 C3 错误

注意

1. 从节省 RAM 资源的角度考虑，*pC2LastPart* 结构体中的 *pbData* 可以与 *pDataLastPart* 共享同一片存储区。这样使用时，当原始数据需要重用的话，则请注意备份。

2. *pC2LastPart* 数据的长度，限制范围 0~127 个字节。

3. 函数运行时，将使用 1KB 的 PKU RAM。

快速信息

头文件：声明在 iscrypt_sm2_ext.h。

库文件：使用 iscrypt_sm2_ext.lib。

示例

略

参阅

BIGINTEGER、SM2DecryptInit、SM2DecryptUpdate

3.5.10 SM2Sign

SM2Sign 函数使用当前参数中的私钥，对输入消息进行签名运算，并生成签名结果存放在指定区域。

```
#include “iscrypt_sm2.h”
unsigned char SM2Sign(
    SM2PRIVATEKEYCTX xdata *pSM2PriKey,      // in
    unsigned char xdata *pbDigest,           // in
    SM2SIGNATURE xdata *pSignature           // out
);
```

参数

pSM2PriKey

指向用于签名运算的SM2 私钥结构体的指针 **SM2PRIVATEKEYCTX**。

pbDigest

待签名数据，该数据通常应为消息数据的摘要值。按照《SM2 公钥密码算法标准》中的规定，该值应为$Hash_{256}$（Z_A || M）；其中$Hash_{256}$应选用SM3算法。

pSignature

消息的摘要的签名值结构体指针 **SM2SIGNATURE**，由两部分组成。

返回值

要求函数必须执行成功，返回值为 **SUCCESS**。

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm2.h。

库文件：使用 iscrypt_sm2.lib。

示例

```
#include <string.h>
#include "iscrypt_sm2.h"
#include "iscrypt_sm2_test.h"

extern unsigned char xdata space_for_test[1024];

SM2PUBLICKEYCTX xdata PubKey_A;
SM2PRIVATEKEYCTX xdata PriKey_A;

SM2SIGNATURE xdata Signature;

char code SM2Message[] = "message digest";

char SM2_SIGN_VERIFY_TEST()
{
    unsigned char ret;
    unsigned char xdata *tmp_msg;
    unsigned char xdata tmp_msg_len;

    // prepare key
    memcpy(space_for_test, SM2PublicKey, 64);
    memcpy(space_for_test+64, SM2PrivateKey, 32);

    PubKey_A.x = space_for_test;
    PubKey_A.y = space_for_test+32;
    PriKey_A.d = space_for_test+64;

    ////////////////////////////////////////////////////////////////
    // precompute step 1: Za
```

```
memcpy(space_for_test+0x100, SM2KeyExchangeParam+384, 16);

IDInfo_A.pd = space_for_test+0x100;
IDInfo_A.len = 16;

ret = SM2GetZ(&PubKey_A, &IDInfo_A, space_for_test+0x100);

if (SUCCESS != ret)
{
    goto ErrorExit;
}

tmp_msg = space_for_test+0x100;
tmp_msg_len = 32;

// M~ = ZA || M
memcpy(space_for_test+0x120, SM2Message, strlen(SM2Message));
tmp_msg_len += strlen(SM2Message);


// precompute step 2: e = Hash(Za || M)
SM3Init();

while (tmp_msg_len >= 64)
{
    SM3Update(tmp_msg);

    tmp_msg += 64;
    tmp_msg_len -= 64;
}

SM3Final(tmp_msg, tmp_msg_len, space_for_test+0x100);
```

```
    // SM2 sign
    Signature.r = space_for_test+0x200;
    Signature.s = space_for_test+0x220;

    ret = SM2Sign(&PriKey_A, space_for_test+0x100, &Signature);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }

    ret = SM2Verify(&PubKey_A, space_for_test+0x100, &Signature);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }

    return 0;

ErrorExit:
    return 1;
}
```

参阅

SM2PRIVATEKEYCTX、SM2SIGNATURE、SM2Verify

3.5.11 SM2Verify

SM2Verify 函数使用当前参数中的 SM2 公钥，对输入消息及其签名值，进行验证。

```
#include “iscrypt_sm2.h”
unsigned char SM2Verify(
    SM2PUBLICKEYCTX xdata *pSM2PubKey,// in
    unsigned char xdata *pbDigest,                  // in
    SM2SIGNATURE xdata *pSignature                  // in
);
```

参数

pSM2PubKey

指向用于签名运算的SM2 公钥结构体指针 **SM2PUBLICKEYCTX**。

pbDigest

待签名数据，该数据通常应为消息数据的摘要值。按照《SM2 公钥密码算法标准》中的规定，该值应为$Hash_{256}$（Z_A || M）；其中$Hash_{256}$应选用SM3算法。

pSignature

指向SM2 签名结构体 **SM2SIGNATURE**的指针，表示待验证消息摘要的签名值。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，返回值为 FAIL。

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm2.h。

库文件：使用 iscrypt_sm2.lib。

示例

请参见 **SM2Sign**。

参阅

SM2PUBLICKEYCTX、SM2SIGNATURE、SM2Sign

3.5.12 SM2KeyExchange

SM2KeyExchange 函数使用当前参数进行密钥交换运算，并生成交换密钥输出至用户指定区域。

```
#include “iscrypt_sm2.h”
#include “iscrypt_sm3.h”
unsigned char SM2KeyExchange(
SM2KEYEXCHANGEPARAM xdata *pSM2KeyExchangeParam,      //in
    unsigned char xdata *pbKey                          // out
);
```

参数

pSM2KeyExchangeParam

指向用于传送密钥交换所需要的具体参数的密钥交换结构体 **SM2KEYEXCHANGEPARAM**的指针。

pbKey

密钥交换运算成功后，生成的指定长度的密钥。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，返回 FAIL。

注意

1. 期望交换密钥的长度 *bKeyLenExpected* 通常选用 8、16、32 三种长度，同时支持不大于 128 字节的任意字节长的交换密钥生成。
2. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm2.h。

库文件：使用 iscrypt_sm2.lib。

示例

```
#include <string.h>
#include "iscrypt_sm2.h"
#include "iscrypt_sm2_test.h"

extern unsigned char xdata space_for_test[1024];

SM2PUBLICKEYCTX xdata PubKey_A;
SM2PUBLICKEYCTX xdata PubKey_B;
SM2PUBLICKEYCTX xdata TempPubKey_A;
SM2PUBLICKEYCTX xdata TempPubKey_B;

SM2PRIVATEKEYCTX xdata PriKey_A;
SM2PRIVATEKEYCTX xdata PriKey_B;
SM2PRIVATEKEYCTX xdata TempPriKey_A;
SM2PRIVATEKEYCTX xdata TempPriKey_B;

IDINFO xdata IDInfo_A, xdata IDInfo_B;

SM2KEYEXCHANGEPARAM xdata KeyExchangeParam;

char SM2_KEY_EXCHANGE_TEST()
{
    unsigned char ret;

    unsigned char xdata *pZa, xdata *pZb;

    // Just as alias
    unsigned char xdata *pbKeyAB, xdata *pbKeyBA;

    ////////////////////////////////////////////////////////////////
```

```
memcpy(space_for_test,SM2KeyExchangeParam,sizeof(SM2KeyExchangeParam));

//
TempPubKey_A.x = space_for_test;
TempPubKey_A.y = space_for_test+32;
TempPriKey_A.d = space_for_test+64;

PubKey_A.x = space_for_test+96;
PubKey_A.y = space_for_test+128;
PriKey_A.d = space_for_test+160;

TempPubKey_B.x = space_for_test+192;
TempPubKey_B.y = space_for_test+224;
TempPriKey_B.d = space_for_test+256;

PubKey_B.x = space_for_test+288;
PubKey_B.y = space_for_test+320;
PriKey_B.d = space_for_test+352;

IDInfo_A.pd = space_for_test+384;
IDInfo_A.len = 16;

IDInfo_B.pd = space_for_test+400;
IDInfo_B.len = 16;

// precompute Za
pZa = space_for_test+432;

ret = SM2GetZ(&PubKey_A, &IDInfo_A, pZa);
if (SUCCESS != ret)
{
    goto ErrorExit;
```

```
}

// precompute Zb
pZb = space_for_test+464;

ret = SM2GetZ(&PubKey_B, &IDInfo_B, pZb);
if (SUCCESS != ret)
{
    goto ErrorExit;
}

// A is initiator, and B is responder

//////////////////////////////////////////////////////////////////
memset(&KeyExchangeParam, 0x00, sizeof (KeyExchangeParam));

KeyExchangeParam.role = INITIATOR;
KeyExchangeParam.keyLenExpected = 32;
KeyExchangeParam.pSelfTempPubKey = &TempPubKey_A;
KeyExchangeParam.pSelfTempPriKey = &TempPriKey_A;
KeyExchangeParam.pSelfPubKey = &PubKey_A;
KeyExchangeParam.pSelfPriKey = &PriKey_A;
KeyExchangeParam.pOtherTempPubKey = &TempPubKey_B;
KeyExchangeParam.pOtherPubKey = &PubKey_B;
KeyExchangeParam.pSelfZ = pZa;
KeyExchangeParam.pOtherZ = pZb;

pbKeyAB = space_for_test + 0x200;

ret = SM2KeyExchange(&KeyExchangeParam, pbKeyAB);
if (SUCCESS != ret)
{
```

```
    goto ErrorExit;
}

//////////////////////////////////////////////////////////////////
memset(&KeyExchangeParam, 0x00, sizeof (KeyExchangeParam));

KeyExchangeParam.role = RESPONDER;
KeyExchangeParam.keyLenExpected = 32;
KeyExchangeParam.pSelfTempPubKey = &TempPubKey_B;
KeyExchangeParam.pSelfTempPriKey = &TempPriKey_B;
KeyExchangeParam.pSelfPubKey = &PubKey_B;
KeyExchangeParam.pSelfPriKey = &PriKey_B;
KeyExchangeParam.pOtherTempPubKey = &TempPubKey_A;
KeyExchangeParam.pOtherPubKey = &PubKey_A;
KeyExchangeParam.pSelfZ = pZb;
KeyExchangeParam.pOtherZ = pZa;

pbKeyBA = space_for_test + 0x300;

ret = SM2KeyExchange(&KeyExchangeParam, pbKeyBA);
if (SUCCESS != ret)
{
    goto ErrorExit;
}

//////////////////////////////////////////////////////////////////
ret = memcmp(pbKeyAB, pbKeyBA, 32);
if (0 != ret)
{
    goto ErrorExit;
}
```

```
    return 0;

ErrorExit:
    return 1;
}
```

参阅

SM2KEYEXCHANGEPARAM、SM2GetZ

3.5.13 SM2GetZ

SM2GetZ 函数生成签名验证运算和密钥协商运算中需要使用的数据。

```
#include “iscrypt_sm2.h”
unsigned char SM2GetZ(
    SM2PUBLICKEYCTX xdata *pSM2PubKey,// in
    IDINFO xdata *pIDInfo,                    // in
    unsigned char xdata *pbZValue             // out
);
```

参数

pSM2PubKey

SM2 公钥结构体指针 **SM2PUBLICKEYCTX**。

pIDInfo

用户的ID信息结构体 **IDINFO**指针，分成两部分。

pZValue

用于存放生成签名认证运算和密钥协商运算中需要使用的数据的地址。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，错误代码列举如下。

Error Code	Description
IN_DATA_LENGTH_ERROR	ID 的长度错误。

注意

1. 按照《SM2 椭圆曲线公钥密码算法》，使用 **SM2Sign**和 **SM2Verify**，以及 **SM2KeyExchange**等函数时，需要用户的身份信息的杂凑值的参与。其中用户身份信息的杂凑值计算方式为 $Z_A=H_{256}(ENTL_A||ID_A||a||b||x_G||y_G||x_A||y_A)$。

2. *pIDInfo* 中 ID 的长度，限制范围 1~32 个字节。

3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm2.h。

库文件：使用 iscrypt_sm2.lib。

示例

```
#include <string.h>
#include "iscrypt_sm2.h"
#include "iscrypt_sm2_test.h"

extern unsigned char xdata space_for_test[1024];

SM2PUBLICKEYCTX xdata PubKey_A;
IDINFO xdata IDInfo_A;

char SM2_KEY_GET_Z_TEST()
{
    unsigned char ret;
    unsigned char xdata *pbZ;

    memcpy(space_for_test,SM2KeyExchangeParam,sizeof(SM2KeyExchangeParam));

    PubKey_A.x = space_for_test+96;
    PubKey_A.y = space_for_test+128;
    PriKey_A.d = space_for_test+160;

    IDInfo_A.pd = space_for_test+384;
    IDInfo_A.len = 16;

    pbZ = space_for_test+0x200;
```

```
    ret = SM2GetZ(&PubKey_A, &IDInfo_A, pbZ);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }

    return 0;

ErrorExit:
    return 1;
}
```

参阅

SM2PUBLICKEYCTX、IDINFO

3.6 SM3

3.6.1 SM3Init

SM3Init 函数用于完成基于 SM3 算法的初始化操作。

#include “iscrypt_sm3.h”

void SM3Init();

参数

无

返回值

无

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm3.h。

库文件：使用 iscrypt_sm3.lib。

示例

请参见 **SM3Final**一节。

参阅

SM3Update，SM3Final

3.6.2 SM3Update

SM3Update 用以对一个 SM3 的消息分组中间块儿进行处理。

```
#include “iscrypt_sm3.h”
void SM3Update(
    unsigned char xdata *pbMessage      //in
);
```

参数

pbMessage

需要被哈希处理的消息分组，对于 SM3 算法来说为 64 字节为一组。

返回值

无

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm3.h。

库文件：使用 iscrypt_sm3.lib。

示例

请参见 **SM3Final**一节。

参阅

SM3Init、SM3Final

3.6.3 SM3Final

SM3Final 用以对一个 SM3 的末块儿消息分组进行处理。

```
#include “iscrypt_sm3.h”
unsigned char SM3Final (
    unsigned char xdata *pbMessage      //  in
    unsigned char bLength,              //  in
    unsigned char xdata *pbDigest       //  out
);
```

参数

pbMessage

需要被哈希处理的末块儿消息分组。

bLength

末块儿消息分组的字节长度，小于 64，可以为 0。

pbDigest

SM3 处理结束后，摘要的存储区地址。SM3 摘要长度为 32 字节（256 比特）。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，错误代码列举如下。

Error Code	Description
HASH_OBJECT_ERROR	哈希对象错误

注意

1. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm3.h。

库文件：使用 iscrypt_sm3.lib。

示例

```
#include <string.h>
#include "iscrypt_sm3.h"
#include "iscrypt_sm3_test.h"

unsigned char xdata space_for_test[1024];

char SM3_TEST()
{
    unsigned char xdata ret;
    memcpy(space_for_test, SM3_VECTOR, 64);

    ////////////////////////////////////////////////////////////////
    SM3Init();

    ret = SM3Final(space_for_test, 3, space_for_test+0x200);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }

    ret = memcmp(space_for_test+0x200, DIGEST_SM3_3, 32);
    if (0 != ret)
    {
        goto ErrorExit;
    }

    ////////////////////////////////////////////////////////////////
    SM3Init();
```

```
ret = SM3Final(space_for_test, 56, space_for_test+0x200);
if (SUCCESS != ret)
{
    goto ErrorExit;
}

ret = memcmp(space_for_test+0x200, DIGEST_SM3_56, 32);
if (0 != ret)
{
    goto ErrorExit;
}

////////////////////////////////////////////////////////////////
SM3Init();

SM3Update(space_for_test);

ret = SM3Final(space_for_test, 0, space_for_test+0x200);
if (SUCCESS != ret)
{
    goto ErrorExit;
}

ret = memcmp(space_for_test+0x200, DIGEST_SM3_64, 32);
if (0 != ret)
{
    goto ErrorExit;
}

////////////////////////////////////////////////////////////////
SM3Init();
```

```
SM3Update(space_for_test);

SM3Update(space_for_test);

ret = SM3Final(space_for_test, 0, space_for_test+0x200);
if (SUCCESS != ret)
{
    goto ErrorExit;
}

ret = memcmp(space_for_test+0x200, DIGEST_SM3_128, 32);
if (0 != ret)
{
    goto ErrorExit;
}

////////////////////////////////////////////////////////////////
SM3Init();

SM3Update(space_for_test);

SM3Update(space_for_test);

ret = SM3Final(space_for_test, 56, space_for_test+0x200);
if (SUCCESS != ret)
{
    goto ErrorExit;
}

ret = memcmp(space_for_test+0x200, DIGEST_SM3_184, 32);
if (0 != ret)
```

```
    {
        goto ErrorExit;
    }

    return 0;
ErrorExit:
    return 1;
}
```

参阅

SM3Init、SM3Update

3.7 SM4

3.7.1 SM4Crypt

SM4Crypt 函数用于提供基于 SM4 算法的 ECB 和 CBC 模式的加解密功能操作。

```
#include “iscrypt_sm4.h”
unsigned char SM4Crypt (
    BLOCKCIPHERPARAM xdata *pBlockCipherParam        //  in,out
);
```

参数

pBlockCipherParam

指向分组密码算法结构体 **BLOCKCIPHERPARAM**，用以传送SM4 密钥、操作模式、输入输出等数据信息。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，错误代码列举如下。

Error Code	Description
NOT_SUPPORT_YET_ERROR	暂不支持
MEMORY_ERROR	内存操作错误

注意

1. SM4 为以 SM4_BLOCK_SIZE（定义在头文件中）个字节为一分组。
2. 当前算法仅支持 ECB 和 CBC 模式，其他模式不支持。
3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_sm4.h。

库文件：使用 iscrypt_sm4.lib。

示例

```
#include <string.h>
#include "iscrypt_sm4.h"

extern unsigned char xdata space_for_test[1024];
extern BLOCKCIPHERPARAM xdata BlockCipherParam;

char SM4_TEST()
{
    unsigned char ret;

    memcpy(space_for_test, BLOCK_CRYPT_KEY, SM4_BLOCK_SIZE);
    memcpy(space_for_test+SM4_BLOCK_SIZE,BLOCK_CRYPT_IV,SM4_BLOCK_SIZ
E);

    memcpy(space_for_test+2*SM4_BLOCK_SIZE,BLOCK_CRYPT_PLAINTEXT,
5*SM4_BLOCK_SIZE);

    BlockCipherParam.bOpMode = SYM_ECB | SYM_ENCRYPT;
    BlockCipherParam.pbKey = space_for_test;
    BlockCipherParam.pbIV = space_for_test+SM4_BLOCK_SIZE;

    BlockCipherParam.pbInput = space_for_test+2*SM4_BLOCK_SIZE;
    BlockCipherParam.bTotalBlock = 5;
    BlockCipherParam.pbOutput = space_for_test+2*SM4_BLOCK_SIZE;

    ret = SM4Crypt(&BlockCipherParam);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
    }
```

```
        ret=memcmp(space_for_test+2*SM4_BLOCK_SIZE,BLOCK_CRYPT_CIPHERTEXT
_SM4_ECB, 5*SM4_BLOCK_SIZE);

        if (0 != ret)
        {
            goto ErrorExit;
        }

        return 0;
    ErrorExit:
        return 1;
    }
```

参阅

BLOCKCIPHERPARAM、SM1Crypt

3.8 RNG

3.8.1 GenRandom

生成指定长度的随机数。

```
#include “iscrypt_random.h”
void GenRandom(
    unsigned short bRandomLen,          // in
    unsigned char xdata *pbRandom       // out
);
```

参数

bRandomLen

需要生成随机数的字节长度。

pbRandom

存储随机数的缓冲区地址。

返回值

无

注意

无

示例

```
#include "iscrypt_random.h"

void main()
{
 unsigned char xdata space_for_test[256] = {0x00};
 GenRandom(16, space_for_test);
```

```
while(1) ;
}
```

参阅

无

3.8.1 GetFactoryCode

读取芯片工厂码。

```
#include “iscrypt_random.h”
void GetFactoryCode(
    unsigned char xdata *pbFtyCode       // out
);
```

参数

pbFtyCode

存储芯片序列号的缓冲区地址，芯片序列号只有 13 个字节。

返回值

无

注意

无

示例

```
#include "iscrypt_random.h"

void main()
{
 unsigned char xdata space_for_test[8];
 GetFactoryCode(space_for_test);
 while(1) ;
}
```

参阅

无

3.9 其他

3.9.1 HASHSetBkpt

```
#include "iscrypt_hash_bkpt_rst.h"
unsigned char HASHSetBkpt(
    unsigned char ChoiceOfHASH,                  // in
    unsigned char xdata *pbRareIV,               // out
    unsigned char xdata *pLengthHasProcessed     // out
);
```

参数

ChoiceOfHASH

HASH 算法选择，支持以下几种。

Value	Description
SHA1	SHA-1
SHA256	SHA-256
SM3	国标 SM3 哈希算法

pbRareIV

当前 HASH 算法的中间向量，不同 HASH 算法的中间向量的长度不同。

pMsgLengthHasProcessed

当前 HASH 算法已经处理过的消息数据的长度，不同 HASH 算法的长度表示范围和所占用的空间大小不同。

返回值

无

注意

针对不同的 HASH 算法，对其进行中断续处理时，需要备份的数据的长

度不同。因此，函数调用者需保证已经申请了足够的空间进行这些内容的保护。

示例

请参见 **HASHRestore**的示例

参阅

HASHRestore

3.9.1 HASHRestore

```
#include "iscrypt_hash_bkpt_rst.h"
unsigned char HASHRestore(
    unsigned char ChoiceOfHASH,                    // in
    unsigned char xdata *pbRareIV,                 // in
    unsigned char xdata *pLengthHasProcessed       // in
);
```

参数

ChoiceOfHASH

HASH 算法选择，支持以下几种。

Value	**Description**
SHA1	SHA-1
SHA256	SHA-256
SM3	国标 SM3 哈希算法

pbRareIV

当前 HASH 算法的中间向量，不同 HASH 算法的中间向量的长度不同。

pMsgLengthHasProcessed

当前 HASH 算法已经处理过的消息数据的长度，不同 HASH 算法的长度表示范围和所占用的空间大小不同。

返回值

无

注意

无

示例

```
#include <string.h>
#include "iscrypt_sha.h"
#include “iscrypt_sm3.h”
#include "iscrypt_hash_bkpt_rst.h"
unsigned char xdata space_for_test[1024];

char HASH_BKPT_RST_TEST()
{
    unsigned char xdata ret;
    unsigned char xdata sha_backup_iv[64];
    unsigned char xdata sha_backup_msg_length[16];

    memcpy(space_for_test, SHA_VECTOR, 64);

    ////////////////////////////////////////////////////////////
    //SHA1
    SHA1Init();

    SHA1Update(space_for_test);
    ret = HASHSetBkpt(SHA1, sha_backup_iv, sha_backup_msg_length);

    // Do other things
    SHA256Init();

    //
    ret = HASHRestore(SHA1, sha_backup_iv, sha_backup_msg_length);
    SHA1Update(space_for_test);

    ret = SHA1Final(space_for_test, 0, space_for_test+0x200);
    if (SUCCESS != ret)
    {
        goto ErrorExit;
```

```
    }

    ret = memcmp(space_for_test+0x200, DIGEST_SHA1_128, 20);
    if (0 != ret)
    {
        goto ErrorExit;
    }

    return SUCCESS;

ErrorExit:
    return FAIL;
}
```

参阅

HASHSetBkpt

3.10 ECC

3.10.1 ECC_Init

```
#include "iscrypt_ecc.h"
unsigned char ECC_Init(
    EC_PARA xdata *pEC   // in, out
);
```

参数

pEC

指向椭圆曲线参数结构体 **EC_PARA**，用以传送ECC参数信息。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，返回值为 NOT_SUPPORT_YET_ERROR。

注意

1. 函数调用者需要通过结构体参数中的成员 *bitLen*、*p*、*a*、*b*、*Gx*、*Gy*、*n* 为传入使用的曲线位长和曲线参数，其它的成员由函数内部计算，函数调用者不需关心，但不能修改，否则可能引起计算错误；
2. 所有数据成员的数序均为自然序；
3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_ecc.h。

库文件：使用 iscrypt_ecc.lib。

示例

```
unsigned char code ECC_160_PARA[]={
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xf4,0x8b,
```

```
0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xf4,0x88,
0x41,0x7f,0x0e,0x7c,0x28,0x25,0x00,0x0c,0x83,0x8b,0x7c,0x95,0x15,0xd1,0xff,0xf3,0x69,0x
b5,0x0e,0x16,0x4f,0x3a,0x60,0x9a,0xec,0x06,0xde,0x6e,0x46,0x6e,0x2d,0xb5,0x2f,0x0e,0xd5
,0x36,0xc1,0xff,0x87,0xf4,0x85,0x59,0xef,0x2a,0xdd,0xa2,0x00,0x56,0xc4,0xbe,0x54,0xe3,0x
06,0x5d,0xe5,0xfe,0x39,0x52,0xff,0x22,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xff,0xfe,0xa8,0
xc8,0x95,0xf7,0x67,0x85,0xfc,0x17,0x91,0x71};

   memset(&ec_para_160, 0x00, sizeof(ec_para_160));
   ec_para_160.bitLen = 160;
   memcpy(ec_para_160.p, ECC_160_PARA, 20);
   memcpy(ec_para_160.a, ECC_160_PARA+20, 20);
   memcpy(ec_para_160.b, ECC_160_PARA+40, 20);
   memcpy(ec_para_160.Gx, ECC_160_PARA+60, 20);
   memcpy(ec_para_160.Gy, ECC_160_PARA+80, 20);
   memcpy(ec_para_160.n, ECC_160_PARA+100, 20);

   ret = ECC_Init(&ec_para_160);
   if (0 != ret)
   {
       return 1;
   }
    return 0;
```

参阅

EC_PARA

3.10.2 ECC_PointMult

```
#include "iscrypt_ecc.h"
unsigned char ECC_PointMult(
    EC_PARA xdata *pEC,        // in
    unsigned char xdata *k,    // in
    ECCPOINT xdata *pPoint     // in, out
);
```

参数

pEC

指向椭圆曲线参数结构体 **EC_PARA**，用以传送ECC参数信息。

k

指向计算椭圆曲线标量乘所需要的标量。

pPoint

指向椭圆曲线上点的结构体 **ECCPOINT**，提供计算椭圆曲线标量乘所需要的椭圆曲线上的点。

返回值

如果函数执行成功，返回值为 SUCCESS。如果失败，返回值为 NOT_SUPPORT_YET_ERROR。

注意

1. 确保在调用该函数之前已调用 ECC_Init；
2. 所有输入输出数据的数序均为自然序；
3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_ecc.h。

库文件：使用 iscrypt_ecc.lib。

示例

```
unsigned char xdata space_for_test[128];
ECCPRIKEYCTX xdata pSelfECCPriKey;
ECCPUBLICKEYCTX xdata pSelfECCPubKey;
ECCPOINT xdata pECCPoint;

unsigned char xdata selfPrikey[]={0x3a,0xc0,0xe7,0x17,0xeb,0x61,0x60,0x2e,
0xfc,0xbb,0x1d,0xe8,0x1a,0xa1,0x44,0xa2,0x72,0xb4,0x4b,0xa1};
unsigned char xdata SelfPubkey[] = {0xdf,0xfa,0x23,0xf8,0x08, 0x51,0x48,0xf4,0x6e,
0x87,0x56,0xee,0x54,0x0a,0x5a,0xe1,0xf5,0x4c,0x16,0xfb,0x8e,0x34,0x50,0x5d,0x2d,0x08,0xc8
,0x95,0xf7,0x23,0x30,0x80,0xf3,0xe1,0x07,0x8f,0x5d,0x60,0xaf,0x41};
memcpy(space_for_test, ec_para_160.Gx, 20);
memcpy(space_for_test+20, ec_para_160.Gy, 20);

pSelfECCPubKey.x = space_for_test;
pSelfECCPubKey.y = space_for_test+20;

ret = ECC_PointMult(&ec_para_160, SelfPrikey, &pSelfECCPubKey);
if (0 != ret)
{
    return 1;
}

ret = memcmp(space_for_test, SelfPubkey, 40);
if (0 != ret)
{
    return 1;
 }
return 0;
```

参阅

EC_PARA、ECCPUBLICKEYCTX

3.10.3 ECC_ECDSA_Sign

```
#include "iscrypt_ecc.h"
unsigned char ECC_ECDSA_Sign(
    EC_PARA xdata *pEC,                     // in
    ECCPRIKEYCTX  xdata   *pECCPriKey,      // in
    unsigned char xdata *pbDigest,          // in
    ECCSIGNATURE  xdata   *pSignature       // out
);
```

参数

pEC

指向椭圆曲线参数结构体 **EC_PARA**，用以传送ECC参数信息。

pECCPriKey

指向椭圆曲线私钥结构体指针 **ECCPRIKEYCTX**。

pbDigest

待签名数据，数据长度与 *pEC* 中 *bitLen* 一致。

pSignature

消息的摘要的签名值结构体指针 **ECCSIGNATURE**，由两部分组成。

返回值

要求函数必须执行成功，返回值为 **SUCCESS**。

注意

1. 确保在调用该函数之前已调用 ECC_Init；
2. 所有输入输出数据的数序均为自然序；
3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_ecc.h。

库文件：使用 iscrypt_ecc.lib。

示例

```
unsigned char xdata space_for_test[128];
ECCPRIKEYCTX xdata pSelfECCPriKey;
ECCPUBLICKEYCTX xdata pSelfECCPubKey;
ECCSIGNATURE xdata pSignature;

unsigned char xdata selfPrikey[]={0x3a,0xc0,0xe7,0x17,0xeb,0x61,0x60,0x2e, 0xfc,0xbb, 0x1d, 0xe8,0x1a,0xa1,0x44,0xa2,0x72,0xb4,0x4b,0xa1};
unsigned char xdata SelfPubkey[] = {0xdf,0xfa,0x23,0xf8,0x08, 0x51,0x48,0xf4,0x6e, 0x87,0x56,0xee,0x54,0x0a,0x5a,0xe1,0xf5,0x4c,0x16,0xfb,0x8e,0x34,0x50,0x5d,0x2d,0x08,0xc8 ,0x95,0xf7,0x23,0x30,0x80,0xf3,0xe1,0x07,0x8f,0x5d,0x60,0xaf,0x41};

memcpy(space_for_test, ec_para_160.Gx, 20);
memcpy(space_for_test+20, ec_para_160.Gy, 20);

pSelfECCPubKey.x = space_for_test;
pSelfECCPubKey.y = space_for_test+20;

pSelfECCPriKey.d = SelfPrikey;
pSelfECCPubKey.x = SelfPubkey;
pSelfECCPubKey.y = SelfPubkey+20;

pSignature.r = space_for_test;
pSignature.s = space_for_test+32;

ret = ECC_ECDSA_Sign(&ec_para_160, &pSelfECCPriKey, Digest, &pSignature);
if (0 != ret)
{
    return 1;
```

```
}

ret = ECC_ECDSA_Verify(&ec_para_160, &pSelfECCPubKey, Digest, &pSignature);
if (0 != ret)
{
    return 1;
}
  return 0;
```

参阅

EC_PARA、ECCPRIKEYCTX、ECCSIGNATURE、ECC_ECDSA_Verify

3.10.4 ECC_ECDSA_Verify

```
#include "iscrypt_ecc.h"
unsigned char ECC_ECDSA_Verify(
    EC_PARA xdata *pEC,                          // in
    ECCPUBLICKEYCTX xdata *pECCPubKey,           // in
    unsigned char xdata *pbDigest,               // in
    ECCSIGNATURE xdata *pSignature               // in
);
```

参数

pEC

指向椭圆曲线参数结构体 **EC_PARA**，用以传送ECC参数信息。

pECCPubKey

指向椭圆曲线公钥结构体指针 **ECCPUBLICKEYCTX**。

pbDigest

待签名数据，数据长度与 *pEC* 中 *bitLen* 一致。

pSignature

消息的摘要的签名值结构体指针 **ECCSIGNATURE**，由两部分组成。

返回值

要求函数必须执行成功，返回值为 **SUCCESS**。

注意

1. 确保在调用该函数之前已调用 ECC_Init；
2. 所有输入输出数据的数序均为自然序；
3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_ecc.h。

库文件：使用 iscrypt_ecc.lib。

示例

参考 ECC_ECDSA_Sign 中的示例。

参阅

EC_PARA、ECCPUBLICKEYCTX、ECCSIGNATURE、ECC_ECDSA_Sign

3.10.5 ECC_ECES_Encrypt

```
#include "iscrypt_ecc.h"
unsigned char ECC_ECES_Encrypt(
    EC_PARA xdata *pEC,                          // in
    ECCPUBLICKEYCTX xdata *pECCPubKey,           // in
    ECCPLAINTEXT xdata *pPlaintext,              // in
    ECCCIPHERTEXT xdata *pCiphertext             // out
);
```

参数

pEC

指向椭圆曲线参数结构体 **EC_PARA**，用以传送ECC参数信息。

pECCPubKey

指向椭圆曲线公钥结构体指针 **ECCPUBLICKEYCTX**。

pPlaintext

指向椭圆曲线明文结构体指针 **ECCPLAINTEXT**，数据长度与*pEC*中*bitLen*一致。

pCiphertext

指向椭圆曲线的密文结构体指针 **ECCCIPHERTEXT**。

返回值

要求函数必须执行成功，返回值为 **SUCCESS**。

注意

1. 确保在调用该函数之前已调用 ECC_Init；
2. 所有输入输出数据的数序均为自然序；
3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_ecc.h。

库文件：使用 iscrypt_ecc.lib。

示例

```
unsigned char xdata space_for_test[128];
ECCPRIKEYCTX xdata pSelfECCPriKey;
ECCPUBLICKEYCTX xdata pSelfECCPubKey;
ECCPOINT xdata pECCPoint;
 ECCPLAINTEXT xdata pPlaintext;
 ECCCIPHERTEXT xdata pCiphertext;
```

unsigned char xdata selfPrikey[]={0x3a,0xc0,0xe7,0x17,0xeb,0x61,0x60,0x2e, 0xfc,0xbb, 0x1d, 0xe8,0x1a,0xa1,0x44,0xa2,0x72,0xb4,0x4b,0xa1};

unsigned char xdata SelfPubkey[] = {0xdf,0xfa,0x23,0xf8,0x08, 0x51,0x48,0xf4,0x6e, 0x87,0x56,0xee,0x54,0x0a,0x5a,0xe1,0xf5,0x4c,0x16,0xfb,0x8e,0x34,0x50,0x5d,0x2d,0x08,0xc8 ,0x95,0xf7,0x23,0x30,0x80,0xf3,0xe1,0x07,0x8f,0x5d,0x60,0xaf,0x41};

```
////////////////////////////////
memset(space_for_test, 0x11, 20);

pSelfECCPriKey.d = SelfPrikey;
pSelfECCPubKey.x = SelfPubkey;
pSelfECCPubKey.y = SelfPubkey+20;

pPlaintext.pd = space_for_test;
pPlaintext.len = 20;

pECCPoint.x = space_for_test+32;
pECCPoint.y = space_for_test+52;

pCiphertext.p = &pECCPoint;
```

```
    pCiphertext.c = space_for_test+96;
    pCiphertext.h = NULL;
    pCiphertext.clen = 0;

    ret = ECC_ECES_Encrypt(&ec_para_160, &pSelfECCPubKey, &pPlaintext,
&pCiphertext);
    if (0 != ret)
    {
        return 1;
    }

    ret = ECC_ECES_Decrypt(&ec_para_160, &pSelfECCPriKey, &pCiphertext,
&pPlaintext);
    if (0 != ret)
    {
        return 1;
    }

    for (i = 0; i < 20; i++)
    {
        if (0x11 != space_for_test[i])
        {
            return 1;
        }
    }
    return 0;
```

参阅

EC_PARA、ECCPUBLICKEYCTX、ECCPLAINTEXT、ECCCIPHERTEXT、ECC_ECES_Decrypt

3.10.6 ECC_ECES_Decrypt

```
#include "iscrypt_ecc.h"
unsigned char ECC_ECES_Decrypt(
    EC_PARA xdata *pEC,                    // in
    ECCPRIKEYCTX xdata *pECCPriKey,  // in
    ECCCIPHERTEXT xdata * pCiphertext, // in
    ECCPLAINTEXT xdata * pPlaintext      // out
);
```

参数

pEC

指向椭圆曲线参数结构体 **EC_PARA**，用以传送ECC参数信息。

pECCPriKey

指向椭圆曲线私钥结构体指针 **ECCPRIKEYCTX**。

pCiphertext

指向椭圆曲线密文结构体指针 **ECCCIPHERTEXT**。

pPlaintext

指向椭圆曲线的明文结构体指针 **ECCPLAINTEXT**。

返回值

要求函数必须执行成功，返回值为 **SUCCESS**。

注意

1. 确保在调用该函数之前已调用 ECC_Init；
2. 所有输入输出数据的数序均为自然序；
3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_ecc.h。

库文件：使用 iscrypt_ecc.lib。

示例

参考 ECC_ECES_Encrypt 中的示例。

参阅

EC_PARA、
ECCPRIKEYCTX、ECCPLAINTEXT、ECCCIPHERTEXT、
ECC_ECES_Encrypt

3.10.7 ECC_ECDH

```
#include "iscrypt_ecc.h"
unsigned char ECC_ECDH(
    EC_PARA xdata *pEC,                          // in
    ECCPRIKEYCTX xdata *pSelfPriKey,    // in
    ECCPUBLICKEYCTX xdata *pOhterPubKey,// in
    ECCPUBLICKEYCTX xdata *pAgreementKey// out
);
```

参数

pEC

指向椭圆曲线参数结构体 **EC_PARA**，用以传送ECC参数信息。

pSelfPriKey

指向椭圆曲线私钥结构体指针 **ECCPRIKEYCTX**。

pOhterPubKey

指向椭圆曲线公钥结构体指针 **ECCPUBLICKEYCTX**。

pAgreementKey

指向椭圆曲线公钥结构体指针 **ECCPUBLICKEYCTX**。

返回值

要求函数必须执行成功，返回值为 **SUCCESS**。

注意

1. 确保在调用该函数之前已调用 ECC_Init；
2. 所有输入输出数据的数序均为自然序；
3. **函数运行时，将使用 1KB 的 PKU RAM。**

快速信息

头文件：声明在 iscrypt_ecc.h。

库文件：使用 iscrypt_ecc.lib。

示例

```
unsigned char xdata space_for_test[128];
ECCPRIKEYCTX xdata pSelfECCPriKey;
ECCPUBLICKEYCTX xdata pSelfECCPubKey;

ECCPRIKEYCTX xdata pOtherECCPriKey;
ECCPUBLICKEYCTX xdata pOtherECCPubKey;
ECCPUBLICKEYCTX xdata pAgreementKey;
```

unsigned char xdata selfPrikey[]={0x3a,0xc0,0xe7,0x17,0xeb,0x61,0x60,0x2e,0xfc,0xbb, 0x1d, 0xe8,0x1a,0xa1,0x44,0xa2,0x72,0xb4,0x4b,0xa1};

unsigned char xdata SelfPubkey[] = {0xdf,0xfa,0x23,0xf8,0x08, 0x51,0x48,0xf4,0x6e, 0x87,0x56,0xee,0x54,0x0a,0x5a,0xe1,0xf5,0x4c,0x16,0xfb,0x8e,0x34,0x50,0x5d,0x2d,0x08,0xc8 ,0x95,0xf7,0x23,0x30,0x80,0xf3,0xe1,0x07,0x8f,0x5d,0x60,0xaf,0x41};

unsigned char xdata OtherPrikey[] = {0x25,0xfb,0xb3,0x2e,0xfb,0xec,0x6e,0xcb,0x13,0x14, 0x33,0x2a,0x02,0x65,0x82,0xdb,0x7b,0xe0,0x0c,0x05};

unsigned char xdata OtherPubkey[] = {0x95,0xa4,0x13,0x6e,0xd7,0x7d,0x08,0x73,0x97, 0x07,0xf3,0x6c,0x2d,0x0a,0x6a,0x8c,0x38,0x17,0xc6,0x6e,0x25,0x84,0x4d,0xee,0xbd,0xad,0x1a ,0x7f,0xdf,0x13,0xda,0x1a,0xfa,0x51,0x4a,0xae,0xb1,0x2f,0x80,0xe2};

```
////////////////////////////////
pSelfECCPriKey.d = SelfPrikey;
pSelfECCPubKey.x = SelfPubkey;
pSelfECCPubKey.y = SelfPubkey+20;

pOtherECCPriKey.d = OtherPrikey;
pOtherECCPubKey.x = OtherPubkey;
pOtherECCPubKey.y = OtherPubkey+20;

pAgreementKey.x = space_for_test;
pAgreementKey.y = space_for_test+20;
```

```
    ret = ECC_ECDH(&ec_para_160, &pSelfECCPriKey, &pOtherECCPubKey,
&pAgreementKey);
    if (0 != ret)
    {
        return 1;
    }

    pAgreementKey.x = space_for_test+64;
    pAgreementKey.y = space_for_test+84;

    ret = ECC_ECDH(&ec_para_160, &pOtherECCPriKey, &pSelfECCPubKey,
&pAgreementKey);
    if (0 != ret)
    {
        return 1;
    }

    ret = memcmp(space_for_test, space_for_test+64, 40);
    if (0 != ret)
    {
        return 1;
    }

    return 0;
```

参阅

EC_PARA、ECCPRIKEYCTX、ECCPUBLICKEYCTX

附　录

附录 1　结构体定义说明

CSDK 中提供的算法函数使用过程中，需要定义相应的算法结构体，以下是需要被使用的结构体定义。

表 2　CSDK 算法结构体列表

算法类别	结构体名	使用函数
对称算法	BLOCKCIPHERPARAM	DESCrypt TDESCrypt SM1Crypt SM4Crypt
非对称算法 RSA	RSAPUBLICKEYCTX	RSAGenKeyPair RSAEncrypt RSAVerify
	RSAPRIVATEKEYCTX	RSAGenKeyPair RSADecrypt RSASign
	RSAPRIVATEKEYCRT	RSAGenKeyPair RSADecrypt RSASign
	RSAPRIVATEKEYSTD	RSAGenKeyPair RSADecrypt RSASign
	RSAKEYCTX	保留
非对称算法 SM2	SM2PUBLICKEYCTX	SM2GenKeyPair SM2Encrypt SM2EncryptInit SM2Verify
	SM2POINT	SM2EncryptInit SM2DecryptInit

	SM2PRIVATEKEYCTX	SM2GenKeyPair SM2Decrypt SM2DecryptInit SM2Sign
	SM2PLAINTEXT	SM2Encrypt SM2Decrypt
	SM2CIPHERTEXT	SM2Encrypt SM2Decrypt
	SM2SIGNATURE	SM2Sign SM2Verify
	SM2KEYEXCHANGEPARAM	SM2KeyExchange
	IDINFO	SM2GetZ
保留	BIGINTEGER	SM2EncryptFinal SM2DecryptFinal

BLOCKCIPHERPARAM

```
typedef struct Struct_BLOCKCIPHERPARAM
{
    unsigned char bOpMode;                  //  in
    unsigned char xdata *pbKey;             //  in
    unsigned char xdata *pbIV;              //  in
    unsigned char xdata *pbInput;           //  in
    unsigned char bTotalBlock;              //  in
    unsigned char xdata *pbOutput;          //  out
    unsigned char xdata *pbReserve;         //  reserved
}BLOCKCIPHERPARAM, *PBLOCKCIPHERPARAM;
```

成员

bOpMode

操作模式类型选择，根据当前所使用的对称算法进行选择。

Value	Description
SYM_ECB	ECB 模式
SYM_CBC	CBC 模式
SYM_CFB	CFB 模式
SYM_OFB	OFB 模式
SYM_CTR	CTR 模式
SYM_ENCRYPT	加密操作
SYM_DECRYPT	解密操作

pbKey

指向所使用的对称加密算法的密钥。

pbIV

如果对称算法使用 CBC/CFB 等模式进行加解密，则需要对此参数赋值，指向初始向量。

pbInput

输入数据的存放地址，加密时指向明文数据，解密时指向密文数据。

bTotalBlock

输入数据的分组数，分组长度由当前选用的对称算法决定，当该参数为 0x00 时，算法库内部按照 256 个分组执行。

pbOutput

输出数据的存放地址，加密时指向密文的存放地址，解密时指向明文的存放地址。

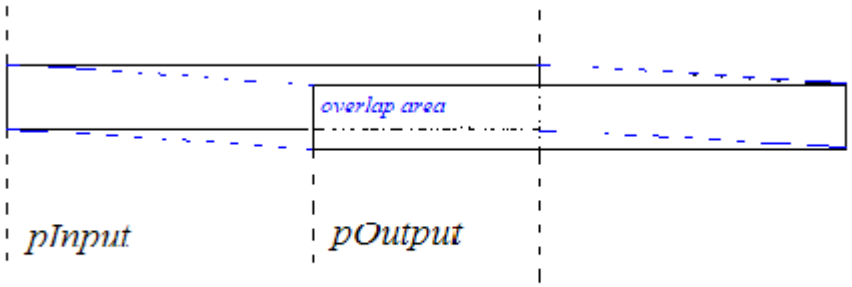
pbReserve

保留指针，留待以后扩展使用。

注意

对于对称算法来说，ECB 模式进行操作时，*pbInput* 可以与 *pbOutput* 相同；CBC 模式操作时，二者不能相同。

同时要求，*pbOutput* 指向的存储区域不可与 *pbInput* 指向的存储区域有前向重叠。下图所示情况，在对称算法进行 ECB 和 CBC 模式的加解密时是不允许的，否则不保证处理结果的正确性。



快速信息

头文件：声明在 iscrypt_symmetric.h

参阅

DESCrypt、TDESCrypt、SM1Crypt、SM4Crypt

RSAPRIVATEKEYCRT

```
typedef struct Struct_RSAPrivateKeyCRT
{
    unsigned char xdata *p;
    unsigned char xdata *q;
    unsigned char xdata *dP;
    unsigned char xdata *dQ;
    unsigned char xdata *qInv;
}RSAPRIVATEKEYCRT;
```

成员

p

使用 CRT 的 RSA 私钥形式中的素数 *p*。

q

使用 CRT 的 RSA 私钥形式中的素数 *q*。

dP

使用 CRT 的 RSA 私钥形式中素数 *p* 的 CRT 指数。

dQ

使用 CRT 的 RSA 私钥形式中素数 *q* 的 CRT 指数。

qInv

使用 CRT 的 RSA 私钥形式中的 CRT 协因子。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好，分配空间的大小需要根据当前使用的 RSA 位长决定。一般地，设所选 RSA 位长为 *n* 比特（指的是 RSA 的模长），则 CRT 的 RSA 私钥形式中的各成员所需的存储

空间分别为 *n*/2 比特。

快速信息

头文件：声明在 iscrypt_rsa.h

参阅

RSAPRIVATEKEYSTD、RSAPRIVATEKEYCTX、RSAGenKeyPair、RSA Decrypt、RSASign.

RSAPRIVATEKEYSTD

```
typedef struct Struct_RSAPrivateKeySTD
{
    unsigned char xdata *d;
    unsigned char xdata *n;
}RSAPRIVATEKEYSTD;
```

成员

d

使用标准的 RSA 私钥形式中的私钥指数 *d*。

n

使用标准的 RSA 私钥形式中的模数 *n*。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好，分配空间的大小需要根据当前使用的 RSA 位长决定。一般地，设所选 RSA 位长为 *n* 比特（指的是 RSA 的模长），则标准的 RSA 私钥形式中的各成员所需的存储空间分别为 *n* 比特。

快速信息

头文件：声明在 iscrypt_rsa.h

参阅

RSAPRIVATEKEYCRT、RSAPRIVATEKEYCTX、RSAGenKeyPair、RSA Decrypt、RSASign.

RSAPRIVATEKEYCTX

#define RSAPRIVATEKEYCTX void

该定义是为统一 RSA 的相关函数参数定义时所设置，并非单独的结构体，可根据所使用的 RSA 的模式查看相应的私钥形式。

参阅

RSAGenKeyPair、RSAKEYCTX 、RSADecrypt、RSASign.

RSAPUBLICKEYCTX

```
typedef struct Struct_RSAPublicKey
{
    unsigned char xdata *e;
    unsigned char xdata *n;
}RSAPUBLICKEYCTX;
```

成员

e

RSA 公钥中的指数 *e*。

n

RSA 公钥中的模数 *n*。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好，分配空间的大小需要根据当前使用的 RSA 位长决定。

公钥指数 *e* 的存储空间需要四个字节，需事先指定好，通常推荐的公钥指数选取 *e*=65537。

快速信息

头文件：声明在 iscrypt_rsa.h

参阅

RSAKEYCTX、RSAGenKeyPair、RSAEncrypt、RSAVerify

RSAKEYCTX

```
typedef struct Struct_RSAKey
{
    RSAPUBLICKEYCTX xdata *pRSAPubKey;
    RSAPRIVATEKEYCTX xdata *pRSAPriKey;
}RSAKEYCTX;
```

成员

pRSAPubKey

指向 RSA 公钥结构体 RSAPUBLICKEYCTX。

pRSAPriKey

指向 RSA 私钥结构体 RSAPRIVATEKEYCTX。

快速信息

头文件：声明在 iscrypt_rsa.h

参阅

RSAPRIVATEKEYCTX、RSAPUBLICKEYCTX、RSAGenKeyPair

SM2PUBLICKEYCTX

```
typedef struct Struct_SM2_PubKey
{
    unsigned char xdata *x;
    unsigned char xdata *y;
}SM2PUBLICKEYCTX, SM2POINT;
```

成员

x

SM2 公钥中的 *x* 坐标。

y

SM2 公钥中的 *y* 坐标。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好，分配空间的大小需要根据当前使用的曲线位长决定。SM2 使用的 256 位的素域曲线，因此各成员变量所需的存储空间为 256 比特。

快速信息

头文件：声明在 iscrypt_sm2.h

参阅

SM2GenKeyPair、SM2Encrypt、SM2EncryptInit、SM2DecryptInit、SM2Sign、SM2EncryptInit 、SM2GetZ

SM2PRIVATEKEYCTX

```
typedef struct Struct_SM2_PriKey
{
    unsigned char xdata *d;
}SM2PRIVATEKEYCTX;
```

成员

d

SM2 私钥标量 *d*。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好，分配空间的大小需要根据当前使用的曲线位长决定。SM2 使用的 256 位的素域曲线，因此各成员变量所需的存储空间为 256 比特。

快速信息

头文件：声明在 iscrypt_sm2.h

参阅

SM2GenKeyPair、SM2Decrypt、SM2Sign、SM2DecryptInit

SM2PLAINTEXT

```
typedef struct Struct_BigInteger
{
    unsigned char xdata *pd;
    unsigned char len;
}BIGINTEGER, SM2PLAINTEXT, IDINFO;
```

成员

pd

指向明文数据。

len

明文数据的长度。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好.

该结构体用于 SM2Encrypt 和 SM2Decrypt 函数中，仅针对 32 字节的数据的加解密。

快速信息

头文件：声明在 iscrypt_sm2.h

参阅

SM2Encrypt、SM2Decrypt、SM2EncryptFinal、SM2DecryptFinal、SM2GetZ

SM2CIPHERTEXT

```
typedef struct Struct_SM2_Ciphertext
{
    SM2POINT xdata *p;
    unsigned char xdata *c;
    unsigned char xdata *h;
    unsigned char clen;
}SM2CIPHERTEXT;
```

成员

p

指向 SM2 曲线上一点的结构体的指针，其形式与 SM2 公钥结构体一致。

c

密文数据的存储地址。

h

指向校验值。

clen

密文数据的长度，即成员 ***c*** 的长度。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好，分配空间的大小需要根据当前使用的曲线位长决定。SM2 使用的 256 位的素域曲线，因此各成员变量所需的存储空间为 256 比特。

该结构体用于 SM2Encrypt和 SM2Decrypt函数中，仅针对 32 字节的数据的加解密。

快速信息

头文件：声明在 iscrypt_sm2.h

参阅

SM2Encrypt、SM2Decrypt、SM2POINT

SM2SIGNATURE

```
typedef struct Struct_SM2_Signature
{
    unsigned char xdata *r;
    unsigned char xdata *s;
}SM2SIGNATURE;
```

成员

r

指向签名值 *r*。

s

指向签名值 *s*。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好，分配空间的大小需要根据当前使用的曲线位长决定。SM2 使用的 256 位的素域曲线，因此各成员变量所需的存储空间为 256 比特。

快速信息

头文件：声明在 iscrypt_sm2.h

参阅

SM2Sign、SM2Verify

SM2KEYEXCHANGEPARAM

```
typedef struct Struct_SM2_KeyExchange
{
    unsigned char role;
    unsigned char keyLenExpected;
    SM2PUBLICKEYCTX xdata *pSelfTempPubKey;
    SM2PRIVATEKEYCTX xdata *pSelfTempPriKey;
    SM2PUBLICKEYCTX xdata *pSelfPubKey;
    SM2PRIVATEKEYCTX xdata *pSelfPriKey;
    SM2PUBLICKEYCTX xdata *pOtherTempPubKey;
    SM2PUBLICKEYCTX xdata *pOtherPubKey;
    unsigned char xdata *pSelfZ;
    unsigned char xdata *pOtherZ;
}SM2KEYEXCHANGEPARAM;
```

成员

role

密钥交换中的角色，分为两种：

Value	Description
INITIATOR	发起方
RESPONDER	响应方

keyLenExpected

期望交换密钥的长度。

pSelfTempPubKey

己方临时公钥的结构体指针。

pSelfTempPriKey

己方临时私钥的结构体指针。

pSelfPubKey

己方固有公钥的结构体指针。

pSelfPriKey

己方固有私钥的结构体指针。

pOtherTempPubKey

对方临时公钥的结构体指针。

pOtherPubKey

对方固有公钥的结构体指针。

pSelfZ

指向己方的预计算数据 Z 值。

pOtherZ

指向对方的预计算数据 Z 值。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好，分配空间的大小需要根据当前使用的曲线位长决定。SM2 使用的 256 位的素域曲线，因此各成员变量所需的存储空间为 256 比特。

keyLenExpected 一般选择 8、16、32 三种长度，但支持不大于 128 字节的任意长度密钥截取。

*pSelfZ*和*pOtherZ*所指向的Z值需要事先调用 SM2GetZ进行计算，其长度均为 256 比特。

快速信息

头文件：声明在 iscrypt_sm2.h

参阅

SM2PUBLICKEYCTX、SM2PRIVATEKEYCTX、SM2KeyExchange、SM2GetZ

EC_PARA

```
typedef struct struct _EC_Para_st
{
    unsigned int bitLen;

    unsigned char p[32];
    unsigned char a[32];
    unsigned char b[32];
    unsigned char Gx[32];
    unsigned char Gy[32];
    unsigned char n[32];

    unsigned char mona[32];

    unsigned char pmc[8];
    unsigned char pr[32];
    unsigned char prr[32];

    unsigned char nmc[8];
    unsigned char nr[32];
    unsigned char nrr[32];
}EC_PARA, *pEC_PARA;
```

成员

bitLen

椭圆曲线的位比特长度，这里支持四种：160，192，224，256。

p

椭圆曲线的模数。

a,b

椭圆曲线常数。

Gx

椭圆曲线基点的 X 坐标。

Gy

椭圆曲线基点的 Y 坐标。

n

椭圆曲线的阶。

其他参数为算法内部使用，外部用户不需关心，此处不赘述。

注意

无。

快速信息

头文件：声明在 iscrypt_ecc.h

参阅

ECC_ECDH、ECC_ECES_Encrypt、ECC_ECDSA_Verify.

ECCPUBLICKEYCTX

```
typedef struct _EC_Point_st
{
    unsigned char xdata *x;
    unsigned char xdata *y;
} ECCPOINT, ECCPUBLICKEYCTX;
```

成员

x

ECC 公钥中的 *x* 坐标。

y

ECC 公钥中的 *y* 坐标。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好，分配空间的大小需要根据当前使用的曲线位长决定。

快速信息

头文件：声明在 iscrypt_ecc.h

参阅

ECC_Init、ECC_ECDH、ECC_ECDSA_Sign、ECC_ECDSA_Verify、ECC_ECES_Encrypt、ECC_ECES_Decrypt、ECC_PointMult.

ECCPRIKEYCTX

```
typedef struct _EC_Prikey_st
{
    unsigned char xdata *d;
} ECCPRIKEYCTX;
```

成员

d

ECC 私钥标量 *d*。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好，分配空间的大小需要根据当前使用的曲线位长决定。

快速信息

头文件：声明在 iscrypt_ecc.h

参阅

ECC_ECDSA_Sign、ECC_ECES_Decrypt、ECC_ECDH

ECCSIGNATURE

```
typedef struct Struct_ECC_Signature
{
    unsigned char xdata *r;
    unsigned char xdata *s;
}ECCSIGNATURE;
```

成员

r

指向签名值 *r*。

s

指向签名值 *s*。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好，分配空间的大小需要根据当前使用的曲线位长决定。

快速信息

头文件：声明在 iscrypt_ecc.h

参阅

ECC_ECDSA_Sign、ECC_ECDSA_Verify。

ECCPLAINTEXT

```
typedef struct Struct_BigInteger
{
    unsigned char xdata *pd;
    unsigned char len;
} ECCPLAINTEXT;
```

成员

pd

指向明文数据。

len

明文数据的长度。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好.

快速信息

头文件：声明在 iscrypt_ecc.h

参阅

ECC_ECES_Encrypt、ECC_ECES_Decrypt.

ECCCIPHERTEXT

```
typedef struct Struct_ECC_Ciphertext
{
    ECCPOINT xdata *p;
    unsigned char xdata *c;
    unsigned char xdata *h;
    unsigned char clen;
}ECCCIPHERTEXT;
```

成员

p

指向 ECC 曲线上一点的结构体的指针，其形式与 ECC 公钥结构体一致。

c

密文数据的存储地址。

h

预留。

clen

密文数据的长度，即成员 ***c*** 的长度。

注意

由于结构体成员均为指针，因此存储空间需要外部调用者事先分配好，分配空间的大小需要根据当前使用的曲线位长决定。

快速信息

头文件：声明在 iscrypt_ecc.h

参阅

ECC_ECES_Encrypt、ECC_ECES_Decrypt、ECCPUBLICKEYCTX.

附　表

附表 A　密码算法接口错误定义和说明

表 3　算法函数返回值定义说明

宏描述	预定义值	说明
SUCCESS	0x00	成功
FAIL	0x01	失败
UNKNOWN_ERROR	0x02	未知错误
NOT_SUPPORT_YET_ERROR	0x03	功能不支持的错误
NOT_INITIALIZE_ERROR	0x04	未初始化错误
OBJECT_ERROR	0x05	对象错误
MEMORY_ERROR	0x06	内存错误
IN_DATA_LENGTH_ERROR	0x07	输入数据错误
IN_DATA_ERROR	0x08	输入数据的长度错误
HASH_OBJECT_ERROR	0x09	哈希对象错误
HASH_ERROR	0x0A	哈希运算错误
HASH_NOT_EQUAL_ERROR	0x0B	哈希值不相等错误
OUT_OF_RANGE_ERROR	0x0C	消息数据超出约束范围错误
PARAMETER_ERROR	0x0D	无效的参数
NO_ROOM	0x30	空间不足