S32K1xx 系列 MCU 应用指南之 CSEc 硬件加密模块使用手册

By：China AMP GPIS AE

修改记录

版本	日期	修改人	修改内容
Rev_1.0	2019.06.25	Yunchuan.wang@nxp.com	初始版本，包含完整应用指南内容，内部审核。
Rev_2.0	2019.07.22	Yunchuan.wang@nxp.com	第二次内部审核版本。
Rev_2.1	2019.08.01	Yunchuan.wang@nxp.com	发布版本，修改了文中笔误和有歧义的地方。
Rev_2.2	2019.08.08	Yunchuan.wang@nxp.com	更新了安全引导相关内容。

目录

1 前言

S32K1xx 的 CSEc 模块是一个硬件加密模块，全称 Cryptographic Service Engine – Compressed，符合 HIS-SHE specification 1.1 rev 439 和 GM-SHE+ 安全规范标准，本文不对这些规范本身做任何介绍，若想了解更多协议本身的内容，请参考相关协议规范。

S32K1xx 的 CSEc 硬件加密模块（以下统一简称 CSEc 模块）在汽车电子系统中运用非常广泛，下面为其中的 5 中典型应用:

1) **里程数据保护：**在存储里程数据前，通过使用 CSEc 模块加密，然后将加密后的数据存入到 EEPROM 或者 Flash 等存储设备中，当需要使用时，取出这段加密后的数据，再通过 CSEc 模块解密，这样能够对里程等重要数据进行保护。

2) **汽车防盗：** 通过 CSEc 模块的真随机数和加密解密功能，就可以实现一个简单的汽车防盗功能。当任意设备尝试开启汽车时，均需要通过 CSEc 模块产生一个随机数，并通过仅双方已知的密钥进行加密后发送给汽车，只有使用了正确的密钥时，汽车端才能解密出正确的数据，从而达到防盗的效果。

3) **电子组件验证和保护：**CSEc 模块中固化了一个 15 个字节长度的唯一 UID（Unique ID），可以通过这个 UID 实现汽车电子组件的保护和验证，防止其他未经验证的设备接入到汽车中，同时也能防止汽车部件非法的流入到零部件市场中。

4) **固件更新保护：**CSEc 模块通过对消息认证码（CMAC）计算从而支持安全的 Flash 编程。应用程序通过计算接收到的固件数据的 CMAC 值并与离线计算的固件的 CMAC 值做比较，来验证需要更新的每个程序块。只有在使用相同的密钥进行 CMAC 计算时，两者才会相同，此验证才会通过。详情请参考应用笔记 AN4235-Using CSE to protect your application via a circle of trust。

5) **通信安全保护：**当中央 ECU(Electronic Control Unit，电控单元)需要与任何其他 ECU 通信时，中央 ECU 将其 CSEc 模块生成的随机数发送给另一个 ECU。另一个 ECU 接收随机数，将其与数据一起加密，并将加密的消息返回给中央 ECU。中央 ECU 对收到的数据进行解密并将接收到的随机数与之前生成的随机数进行比较，以验证其正确性。随机数可以防止连续的攻击，通信加密能够防止通信被窃听，因此，整个机制结合确保了数据的完整性和真实性。

在正式介绍 S32K1xx 系列的 CSEc 模块前，先对本文用到的术语进行统一的介绍，如下表，在正文中第一次出现时，也会进行注解。

CSEc	Cryptographic Service Engine – Compressed
HIS	Hersteller Initiative Software
SHE	Secure Hardware Extension
AES	Advanced Encryption Standard
CMAC	Cipher based message authentication code
ECB	Electronic Code Book
CBC	Cipher Block Chaining
FTFC	Flash Memory Module of S32K1xx devices
CCOB/ FCCOB	Flash Common Command Object
PRAM	Parameter space Random Access Memory
SRAM	System Random Access Memory
PGMPART	Program Partition Command

EEEPROM **Emulated-EEPROM**	Emulated Electrically Erasable Programmable Read-Only Memory
RNG	Random Number Generator
PRNG	Pseudo Random Number Generator
TRNG	True Random Number Generator
SFE	Security Flag Extension

2 S32K1xx 系列 MCU 的 CSEc 硬件加密模块介绍

2.1 CSEc 硬件加密模块功能概述

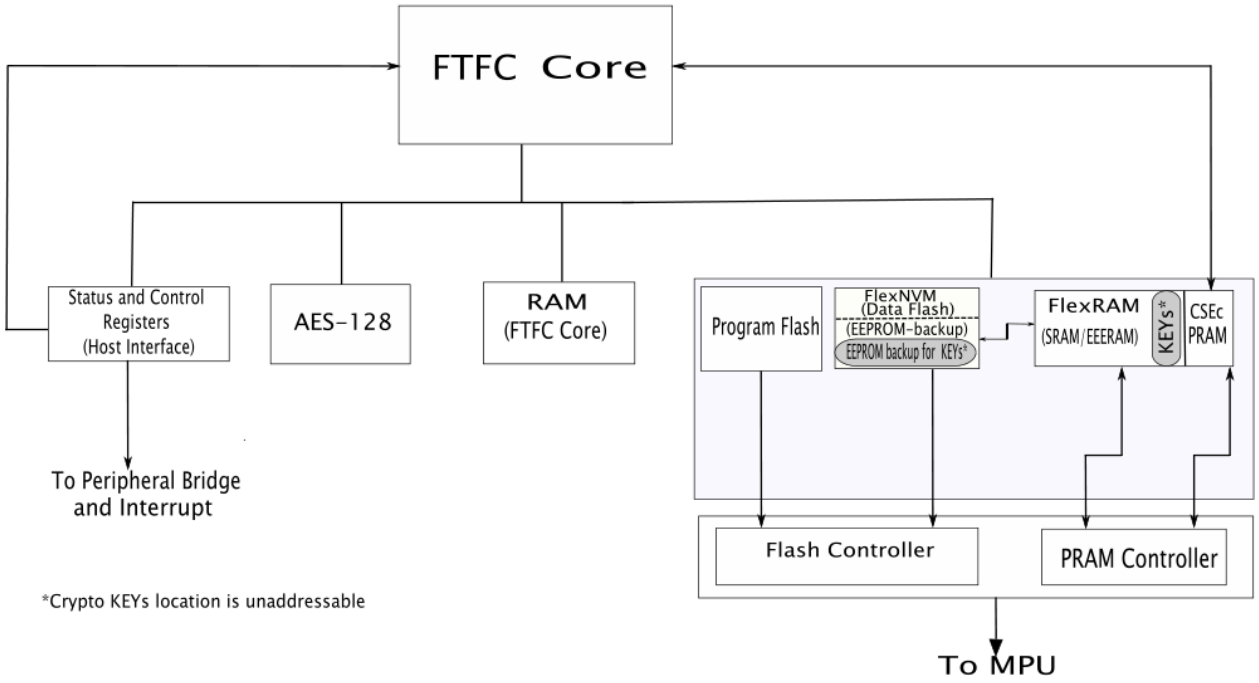
S32K1xx 系列 MCU 的 CSEc 硬件加密模块具备以下的功能：

- 符合 HIS-SHE specification 1.1 rev 439 和 GM-SHE+ 安全规范标准。
- 具备多达 17 个的用户密钥存储。
- 支持 AES(Advanced Encryption Standard)-128 算法加密和解密。
- 支持 AES-128 CMAC (Cipher-based Message Authentication Code)生成和验证。
- 支持 ECB (Electronic Cypher Book) 模式的加密和解密。
- 支持真假随机数的生成。
- 提供 Miyaguchi-Preneel 压缩算法 API。
- 支持三种安全引导（Secure Boot）模式：
  - 顺序安全引导模式（Sequential Boot Mode）。
  - 严格的顺序安全引导模式（Strict Sequential Boot Mode）（设置后不能更改）。
  - 并行安全引导模式（Parallel Boot Mode）。

注意：CSEc 模块并不对存储在 Flash 中的数据和代码进行保护，如需对 Flash 中的数据和代码进行加密保护，请参考 Flash 安全（Flash security）相关文档。

2.2 CSEc 模块与 FTFC 模块的关系

CSEc 模块包含在 FTFC（Flash Memory Module）模块内，Flash 模块和 CSEc 模块均由 FTFC 模块控制管理，CSEc 模块和 FTFC 模块关系如下图所示：



上图中的各个模块功能如下：

- FTFC Core：FTFC 模块的核心，包含硬件核心和 FTFC 的软件算法。
- Host Interface：系统与 FTFC 模块的通信接口。
- AES-128：AES-128 为一个硬件子模块，提供 AES-128 加密、解密算法和 CMAC 生成、验证算法。
- RAM(FTFC Core)：一块对用户不可见的 RAM 区域，用于提高 FTFC Core 的算法性能。
- Flash Controller：提供访问 Flash 的接口，由于 CSEc 的密钥是存储在 E-Flash（模拟 EEPROM）中，并且使用安全引导功能的时候，会对 Program Flash 中的数据进行 CMAC 验证，所以需要 Flash 控制器提供访问 Flash 的接口。
- PRAM(Parameter space Random Access Memory) Controller：应用程序与 CSEc 模块的通信接口，应用程序加载命令和数据至 FTFC 模块、FTFC 模块返回操作结果和数据至应用程序均通过 PRAM 接口实现。
- Program Flash：程序存储器，用于存储应用程序代码。
- FlexNVM：可配置的 Flash 存储器，可被配置为正常的 Flash 存储器用于存储数据或应用代码，也可被配置为模拟 EEPROM 存储器，作为 EEPROM 的数据保存区域。
- FlexRAM：可灵活配置的 RAM，可被配置为正常的 RAM，在程序运行中和 SRAM(System Random Access Memory)一样的被使用，也可被配置为模拟 EEPROM 存储器。由于 RAM 掉电数据即会被丢失，所以当 FlexRAM 被配置为模拟 EEPROM 时，需要 FlexNVM 配置出一块区域作为模拟 EEPROM 的数据保存区域与 FlexRAM 共同配合才能完成模拟 EEPROM 的功能，当写入和读出数据时，只需要按照操作 RAM 的方式，硬件会自动同步 FlexRAM 和 FlexNVM 中的模拟 EEPROM 数据。

通过 Host Interface，可使用 FTFC 模块的分区操作命令 PRGPART(Program Partition Command)，根据用户需要分配模拟 EEPROM 的大小，包括模拟 EEPROM 在 FlexRAM 中和在 FlexNVM 中的大小。执行 FTFC

的命令通过 FCCOB(Flash Common Command Objects)寄存器实现。

由于 CSEc 模块的密钥（Key）是存储在模拟 EEPROM 的区域，所以如果要使用 CSEc 模块的功能，芯片必须配置模拟 EEPROM 分区，通过 PRGPART 命令配置模拟 EEPROM 分区大小和存储在模拟 EEPROM 中 CSEc 模块密钥数量。模拟 EEPROM 的最后 128/256/512 bytes 区域则为存储 CSEc 模块密钥的区域，这段区域是不可见的。并且不能被通过 CSEc 模块以外的其他方式读取、擦除和更改。

根据 HIS-SHE 规范，在芯片连接调试器的情况下，CSEc 的一些功能会被关闭（详情请参考 HIS-SHE 规范文档）。一旦用户配置了 CSEc 模块并加载了用户密钥，芯片就可以进行 HIS-SHE 和 GM-SHE+规范中所述的任何安全相关操作。

CSEc 模块的 PRAM 接口是用于支持安全功能相关操作的命令（Command Header）和数据交互，应用程序通过 PRAM 传输命令和数据到 CSEc 模块，当 CSEc 模块完成操作后，再通过 PRAM 返回数据到应用程序。传入到 PRAM 的数据大小均为 128 位，如果实际数据小于 128 位，则需要由应用程序填充。不过，由于 CMAC 相关操作会在 CSEc 模块内部填充，所以进行 CMAC 相关操作时不需要填充。当 CSEc 命令头被写入时，CSEc 模块就会开始执行命令，并锁住 CCOB 接口、EEERAM 和 CSEc PRAM 接口。在 CSEc 模块上一条命令执行完之前，不能再启动其他的命令，可通过查看 Flash 状态寄存器（FSTAT）和 CSEc 状态寄存器（FCSESTAT）获取 CSEc 模块的状态。也可以设置 CSEc 模块命令执行完成后触发中断，命令执行期间发生的任何错误都会记录在 CSEc Pram 的错误标志位上。

注意：

1、当 MCU 处于 VLPR（极低功率运行）和 HSRUN（高速运行）模式时，不能使用 CCOB 或 CSEc 相关命令，CCOB 和 CSEc 命令操作只能在处理器处于 RUN（运行）模式时执行。建议在切换到任何模式之前，必须等待所有命令执行完成（可通过轮询 FSTAT[CCIF]标志检查）。如果 MCU 在 HSRUN/VLPR 模式下运行，并且需要进行 CCOB/CSEc 操作，则必须先切换到 RUN（运行）模式才能进行写入操作，完成后，用户可以切换回 HSRUN/VLPR 模式。

2、 不能同时执行 CCOB 命令（编程、擦除 Flash 等命令）和 CSEc 模式的命令。

3、 在执行的 CSEc 模块命令期间，不能再执行不同的 CSEc 模块命令。

4、 在执行 CSEc 模块命令的过程中发出 CCOB 命令时，会导致正在执行的 CSEc 命令被取消。

5、 当处于擦除挂起（ERSSUSP）状态时执行 CSEc 模块命令将导致挂起的擦除操作中止并无法恢复。

6、 开始执行 CCOB 或 CSEc 模块命令时 CCOB、EEERAM 和 CSEc PRAM 接口会被锁定，直到 CCOB 或 CSEc 命令完成后，才会被解锁。

2.3 CSEc 模块的密钥详解

本章节对 CSEc 模块的密钥进行详细介绍，主要包含 CSEc 模块的密钥类型、密钥 ID、密钥计数器和密钥的属性。

2.3.1 CSEc 模块密钥概述

在进行加密和解密数据操作时，需要使用到加密解密密钥。CSEc 模块一共可存储 24 个密钥，CSEc 模块的密钥和 RAM_KEY 如下表。

CSEc 模块的密钥

密钥名	密钥 ID		存储位置	密钥长度（字节）	密钥计数器长度（位）	密钥属性						出厂状态
	KBS	KEY IDs				写保护	Boot 保护	调试保护	密钥用途	通配符	仅用于 MAC 验证	
Secret_Key	X	0x0	ROM	16	-	-	√	√	-	-	-	NXP 定义
UID	X	0x0	ROM	15	-	-	-	-	-	-	-	NXP 定义
MASTER_ECU_KEY	X	0x1	Non-Volatile	16	28	√		√		√		空白
BOOT_MAC_KEY	X	0x2	Non-Volatile	16	28	√		√		√		空白
BOOT_MAC	X	0x3	Non-Volatile	16	28	√		√		√		空白
KEY_01-KEY_10	1'b0	0x4-0xD	Non-Volatile	16	28	√	√	√	√	√	√	空白
KEY_11-KEY_17	1'b1	0x4-0xA	Non-Volatile	16	28	√	√	√	√	√	√	空白
Reserved	1'b1	0xE	-	16	-	-	-	-	-	-	-	空白
Reserved	1'b0	0xE	-	-	-	-	-	-	-	-	-	空白
RAM_KEY	X	0xF	Volatile	16	-	-	-	-	-	-	-	空白

其他密钥			
密钥名	**地址**	**存储类型**	**长度（字节）**
PRNG_KEY	N/A	RAM	16
PRNG_STATE	N/A	RAM	16

注:“√”代表可以使用的属性

FTFC 模块为 HIS-SHE 功能规范中描述的加密解密密钥提供了安全可靠的存储。如上表所示的前五个密钥有专用功能，其余的密钥为应用程序中所使用的用户密钥。RAM_KEY 具备明文密钥的属性，需要使用明文密钥加密时，通过加载密钥至 RAM_KEY 中实现。RAM_KEY 以明文形式存储在 RAM_KEY 区域中。但 RAM_KEY 区域对用户仍然是不可见的。下面对 CSEc 模块中的所有密钥进行详细解释:

- SECRET_KEY：在器件制造过程中写入的一个随机值，其值不可见且不能更改，在内部用于生成特殊的密钥，如伪随机数生成器的密钥（PRNG_KEY）。
- UID：唯一标识号对于每个芯片都是唯一的，并且在出厂时被写入到芯片中，可通过 CSEc 模块相关命令查询到这个值，但是不能更改。
- MASTER_ECU_KEY：可以使用 MASTER_ECU_KEY 作为授权密钥，用于更新其他所有的用户密钥和恢复 CSEc 模块的密钥至出厂状态。
- BOOT_MAC_KEY：用于 Secure boot（安全引导）功能中计算验证 Bootloader 代码的 MAC 值。

- BOOT_MAC：用于 Secure boot（安全引导）功能，为 Bootloader 的 MAC 值，这个值可以通过 CSEc 模块自动加载也可以用户手动加载更新。
- KEY_01 ~ KEY_17：存储在 EEERAM 区域的 17 个用户可配置密钥，实际使用的密钥数量可以在进行 D-Flash 分区时指定。
- RAM_KEY：存储在 RAM 区域的可随时更改的密钥，可用于任何需要的操作。虽然 RAM_KEY 的存储区域用户也不可见，但是这个区域并不受 CSEc 模块的控制，所以不够安全。
- PRNG_KEY 和 PRNG_STATE：随机数生成器内部使用的密钥，用户不能访问。

2.3.2 CSEc 模块密钥 ID

每个密钥都有一个自己的标识号 Key ID，Key ID 由两部分组成，KBS（密钥块编号）和 Key ID，在使用、更新或授权更新密钥时，CSEc 模块都是通过 Key ID 来决定使用存储在存储器中的哪一个密钥。由于 S32K1xx 系列 MCU CSEc 模块的密钥比 HIS-SHE 标准中指定的要多，所以引入了 KBS 进行密钥所在块的选择，使用 KBS 和 Key ID 就可以对所有的 17 个密钥进行索引。

其中密钥 1~密钥 10 在块 0（Bank0）区域，密钥 11~密钥 17 在块 1（Bank1）区域。密钥 1 的 Key ID 则为 0x04（0x0,0x4），密钥 11 的 Key ID 则为 0x14（0x1,0x4）。高 4 位代表所在 KBS，低四位代表 Key ID。

2.3.3 CSEc 模块密钥计数器

每个用户密钥都有一个 28 位的计数器（Key Counter）用于记录每次更新。每次更新密钥时都必须增加此计数器值。在更新密钥时，这个计数值会被加入到计算中，如果更新密钥时没有同时更新计数器的值，将会导致出错。

2.3.4 CSEc 模块密钥属性

每个密钥都有 6 个可配置属性。这些属性决定了如何使用以及在什么条件下能够使用该密钥。这些属性将会在加载、更新密钥的时候用到。

1) 写保护 (WRITE_PROT)

当设置了写保护功能（置 1），则这个密钥不允许再被改变。所以，在设置此功能时，必须非常的小心，因为此操作不可逆，设置此操作后，则任何方式都无法再对此密钥进行更改，删除。甚至连通过恢复 CSEc 模块出厂状态的命令来擦除该密钥都不会成功。同时，由于 CSEc 的密钥是存储在芯片的模拟 EEPROM 中，所以当设置此功能后，也不能再对 D-Flash 进行分区和擦除（对 P-Flash 的操作不受影响）。所以**设置写保护属性的时候务必要非常小心！**

2) Boot 保护(BOOT_PROT)

如果设置了 Boot 保护属性（置 1），则如果安全引导步骤中计算的 MAC 值与 CSEc 模块中存储的 BOOT_MAC 值不匹配，则无法使用该密钥。也就是说，如果安全引导失败，则所有标记为 BOOT_PROT 的密钥都将保持锁定(失效)，在应用程序中无法使用它们。

3) 调试端口保护(DEBUG_PROT)

如果设置了 DEBUG_PROT 属性（置 1），则下次复位后，如果有调试器连接（或已经连接）到

MCU，则无法使用该密钥。

4) 密钥功能设置(KEY_USAGE)

此属性确定密钥是用于加密/解密还是用于 CMAC 生成/验证。如果设置此属性（置 1），则密钥用于生成和验证 CMAC 值。如果不设置此属性，则该密钥用于加密/解密。

5) 通配符保护 (WILDCARD)

如果设置了此属性（置 1），则无法通过提供特殊的通配符 UID（即 UID=0）来更新密钥（更新密钥时需要使用到 UID，如果设置了此属性，则必须获取真实的芯片 UID；不设置此属性的话，就可以使用 0 作为 UID）。

6) 密钥仅用于 CMAC 验证(VERIFY_ONLY)

此标志是根据 GM-SHE+规范建立。如果需要启用该功能，则需要在进行 D-Flash 分区，使用 PGMART 命令时（使用 SDK API，配置传入参数即可），配置 SFE(Security Flag Extension)=0x01。

如果启用了该功能，则这个 Key 不能用于生成 MAC（GENERATE_MAC）命令，而只能用于验证 MAC（VERIFY_MAC）命令，仅仅作为 CMAC 验证密钥。当 KEY_USAGE 设置为 0 的情况下，该功能将无效。

2.4 CSEc 的 PRAM 接口介绍

PRAM 是 CSEc 模块与应用程序进行数据交互的接口，这样的方式使得访问 CSEc 模块更加安全。CSEc 的 PRAM 由 8 个 128 位的 RAM 页组成，支持一个字（4 个字节）或一个字节的访问。

CSEC 的通用 PRAM 接口

Bits	[127:0]															
Bits	31:24	23:17	15:8	7:0	31:24	23:17	15:8	7:0	31:24	23:17	15:8	7:0	31:24	23:17	15:8	7:0
WD	Word 0				Word 1				Word 2				Word 3			
Byte	3	2	1	0	7	6	5	4	B	A	9	8	F	E	D	C
页 0	FuncID	Format Func	CallSeq	KEYID	Error Bits				Command Specific							
备注	命令头（Command Header）								控制字段							
页 1	与 CSEc 模块交互的数据（输入输出）															
页 2	与 CSEc 模块交互的数据（输入输出）															
…	……															
页 6	与 CSEc 模块交互的数据（输入输出）															
页 7	与 CSEc 模块交互的数据（输入输出）															

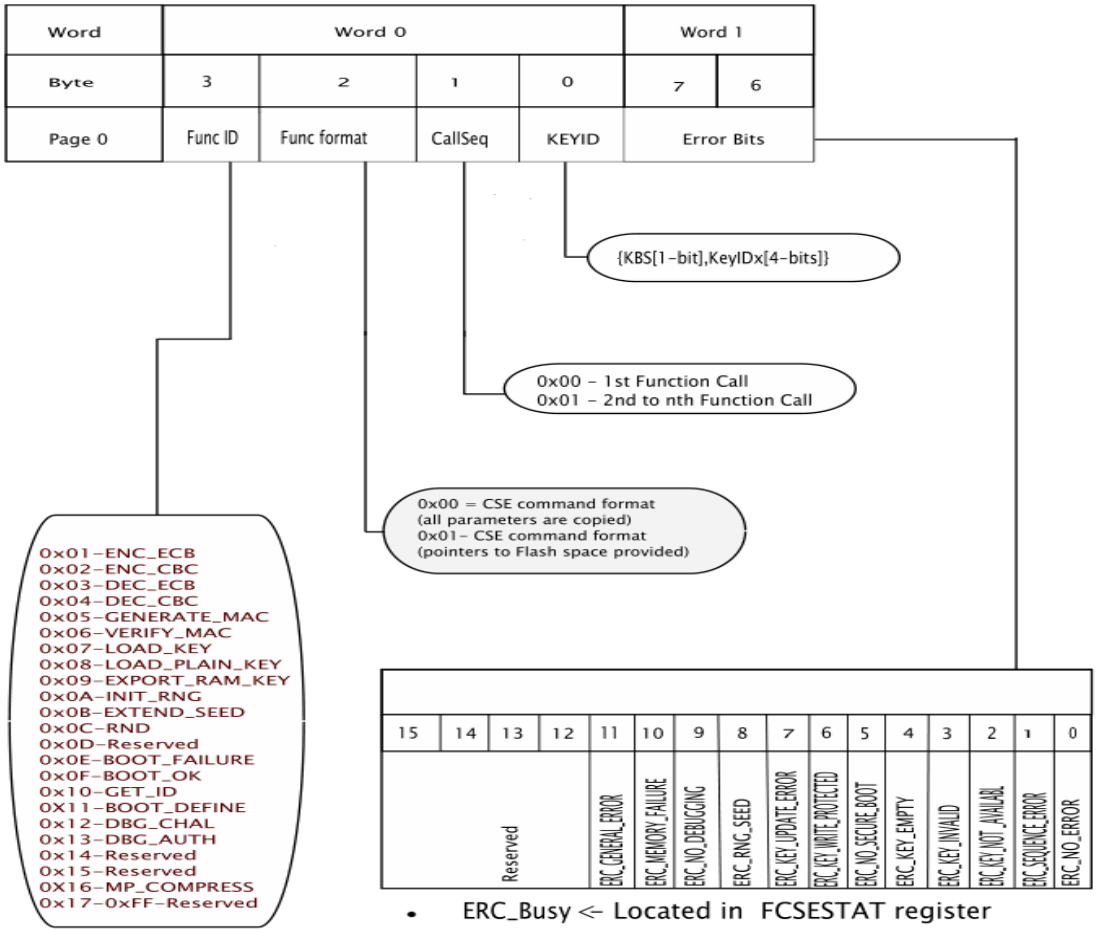
如上所示，这里重点介绍页 0 的数据含义，前 6 个字节为命令头（Command Header），接着的 10 个字节为控制字段。其余页的数据均用于输入/输出数据信息。当写入命令头时，会触发锁定 PRAM，同时 CSEc 模块读取 PRAM 接口数据，然后开始执行输入的命令。所以，要发送 CSEc 命令前，应当首先输入数据信息，然后输入数据消息长度信息，最后写入命令头，触发 CSEc 操作。

一旦写入命令头，命令完成中断标志（CCIF）就会被重置，CSEc 模块便开始执行 PRAM 中的命令。执行完成后将 FSTAT[CCIF]置 1。可以通过此寄存器查询 CSEc 命令是否执行完成，当命令执行完成后，就可以从 PRAM 读取 CSEc 返回的结果和数据。

当数据较大不能一次性加入到 CSEc PRAM 中时，就需要通过重复相同命令的方式来加载新数据。在 CSEc PRAM 中写入剩余的数据，然后再写入更新的命令头即可。在命令头中，只需要更改“CallSeq”字段（见后文），其他信息必须保持不变。如果在继续上一个命令的同时更改 FuncId 字段的话将会导致错误发生。

2.4.2 命令头（Command Header）格式详解

CSEc 模块的所有命令必循包含 6 个字节的命令头，命令头的各个字节的含义如下图：



- **FuncID**：长度为 8 位的功能标识字段，根据 HIS-SHE 命令定义指定 CSEc 模块要执行的命令，如 FunID 为 0x01 时，则意味着进行 ECB 模式的加密操作。有效值范围为 0x01~0x16。
- **FuncFormat:** FuncFormat 指定数据如何在 CSEc 模块和应用程序之间传输。设置为 0x00 意味着使用最常见的一种方法：所有数据和命令通过复制的方式传入到 CSEc 模块。设置为 0x01 则是使用指针的方法将数据和命令传入 CSEc 模块（仅适用于 CAMC 操作的两个命令 GENERATE_MAC 和 VERIFY_MAC）。
- **CallSeq:** CallSeq 用于当数据量超过 PRAM 大小（128 个字节），数据需要分多次写入到 PRAM 时，用于指定当前数据是第一次写入还是后续的写入，取值如下：

第一次数据写入：0x00

第二次到第 n 次数据写入：0x01

假设现在需要对 128 字节数据做 CMAC 验证时，由于实际一次性传入 PRAM 的数据只有 112 字节（除去页 0 的命令头和消息控制字段），那么就需要分两次传入到 CSEc 块，这样在第一次写入时将 CallSeq 字段设置为 0x00，第二次写入时设置为 0x01 即可，其余信息必须保持不变。

- **KeyID: {KBS, KeyIDx}:**它分为两部分，KeyIDx 是指向要使用的密钥的值（4 Bits），KBS 是密钥所在区域的选择，位定义如下：

Bits	7	6	5	4	3	2	1	0
Value	0	0	0	KBS	Key Id			

这里需要注意的是，Bits [7-5]必须始终为 0。写入 1 将会导致寻址错误。如果 KBS 不用于特定密钥，则用户必须将 0 写入 Bits [4]。

- **Error Bits:** 在命令执行后所有的错误信息将会通过这 16bits 返回。当检查到命令执行完成后，通过读取这 16 位的错误信息字段，检查执行过程中是否产生了错误。

2.4.3 CSEc 模块状态、错误和中断信息

CSEc 模块开始执行命令前，会清除 FSTAT[CCIF]标志，并在完成后立即设置 FSTAT[CCIF]标志。可以通过 Flash 配置寄存器（FCNFG[CCIE]）设置当 CSEc 模块完成操作后产生中断。

CSEc 模块执行命令期间发生的任何错误都会记录在命令头的“Error Bits”字段中。

FTFC 模块的 CSEc 状态寄存器（FCSESTAT）记录 CSEc 模块的状态。IDB 和 EDB 位用于设置是否启用内部和外部调试功能。RIN 位是在随机数生成器初始化时设置的。当 CSEc 模块正在执行命令时会设置 BSY 位。BOK、BFN、BIN 和 SB 位是安全引导功能特定的位，在 4.5.3.2 自动添加 BOOT_MAC 中进行了详细描述。

FCSESTAT 寄存器

Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Field	IDB	EDB	RIN	BOK	BFN	BIN	SB	BSY

3 CSEc 模块应用开发详解

前面章节已经介绍了 CSEc 模块的功能，密钥等，本章节开始介绍如何将 CSEc 模块应用到工程中，在使用 CSEc 模块的功能前，应该先按照顺序对 Flash 分区，然后加载密钥，而加载密钥并不是像存储数据到 Flash 那样，而是需要进行一系列的加密计算，以保证密钥本身的安全。

3.1 CSEc 模块应用开发步骤

使用 CSEc 模块应按照以下步骤进行。

1、 对 Flash 进行分区操作

由于 CSEc 模块所使用的密钥实际是存储在模拟 EEPROM 区域（见 2.2 CSEc 模块与 FTFC 模块的关

系），只不过这段区域是由 CSEc 模块管理，以保证密钥的安全，所以在使用 CSEc 模块前，必须在 D-Flash 中划分出 EEPROM 区域，同时使能 CSEc 密钥存储区域。

2、 加载密钥到 CSEc 模块

由于初次使用，用户密钥均为空白，所以必须先加载密钥到 CSEc 模块，否则无法直接使用用户密钥进行后续操作，这里有个非常值得注意的地方，由于 CSEc 模块的密钥实际是存储在 E-Flash（模拟 EEPROM）中的，并且对用户来说不可见，只能由 CSEc 模块操作，其他任何模块，均无法操作。所以当使能了 CSEc 模块密钥存储区域后，则不能通过 Flash 的命令对 D-Flash 进行 Erase All 操作，也就是说，这个时候不能再通过除 CSEc 模块外的其他任何命令恢复 D-Flash 至出厂状态。这个时候如果需要重新分区，则必须使用 CSEc 提供的恢复 Flash(D-Flash)出厂状态的命令。

3、 使用加载的密钥进行加密/解密、生成随机数、生成/验证 CMAC 等操作。

完成上面的操作后，就可以使用加载到 CSEc 模块中的密钥进行加密/解密等一系列操作了。

3.2 CSEc 模块的授权密钥使用

为了保证密钥的安全，HIS-SHE 标准要求在尝试更新密钥之前必须提供一个授权密钥。可以使用这个密钥原来的值或 MASTER_ECU_KEY 作为授权密钥（两者使用一个即可）。MASTER_ECU_KEY 可以作为所有密钥的授权密钥，用于操作其他密钥（更新 RAM_KEY 不需要授权密钥）。比如，可以使用 MASTER_ECU_KEY 作为授权密钥来更新 BOOT_MAC_KEY, BOOT_MAC 和用户的 KEY_1 to KEY_17。各个密钥的授权密钥如下表所示。

授权密钥

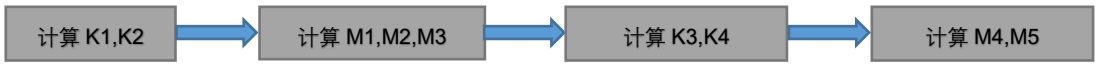
待更新的密钥	授权密钥				
	MASTER_ECU_KEY	BOOT_MAC_KEY	BOOT_MAC	KEY_<N>	RAM_KEY
MASTER_ECU_KEY	√				
BOOT_MAC_KEY	√	√			
BOOT_MAC	√	√			
KEY_<N>	√			√	
RAM_KEY					

3.3 CSEc 模块 M1~M5 的计算方法

加载用户密钥至 CSEc 模块中时，并不是直接将密钥明文加载至 CSEc 模块中（除了 RAM_KEY），为了确保密钥的机密性、完整性、真实性，并防止重复攻击，添加密钥必须使用 HIS-SHE 规范中定义的协议（HIS-SHE 功能规范，v1.1，9.1 Description of memory update protocol）。通过对原始的密钥进行一系列的计算，生成 M1,M2,M3,M4,M5 共 5 组数据，其中 M1~M3 作为输入，加载到 CSEc 模块中，加载完成后 CSEc 模块会返回 M4,M5，通过返回的 M4,M5 与之前计算的 M4,M5 进行比较，如果返回的数据与之前计算的数据相同，则说明密钥已经正确的加载至 CSEc 模块中。

由于通过原始密钥计算出 M1~M5 值的这个过程是不可逆的，所以整个过程是安全的。举个例子，整车厂定义了自己的 CSEc 密钥，在交与供应商进行开发或者工厂进行批量生产时，这个密钥明文是不能够透露的，但是可以通过离线计算出密钥的 M1~M5 值，由于计算过程不可逆，供应商并不能推算出密钥明文。这样可以很大程度的防止整车厂密钥泄露。

M1~M5 计算需要编写函数完成，本文提供了相应的参考代码（详见 附录 2 计算 M1~M5 的参考函数），计算原始密钥的 M1~M5 通过以下步骤完成：



1、 KDF 函数

KDF 函数是自己定义的一个用于计算 M1~M5 值的工具函数，本文设计的 KDF 函数见附录 2 计算 M1~M5 的参考函数。

```
1.  KDF(AuthKey, Constant, K_out)
```

参数 AuthKey 和 Constant 均为 16 个字节长度的输入参数，K_out 为 32 个字节长度的输出，KDF 函数功能是将参数 AuthKey 和 Constant 连接后，使用 CSEc 模块的 **CSEC_DRV_MPCompress()**压缩算法 API，得到输出的 K_out。

2、 计算 K1 和 K2

K1 = KDF(AuthKey, KEY_UPDATE_ENC_C)

K2 = KDF(AuthKey, KEY_UPDATE_MAC_C)

- AuthKey：授权密钥，出厂状态下，使用所有 Bit 位为 1 的空密钥作为授权密钥。
- KEY_UPDATE_ENC_C：0x01015348_45008000_00000000_000000B0（HIS-SHE 规范定义）。
- KEY_UPDATE_MAC_C：0x01025348_45008000_00000000_000000B0（HIS-SHE 规范定义）。
- 计算出的 K1 和 K2 长度均为 32 Bytes。

3、 计算 M1

$M1 = UID \bowtie ID \bowtie AuthKey_{ID}$

- UID：芯片 120 bits 的唯一标识符。对于出厂状态或该密钥的通配符(WILDCARD)属性未使能。那么 UID 可以使用 0 来代替。
- ID：待更新密钥的密钥 ID，这里不需要考虑 KBS 域。
- $AuthKey_{ID}$：授权密钥的 ID，如果使用原密钥作为授权密钥，则与 ID 参数的值一样即可，如果使用 MASTER_ECU_KEY 来更新，则 $AuthKey_{ID}$ 为 MASTER_ECU_KEY 的 ID（KeyID = 0x1），授权密钥 ID 不用考虑 KBS 值。
- **$\bowtie$：代表将数据按位拼接至一起，下同**，比如：$0xF5 \bowtie 0xA$ 则拼接后数据为 $0xF5A$。
- M1 的长度为 128 bits。

4、 计算 M2

M2 的长度为 256 bits。

如果进行 D-Flash 分区时，设置 SFE==0x0，则：

$M2 = ENC_{CBC}(K1, IV=0,(CID \bowtie FID \bowtie 0_{bit1}...0_{bit95} \bowtie KEY))$

如果进行 D-Flash 分区时，设置 SFE==0x1，则：

$M2 = ENC_{CBC}(K1, IV=0,(CID \bowtie FID \bowtie 0_{bit1}...0_{bit94} \bowtie KEY))$

- M2 的值是在将 K1 作为加密密钥，IV 参数为 0 的情况下，对由 CID，FID，0 填充位和密钥 ID 组成的数据进行 AES-128 CBC 模式加密得到。
- CID：密钥计数器的值（28 bits），从 1 开始计数，每对该密钥进行一次更新就将计数器值加 1，如果对 CSEc 密钥进行了恢复出厂设置操作，则计数器值重新从 1 开始计数。
- FID：密钥属性
  - **SFE 为 0 时(FID 为 5 bits)：**

    FID = WRITE_PROT ⋈ BOOT_PROT ⋈ DEBUG_PROT ⋈ KEY_USAGE ⋈ WILD_CARD
  - **SFE 为 1 时(FID 为 6 bits)：**

    FID = WRITE_PROT ⋈ BOOT_PROT ⋈ DEBUG_PROT ⋈ KEY_USAGE ⋈ WILD_CARD ⋈ **VERIFY_ONLY**
- $0_{bit1}...0_{bit94/bit95}$：如果 SFE 等于 0 则添加 95 个 0 填充位，如果等于 1 则添加 94 个 0 填充位。
- KEY：待添加或更新的密钥(128 bits)。

5、 计算 M3

$M3 = CMAC_{K2}(M1 \bowtie M2)$

将 M1 和 M2 连接在一起，使用 K2 作为密钥，计算出的 CMAC 值，即为 M3 值。M3 长度为 128 bits。

注意：实际应用代码中，不应当出现密钥的明文，可以通过脱机计算的方式得到密钥的 M1~M5 值，这样可以防止密钥泄露。

6、 计算 K3 和 K4

$K3 = KDF(KEY_{ID}, KEY\_UPDATE\_ENC\_C)$

$K4 = KDF(KEY_{ID}, KEY\_UPDATE\_MAC\_C)$

- KEY_{ID}：待更新密钥的 ID，不考虑 KBS 域。
- KEY_UPDATE_ENC_C:0x01015348_45008000_00000000_000000B0（HIS-SHE 规范定义）。
- KEY_UPDATE_MAC_C:0x01025348_45008000_00000000_000000B0（HIS-SHE 规范定义）。

7、 计算 M4

$M4 = UID \bowtie ID \bowtie AuthKey_{ID} \bowtie M4^*$

- UID：芯片的 120 位的唯一标识符。
- ID：需要被更新的密钥 ID（不包含 KBS 域）。
- $AuthKey_{ID}$：需要更新密钥的授权码（不包含 KBS 域）。

- M4*：M4* = $ENC_{ECB}(K3, (CID \bowtie 1_{bit1} \bowtie 0_{bit1}...0_{bit99}))$，即以 K3 作为加密密钥，CID 后添加 1 个 1 填充位，99 个 0 填充位，进行 AES-128 ECB 加密操作。
- M4 长度为 256 bits。

8、 计算 M5

$M5 = CMAC_{K4}(M4)$

- 用 K4 作为密钥，获取 M4 的 CMAC 值，即得到 M5 值。
- M5 长度：128 bits。

M4 和 M5 为添加或更新密钥时，CSEc 模块的返回值，如果该返回值与脱机计算得到的值相同，说明操作成功，如果两者不相同，说明脱机计算得到的值不正确。

9、 参考函数

本文所设计计算 M1~M5 的参考函数见 附录 2 计算 M1~M5 的参考函数。

4 基于 SDK 的 CSEc 模块应用开发

4.1 SDK API 介绍

本节以 SDK3.0.0RTM 版本为例，介绍 SDK 中部分 API 的功能。API 详细介绍见 附录 4 SDK 3.0.0RTM 版本 API 说明

4.2 D-Flash 分区 API

使用 D-Flash 分区的 API 前，需要配置添加 SDK 的 Flash 组件，初始化 Flash 后就可以进行分区操作，Flash 模块的 D-Flash 分区 API 如下：

```
status_t FLASH_DRV_DEFlashPartition(const flash_ssd_config_t * pSSDConfig,
                                    uint8_t uEEEDataSizeCode,
                                    uint8_t uDEPartitionCode,
                                    uint8_t uCSEcKeySize,
                                    bool uSFE,
                                    bool flexRamEnableLoadEEEData);
```

D-Flash 分区函数详细的参数如下表：

uEEEDataSizeCode	配置模拟 EEPROM 大小。 **对于 2KB 大小 FlexRAM 的 MCU：** ● 0x0F：不配置模拟 EEPROM 功能，FlexRAM 作为系统 RAM 使用。 ● 0x03：配置 2KB 的 FlexRAM 为模拟 EEPROM 功能。 **对于 4KB 大小 FlexRAM 的 MCU：** ● 0x0F：不配置模拟 EEPROM 功能

<table>
<tr><td></td><td>

- 0x02：配置 4KB 的 FlexRAM 为模拟 EEPROM 功能。

</td></tr>
<tr><td>uDEPartitionCode</td><td>

配置 D-Flash 分区。

- 对于 128KB 和 256KB 的 P-Flash 空间（S32K11x 系列），32KB 大小 FlexNVM 的 MCU：

<table>
<tr><th>分区参数</th><th>D-Flash 大小（Kbytes）</th><th>E-Flash 大小（Kbytes）</th></tr>
<tr><td>0x00</td><td>32</td><td>0</td></tr>
<tr><td>0x03</td><td>0</td><td>32</td></tr>
<tr><td>0x08</td><td>0</td><td>32</td></tr>
<tr><td>0x09</td><td>8</td><td>24</td></tr>
<tr><td>0x0B</td><td>32</td><td>0</td></tr>
</table>

- 对于 256KB，512KB 和 1M 的 P-Flash 空间（S32K14x 系列），64KB 大小 FlexNVM 的 MCU：

<table>
<tr><th>分区参数</th><th>D-Flash 大小（Kbytes）</th><th>E-Flash 大小（Kbytes）</th></tr>
<tr><td>0x00</td><td>64</td><td>0</td></tr>
<tr><td>0x03</td><td>32</td><td>32</td></tr>
<tr><td>0x04</td><td>0</td><td>64</td></tr>
<tr><td>0x08</td><td>0</td><td>64</td></tr>
<tr><td>0x0A</td><td>16</td><td>48</td></tr>
<tr><td>0x0B</td><td>32</td><td>32</td></tr>
<tr><td>0x0C</td><td>64</td><td>0</td></tr>
</table>

- 对于 2M 的 P-Flash 空间（S32K148），64KB 大小 FlexNVM 的 MCU：

<table>
<tr><th>分区参数</th><th>D-Flash 大小（Kbytes）</th><th>E-Flash 大小（Kbytes）</th></tr>
<tr><td>0x00</td><td>64</td><td>0</td></tr>
<tr><td>0x04</td><td>32</td><td>32</td></tr>
<tr><td>0x0F</td><td>0</td><td>64</td></tr>
</table>

</td></tr>
<tr><td>uCSEcKeySize</td><td>

设置 CSEc 密钥数量：

- 0x00：不使用 CSEc 密钥，不占用模拟 EEPROM 的空间。
- 0x01：使用 1 至 5 个密钥，占用 128 字节模拟 EEPROM 空间。
- 0x02：使用 1 至 10 的密钥，占用 256 字节模拟 EEPROM 空间。
- 0x03：使用 1 至 24 个密钥，占用 512 字节模拟 EEPROM 空间。

</td></tr>
<tr><td>uSFE</td><td>

Verify Only 属性设置：

- 0x1：密钥的 Verify Only 属性配置功能打开。
- 0x0：密钥的 Verify Only 属性配置功能关闭。

</td></tr>
<tr><td>flexRamEnableLoadEEEData</td><td>

MUC 启动时是否自动加载 E-Flash 中的至 FlexRAM 中。使用 CSEc 模块功能时，必须设置为自动加载模式。

- 0x1：MUC 启动时自动加载 E-Flash 中的值到 FlexRAM 中。
- 0x0：不自动加载 E-Flash 中的值到 FlexRAM 中。

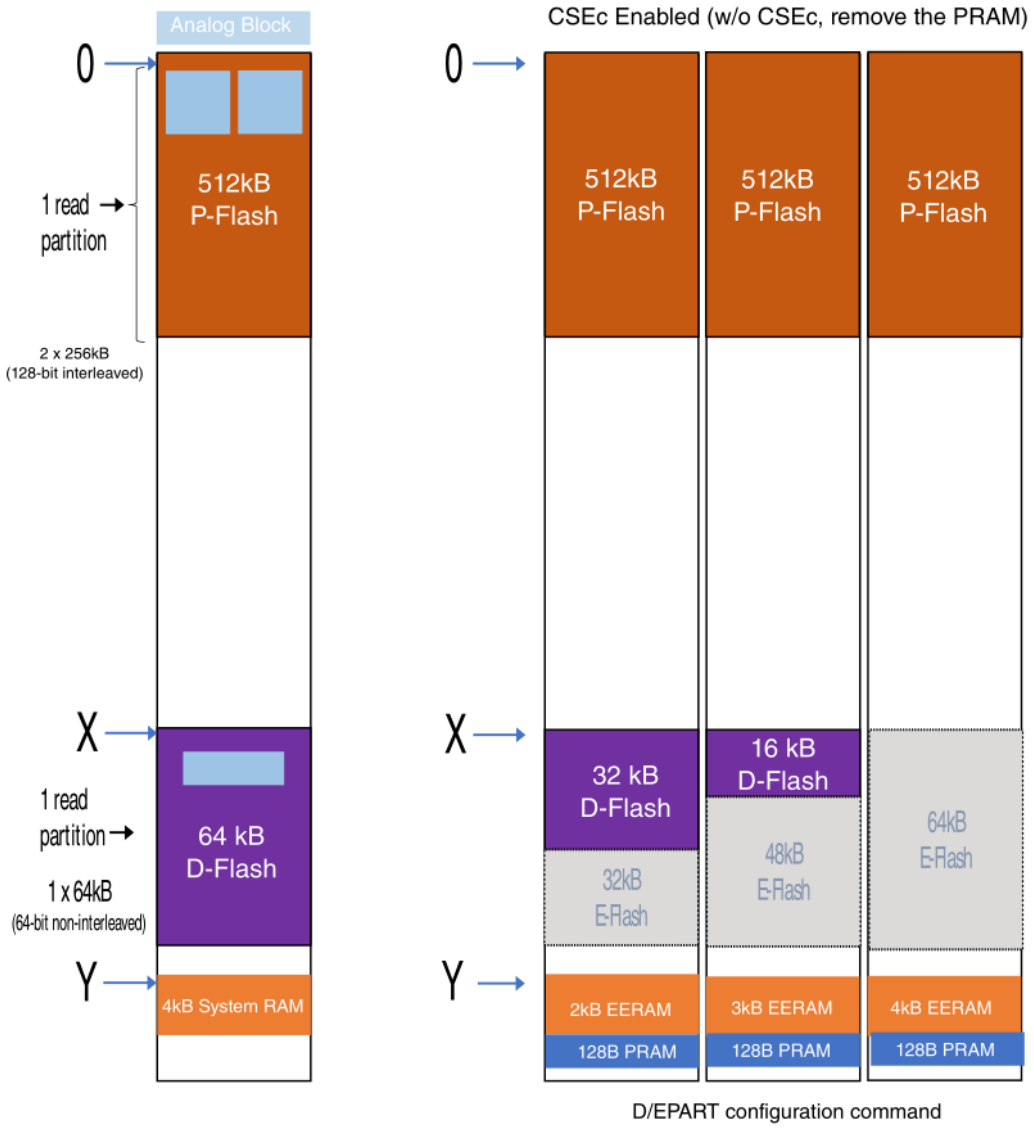
使用 CSEc 功能时，必须设置为自动自动加载。

</td></tr>
</table>

在进行 D-Flash 分区时，模拟的 EEPROM 备份大小应至少为 EEERAM 的 16 倍。更多关于配置模拟 EEPROM 的内容请参考芯片手册和相关应用笔记（AN11983）等，本文对此不做详细介绍。

对于配置 CSEc 模块来说，主要需要关心的是参数是 uCSEcKeySize 和 uSFE，uCSEcKeySize 参数用于设置需要存储的密钥数量，uSFE 参数用于设置是否开启附加的属性配置。

以 S32K144/512KB 为例，进行不同的分区设置后，Flash 实际的功能如下图所示：



4.3 CSEc 模块密钥管理

密钥管理包括向 CSEc 硬模块中添加密钥、更新密钥和擦除密钥等操作。CSEc 存储密钥的区域除了 CSEc 模块可以访问外，其余模块均没有访问权限。

4.3.1 加载密钥

在使用 CSEc 模块一些功能的时候，必须先加载密钥到 CSEc 模块，通过前面章节描述的方法，计算得到 M1~M5 后，调用 CSEc 模块的 API：

```
1.  status_t CSEC_DRV_LoadKey(csec_key_id_t keyId, const uint8_t *m1,
2.                            const uint8_t *m2, const uint8_t *m3, uint8_t *m4, uint8_t *m5);
```

检查返回的 `status_t` 判断函数是否成功执行，对比函数返回的 m4 和 m5 与计算的 M4 和 M5 是否一一匹配，如果匹配的，则说明密钥已经成功的加载到 CSEc 模块中了，如果不匹配，则说明加载的密钥有误，这有可能是计算 M1~M5 的方法有误导致。

注意：如果要更新的密钥未设置通配符保护（WILDCARD == 0），则可以在生成 M1~M5 时使用 UID=0。否则，需要得到 MCU 真实的 UID，并将其用于生成 M1~M5 的计算中。可以按照 4.4.1 获取 UID 中的描述的方法获取 UID 值。

对于出厂的默认密钥均为空密钥（blank key），空密钥值为全 1，这时授权密钥值即为空密钥，比如初次添加 MASTER_ECU_KEY 时，就需要使用空密钥作为授权密钥进行添加。其余任何时候，授权密钥都应当是当前密钥或者非空的 MASTER_ECU_KEY。

请参考本文的 Demo 工程，了解详细的流程。

4.3.2 更新密钥

当需要对已经加载至 CSEc 模块中的密钥进行更改时，与加载密钥不同的是，需要每更新一次密钥时，将密钥的计数器加 1，如果这个密钥使能了写保护（WRITE_PROT）属性，则这个密钥不能再进行任何更改，所以设置密钥属性的时候要慎重！详见 2.3.4 CSEc 的密钥属性。

请参考本文的 Demo 工程，了解详细的流程。

4.3.3 删除密钥

不能单独的对任何一个密钥进行擦除操作，只能通过 CSEc 模块恢复出厂设置的命令，擦除所有的密钥。参见 4.6 恢复出厂设置编程。

4.4 CSEc 模块基本功能使用

本节将介绍一些 CSEc 模块的基本操作，如 UID 检索、AES-128 编码和解码、CMAC 生成和验证以及随机数生成。

4.4.1 获取 UID

唯一标识号（UID）对于每片 S32K1xx MCU 都是唯一的，并且在出厂时被写入到 CSEc 模块中。UID 的长度为 120 位。在 ECU 之间的通信过程中，可以使用 UID 来确认部件的唯一性。UID 也用于 CSEc 模块密钥重置为出厂状态的操作中。可通过 CSE_GET_ID 命令或者 SDK 中的 API **CSEC_DRV_GetID()** 来获取 MCU 的 UID。

不过在 MASTER_ECU_KEY 为空时，获取得到的 UID 为 0。使用 CSEc 模块时，建议应该首先加载 MASTER_ECU_KEY。

4.4.2 AES-128 加密解密

CSEc 模块支持在 ECB（电子码本）和 CBC（密码块链）操作模式下进行 AES-128 加密和解密。加密解密操作必须选择 CSEc 模块中的一个密钥（KEY_USAGE =0）作为加密解密密钥。

如果未添加任何密钥，也可以使用 CMD_LOAD_PLAIN_KEY 命令或 API 将明文密钥加载到 RAM_KEY 中，使用 RAM_KEY 作为加密解密的密钥，但是，由于此方法意味着潜在的安全风险，因此它可能只对开发或调试有用。

由于命令的长度是以页长度为单位的，因此数据长度以 128 位为一个单位处理。如果不满足，则应由应用程序进行填充。列如在使用 CMD_ENC_ECB, CMD_ENC_CBC, CMD_DEC_ECB 和 CMD_DEC_CBC 等命令时。在使用 SDK 的情况下，只需要直接调用相应的 API 即可。

4.4.3 生成随机数

PRNG 会通过 SECRET_KEY 派生出用于生成随机数的密钥，然后使用 AES 模块生成伪随机值。RND 命令用于更新 PRNG 的状态并返回 128 位随机值。CMD_EXTEND_SEED 命令可用于向 PRNG 状态添加种子。每次复位后，必须先使用 CMD_INIT_RNG 命令初始化，初始化后 FCSESTAT[RIN]将会被设置为“1”，该命令会使用内部 TRNG 为 PRNG 生成 128 位种子值。根据 HIS-SHE 规范和 AIS20 标准，PRNG 使用 PRNG_STATE/KEY 和种子。使用 CMD_RND 命令以生成随机数。

注意：在初始化 PRNG 之前必须先初始化 CMD_EXTEND_SEED, CMD_RND 和 CMD_DBG_CHAL 命令。使用 SDK 时，需要先调用 **CSEC_DRV_InitRNG()** API 初始化 CSEc 模块中的随机数生成器。

4.4.4 CMAC 值的生成和验证

CSEc 模块使用 AES-128 CMAC 算法进行消息身份验证。从 CSEc 模块的密钥中（KEY_USAGE =1）选择一个进行作为生成 CMAC 的密钥。

如果未添加任何密钥，也可以使用 CMD_LOAD_PLAIN_KEY 命令将明文密钥加载到 RAM_KEY 中，使用 RAM_KEY 作为 CMAC 操作的密钥，但是，由于此方法意味着潜在的安全风险，因此它可能更适合开发和调试阶段使用。

CMD_VERIFY_MAC 命令用于将计算的 MAC 与输入的 MAC 值进行比较。

4.5 安全引导程序

CSEc 模块允许用户在 Flash 中验证引导代码。通过对 MCU 的配置，可以在每次 MCU 启动的时候，对某段代码（如 Bootloader）进行验证，确保程序未被破坏。需要说明的是，CSEc 只会验证设定区域的数据是否被篡改，如发现程序被篡改，CSEc 模块根据配置的安全引导模式决定是否中断程序的运行，也可以配置为当 CSEc 模块检查到设定区域的代码被篡改后，使某些密钥失效，详见 2.3.4 CSEc 的密钥属性。

4.5.1 几种安全引导程序模式

S32K1xx 系列处理器支持 3 种安全引导程序模式

A. **顺序启动模式（Sequential Boot Mode）：**复位或上电启动后，先执行 CSEc 模块的安全引导相关的代码，验证完毕后，才会按照正常的流程执行用户代码。如果验证失败后，则在用户代码中无法使用设置了 BOOT_PROT 属性的密钥。应用程序不会被中断。

B. **严格的顺序启动模式（Strict Sequential Boot Mode）：**复位或上电启动后，先执行 CSEc 模块的安全引导相关的代码，验证完毕后，才会按照正常的流程执行用户代码。如果验证失败后，那么 CPU

将一直保持复位状态（不会再继续执行应用程序）。

注意：**此模式设置是不可逆的**，一旦设置为严格的顺序启动模式后，就不能更改为其他启动模式。在设置此模式之前，必须计算并存储 BOOT_MAC 值，否则设备将保持重置状态。（自动 BOOT_MAC 计算不能在此模式下运行）

C. **并行启动模式（Parallel Boot Mode）：**复位或上电启动后， CSEc 模块执行安全引导相关的代码，CPU 同时执行应用程序，两者互不影响，行安全引导验证完毕后，CSEc 模块会设置相应状态寄存器，通过读取状态值，查看安全引导验证是否成功。如果验证失败后，则在用户代码中无法使用设置了 BOOT_PROT 属性的密钥。应用程序不会被中断。

注意：安全引导功能仅仅使用于验证 Flash 中的程序。

4.5.2 使能安全引导程序

使用 CMD_BOOT_DEFINE 命令或 SDK 中的 **CSEC_DRV_BootDefine()** API 用于配置有效的安全引导程序模式和 Boot 程序的大小。

BOOT_MAC_KEY 是用于验证启动代码的密钥。如果没有添加 BOOT_MAC_KEY，则设备会中止安全引导过程并清除 FCSESTAT[SB]位。

一旦配置了安全引导，每次复位时，安全引导模块都会计算从地址 0 开始到设定结束区域这段代码的 MAC 值，并与存储在 CSEc 模块中的 BOOT_MAC 进行比较。如果 BOOT_MAC 为空，那么 CSEc 模块会自动存储计算的 BOOT_MAC_KEY（4.5.3.2 自动添加 BOOT_MAC），并中止当前安全引导过程，设置 FCSESTAT[BIN]为 1。

如果计算出的 MAC 值与存储在 CSEc 模块中的 BOOT_MAC 一致，那么将设置 FCSESTAT[BOK]为 1。安全引导过程结束后，应用程序应当将执行 CMD_BOOT_OK（或 SDK 的 API **CSEC_DRV_BootOK()**）命令。这会设置 FCSESTAT[BFN]为 1 来标记安全引导过程的结束。

在用户程序中，应当检查 FCSESTAT[BOK]是否为 1，如果不为为 1，说明所设定安全引导区域的代码已经被篡改，则应当执行 CMD_BOOT_FAILURE 命令（或 API **CSEC_DRV_BootFailure()**）使 CSEc 模块所有的密钥都失效，以保证安全。CMD_BOOT_FAILURE 与 CMD_BOOT_OK 应当在每次上点/复位后，在安全引导完成后立即根据安全引导结果执行这两个命令中的一个。

如果安全引导过程成功，应用程序可以正常的使用所有密钥。否则，所有设置了 BOOT_PROT 属性的密钥将保持锁定，应用程序无法使用。

注意：使用自动安全引导过程最多可以验证 512KB 的代码。但是，可以遵循应用笔记文档 AN4235 中描述的 circle of trust（循环验证）方法：使用 CSEc 通过循环验证的方法来验证整个应用程序代码（可以大于 512 kb）。

4.5.3 首次添加 BOOT_MAC

BOOT_MAC 可以通过两种方式添加至 CSEc 模块中：

1、 手动添加 BOOT_MAC 值

2、 使用 CSEc 自动添加 BOOT_MAC 值

4.5.3.1 手动添加 BOOT_MAC

可按照以下步骤手动添加 BOOT_MAC

1、 编写需要保护的 Boot 程序，写入到 P-Flash 中。

2、 将 BOOT_MAC_KEY 写入到 CSEc 模块中（其他用户的密钥也可以在此时进行写入）。

3、 使用 CMD_BOOT_DEFINE 命令或调用 API 定义安全引导的模式和 Boot 大小。

4、 使用 BOOT_MAC_KEY 计算要保护代码的 MAC 值。可以脱机计算，也可以使用 CSEc 模块 RAM_KEY 的方式计算。

5、加载计算的 MAC 到 BOOT_MAC 中。

6、复位设备。CSEc 模块会验证定义的 Boot 程序段的 MAC 和存储的 BOOT_MAC 值是否匹配，如果匹配，设置 FCSESTAT[BOK]=1（安全引导验证成功），如果不相同，则设置为 0，意味着安全引导验证失败。

注意：手动计算 BOOT_MAC 时，与计算一般的 MAC 值有以下两点不同**（非常重要）**:

1. 计算 BOOT_MAC 需要在数据前加上 128bits 的头数据（96bits'0'+32bits BootSizeValue）,也就是计算数据实际长度为 BootSizeValue+128bits。这样无法直接使用 SDK 中的 API 来计算 BOOT_MAC。

2. CSEc 模块自动计算 BOOT_MAC 时，是按照四字节的方式访问，所以在手动加载计算 BOOT_MAC 时也应当以这种方式才能计算出与之匹配的 BOOT_MAC 值。举个例子：以地址 0 开始依次数据为[0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88]。如果使用 SDK 中的 API **CSEC_DRV_GenerateMAC(),**将地址 0 作为参数传入，则实际计算的数据流与之对应。但 CSEc 模块自动在计算这段数据的 BOOT_MAC 时，采用四字节的访问方式，由于 S32K1xx 使用的小端模式，那么实际计算的数据流则为[0x44,0x33,0x22,0x11,0x88,0x77,0x66,0x55]。这种方式仅限于计算 BootMAC 值得时候。

   对于以上的问题，可以将 PFlash 中的数据拷贝到 RAM 中，再根据上述规则调整数据即可，这种方法的局限就是如果 BootSize 太大了，则无法完全拷贝到 RAM 计算。另一种方法就是根据规则，自定义一个计算 BOOT_MAC 的函数，按照上述方式读取数据即可。以上两种方法在本文的 Demo 工程 4 中具有介绍和代码，请参考 Demo 工程。

4.5.3.2 自动添加 BOOT_MAC

芯片出厂时 CSEc 模块中不会有任何用户密钥也不会有 BOOT_MAC 值。可通过以下步骤，让 CSEc 模块自动计算并添加 BOOT_MAC 到 CSEc 模块中:

1、 编写需要保护的 Boot 程序，写入到 P-Flash 中。

2、 将 BOOT_MAC_KEY 写入到 CSEc 模块中（其他用户的密钥也可以在此时进行写入）。

3、 使用 CMD_BOOT_DEFINE 命令或调用 API 定义安全引导的模式和 Boot 大小。

4、 复位设备。CSEc 模块将自动计算 BOOT_MAC 并将其存储在 CSEc 模块中。

再次复位设备时，CSEc 验证定义的 Boot 程序段的 MAC 和存储的 BOOT_MAC 值是否相同，如果相同，设置 FCSESTAT[BOK]=1（安全引导验证成功），如果不相同，则设置为 0，意味着安全引导验证失败。

安全引导的状态可以在 Flash CSEc 状态寄存器（FCSESTAT）中查看，如下表所示。

Boot/Mac 在 FCSESTAT 中的标志位

实际情况	SB	BIN	BFN	BOK	Error Code	详解
未使用安全引导	0	0	0	0	0	未使用安全引导功能。
BOOT_MAC 为空	1	1	1	0	未返回错误	安全引导程序计算 BOOT_MAC 并将其加载到 CSEc 模块 BOOT_MAC 区域未成功。
BOOT_MAC 不匹配	1	0	1	0	未返回错误	安全引导过程成功，但是验证失败。用户应用程序需要运行 BOOT_OK 命令来设置 bfn 位并启用对设置了 BOOT_PROT 密钥的访问。
BOOT_MAC 验证成功	1	0	0	1	未发生错误	安全引导过程成功，BOOT_MAC 验证成功。
BOOT_MAC_KEY 为空	0	0	1	0	NO_SECURE_BOOT	安全引导进程退出，并返回错误代码。

4.5.4 更新 BOOT_MAC

在软件开发期间以及在 ECU 生命周期中，可能需要更改已经经过安全引导验证的程序代码。这时，CSEc 计算的 BOOT_MAC 将与 CSEc 模块中存储的原来的 BOOT_MAC 值不匹配。这就会导致所有设置了 BOOT_PROT 属性的密钥不能再用于加密解密。这时必须更新存储在 CSEc 模块中的 BOOT_MAC，才能避免出现这种情况。根据实际密钥属性设置的不同，有两种方法可以更新 BOOT_MAC 值

4.5.4.1 情况 1：所有密钥均可以擦除

这种情况是 CSEc 模块中的所有密钥都没有设置写保护（WRITE_PROT），在这种情况下，可以使用 CMD_DBG_CHAL 和 CMD_DBG_AUTH 命令将 CSEc 模块中的所有密钥恢复至出厂状态。CMD_DBG_CHAL 和 CMD_DBG_AUTH 命令只能在没有标记写保护属性的密钥的设备上工作。此过程将删除所有密钥（详见 4.6 恢复出厂设置的编程）。恢复出厂设置成功后，芯片 CSEc 模块的所有密钥将恢复到出厂状态。这时就可以按照前文所描述的步骤，将新的密钥加载到 CSEc 模块中。然后再按照前文添加 BOOT_MAC 的步骤，更新 BOOT_MAC。

4.5.4.2 情况 2：任一密钥无法擦除

如果有任意一个或多个密钥设置了写保护属性（WRITE_PROT），或者忘记了所有密钥，这就导致了无法使用恢复出厂设置的方式更新 BOOT_MAC。在这种情况下。有两种方法可用于生成新的 BOOT_MAC:

1. 使用 RAM_KEY 的方式生成新的 BOOT_MAC：将 BOOT_MAC_KEY 加载到 RAM_KEY 区域，然后使用 RAM_KEY 生成新的 BOOT_MAC。
2. 通过离线计算的方式得到新的 BOOT_MAC 值。

使用上述两种方式得到 BOOT_MAC 值后，再使用 4.5.3 首次添加 BOOT_MAC 中描述的手动添加 BOOT_MAC 的方法将新的 BOOT_MAC 加载到 CSEc 模块中。

4.6 恢复出厂设置编程

HIS-SHE 规范中有恢复安全模块密钥为出厂状态的机制。但该机制仅适用于没有写保护的用户密钥。所以，**如果有任何密钥设置了写保护属性，则无法将密钥重置为出厂状态**。

CSEc 模块通过 CMD_DEBUG_CHAL 和 CMD_DEBUG_AUTH 2 个命令来实现这个机制。不过由于需要 PRNG 生成一个随机数值用于这个操作，所以在使用 CMD_DEBUG_CHAL 命令之前初始化 PRNG。恢复 CSEc 模块密钥为出厂状态的具体操作步骤如下：

1、 通过 CMD_DBG_CHAL 命令请求一组 128bits 的随机数。

2、 计算授权码

$$K = KDF(KEY_{MASTER\_ECU\_KEY}, DEBUG\_KEY\_C)$$

- $KEY_{MASTER\_ECU\_KEY}$：MASTER_ECU_KEY。
- KEY_UPDATE_MAC_C：0x01035348 45008000 00000000 000000B0（HIS-SHE 规范定义）。
- AUTHORIZATION = $CMAC_K$ (CHALLENGE ⋈ UID)。

计算授权码的参考代码参见附录 3 计算恢复出厂设置的授权码参考函数。

3、 通过 CMD_DBG_AUTH 命令和授权码，请求恢复 CSEc 模块出厂状态。

4、 复位芯片。

CMD_DBG_CHAL 命令后面必须跟 CMD_DBG_AUTH 命令，如果中途使用了其他的命令，则需要再重新发送 CMD_DBG_CHAL 命令后才能再发送 CMD_DBG_AUTH 命令，使用 SDK 的 API 时同理。

当以上执行成功，复位设备后：

1、 所有密钥都被清除。

2、 FlexRAM 重置为普通的 RAM 功能（FCNFG[RAMRDY] == 1）。

3、 FlexNVM 重置为 DFlash（FCNFG[EEERDY] == 0）。

如果使用 SDK 编程，则按顺序先后调用 **CSEC_DRV_DbgChal()** 和 **CSEC_DRV_DbgAuth()** 两个 API 就行了。

请参考本文的 Demo 工程，了解详细的流程。

5 示例工程介绍

为了方便用户评估和使用CSEc硬件加密模块功能，本文配套提供了5个基于S32K1xx SDK RTM3.0.0的示例工程：

1. Example1_Configure_part_and_Load_keys_SDK3_0_0

2. Example2_Update_user_keys_sdk_SDK3_0_0
3. Example3_Basic_operations_SDK3_0_0
4. Example4_secure_boot_add_BOOT_MAC_manual_SDK3_0_0
5. Example5_Resetting_flash_to_the_factory_state_SDK3_0_0

使用 Demo 工程时，应当先使用 Example1 进行分区和加载密钥，再使用其他的 CSEc 模块功能，需要使用 Example5 恢复 D-Flash 出厂设置时，应当保证 CSEc 模块已经加载了密钥。

CSEc 模块相关的工具函数位于 **根目录/ Sources/ CSEcFunctions**。如果不想使用 SDK 进行编程，可参考 AN5401 文档的 Demo 工程，详情见 6 参考文档。

6 参考文档

1. AN5401 Getting Started with CSEc Security Module
2. Application Note Software for AN5401
3. S32K RM S32K1xx Series Reference Manual
4. AN4234 Using the Cryptographic Service Engine (CSE)
5. AN4235 Using CSE to protect your Application Code via a Chain of Trust

附录 1 CSEc 模块应用常见问题（FAQ）

1. 使用 Flash 分区后，无法使用 Erase All 擦除 D-Flash，也无法重新进行分区

如果在使用 D-Flash 分区时，使能了 CSEc 的密钥功能，那么模拟 EEPROM 中就会划分出一段区域用于存储 CSEc 模块的密钥，而这段区域又只能又 CSEc 模块控制，所以导致无法擦除 D-Flash。这个时候必须要使用 CSEc 模块恢复 Flash 至出厂状态的命令才能擦除调模拟 EEPROM 中 CSEc 模块存储密钥的区域。

2. 在遇到 1 中的情况后，使用 CSEc 模块恢复 Flash 至出厂状态不成功

当仅仅只是使能了 CSEc 模块的密钥存储功能，随后便使用 CSEc 模块恢复 Flash 至出厂状态的命令是无法执行成功的，因为这个时候所有的密钥均为空。正确的做法是，先加载 MASTER_ECU_KEY 至 CSEc 模块，然后再使用 MASTER_ECU_KEY 计算出授权码，最后使用恢复命令即可。

3. 密钥无法更新

如果密钥设置了 WRITE_PROTECT 属性，则这个不能再更新或者擦除这个密钥，仅在确定这个密钥需要更新的情况下，才能设置写保护属性，否则这个密钥永远都无法被更新和擦除。

4. 加载密钥时，返回的 M4，M5 值与计算的不相同

注意检查更计算 M1~M5 值得过程中，CID 值有没有随着密钥更新次数的增加而增加。是否密钥的通配符保护属性设置开启，但计算中确实使用的 0 作为 UID 等。

5. 使能 CSEc 时，D-Flash 分区 API 执行不成功

在分区时一定要使能模拟 EEPROM 数据自动加载到 FlexRAM 功能，因为实际 CSEc 模块读取密钥是从 FlexRAM 中读取，如果不使用自动加载功能，就无法读取到正确的密钥。在使能 CSEc 模块时如果不使能自动加载功能，则分区函数无法成功执行，可查看 S32K1xx 模拟 EEPROM 实现原理有关的文档了解详情。

6. 使用 SDK 中的 CSEC_DRV_GenerateMAC()和 CSEC_DRV_GenerateMACAddrMode()计算同一数据结果不一致

这是由于 CSEc 模块访问数据时是使用四字节的方式访问的，**AddrMode** 方式与 CSEc 模块自动计算 BootMAC 的方式相同，比如 S32K1xx 内存中的一段数据为[0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88], 在使用前者计算的时候，SDK 代码中按照地址从小到大取数据到 CSEc 模块，实际计算的数据流与内存中的数据一致，这种方式也和正常计算 MAC 值的方式相同。而 **AddrMode** 方式则是将地址传给 CSEc 模块，CSEc 模块在取数据时按照四字节的方式取值，由于 S32K1xx 使用的是小端模式，那么 CSEc 模块实际取的数据按照单字节的排列则为[0x44,0x33,0x22,0x11,0x88,0x77,0x66,0x55]，正是因为这种差异造成了计算结果的不同。值得注意的是，由于 CSEc 模块自动计算 BootMAC 的时候是采用的这种方式，所以在使用安全引导相关功能的时候应当遵循这种方式，而进行其他数据的 MAC 值生成等，应当使用 **CSEC_DRV_GenerateMAC()**的方式来实现。

PS:若发现任何错误，或有任何意见和问题，欢迎联系作者更正修改。

附录 2 计算 M1~M5 参考函数

- **KDF 函数**

```
1.  static uint8_t KDF(const uint8_t *AuthKey, const uint8_t *Constant,uint8_t *K_out)
2.  {
3.      uint8_t Res,i,Concat[32u];
4.      for (i = 0u; i < 16u; i++)
5.      {
6.          Concat[i]  = AuthKey[i];
7.         Concat[i+16] = Constant[i];
8.      }
9.      Res = CSEC_DRV_MPCompress(Concat, 2u, K_out, 2u);
10.     return Res;
11. }
```

- **计算 M1~M5 的函数**

注意： 这里 attribute_flags 参数位定义如下：

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
WRITE_PROT	**BOOT_PROT**	**DEBUG_PROT**	**KEY_USAGE**	**WILD_CARD**	**VERIFY_ONLY/0**	**0**	**0**

例如：设置 **KEY_USAGE** 为 1，其余全为 0，那么 attribute_flags 应当为 0x10。

```
1.  uint16_t CalculateM1ToM5(uint8_t *authorizing_key, uint8_t *new_key, uint8_t auth_key_id,
    uint8_t key_id, uint32_t counter, uint8_t attribute_flags,
2.  uint8_t *M1_out, uint8_t *M2_out, uint8_t *M3_out, uint8_t *M4_out, uint8_t *M5_out)
3.  {
4.      uint8_t UID[16u] = {0u}, UID_MAC[16u] = {0u};
5.      uint8_t M4_star[16u] = {0u},M4_tmp[16u]  = {0u} ,M1M2[48u] = {0u} ,K_out[16u] = {0u};
6.      uint8_t Iv[16u]      = {0u},PLAIN_TEXT[32u] = {0u};
7.      uint8_t Challenge[16u]  = {0u,1u,2u,3u,4u,5u,6u,7u,8u,9u,10u,11u,12u,13u,14u,15u};
8.      uint8_t Sreg=0u,i = 0u,Res=0u;;
9.     //For keys in the other bank, don't consider the KBS bit in M1 & M4 calculation.
    Follow traditional SHE specs
10.    /****Calculate M1****/
11.    for(i=0u;i<15u;i++)
12.        M1_out[i] = UID[i];
13.    M1_out[15u] = ((key_id & 0x0Fu)<<4u) | (auth_key_id & 0x0Fu);
14.    /****Calculate M2****/
15.    //First, calculate K1 and load K1 into RAM
16.    Res = KDF(authorizing_key, KEY_UPDATE_ENC_C,K_out);
17.    Res = CSEC_DRV_LoadPlainKey(K_out);
18.    //Second, concatenate the input data
```

```
19.     //0bit~27bit:CID  28bit~32bit:FID
20.     PLAIN_TEXT[0u] = (uint8_t)((counter &  0x0FF00000u) >> 20u);
21.     PLAIN_TEXT[1u] = (uint8_t)((counter &  0x000FF000u) >> 12u);
22.     PLAIN_TEXT[2u] = (uint8_t)((counter &  0x00000FF0u) >> 4u);
23.     PLAIN_TEXT[3u] = (uint8_t)(((counter & 0x0000000Fu) << 4u)|((attribute_flags &
    0xF0u) >> 4u));
24.     PLAIN_TEXT[4u] = (uint8_t)(attribute_flags & 0x0Fu) << 4u;
25.     //  ~127bit fill 0
26.     for(i=0u;i<11u;i++)
27.         PLAIN_TEXT[5u+i] = 0u;
28.     //128bit~255bit key id
29.     for(i=0u;i<16u;i++)
30.         PLAIN_TEXT[16u+i] = new_key[i];
31.     Res = CSEC_DRV_EncryptCBC(CSEC_RAM_KEY,PLAIN_TEXT,32u,Iv,M2_out,2u);
32.     /****Calculate M3****/
33.     //First, calculate K2 load it as RAM_KEY
34.     Res = KDF(authorizing_key, KEY_UPDATE_MAC_C,K_out);
35.     Res = CSEC_DRV_LoadPlainKey(K_out);
36.     //concatenate M1 and M2
37.     for(i=0u;  i<16u; i++)
38.         M1M2[i] = M1_out[i];
39.     for(i=16u; i<48u; i++)
40.         M1M2[i] = M2_out[i-16u];
41.     Res = CSEC_DRV_GenerateMAC(CSEC_RAM_KEY,M1M2,384u,M3_out,2u);
42.     /****Calculate M4****/
43.     Res = CSEC_DRV_GetID(Challenge,UID,&Sreg,UID_MAC);
44.     //First, calculate K3 load it as RAM_KEY
45.     Res = KDF(new_key, KEY_UPDATE_ENC_C,K_out);
46.     Res = CSEC_DRV_LoadPlainKey(K_out);
47.     //calculate K4*
48.     M4_tmp[0u] = (uint8_t)((counter &  0x0FF00000u) >> 20u);
49.     M4_tmp[1u] = (uint8_t)((counter &  0x000FF000u) >> 12u);
50.     M4_tmp[2u] = (uint8_t)((counter &  0x00000FF0u) >> 4u);
51.     M4_tmp[3u] = (uint8_t)(((counter & 0x0000000Fu) << 4u)|0x08u);
52.     Res = CSEC_DRV_EncryptCBC(CSEC_RAM_KEY,M4_tmp,16u,Iv,M4_star,2u);
53.     //0bit ~ 119bit UID
54.     for(i=0u;i<15u;i++)
55.         M4_out[i] = UID[i];
56.     //120bit ~ 1237bit KeyID(4bits)|AuthID(4bits)
57.     M4_out[15u] = (key_id << 4u)|(auth_key_id & 0x0Fu);
58.     //128bit ~ 255bit M4*
59.     for(i=0u;i<16u;i++)
60.         M4_out[16u+i] = M4_star[i];
61.     /****Calculate M5****/
```

```
62.      Res = KDF(new_key, KEY_UPDATE_MAC_C,K_out);
63.      Res = CSEC_DRV_LoadPlainKey(K_out);
64.      Res = CSEC_DRV_GenerateMAC(CSEC_RAM_KEY,M4_out,256u,M5_out,2u);
65.      return Res;
66. }
```

附录 3 计算恢复出厂设置授权码参考函数

```
1.  uint16_t CalculateDbgAuth(uint8_t* MasterEcuKey,uint8_t * Challenge,uint8_t *DbgAuthOut)
2.  {
3.      uint8_t K_out[16u],UID_MAC[16u] = {0u},DATA[32u];
4.      uint8_t UID[16u] = {0u};//UID just have 15 bytes
5.      uint8_t UidChallenge[16u]={0u,1u,2u,3u,4u,5u,6u,7u,8u,9u,10u,11u,12u,13u,14u,15u};
6.      uint16_t Res=0u;
7.      uint8_t Sreg=0u,i=0u;
8.      //step1 Calculate the Authorization
9.      Res = KDF(MasterEcuKey, DEBUG_KEY_C,K_out);
10.     Res = CSEC_DRV_LoadPlainKey(K_out);
11.     Res = CSEC_DRV_GetID(UidChallenge,UID,&Sreg,UID_MAC);
12.     for(i=0u; i<16u; i++)
13.         DATA[i] = Challenge[i];
14.     for(;i<32u;i++)
15.         DATA[i] = UID[i-16u];
16.     Res = CSEC_DRV_GenerateMAC(CSEC_RAM_KEY,DATA,248u,DbgAuthOut,2u);
17.     return Res;
18. }
```

附录 4 量产建议

由于 CSEc 模块所有密钥都只能 CSEc 模块管理，这样才能确保密钥安全的安全性。虽然密钥实际存储区域是在模拟 EEPROM 中，但由于这段区域是由 CSEc 管理并且对用户是不可见的，所以也无法通过直接对 D-Flash 编程的方式写入密钥。在量产时，可以使用一份专用于写入密钥的程序，通过先刷写这份程序，等写入密钥后，再刷写正式的应用程序即可。需要注意的时，两份程序需要有一致的 D-Flash 分区策略。

当然也可以通过诊断服务的方式加载密钥。先脱机计算出密钥的 M1~M5 值，然后通过诊断服务（如 2Eh 服务）将 M1~M3 加载至 CSEc 模块中，并通过 M4 和 M5 验证是否加载成功。由于不能通过 M1~M3 值逆推算出密钥值，所以这种方式也是安全的。

由于使能 CSEc 模块后，只能通过 CSEc 模块的恢复命令（使用调试器也不行）才能擦除 D-Flash 数据并恢复 D-Flash 至出厂状态，所以如果在量产后需要再次擦除 D-Flash，或者重新对 D-Flash 进行分区等，可以在用户代码中加入相关程序，通过触发这段程序来使用 CSEc 模块的命令擦除 D-Flash。CSEc 模块对 P-Flash 的重新编程没有影响。

附录 5 SDK 3.0.0RTM 版本 API 说明

函数名	参数	说明
CSEC_DRV_EncryptECB	keyIdL：加密使用的密钥 ID plaintext：待加密的明文 length：待加密的明文长度 cipherText：加密后输出的密文 timeout：超时时间	ECB 模式下的 AES-128 加密
CSEC_DRV_DecryptECB	keyId：解密使用的密钥 ID cipherText：待解密的密文 length：密文长度 plainText： 解密后的输出的明文 timeout：超时时间	ECB 模式下的 AES-128 解密
CSEC_DRV_EncryptCBC	keyId：加密使用的密钥 ID plaintext：待加密的明文 length：待加密的明文长度 iv：初始化向量 cipherText：加密后输出的密文 timeout：超时时间	CBC 模式下的 AES-128 加密
CSEC_DRV_DecryptCBC	keyId：解密使用的密钥 ID cipherText,：待解密的密文 length：密文长度 iv：初始化向量 plaintext：解密后的输出的明文 timeout：超时时间	CBC 模式下的 AES-128 解密
CSEC_DRV_GenerateMAC	keyId：使用的密钥 ID msg：待计算的数据 msgLen：待计算的数据长度（bits） cmac：计算输出的 MAC 值 timeout：超时时间	计算数据的 MAC 值
CSEC_DRV_VerifyMAC	keyId：使用的密钥 ID msg：待验证 MAC 的数据 msgLen：待验证 MAC 的数据长度 mac：待验证 MAC 数据的 MAC 值 macLen：待验证 MAC 数据的 MAC 值长度 verifStatus：输出的验证结果 timeout：超时时间	验证数据的 MAC 值
CSEC_DRV_LoadKey	keyId: 使用的密钥 ID m1~m3：密钥的 M1~M3 值 m4~m5：CSEc 模块输出的 M4~M5 值，用于验证加载密钥是否成功	加载密钥到 CSEc 模块
CSEC_DRV_LoadPlainKey	plainKey：加载到 RAM_KEY 区域的密钥明文	以明文的方式加载密钥到 RAM_KEY 区域
CSEC_DRV_ExportRAMKey	m1~m5：导出的 M1~M5	导出 RAM_KEY 中的密钥导出为 M1~M5 的格式

CSEC_DRV_GenerateRND	Rnd：输出的随机值	生成随机数
CSEC_DRV_BootDefine	bootSize：设置 Boot 大小（起始地址从 0 开始计算） bootFlavor：安全引导模式	定义 Boot 区域
CSEC_DRV_GetID	challenge：查询数据输入（可使用0或一个随机数） uid：输出的UID sreg：输出的状态寄存器值 mac：输出 challenge,uid 和 sreg 合成数据的 MAC 值，详见芯片手册 CMD_GET_ID 部分	获取 MCU 的 UID
CSEC_DRV_DbgChal	Challenge：输出的用于恢复出厂设置的随机值，随后通过这个随机值，计算出授权码	获取用于恢复出厂设置的随机值
CSEC_DRV_DbgAuth	Authorization：输入用于恢复出厂设置的授权码	恢复出厂设置