注意：该版本是临时版本，因为国密CA体系还没有上线，故所有使用到证书的地方都暂由RSA证书体系替换。

# 数据结构定义

## Flag定义

#define ALGID\_AES 0x00000001 // AES密钥

#define ALGID\_SM4 0x00000002 // SM4密钥

#define ALGID\_RSA\_PUB 0x00010100 // RSA公钥

#define ALGID\_RSA\_PRI 0x00020100 // RSA私钥

#define ALGID\_SM2\_PUB 0x00010200 // SM2公钥

#define ALGID\_SM2\_PRI 0x00020200 // SM2私钥

## KeyChain创建请求包

typedef struct \_KEYCHAIN\_CREATE\_REQ

{

unsigned int u32Magic; // 魔数，值为0x6b636363，后续数据结构与此相同

unsigned int u32Version; // 版本号，值为0x10000，后续数据结构与此相同

unsigned int u32Flags;

unsigned int u32TimeStamp; //时间戳，单位秒

unsigned char au8ID[16]; //KeyChain的ID

unsigned char au8KeyBagID[8]; //KeyBagID

unsigned char au8AccessCodePubKey[64]; //SM2公钥，前32个字节为X分量，后32个字节为Y分量

unsigned char au8Signature[256]; //Firmail服务器私钥对以上字段的签名，目前版本使用SM3withSM2签名

} KEYCHAIN\_CREATE\_REQ;

## KeyChain创建码

KeyChain创建码中的数据除签名部分外，其他字段数据从KeyChain创建请求包中原样拷贝即可。

typedef struct \_KEYCHAIN\_CREATE\_CODE

{

unsigned int u32Magic; // 魔数，值为0x6b636363，后续数据结构与此相同

unsigned int u32Version; // 版本号，值为0x10000，后续数据结构与此相同

unsigned int u32Flags;

unsigned int u32TimeStamp; //时间戳，单位秒

unsigned char au8ID[16]; //KeyChain的ID

unsigned char au8KeyBagID[8]; //KeyBagID

unsigned char au8AccessCodePubKey[64]; //SM2公钥，前32个字节为X分量，后32个字节为Y分量

unsigned char au8Signature[256]; //加密机服务器私钥对以上字段的签名，目前版本使用SM3withSM2签名

} KEYCHAIN\_CREATE\_CODE;

## 绑定验证码

typedef struct \_KEYBAG\_BIND\_CODE

{

unsigned int u32Magic;

unsigned int u32Version;

unsigned int u32Flags;

unsigned int u32TimeStamp;

unsigned char au8KeyBagID[8];

unsigned char au8PhoneNumber[16]; //手机号码，如不足16字节，后面以0填充

unsigned char au8BindCode[256]; //使用加密机证书加密的BindCode密文（SM2），其明文为16字节的ASCII字符，若不足16字节后面以0填充

unsigned char au8Signature[256]; //使用KeyBag设备私钥对以上字段的签名（SM3withSM2）

} KEYBAG\_BIND\_CODE;

## 绑定验证码校验包

typedef struct \_KEYBAG\_BIND\_CODE\_VERIFY

{

unsigned int u32Magic;

unsigned int u32Version;

unsigned int u32Flags;

unsigned int u32TimeStamp;

unsigned char au8KeyBagID[8];

unsigned char au8PhoneNumber[16]; //手机号码，如不足16字节，后面以0填充

unsigned char au8BindCode[16]; //BindCode明文，为16字节的ASCII字符，若不足16字节用0补齐

unsigned char au8Mac[32]; //使用Kmac密钥对以上字段进行HMAC-SM3运算得到的MAC值

} KEYBAG\_BIND\_CODE\_VERIFY;

## 创建Circle请求包

typedef struct \_KEYBAG\_CREATE\_CIRCLE\_REQUEST

{

unsigned int u32Magic;

unsigned int u32Version;

unsigned int u32Flags;

unsigned int u32TimeStamp;

unsigned char au8KeyBagID[8];

unsigned char au8PhoneNumber[16]; //手机号码，如不足16字节，后面以0填充

unsigned char au8BindCode[256]; //使用加密机证书加密的BindCode密文（SM2），其明文为16字节的ASCII字符，若不足16字节后面以0填充

unsigned char au8SyncPubKey[64]; //同步公钥（SM2），前32个字节为X分量，后32个字节为Y分量

unsigned char au8Signature[64]; //同步私钥对以上字段的签名（SM3withSM2），使用同步公钥验签（即au8SyncPubKey字段）

} KEYBAG\_CREATE\_CIRCLE\_REQUEST;

## Circle数据结构

typedef struct \_KEYBAG\_CIRCLE\_PUBKEY

{

unsigned char au8KeyBagID[8]; //KeyBagID

unsigned char au8SyncPubKey[64]; //KeyBag的同步公钥

} KEYBAG\_CIRCLE\_PUBKEY;

typedef struct \_KEYBAG\_CIRCLE

{

unsigned int u32Magic;

unsigned int u32Version;

unsigned int u32Flags;

unsigned int u32TimeStamp;

unsigned char au8PhoneNumber[16]; //手机号码，如不足16字节，后面以0填充

unsigned char au8Uuid[16]; //CircleID，绑定第一个keyBag时由加密机内部生成的一个UUID，如果是添加第二个第三个...时，则来自老Circle包中

unsigned int u32Count; //KeyBag同步公钥数组中元素个数

KEYBAG\_CIRCLE\_PUBKEY \*kcPubKey; //KeyBag同步公钥数组，一个Circle里可以有多个KeyBag

unsigned char au8Signature[256]; //加密机服务器私钥对以上字段的签名，目前版本使用SM3withSM2签名

} KEYBAG\_CIRCLE;

## 加入Circle审批包

typedef struct \_KEYBAG\_JOIN\_CIRCLE\_APPROVE

{

unsigned int u32Magic;

unsigned int u32Version;

unsigned int u32Flags;

unsigned int u32TimeStamp;

unsigned char au8KeyBagID[8];

unsigned char au8PhoneNumber[16];

unsigned char au8Uuid[16]; //老Circle的UUID

unsigned char au8BindCode[256]; //使用加密机证书加密的BindCode密文（RSA2048），其明文为16字节的ASCII字符，若不足16字节后面以0填充

unsigned char au8SyncPubKey[64]; //新添加进来的KeyBag的同步公钥（SM2）

unsigned char au8KeyBagIDApporver[8]; //审批的KeyBag的ID，需要通过该ID去老的Circle包中找到公钥来验签

unsigned char au8Signature[64]; //使用审批的KeyBag的同步私钥的签名

} KEYBAG\_JOIN\_CIRCLE\_APPROVE;

## 板卡初始化请求包

struct CHIP\_INIT\_REQ {

uint32 Version; //版本号

uint8 chipId[16]; //板卡ID

uint32 Flag; //用于签名的密钥类型、算法

uint32 bits; //用于签名的密钥bit长度

uint8 Signaute[256]; //使用板卡的设备私钥对以上数据的签名（SM3withSM2算法）

};

## 板卡初始化命令包

struct CHIP\_INIT\_CMD\_INNER {

uint8 chipId[16]; //板卡ID

uint8 Kseed[32]; //用于生成Kenc和Kmac的种子码

uint8 CryptorPri[64]; //加密机私钥（SM2）

uint32 Flag; //用于签名的密钥类型、算法

uint32 bits; //用于签名的密钥bit长度

uint8 Signaute[256]; //使用加密机私钥对以上字段的签名

};

struct CHIP\_INIT\_CMD {

uint32 Version; //版本号

uint32 Flag; //用于加密会话密钥的密钥类型、算法

uint32 bits; //用于加密会话密钥的密钥bit长度

uint32 sessionKeyFlag; //会话密钥类型、算法

uint32 sessionKeyBits; //会话密钥bit长度

uint8 sessionKeyCipher[256]; //会话密钥密文，使用板卡设备证书加密（SM2），此处会话密钥的明文为[IV+KEY],其中IV长度固定16字节,KEY长度为（sessionKeyBits/8）,下面是加密CHIP\_INIT\_CMD\_INNER的算法为SM4\_CBC

uint32 cmdCipherLen; // CHIP\_INIT\_CMD\_INNER的密文长度

uint8 cmdCipher[512]; //CHIP\_INIT\_CMD\_INNER的密文，使用会话密钥加密

};

## 板卡认证管理员锁数据包

struct AUTH\_ADM\_KEY\_INNER {

uint8 rand[32]; //随机数

uint8 Mac[32]; //使用Kmac密钥对以上字段计算的MAC值(HMAC-SM3)

};

struct AUTH\_ADM\_KEY {

uint32 Version; //版本号

uint32 Flag; // 密钥类型，算法

uint32 bits; // 密钥bit长度

uint32 cipherLen; //密文长度

uint8 cipher[128]; //使用Kenc对AUTH\_ADM\_KEY\_INNER加密得到的密文（SM4\_ECB算法）

};

## 管理员锁公共数据包

struct ADMIN\_KEY\_COMN {

uint32 Version; //

uint32 rsaRootPubLen; //RSA证书体系的根公钥长度

uint8 rsaRootPub[768]; //RSA证书体系的根公钥

uint32 sm2RootPubLen; //SM2证书体系的根公钥长度

uint8 sm2RootPub[64]; //SM2证书体系的根公钥

uint8 Hash[32]; //对以上字段的SM3 HASH值

}

## 管理员锁复制请求包

struct ADMIN\_KEY\_COPY\_REQ {

uint32 Version; //版本号

uint8 rand[32]; //随机数

uint8 hash[32]; //对(rand | 固定种子1 | 固定种子2)的SM3值

uint32 Flag; //用于签名的密钥类型、算法

uint32 bits; //用于签名的密钥bit长度

uint8 Signaute[256]; //使用空白管理员锁的设备私钥对以上数据的签名（SHA256withRSA算法）

};

## 管理锁复制命令包

struct ADMIN\_KEY\_COPY\_CMD\_INNER {

uint8 Kseed[32]; //用于生成Kenc和Kmac的种子码

uint8 CryptorPri[64]; //加密机私钥（SM2）

uint32 Flag; //用于签名的密钥类型、算法

uint32 bits; //用于签名的密钥bit长度

uint8 Signaute[256]; //使就绪管理员锁的设备私钥对以上数据的签名（SHA256withRSA算法）

};

struct ADMIN\_KEY\_COPY\_CMD {

uint32 Version; //版本号

uint32 CACertLen; //签发加密机证书的CA证书的长度

uint8 CACert[512]; //签发加密证书的CA的证书(SM2)

uint32 CryptorCertLen; //加密机证书长度

uint8 CryptorCert[512];//加密机证书

uint32 Flag; //用于加密会话密钥的密钥类型、算法

uint32 bits; //用于加密会话密钥的密钥bit长度

uint32 sessionKeyFlag; //会话密钥类型、算法

uint32 sessionKeyBits; //会话密钥bit长度

uint8 sessionKeyCipher[256]; //会话密钥密文，使用就绪管理员锁的设备证书加密（RSA），此处会话密钥的明文为[IV+KEY],其中IV长度固定16字节,KEY长度为（sessionKeyBits/8）,下面是加密ADMIN\_KEY\_COPY\_CMD\_INNER的算法为SM4\_CBC

uint32 cmdCipherLen; //ADMIN\_KEY\_COPY\_CMD\_INNER的密文长度

uint8 cmdCipher[376]; //ADMIN\_KEY\_COPY\_CMD\_INNER的密文，使用会话密钥加密

};

# 业务接口定义

## 创建KeyChain

/\*\*

\* 创建KeyChain

\* @param keychain\_req\_pkg KeyChain创建请求包

\* @param keychain\_req\_pkg\_len KeyChain创建请求包长度

\* @param ca\_cert 中级CA证书，这里应该是SoftwareCA证书

\* @param ca\_cert\_len 中级CA证书长度

\* @param cert Firmail服务器证书

\* @param cert\_len Firmail服务器证书长度

\* @param keybag\_id KeyBag ID

\* @param keychain\_create\_code\_pkg KeyChain创建码。调用方释放内存

\* @param keychain\_create\_code\_pkg\_len KeyChain创建码长度

\* @return

\*/

int kb\_gen\_keychain\_create\_code(

IN unsigned char \*keychain\_req\_pkg,

IN int keychain\_req\_pkg\_len,

IN unsigned char \*ca\_cert,

IN int ca\_cert\_len,

IN unsigned char \*cert,

IN int cert\_len,

IN char \*keybag\_id,

OUT unsigned char \*\*keychain\_create\_code\_pkg,

OUT int \*keychain\_create\_code\_pkg\_len);

说明：

1. 按数据结构解析KeyChain创建请求包
2. 使用预设的RootCA证书验证中级CA证书（SM3withSM2），使用中级CA证书验证Firmail服务器证书（SM3withSM2）
3. 使用Firmail服务器证书验证请求包中的签名（SM3withSM2）
4. 按照数据接口构造KeyChain创建码包
5. 把KeyBag ID填充到KeyChain创建码包中
6. 使用加密机私钥对KeyChain创建码包进行签名（SM3withSM2），并把签名结构附加到签名字段

## 签发BindCode

/\*\*

\* 签发BindCode

\* @param bind\_code\_pkg BindCode原始数据包

\* @param bind\_code\_pkg\_len BindCode原始数据包长度

\* @param ca\_cert 中级CA证书，这里应该是DeviceCA证书

\* @param ca\_cert\_len 中级CA证书长度

\* @param cert KeyBag的设备证书

\* @param cert\_len KeyBag设备证书长度

\* @param bind\_code\_plain BindCode明文，最大长度是可知的，由调用方自己声明缓冲区

\* @param phone\_number 电话号码，最大长度是可知的，由调用方自己声明缓冲区

\* @param bind\_code\_vrf\_pkg\_cipher BindCode校验包密文，调用方释放内存

\* @param bind\_code\_vrf\_pkg\_cipher\_len BindCode校验包密文长度

\* @return

\*/

int kb\_gen\_keybag\_bind\_code(

IN unsigned char \*bind\_code\_pkg,

IN int bind\_code\_pkg\_len,

IN unsigned char \*ca\_cert,

IN int ca\_cert\_len,

IN unsigned char \*cert,

IN int cert\_len,

OUT char \*bind\_code\_plain,

OUT char \*phone\_number,

OUT unsigned char \*\*bind\_code\_vrf\_pkg\_cipher,

OUT int \*bind\_code\_vrf\_pkg\_cipher\_len);

说明：

1. 按照数据接口解析BindCode数据包
2. 使用预设的RootCA证书验证中级CA证书（SM3withSM2），使用中级CA证书验证KeyBag设备证书（SM3withSM2）
3. 使用KeyBag设备证书验证BindCode数据包中的签名（SM3withSM2）
4. 使用加密机私钥解密BindCode数据包中的绑定码
5. 按照数据接口构造BindCode校验包，并把绑定码明文填充到校验包的绑定码字段
6. 使用Kmac密钥对校验包进行HMAC-SM3运算，把MAC值附加到校验包中
7. 使用Kenc对BindCode进行加密（SM4\_ECB）

## 创建Circle

/\*\*

\* 创建Circle

\* @param create\_circle\_req\_pkg 创建Circle请求包

\* @param create\_circle\_req\_pkg\_len 创建Circle请求包长度

\* @param bind\_code\_vrf\_pkg\_cipher BindCode校验包密文

\* @param bind\_code\_vrf\_pkg\_cipher\_len BindCode校验包密文长度

\* @param timestamp 时间戳

\* @param circle\_pkg Circle包，调用方释放内存

\* @param circle\_pkg\_len Circle包长度

\* @return

\*/

int kb\_create\_circle(

IN unsigned char \*create\_circle\_req\_pkg,

IN int create\_circle\_req\_pkg\_len,

IN unsigned char \*bind\_code\_vrf\_pkg\_cipher,

IN int bind\_code\_vrf\_pkg\_cipher\_len,

OUT unsigned int \*timestamp,

OUT unsigned char \*\*circle\_pkg,

OUT int \*circle\_pkg\_len);

说明：

1. 按照数据结构解析创建Circle请求包
2. 使用Circle请求包中的同步公钥校验Circle请求包的签名（SM2）
3. 使用Kenc解密BindCode校验包（SM4\_ECB）
4. 使用Kmac校验BindCode校验包的MAC（HAMC-SM3）
5. 比较Circle请求包中的绑定码是否等于BindCode校验包中的绑定码，如果不等则退出流程
6. 根据数据结构构造Circle包，使用当前时间的时间戳填充时间戳字段
7. 使用加密机私钥对Circle包进行签名（SM3withSM2）

## 加入Circle（更新Circle）

/\*\*

\* 加入Circle（更新Circle）

\* @param old\_circle\_pkg 旧的Circle包

\* @param old\_circle\_pkg\_len 旧的Circle包长度

\* @param join\_circle\_approve\_pkg 加入Circle审批包

\* @param join\_circle\_approve\_pkg\_len 加入Circle审批包长度

\* @param bind\_code\_vrf\_pkg\_cipher BindCode校验包密文

\* @param bind\_code\_vrf\_pkg\_cipher\_len BindCode校验包密文长度

\* @param timestamp 时间戳

\* @param new\_circle\_pkg 新的Circle包，调用方释放内存

\* @param new\_circle\_pkg\_len 新的Circle包长度

\* @return

\*/

int kb\_update\_circle(

IN unsigned char \*old\_circle\_pkg,

IN int old\_circle\_pkg\_len,

IN unsigned char \*join\_circle\_approve\_pkg,

IN int join\_circle\_approve\_pkg\_len,

IN unsigned char \*bind\_code\_vrf\_pkg\_cipher,

IN int bind\_code\_vrf\_pkg\_cipher\_len,

OUT unsigned int \*timestamp,

OUT unsigned char \*\*new\_circle\_pkg,

OUT int \*new\_circle\_pkg\_len);

说明：

1. 按照数据结构解析旧的Circle包
2. 使用加密机公钥证书校验Circle包的签名
3. 按照数据结构解析加入Circle审批包
4. 根据审批包中的审批者ID从Circle包中查找审批者的同步公钥，若找不到则退出流程
5. 使用审批者同步公钥校验加入Circle审批包的签名
6. 使用Kenc解密BindCode校验包（SM4\_ECB）
7. 使用Kmac校验BindCode校验包的MAC（HAMC-SM3）
8. 比较加入Circle审批包中的绑定码是否等于BindCode校验包中的绑定码，如果不等则退出流程
9. 比较旧的Circle包中的电话号码是否与审批包中的电话号码相同，如果不等则退出流程
10. 比较旧的Circle包中的时间戳是否大于审批包的时间戳，如果大于则退出流程
11. 按照数据结构构造新的Circle包，使用当前时间的时间戳填充时间戳字段，其他数据拷贝自旧的Circle包，将审批包中的新加入的KeyBagID和同步公钥加入到Circle公钥数组中
12. 使用加密机私钥对新的Circle包签名（SM3withSM2）

# 管理功能流程概述

本章节主要通过流程通概述管理功能的几个主要流程，更多细节在【[管理接口定义](#_管理接口定义)】章节中描述

## 板卡初始化

Viso文件过大，不引入文档，见《[板卡初始化.vsdx](file:///C:\Users\m4000e\AppData\Roaming\Microsoft\Word\板卡初始化.vsdx)》

## 管理员锁复制

Viso文件过大，不引入文档，见《[复制管理员锁.vsdx](C:\\Users\\m4000e\\AppData\\Roaming\\Microsoft\\Word\\复制管理员锁.vsdx)》

# 管理接口定义

## 一组获取板卡基本信息的接口

* 板卡状态

#define CHIP\_STATUS\_BLANK 0x00000000

#define CHIP\_STATUS\_ALREADY 0x00000001

* 板卡ID

如有其它需要再补充

## 初始化管理员锁（只有第一把管理员锁需要本章节的流程）

### 初始化-预处理

int init\_prepare(

(OUT)uint8 \*sm2\_csr,

(OUT)uint32 \*sm2\_csr\_len);

说明：

1. 检查锁的状态是否为空白状态，如果否则报错退出流程
2. 生成一对SM2公私钥对，作为加密机公私钥对，并且生成CSR返回

外部得到CSR后需要向CA申请一个证书，在初始化阶段下载到锁内

### 初始化

int init(

(IN)uint8 \*rsa\_root\_cert,

(IN)uint32 \*rsa\_root\_cert\_len,

(IN)uint8 \*sm2\_root\_cert,

(IN)uint32 \*sm2\_root\_cert\_len,

(IN)uint8 \*sm2\_ca\_cert,

(IN)uint32 \*sm2\_ca\_cert\_len,

(IN)uint8 \*sm2\_cryptor\_cert,

(IN)uint32 \*sm2\_cryptor\_cert\_len);

说明：

1. 检查锁的状态是否为空白状态，如果否则报错退出流程
2. 解析RSA根证书，保存根公钥
3. 解析SM2根公钥，保存根公钥
4. 使用SM2根公钥校验SM2中级CA证书，保存中级CA证书
5. 使用SM2中级CA证书校验加密机证书，保存加密机证书
6. 使用锁内加密机私钥对一个随机数签名，检查是否可以用加密机证书正确验签
7. 随机生成内部密钥1，并保存
8. 随机生成Kseed，并使用内部密钥1加密保存(SM4\_ECB)
9. 使用Kseed生成并保存Kenc和Kmac,生成方法如下：

Kenc = SM3(Kseed | 固定种子1)

Kmac = SM3(Kseed | 固定种子2)

根据所需密钥长度，取哈希结果前N个字节作为密钥

其中固定种子1和固定种子2，板卡和管理员锁中必须设置为一致

其中Kenc为用于加密的密钥；Kmac为用于计算MAC的密钥（Kmac使用内部密钥1加密保存为二进制文件）

1. 设置管理员锁为已就绪

## 板卡初始化

### 从板卡获取初始化请求包

int getInitReq(

(OUT)CHIP\_INIT\_REQ req,

(OUT)uint8 \*ca\_cert,

(OUT)uint32 \*ca\_cert\_len,

(OUT)uint8 \*cert,

(OUT)uint32 \*cert\_len);

功能：获取板卡初始化请求包，同时获取板卡的设备证书和DeviceCA证书（SM2体系证书）

说明：

1. 板卡判断当前状态，如果当前已是就绪状态，则返回错误，退出流程
2. 按照数据结构定义组装板卡初始化请求包
3. 使用板卡设备私钥对请求包签名(SM3withSM2)，并将签名填充到请求包的签名字段
4. 读取DeviceCA证书和板卡设备证书
5. 返回板卡初始化请求包、DeviceCA证书、板卡设备证书

### 就绪管理员锁校验证书

int checkCert(

(OUT)uint8 cert\_type,

(OUT)uint8 \*cert,

(OUT)uint32 \*cert\_len);

功能：校验证书

说明：

管理员锁依次校验从板卡中获得的DeviceCA证书、板卡设备证书

用预设的SM2根公钥校验DeviceCA证书通过后，将其缓存在内存中

用内存缓存中的DeviceCA证书校验板卡设备证书通过后，将其缓存在内存中，用于后续校验初始化请求包中的签名

### 从就绪管理员锁获取初始化命令包

int getInitCMD(

(IN)CHIP\_INIT\_REQ req,

(OUT)CHIP\_INIT\_CMD cmd);

功能：从管理员锁中获取板卡初始化命令包

说明：

1. 使用板卡设备证书验证请求包中的签名
2. 使用内部密钥1解密Kseed（SM4\_ECB）
3. 按照数据结构定义构造CHIP\_INIT\_CMD\_INNER数据包，使用管理员锁中的加密机私钥签名并把签名填充到数据包的签名字段（SM3withSM2）
4. 随机生成一个会话密钥（IV+KEY），使用会话密钥加密CHIP\_INIT\_CMD\_INNER数据包(SM4\_CBC)
5. 使用板卡设备证书加密会话密钥(SM2)
6. 按照数据结构定义构造CHIP\_INIT\_CMD数据包
7. 返回命令包

### 读取管理员锁内的SM2体系证书

int getSM2Cert(

(IN)uint8 cert\_type,

(OUT)uint8 \*cert,

(OUT)uint32 cert\_len);

说明：

现需要自己再实现一个接口用于读取保存在管理员中的加密机证书及其上级CA证书

### 板卡执行初始化命令包

int init(

(IN)CHIP\_INIT\_CMD cmd,

(IN)uint8 \*ca\_cert,

(IN)uint32 ca\_cert\_len,

(IN)uint8 \*cert,

(IN)uint32 cert\_len);

功能：板卡执行初始化命令包

说明：

1. 使用板卡设备私钥解密命令包中的会话密钥(SM2),注意解密出来的密钥实际是IV+KEY,前16字节是IV,后32字节为密钥
2. 使用会话密钥解密CHIP\_INIT\_CMD\_INNER密文包（SM4\_CBC）
3. 使用预设SM2根证书验证传入的CA证书
4. 使用CA证书验证传入的加密机证书
5. 使用加密机证书验证CHIP\_INIT\_CMD\_INNER包中签名
6. 对比数据包中chipId是否与当前板卡的chipId一致
7. 使用命令包中的加密机私钥对一个随机数进行签名，校验加密机证书是否能够正确验签
8. 保存加密机私钥和加密机证书
9. 使用数据包中的Kseed生成并保存Kenc和Kmac,生成方法如下：

Kenc = SM3(Kseed | 固定种子1)

Kmac = SM3(Kseed | 固定种子2)

根据所需密钥长度，取哈希结果前N个字节作为密钥

其中固定种子1和固定种子2，板卡和管理员锁中必须设置为一致

其中Kenc为业务接口中用于加密的密钥；Kmac为业务接口中用于计算MAC的密钥

1. 保存Kenc和Kmac（注意：不保存Kseed）
2. 将板卡状态设置为已就绪

## 管理员锁复制

### 获取管理员锁状态接口

* 管理员锁状态

#define ADM\_KEY\_STATUS\_BLANK 0x00000000

#define ADM\_KEY\_STATUS\_ALREADY 0x00000001

### 从板卡获取认证管理员锁数据包

int getAuthPkg(

(OUT)AUTH\_ADM\_KEY pkg);

功能：从管理员锁获取板卡认证管理员锁数据包

说明：

1. 判断当前板卡是否已就绪，否则报错退出流程
2. 板卡内生成随机数
3. 使用Kmac对随机数计算MAC（HMAC-SM3算法），然后按数据结构定义组装AUTH\_ADM\_KEY\_INNER数据包
4. 使用Kenc对AUTH\_ADM\_KEY\_INNER数据包进行加密（SM4\_ECB算法）
5. 按照数据结构定义组装AUTH\_ADM\_KEY数据包，并返回

### 就绪管理员锁认证认证数据包

int auth(

(IN)AUTH\_ADM\_KEY pkg);

功能：管理员锁认证从板卡获得的认证数据包

说明：

1. 使用Kenc解密密文字段（SM4\_ECB算法），得到AUTH\_ADM\_KEY\_INNER包
2. 使用内部密钥1解密Kmac，使用Kmac对rand字段结算MAC（HMAC-SM3算法），并与数据包中的MAC字段对比，是否相同，如果不同返回认证失败，退出流程
3. 相同，则设置管理锁已认证标识为true，该标识保存在内存中，掉电即被重置

### 从就绪管理员锁获取管理员锁公共数据包

int getAdminComn(

(OUT)ADMIN\_KEY\_COMN \*pkg);

### 空白管理员锁下载管理员锁公共数据包

int setAdminComn(

(IN)ADMIN\_KEY\_COMN pkg);

说明：

1. 检查管理员锁是否处于空白状态
2. 校验公共数据包的HASH值（SM3）
3. 保存RSA根公钥和SM2根公钥

### 从空白管理员锁获取复制请求包

int getCopyReq(

(OUT) ADMIN\_KEY\_COPY\_REQ req);

功能：从空白管理员锁获取复制请求包

说明：

1. 检查当前管理员锁是否处于空白状态，否则报错退出流程
2. 随机生成rand
3. 按照数据结构定义计算（rand | 固定种子1 | 固定种子2）的SM3哈希值
4. 使用管理员锁设备私钥对数据包签名
5. 按照数据结构定义组装请求包并返回

### 从就绪管理员锁获取复制命令包

int getCopyCmd(

(IN) ADMIN\_KEY\_COPY\_REQ req,

(OUT) ADMIN\_KEY\_COPY\_CMD cmd);

功能：从就绪管理员锁获取复制命令包。在此之前要先校验空白管理员锁的DeviceCA证书、设备证书，并把设备证书缓存在锁内

说明：

1. 判断当前管理员锁是否是已就绪状态，否则报错退出流程
2. 判断当前管理员锁的认证标识是否为true，否则报错退出流程
3. 使用内存缓存中的空白管理员锁设备证书校验请求包中的签名
4. 按照数据结构定义验证请求包中的hash值
5. 使用内部密钥1解密Kseed（SM4\_ECB），并按照数据结构组装ADMIN\_KEY\_COPY\_CMD\_INNER包
6. 使用就绪管理员锁的设备私钥对ADMIN\_KEY\_COPY\_CMD\_INNER包签名（SHA256withRSA）
7. 随机生成会话密钥（IV+KEY），并使用会话密钥加密ADMIN\_KEY\_COPY\_CMD\_INNER包（SM4\_CBC）
8. 使用空白管理员锁的设备证书加密会话密钥(RSA)
9. 按照数据结构定义组装ADMIN\_KEY\_COPY\_CMD包并返回

### 空白管理员锁执行复制命令包

int execCopyCmd(

(IN) ADMIN\_KEY\_COPY\_CMD cmd);

功能：空白管理员锁执行复制命令包。在此之前要先校验就绪管理员锁的DeviceCA证书、设备证书，并把设备证书缓存在锁内

说明：

1. 使用空白管理员锁设备私钥解密会话密钥（IV+KEY）（RSA)
2. 使用会话密钥解密得到ADMIN\_KEY\_COPY\_CMD\_INNER数据包（SM4\_CBC）
3. 使用就绪管理员锁设备证书校验ADMIN\_KEY\_COPY\_CMD\_INNER数据包中的签名(SHA256withRSA)
4. 使用SM2根公钥验证中级CA证书、使用中级CA证书验证加密机证书
5. 使用加密机私钥对一个随机数进行签名，检查加密机证书是否能正确进行验签（SM3withSM2）
6. 保存SM2根公钥、中级CA证书、加密机私钥和加密机证书
7. 随机生成空白管理员锁的内部密钥1，并保存
8. 使用内部密钥1加密保存Kseed
9. 使用数据包中的Kseed生成并保存Kenc和Kmac,生成方法如下：

Kenc = SM3(Kseed | 固定种子1)

Kmac = SM3(Kseed | 固定种子2)

根据所需密钥长度，取哈希结果前N个字节作为密钥

其中固定种子1和固定种子2，板卡和管理员锁中必须设置为一致

其中Kenc为用于加密的密钥；Kmac为用于计算MAC的密钥

1. 设置管理员锁为已就绪