# X-Cube-SBSFU 使用技巧（之二）——与应用集成

*关键字：安全启动，安全升级，SBSFU，签名校验，加解密*

## 1. 引言

安全启动与安全更新是嵌入式设备的基本安全要求之一：安全启动提供信任根，保证每次设备启动运行的应用程序的完整性与合法性；安全更新则在固件升级环节避免系统软件被恶意修改或替换。

为了方便客户在其嵌入式设备的设计中更加容易地集成安全启动和安全更新功能，STM32 提供了相关的参考实现并支持众多 STM32 MCU 系列。其中 X-CUBE-SBSFU 是针对 Cortex V6/V7M 内核的 STM32 MCU 产品的参考方案，软件包下载地址：*https://www.st.com/x-cube-sbsfu*。

在 X-CUBE-SBSFU 使用技巧的第一篇，我们对软件包及其软件架构等进行了介绍，读者对这个软件包有了初步认识。在这一篇我们将介绍 SBSFU 和用户应用程序集成相关的内容。

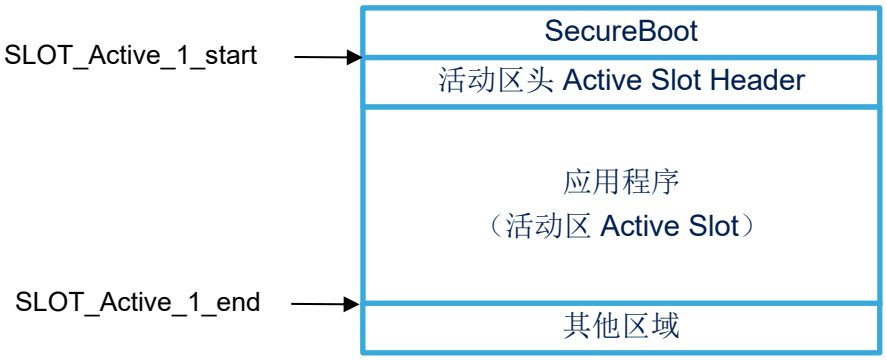
## 2. 应用程序链接文件

### 2.1. Flash

通常一个不带有安全启动或者 bootloader 的应用程序会以内部 Flash 的基地址作为该应用程序对应 vector table 的起始地址，并可以运行于整个内部 Flash 空间。当在应用中集成 SBSFU 参考实现的时候，需要调整应用程序在 Flash 中的占位。

在 X-CUBE-SBSFU 使用技巧的第一篇中我们介绍了以 X-CUBE-SBSFU 为框架的安全启动方案的存储器地址空间布局，单镜像模式、双镜像模式、内部+外部 Flash 模式等等，可能会有不尽相同的空间布局。但是对于应用程序来说，只需要关注其中的 Active Slot 部分，应用程序在 Flash 中的放置位置由 Memory Map 中的 Active Slot 区域决定，如图 1 所示。

图1. 用于指定应用 Flash 地址的 Active Slot




Active Slot 的地址在 SBSFU 参考实现的 Linker_Common 目录下的 mapping_fwimg.x 文件中有定义。那么修改应用 Linker file 的方式有两种，一种方式是根据 mapping_fw.x 中的 Active slot 定义直接在应用程序中 hard code 相关地址。另一种是在应用的 Linker file 中引用 mapping_fw.x 文件，并使用其中的定义来指定 vector table、ROM 区域的起止地址等，这种方式的好处是，应用程序与 SBSFU 引用相同的 linker 定义文件，当需要做地址空间调整的时候只需要统一修改，以避免地址定义不一致的问题。

**注意**：

- 应用程序的实际起始地址并非是 Active Slot 的开始地址，因为当应用程序位于内部 Flash 的时候，Active Slot 包含固件头（header）信息和固件本身，所以应用程序 vector table 的实际起始地址应该是 Active Slot 开始地址（SLOT_Active_1_start）+固件头偏移量。固件头偏移量的大小在不同系列上可能不同，例如 L4 系列的偏移量为 512 字节，而 H7 系列的偏移量大小为 1024 字节，所以这个偏移量需要参考对应系列的 SBSFU 实现中使用的具体定义（例如 UserApp 示例工程的 linker file 中的定义）。如果是应用程序运行于外部 Flash 的情况，Active Slot Header 依旧会位于内部 Flash，而 Active Slot 则在外部 Flash，此时，Active Slot 与 Header 没有重叠部分，外部 Flash 上的应用程序起始地址就是 Active Slot 的开始地址。

stm32h753xx_flash.icf

```
10  include "mapping_sbsfu.icf";
11  include "mapping_fwimg.icf";
12
13  /*-Specials-*/
14  define exported symbol __ICFEDIT_intvec_start__ = __ICFEDIT_SLOT_Active_1_start__ + 1024;
15  /* Cortex-M7 : must be a multiple of 1024 */
```

- 如果应用程序中需要预留一部分 Flash 空间作为 NVM 存储区使用，那么这部分应当与现有 SBSFU 方案定义的几个区域隔开，必须放在 Active Slot，Download Slot，Swap 区以及 SBSFU 区以外的空间。如果 NVM 区放在 Active slot 区以内，则应用运行时一旦修改了 Active Slot 区内的 NVM 数据，将会导致下次启动时待运行的当前版本应用程序无法通过安全启动校验从而启动失败。如果作为参考基础的 SBSFU 示例中的 mapping 定义已经使用了所有内部 Flash 空间，则需要调整 Active Slot，Download Slot 的地址和大小，swap 区可能也需要跟着修改。
- 当 vector table 起始地址在 linker file 中修改时，systemInit 函数中的 VTOR 应当相应修改。可以像参考代码那样，通过引用“__vetor_table”的方式设置 VTOR，例如

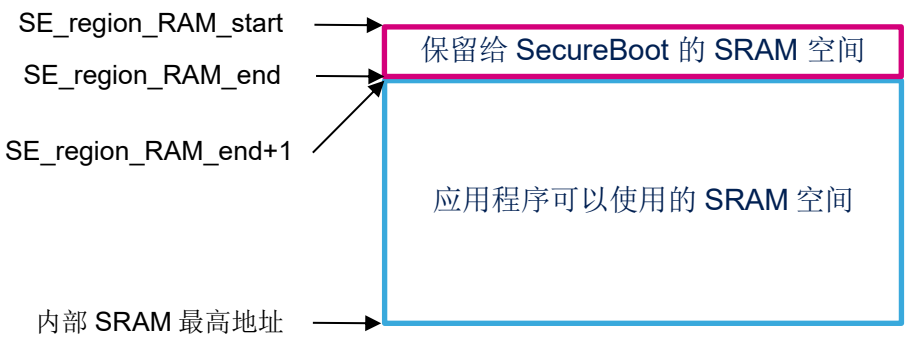
```
#if defined(__ICCARM__)
extern uint32_t __vector_table;
#define INTVECT_START ((uint32_t)& __vector_table)
#elif defined(__CC_ARM)
```

```
extern void * __Vectors;
#define INTVECT_START ((uint32_t) & __Vectors)
#elif defined(__GNUC__)
extern void * g_pfnVectors;
#define INTVECT_START ((uint32_t)& g_pfnVectors)
#endif

void SystemInit (void)
{
  …
  SCB->VTOR = INTVECT_START;
  …
}
```

## 2.2. SRAM

通常情况下，安全启动只在运行期间使用 SRAM，跳转到应用程序之后，SRAM 会被释放给应用程序使用，因此应用程序可以使用所有的 SRAM 空间。但是在某些平台的 SBSFU 参考实现中，部分 SRAM 会保留给安全启动代码使用，主要涉及带有 Firewall 硬件的 L4/L0 系列，以及 SBSFU 中包含 KMS 功能，启动后 SecureEngine 依旧可能给应用程序提供密钥管理服务的情况。这时候应用程序的 linker file 中指定 的 SRAM 的空间应该从 `SE_region_RAM_end` 地址之后开始，如图 2 所示：

图2. 部分 SRAM 保留给 SBSFU 使用的情况

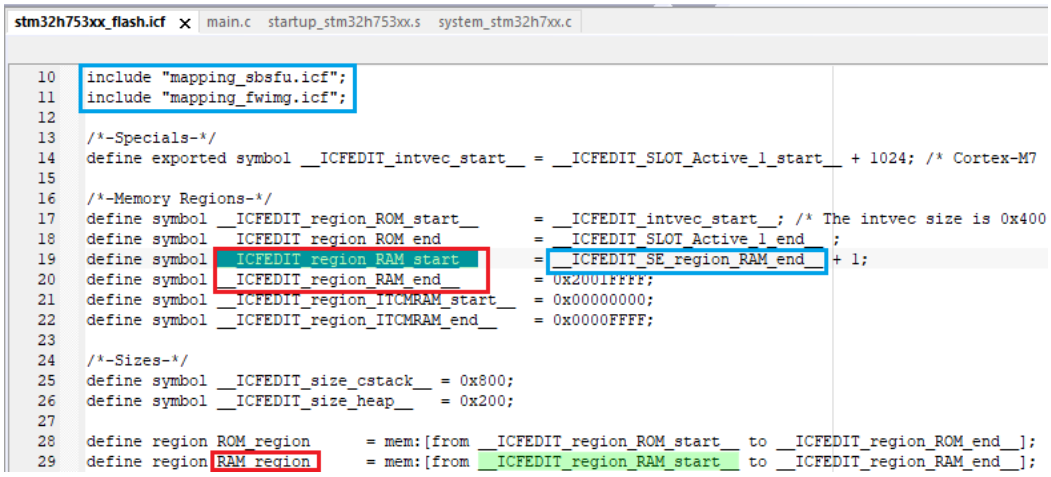



`SE_region_RAM_end` 地址定义在 Linker_Common 目录下的 mapping_sbsfu.x 文件中有定义：

```
/* SE RAM1 region protected area */
define exported symbol __ICFEDIT_SE_region_RAM_start__      = 0x20000000;
define exported symbol __ICFEDIT_SE_region_RAM_stack_top__  = 0x20000400;
define exported symbol __ICFEDIT_SE_region_RAM_end__        = 0x20000FFF;
```

应用程序的 linker file（如图 3）可以通过包含 mapping_sbsfu.x，引用对“SE_region_RAM_end”的定义，来定义应用程序自己的 SRAM 起始地址：

图3. Linker file 中引用 mapping 定义




## 2.3. 应用程序 binary 的填充

为了实现安全启动的固件验证，我们需要对最终的应用程序进行签名、生成固件头、打包等操作，签名打包通常处理的是应用代码的二进制文件，所以在应用程序的工程中除了链接生成 elf 文件外，还需要将 elf 转换成 raw binary 格式。

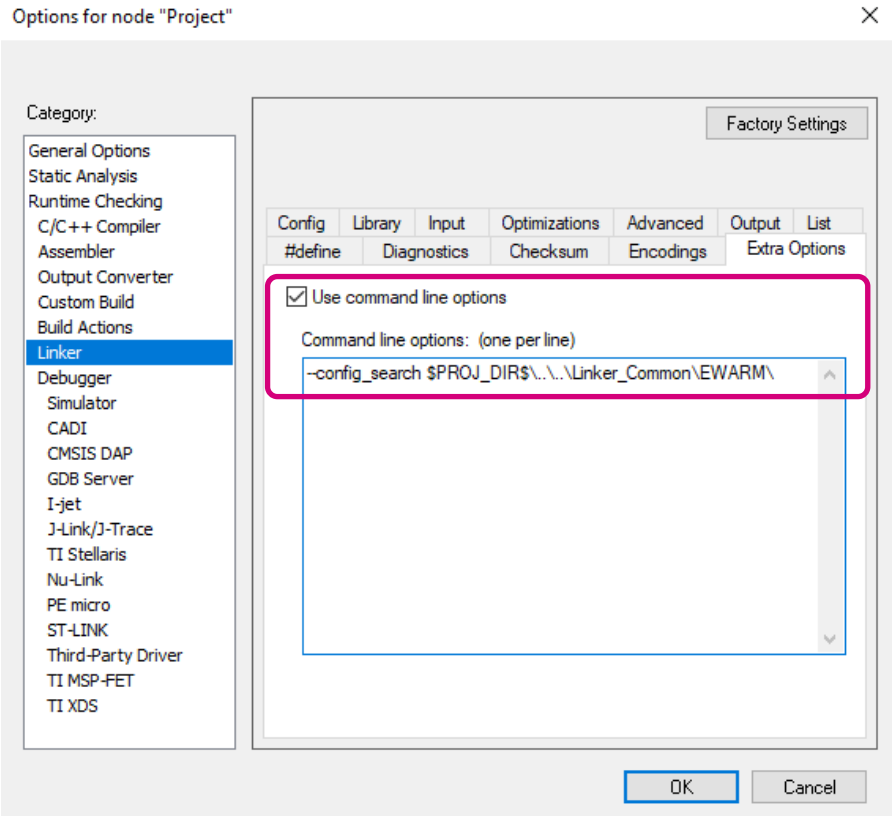
另外，如果选择了带 AES 加密的 Crypto 方案，那么在最终应用程序的签名打包过程中还需要对应用二进制文件进行 AES 加密操作，AES 加密需要保证被加密的二进制文件的大小为 16 字节的倍数，因此，在 linker file 中需要做额外的填充（参见 2.4 节），以便满足这个 binary 大小 16 字节对齐的要求。UserApp 的示例工程中同样有这部分示例。不同的编译器的做法不尽相同，需要参考对应编译器工程的做法（参见 2.4 节）。

## 2.4. 应用工程修改示例

### 2.4.1. IAR

在工程中指明 Linker_Common 文件搜索路径，如图 3 所示，在 Linker 选项的 Extra Options 页勾选“Use command line options”，并在“Command line options:”框中填入“--config_search”命令，后面跟实际的 Linker_Common 目录的路径。

图4. IAR 工程中增加 Linker 搜索路径



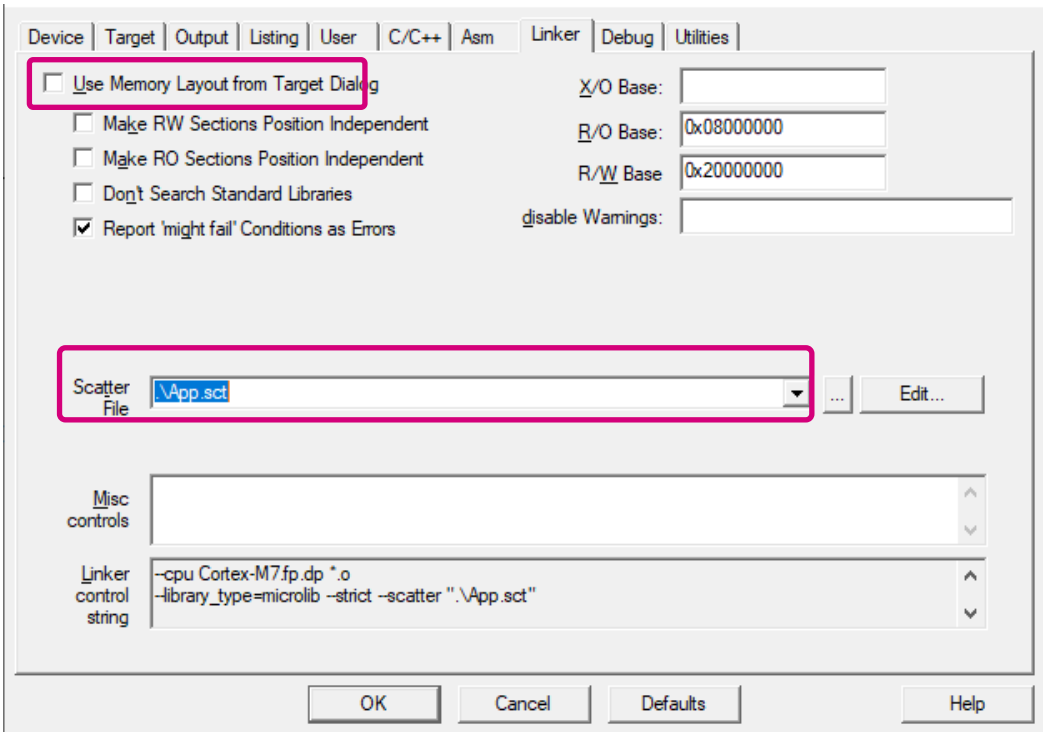

Linker file 的主要修改

```
define memory mem with size = 4G;

/* 包含 Linker Common 目录下的两个 mapping 文件 */
include "mapping_sbsfu.icf";
include "mapping_fwimg.icf";

define exported symbol __ICFEDIT_intvec_start__ = __ICFEDIT_SLOT_Active_1_start__ + 512;
/* 这里 512 字节为 Header size, Header size 可能随不同系列有所不同，例如 H7 的 Header size 为 1024 */

define symbol __ICFEDIT_region_ROM_start__      = __ICFEDIT_intvec_start__;
define symbol __ICFEDIT_region_ROM_end__        = __ICFEDIT_SLOT_Active_1_end__ ;
define symbol __ICFEDIT_region_RAM_start__      = __ICFEDIT_SE_region_RAM_end__ + 1;
/* RAM 的起始地址是否需要在 SE_REGION_RAM_END 之后取决于具体的芯片型号 */

…

/* 如果选择了 AES 加密，需要增加为了 AES 16 字节对齐的填充 */
define root section aes_block_padding with alignment=32
{
udata8 "Force Alignment";
pad_to 32;
};

place in ROM_region   { readonly, last section aes_block_padding };

…
```

## 2.4.2. KEIL

某些应用可能使用缺省的 memory layout 进行链接，而集成 SBSFU 的时候需要指定 linker 文件，如图 5 示例。

图5.　　KEIL 工程中指定链接的 scatter table 文件




## Scatter table 文件示例

```
/* 指定额外的 Linker common 文件的搜索路径，根据实际路径来写 */
#! armcc -E -I.\

; *************************************************************
; *** Scatter-Loading Description File generated by uVision ***
; *************************************************************

/* 包含 Linker Common 目录下的两个 mapping 文件，这里是相对前面-I 指定的路径 */
#include "..\..\Linker_Common\MDK-ARM\mapping_sbsfu.h"
#include "..\..\Linker_Common\MDK-ARM\mapping_fwimg.h"

/* 这里的起始地址为 Active Slot 开始地址+Header size, header 大小根据所选芯片型号不同可能不同 */
LR_IROM1 SLOT_ACTIVE_1_START + 0x200   {    ; load region size_region
  vector_start (SLOT_ACTIVE_1_START + 0x200 ) FIXED VECTOR_SIZE {
  *.o (RESET, +First)
  }
  ROM_region +0 {
    *(InRoot$$Sections)
    .ANY (+RO)
    .ANY (+XO)
  }

/* RAM region 的起始地址是否需要放在 SE_REGION_RAM_END 之后也取决于具体 MCU 型号 */
  SB_RAM_region (SE_REGION_RAM_END + 1) {
    .ANY (STACK)
    .ANY (HEAP)
    .ANY (+RW +ZI)
    }
  }

/* 如果选择了 AES 加密，需要增加为了 AES 16 字节对齐的填充，这里引用到的 ALIGNTOAESBLOCK 定义在
startup_stm32xxx.s 中，*/
LR_ROM1(+0) ALIGN(32) {
  ForAlignment +0 {
    startup_stm32xxx.o (ALIGNTOAESBLOCK,+Last)
  }
```

```
}
```

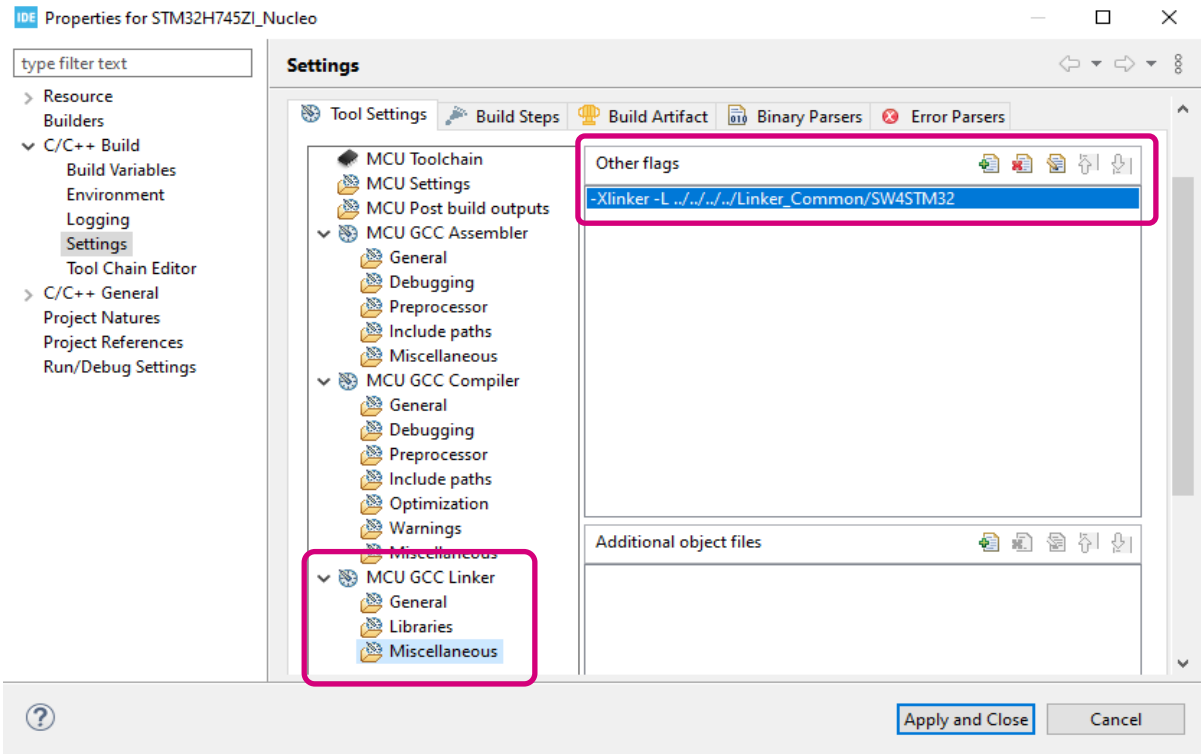
以 H7 为例，填充 16 字节对齐部分引用的 startup_stm32xxx.s 中的代码如下

```
…
;*******************************************************************************
; Element for aligning the Firmware binary size on the AES block size (16 bytes)
; and H7 flash writing unit (32 bytes)
;*******************************************************************************
  AREA ALIGNTOAESBLOCK, DATA, READONLY, MERGE=1, STRINGS
                DCB     "0123456789ABCDEF0123456789ABCDE",0

                END
```

### 2.4.3. CubeIDE

在工程中指明 Linker_Common 文件搜索路径，如图 6 所示

图6. CubeIDE 工程中增加 Linker 搜索路径




Linker 文件的主要修改

```
…

/* 包含 Linker Common 目录下的两个 mapping 文件 */
INCLUDE mapping_fwimg.ld
INCLUDE mapping_sbsfu.ld

/* 根据 mapping 定义增加 vector table 起始地址、ROM 起始地址大小、RAM 起始地址大小的定义 */
__ICFEDIT_intvec_start__   = __ICFEDIT_SLOT_Active_1_start__     + 0x200;
/* 同样header大小需要根据实际选择的芯片型号而定 */
APPLI_region_ROM_start     = __ICFEDIT_SLOT_Active_1_start__     + VECTOR_SIZE + 0x200;
APPLI_region_ROM_length    = __ICFEDIT_SLOT_Active_1_CM7_end__     - APPLI_region_ROM_start + 1;
APPLI_region_RAM_start     = __ICFEDIT_SE_region_RAM_end__ + 1;
APPLI_region_RAM_length     = 0x20020000 - __ICFEDIT_SE_region_RAM_end__ - 1;
```

```
MEMORY
{
 ISR_VECTOR (rx)   : ORIGIN = __ICFEDIT_intvec_start__, LENGTH = VECTOR_SIZE
 FLASH  : ORIGIN = APPLI_region_ROM_start, LENGTH = APPLI_region_ROM_length
 RAM  : ORIGIN = APPLI_region_RAM_start, LENGTH = APPLI_region_RAM_length
 …
}

SECTIONS
{
  /* 指定vector table的位置 */
  .isr_vector :
  {
    . = ALIGN(8);
    KEEP(*(.isr_vector)) /* Startup code */
    FILL(0);
    . = ORIGIN(ISR_VECTOR) + LENGTH(ISR_VECTOR) - 1;
    BYTE(0)
    . = ALIGN(8);
  } >ISR_VECTOR

  /* 其他 section 可以按照原有定义，对于选择了 AES 加密的情况，需要增加填充的部分 */
  …

  .align32 :
  {
    . = . + 1;
    . = ALIGN(32) - 1;
    BYTE(0);
  } > FLASH

  …
}
```

# 3. 应用程序固件的签名加密打包

## 3.1. 根据所选择的 Crypto 方案生成 OEM 自己的密钥

SBSFU 用到的密钥主要有两类，一个是 ECC 密钥（包含公私钥对），用于 ECDSA 签名和验签；一个是 AES 密钥（对称加解密密钥），用于应用程序固件 binary 的加解密。AES 密钥为二进制文件，长度为 16 字节，用户可以自己生成密钥的二进制文件。ECC 密钥为 PEM 格式，可以通过类似 openssl 的 PC 端工具生成 ECC 私钥文件。

SBSFU 软件包也提供了的 prepareimage 工具，可以生成用户自己的密钥，该工具位于 Middlewares\ST\STM32_Secure_Engine\Utilities\KeysAndImages\目录，如果 PC 系统有 Python 工具，可以直接使用 prepareimage.py，没有 python 也可以使用 win 子目录下的 prepareimage.exe。prepareimage 生成密钥的命令用法为

```
prepareimage keygen -k %keyfile% -t %keytype%
```

其中 keyfile 是生成的密钥文件，keytype 是密钥类型，可以是 ecdsa-p256，aes-gcm，aes-cbc，aes-ctr 中的一种。

以下是几个生成密钥的命令的例子

- 生成 ECDSA 签名用的 ECC 密钥

```
prepareimage keygen -k ecdsa-p256.key -t ecdsa-p256
```

生成的 ECC 密钥文件可以用文本的方式查看到其内容，例如

```
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIJLr5GK7SE8iahQEdv8KHd9zb0NOGSYjMFQ5XIMy15PToAoGCCqGSM49
AwEHoUQDQgAEjzZ694RAcGHLM397zXXMzd7lpt4yhxz5naGduuFlibmp/Q6QZwLw
aswJTXOO39tGaTGG3tsyyrLLCSbAy3rCUg==
-----END EC PRIVATE KEY-----
```

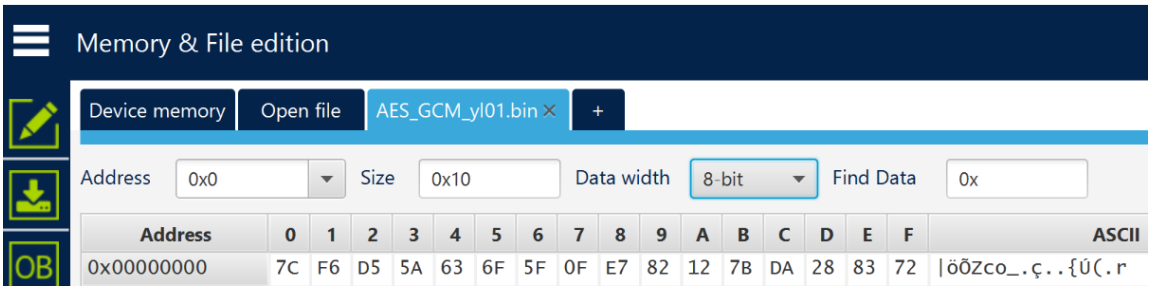
- 生成 AES GCM 密钥

```
prepareimage keygen -k AES_GCM.key -t aes-gcm
```

生成的 AES 密钥文件可以通过 xxd 等工具查看其 16 进制内容，例如

```
$ xxd -g 1 AES_GCM.key
00000000: 5e e8 60 21 7f 3b 89 78 f8 a0 fd 40 06 5a 11 af  ^.`!.;.x...@.Z..
```

也可以把后缀名重命名到 bin，比如 AES_GCM.bin，拖到 STM32CubeProgramer 的 Memory & File edition 窗口查看




- 生成 AES CBC 或 AES CTR 密钥（文件名需要包含 AES_CBC/AES_CTR 字符串）

```
Prepareimage keygen -k AES_CBC.key -t aes-cbc
```

```
Prepareimage keygen -k AES_CTR.key -t aes-ctr
```

密钥内容的查看与 AES GCM 密钥类似，这里不做重复。

哪些密钥需要生成是根据选定的 Crypto 方案决定的，例如选择只做 ECC 签名，不加密，那么只需要生成 ECC 密钥即可；如果选择 AES GCM 方式同时做加密签名，那么只需要生成 AES GCM 密钥，ECC 密钥不需要；如果选择 ECDSA 签名+AES CBC 加密的方式，那么需要生成 ECC 和 AES CBC 两组密钥。生成的文件将会用于后续的应用 Binary 签名、加密、打包处理。

## 3.2. 应用程序固件头内容

安全启动需要对应用程序固件进行校验，因而 Flash 中除了应用固件本身还包含一个固件头，用来提供校验过程所需要的信息，例如固件大小，版本，签名数据等等。固件头的内容根据所选择的 Crypto 方案略有不同，表 1 给出了 SBSFU 参考实现中几种不同 Crypto 方案配置的固件头的数据结构，在 se_def_metadata.h 文件可查看详细定义

**表1. 固件头结构和内容**

| 数据 | 大小(byte) | | | 说明 |
|---|---|---|---|---|
| | ECDSA 签名 +AES CBC 加密 | ECDSA 签名 无加密 | AES GCM Tag+加密 | |
| SFUMagic | 4 | | | Magic number,缺省 active slot 1 为 “SFU1”，active slot 2 为“SFU2”，以此类推 |
| ProtocalVersion | 2 | | | 未使用 |
| FWVersion | 2 | | | 固件版本 |
| FWSize | 4 | | | 被校验的固件大小字节数 |
| PartialFWOffset | 4 | | | 部分升级时的固件偏移量 |
| PartialFWSize | 4 | | | 部分升级时的固件大小 |
| FWTag | 32 (SHA256 HASH) | | 16 (GCM Tag) | 固件 Hash 或 Tag |
| PartialFWTag | 32 (SHA256 HASH) | | 16 (GCM Tag) | 部分升级时的固件 Hash 或 Tag |
| Nonce | 0 (无此项) | | 12 | AES GCM 的 Nonce |
| InitVector | 16 | 0 (无此项) | | AES CBC 的 IV |
| Reserved | 28 | 44 | 112 | 保留字段，使得 Header 从头到 Signature 字段（包含）的总大小为 192 字节 |
| HeaderSignature | 64 (ECDSA 签名) | | 16 (AES GCM Tag) | 固件头签名，被签名部分为头开始到 Reserved 字段结束（黄色高亮部分） |
| FwImageState | 3x32 | | | 固件状态，包含 INAVLID, VALID, VALID_ALL, SELFTEST, NEW 等几种状态 |
| PrevHeaderFingerprint | 32 | | | 前一个版本的固件头指纹，只在升级过程中使用，当出现需要回退到前一个版本的时候，这个字段用于判断此次升级是否来源于之前 Active Slot 中的版本 |

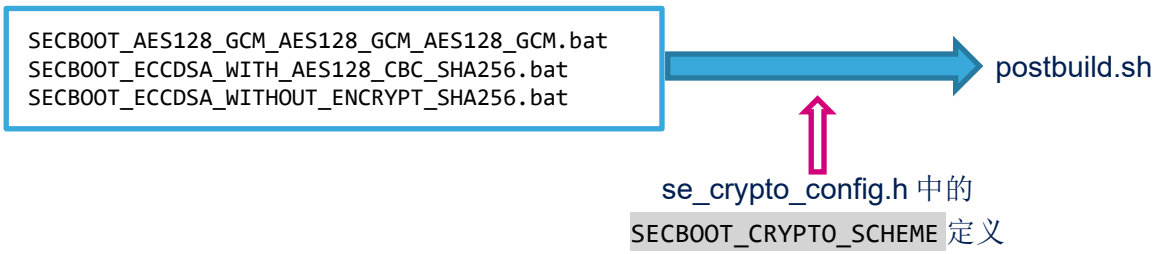
只有构造了正确的固件头信息，安全启动过程才能够成功地对应用程序固件进行校验，完成安全启动的流程。固件头的构造涉及加密、签名、打包等动作，为了方便完成这一系列操作，SBSFU 参考实现中提供了打包工具以及 postbuild 脚本，帮助完成固件头的生成和组装。

## 3.3. 使用 postbuild 脚本

### 3.3.1. Postbuild 脚本的来源

X-CUBE-SBSFU 软件包提供了用户应用程序的签名、加密、打包的参考，每一个 UserApp 的工程中都配合有 build action 以及 postbuild 脚本。Postbuild 脚本来自于

SECoreBin 工程的编译输出，SECoreBin 的 post build action 会根据 Crypto 方案的选择从几个脚本生成最终的 postbuild.bat/sh。




### 3.3.2. 在应用工程中直接调用 postbuild 脚本

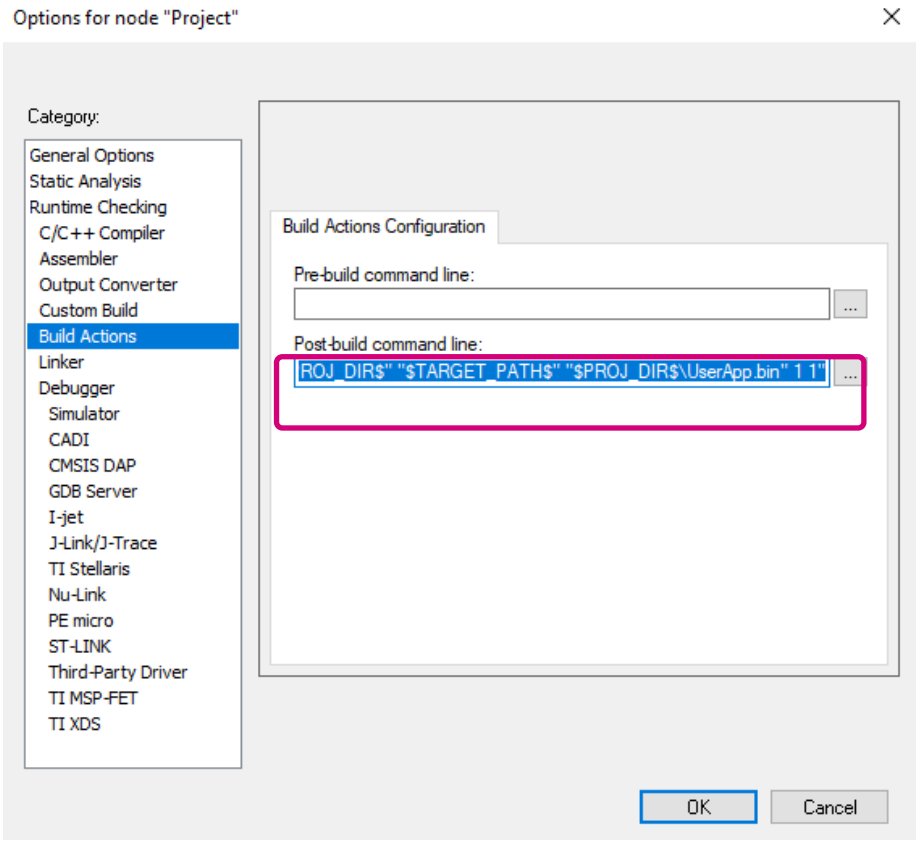
postbuid 脚本可以从 windows command 窗口手动运行，也可以从编译器集成开发环境中直接调用 postbuild 脚本。不同编译器的 build action 和 postbuild 脚本调用方式不同

IAR:

通过 post-build command line 调用 postbuild 脚本，如图 7 所示。

```
cmd /C "cmd /C $PROJ_DIR$\..\..\2_Images_SECoreBin\EWARM\PostBuild.bat "$PROJ_DIR$"
"$TARGET_PATH$" "$PROJ_DIR$\UserApp.bin" 1 1"
```

图7. IAR 工程调用 postbuild 脚本



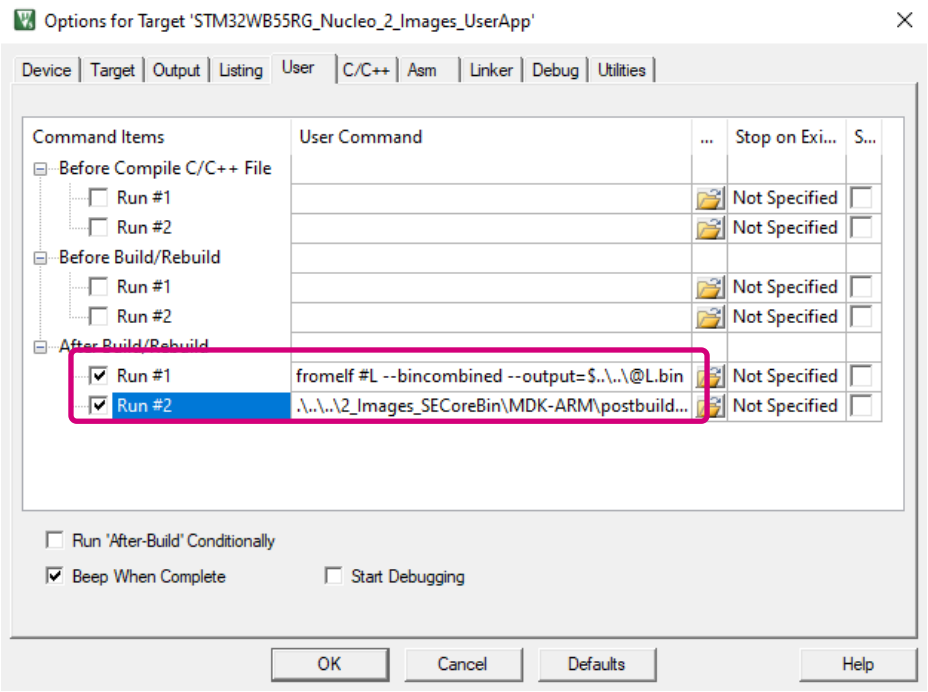
KEIL：

通过 User After Build/Reuild 设置选项调用 postbuild 脚本，如图 8 所示。

其中的第一个步骤是从 elf 生成 binary，第二个步骤是调用 postbuild 脚本，例如：

```
cmd /C "cmd /C $PROJ_DIR$\..\..\2_Images_SECoreBin\EWARM\PostBuild.bat "$PROJ_DIR$"
"$TARGET_PATH$" "$PROJ_DIR$\UserApp.bin" 1 1"
```

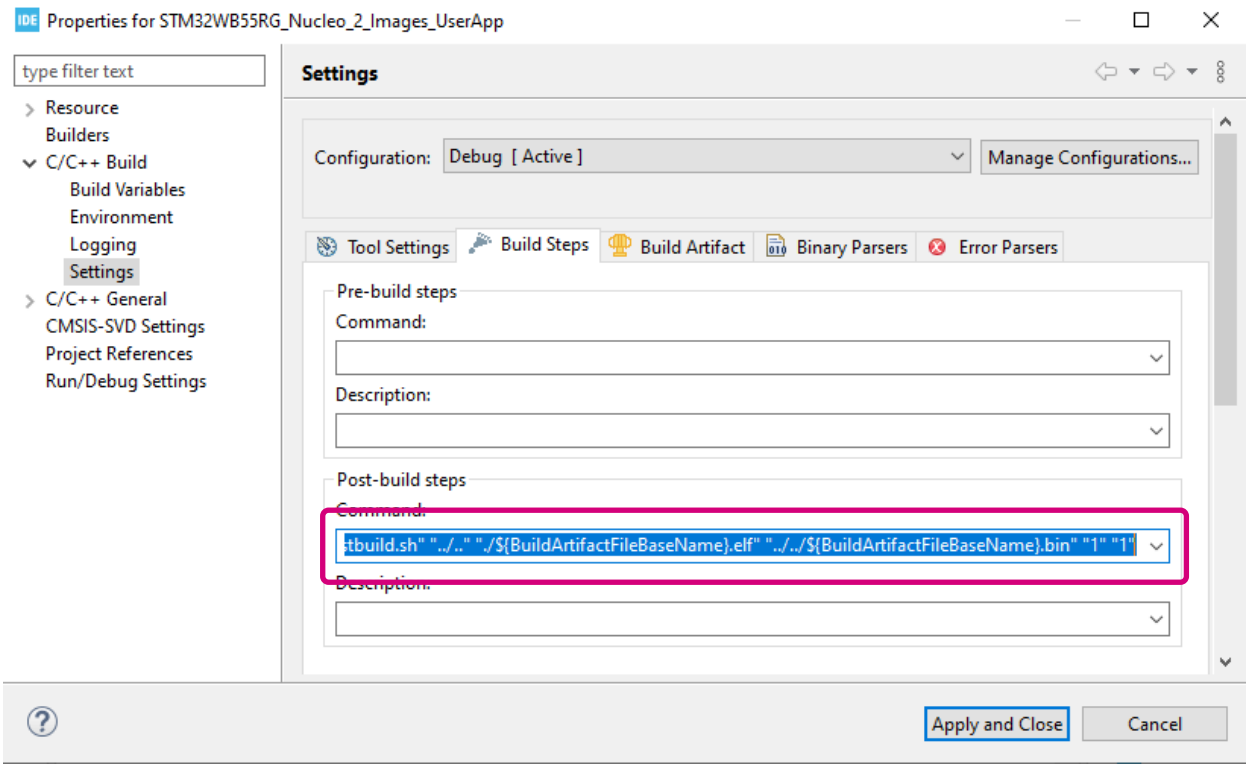
图8. KEIL 工程调用 postbuild 脚本




CubeIDE：

通过 post-build command line 调用 postbuild 脚本，如图 9 所示。

图9. CubeIDE 工程调用 postbuild 脚本




其中的 post-build steps 的命令包含几个动作，包括从 elf 生成 binary，调用 postbuid 脚本，例如

```
arm-none-eabi-objcopy -O binary "${BuildArtifactFileBaseName}.elf"
"../../${BuildArtifactFileBaseName}.bin" &&
arm-none-eabi-size "${BuildArtifactFileName}" &&
"../../../../2_Images_SECoreBin/STM32CubeIDE/postbuild.sh" "../.."
"./${BuildArtifactFileBaseName}.elf" "../../${BuildArtifactFileBaseName}.bin" "1" "1"
```

### 3.3.3. postbuild 脚本的参数和使用

postbuild 脚本通常需要 5 个输入参数

- 参数 1：应用工程目录，
  - 对于 IAR,KEIL 来说就是工程文件所在目录，例如 xxx\EWARM， xxx\MDK-ARM
  - 对于 CubeIDE 来说，就是工程文件所在目录的上一级目录，例如 xxx\STM32CubeIDE
- 参数 2：应用工程编译生成的 elf 文件，包括完整路径+文件名
- 参数 3：应用工程编译生成的 binary 文件，包括完整路径+文件名
- 参数 4：应用固件的 ID，通常为 1
- 参数 5：应用固件的版本，整数（大小为不超过 16bit 能表示的整数）

postbuild 中还会用到 **SBSFU**、**SECoreBin** 以及 Middleware 目录下的其他一些文件，脚本会以第一个参数“应用工程目录“给出的路径为基础，找到相对路径下的其他几个目录和文件，主要涉及：

- 存放应用程序打包输出文件（userAppBinary）的目录：缺省为 %projectdir%\..\Binary\
- 签名加密使用的 Key 所在的目录：缺省为 %projectdir%\..\..\xxx_SECoreBin\Binary
- SBSFU 工程的 elf 文件：缺省为%projectdir%\..\..\2_Images_SBSFU\目录中对应 IDE 目录下的 elf 输出文件
- prepareimage 工具所在目录：prepareimage 是真正完成签名、加密、打包的工具，缺省路径为 %projectdir%\..\..\..\..\..\..\Middlewares\ST\STM32_Secure_Engine\Utilities\KeysAndImages\
- 参考 Appbinary（非必需）：用于基于现有版本生成 partial 升级包，缺省为 %projectdir%\RefUserApp.bin

由此可见，postbuild 脚本假设了目录结构遵循 X-CUBE-SBSFU 软件包中的目录结构，如果应用程序工程并没有像 UserApp 一样与 SBSFU 的其他几个工程在同一个目录下，或者应用工程中的目录层次与 UserApp 下面的层次不同，那么这个 postbuild 脚本不一定能直接正常工作，需要对其中的几个相对路径做修改，确保在脚本中正确设置前面提到的几个路径。

## 3.4. 自行编写脚本

通常在应用程序已经确定采用哪种 Crypto 方案和使用哪种 IDE 的情况下，不需要完全使用 X-CUBE-SBSFU 参考软件包中给出的那样的比较灵活的脚本，用户完全可以自己写一个相对固定但比较简单的脚本。和已有的 postbuild 脚本类似，即可以在应用工程 中直接调用脚本，也可以在 console 窗口手动运行脚本。

脚本中的关键操作主要是运行 prepareimage 工具的命令，完成几个签名、加密、打包的操作，生成最终所需要的镜像文件。运行 prepareiamge -h 命令可以列出该工具支持的所有命令类型。

```
prepareimage -h
usage: prepareimage [-h]

{keygen,trans,getpub,sign,sha256,conf,extract,enc,header,pack,diff,merge,mergev2,appe
nd,inject}
                    ...

positional arguments:

{keygen,trans,getpub,sign,sha256,conf,extract,enc,header,pack,diff,merge,mergev2,appe
nd,inject}
                      subcommand help
  keygen              Generate pub/private keypair
  trans               translate key to execute only code
  getpub              Get public key from keypair
  sign                Sign an image with a private key
  sha256              hash a file with sha256
  conf                get crypto config from .h file
  extract             get value definition from text file
  enc                 encrypt an image with a private key
  header              build installed header file and compute mac according
                      to provided key
  pack                build header file and compute mac according to key
                      provided
  diff                compute differences between 2 binary files
  merge               merge elf appli , install header and sbsfu.elf in a
```

```
                    contiguous binary
  mergev2           merge elf files and binaries in a contiguous binary
  append            append elf appli and install header to an existing
                    binary
  inject            inject key to KMS embedded keys code

optional arguments:
  -h, --help            show this help message and exit
```

接下来我们举例介绍 postbuild 脚本中用到的几个 prepareimage 的基本命令操作。

| enc 命令 | 用于对输入 binary 文件进行 AES 加密 |
|---|---|
| 用法 | prepareiamge enc <选项> <输入被加密 binary> <输出加密后 binary> |
| 选项参数 | -k AES 密钥文件<br>-i IV 文件 （用于 AES CBC）<br>-n nonce 文件（用于 AES GCM）<br>-a address （用于 AES CTR 模式加密 OTFDEC 使用的密文） |
| 示例 | `prepareimage enc -k %aeskey% -i %iv% %bin% %binenc%` |

| sha256 命令 | 用于生成 sha256 hash |
|---|---|
| 用法 | prepareiamge sha256 <输入文件> <输出文件> |
| 选项参数 | 无 |
| 示例 | `prepareimage sha256 %bin% %binsha%` |

| sign 命令 | 用于对输入 binary 文件进行签名生成签名或 tag 文件 |
|---|---|
| 用法 | prepareiamge sign <选项> <输入 binary> <输出签名或 tag binary> |
| 选项参数 | -k ECDSA 签名用的 ECC key 文件或者 AES GCM 密钥文件<br>-n nonce 文件（用于 AES GCM 生成 tag 的情况） |
| 示例 | `prepareimage sign -k %ecckey% %bin% %bintag%` |

| pack 命令 | 用于产生 binary 签名和固件头，并打包生成升级用的.sfb 文件 |
|---|---|
| 用法 | prepareiamge pack <选项>  <输出.sfb 文件> |
| 选项参数 | -m 固件头 magic number，与 SBSFU 代码中检查的 magic number 对应，通常为 SFU1<br>-k 签名密钥（如果是 ECDSA 签名，则为 ECC 密钥，如果是 AES-GCM tag，则为 AES-GCM 密钥）<br>-n nonce 文件（只用于 AES GCM）<br>-i IV 文件（只用于有 AES CBC 加密的情况）<br>-t hash 或 tag 文件（如果是 ECDSA 签名，则为 sha256 命令输出的文件，如果是 AES-GCM tag，则为 sign 命令输出的 tag 文件）<br>-r 固件头保留字段大小（不同 crypto 模式不同，详见固件头说明部分）<br>-f 被签名打包的 binary 文件（如果不带加密，则为原始 App binary 文件，如果签名带加密，则为前一个加密步骤的输出文件）<br>-v 固件版本<br>-p 协议版本，缺省为 1<br>-o 固件头和 App binary 之间的偏移量，缺省为 512 字节<br>-e elf 类型，缺省为 1 |

| | |
|---|---|
| | 另外有几个和 certificate 相关的选项只有使用 STSAFE-A110 的配置时才使用，还有几个选项只有当需要生成部分升级包的时候才用到，具体说明这里省略 |
| 示例 | ECDSA 签名+AES CBC 加密<br>`prepareimage pack -m SFU1 -k %ecckey% -r 28 -v 1 -i %iv% -o 512 -f %binenc% -t %binsha% %binsfb%` |
| | AES GCM tag + 加密<br>`prepareimage pack -m SFU1 -k %aesgcmkey% -r 112 -v 1 -o 512 -n %nonce% -f %binenc% -t %bintag% %binsfb%` |
| | ECDSA 签名无加密<br>`prepareimage pack -m SFU1 -k %ecckey% -r 44 -v 1 -o 512 -f %bin% -t %binsha% %binsfb%` |

| | |
|---|---|
| header 命令 | 用于产生固件头，生成固件头文件，可用于生成 secureboot+app 的完整 binary |
| 用法 | prepareiamge header <选项> <输出 header 文件> |
| 选项参数 | 与 pack 命令参数一致 |
| 示例 | 与 pack 命令用法一致，只是输出为 header binary 文件<br>`prepareimage pack -m SFU1 -k %ecckey% -r 28 -v 1 -i %iv% -o 512 -f %binenc% -t %binsha% %headerbin%` |

| | |
|---|---|
| merge 命令 | 用于将固件头，SecureBoot 和 App 合成一个 binary 文件 |
| 用法 | prepareiamge merge <选项> <输出合成的 binary 文件> |
| 选项参数 | -i header 命令输出的 header 文件<br>-s SBSFU elf 文件<br>-u 应用程序 elf 文件<br>-v padding 填充的字节值，缺省为 0xFF<br>-e elf 文件类型，缺省为 1<br>-x header 地址，用于 header 与固件地址不连续的情况（可选）<br>-p 填充 App binary 的字节大小（可选）<br>-l loader elf 文件（如果有独立的 loader）（可选） |
| 示例 | `prepareimage merge -v 0 -i %headerbin% -s %sbsfuelf% -u %appelf% %singlebin%` |

用户可以编写自己的 postbuild 脚本，通过组合使用几个 prepareimage 工具的命令，完成应用程序固件的签名、加密、打包等过程。

几个典型的 Crypto 方案对应需要使用 prepareimage 工具的命令和步骤如下

- ECDSA 签名+AES CBC 加密

| 步骤 | 目的 | 命令 | 输入 | 输出 |
|---|---|---|---|---|
| 1 | 对 App binary 加密 | enc | App binary | App 加密后的 binary |
| 2 | 生成 App binary 的 hash | sha256 | App binary | App binary 的 HASH |
| 3 | 签名并打包生成升级用的.sfb 文件 | pack | App 加密后的 binary<br>App binary 的 HASH | App 对应的.sfb 升级文件，包含头和密文固件 |
| 4 | 生成固件头 binary | header | App 加密后的 binary | App 安全启动固件头 |

| | | | | |
|---|---|---|---|---|
| | | | App binary 的 HASH | |
| 5 | 生成包含 Secure Boot，固件头和 App 的一个 binary | merge | 固件头，App elf，SBSFU elf | 完整 binary，包含安全启动和 App，用于直接烧入 Flash 运行 |

- ECDSA 签名，无加密

| 步骤 | 目的 | 命令 | 输入 | 输出 |
|---|---|---|---|---|
| 1 | 生成 App binary 的 hash | sha256 | App binary | App binary 的 HASH |
| 2 | 签名并打包生成升级用的.sfb 文件 | pack | App binary<br>App binary 的 HASH | App 对应的.sfb 升级文件，包含头和固件 |
| 3 | 生成固件头 binary | header | App binary<br>App binary 的 HASH | App 安全启动固件头 |
| 4 | 生成包含 Secure Boot，固件头和 App 的一个 binary | merge | 固件头，App elf，SBSFU elf | 完整 binary，包含安全启动和 App，用于直接烧入 Flash 运行 |

- AES GCM 加密+tag 作为签名

| 步骤 | 目的 | 命令 | 输入 | 输出 |
|---|---|---|---|---|
| 1 | 对 App binary 加密 | enc | App binary | App 加密后的 binary |
| 2 | 生成 App binary 的 Tag | sign | App binary | App binary 的 AES GCM Tag |
| 3 | 签名并打包生成升级用的.sfb 文件 | pack | App 加密后的 binary<br>App binary 的 tag | App 对应的.sfb 升级文件，包含头和密文固件 |
| 4 | 生成固件头 binary | header | App 加密后的 binary<br>App binary 的 tag | App 安全启动固件头 |
| 5 | 生成包含 Secure Boot，固件头和 App 的一个 binary | merge | 固件头，App elf，SBSFU elf | 完整 binary，包含安全启动和 App，用于直接烧入 Flash 运行 |

说明：

关于 pack 命令生成的 Header+App 的.sfb 升级文件与 header+merge 命令生成的 SecureBoot+Header+App 的.bin 区别见表 2。

**表2. .sfb 升级包与.bin 完整启动 binary 的内容区别**

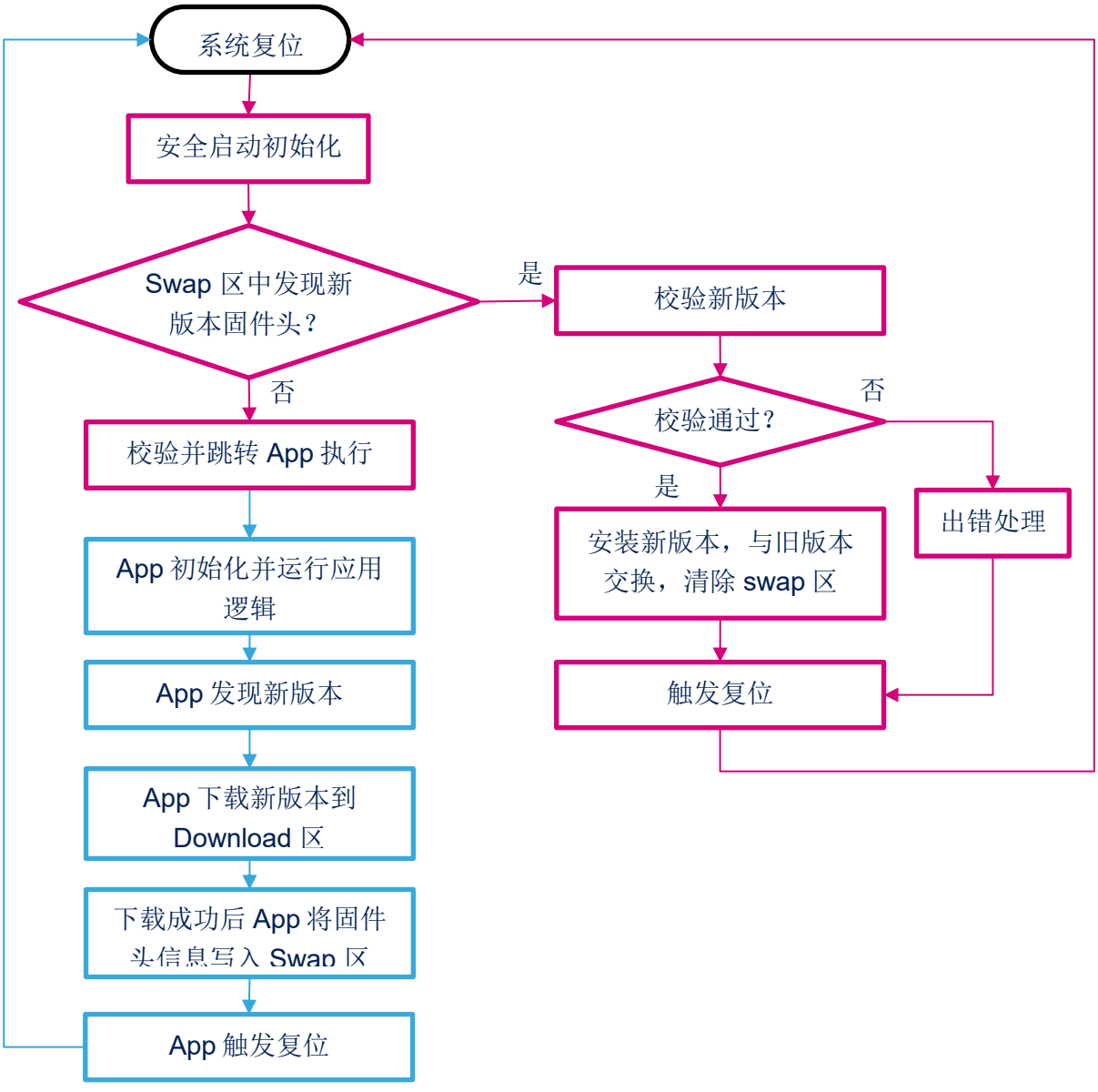
| | .sfb（配合 SBSFU 校验机制的应用程序升级包 binary） | .bin（安全启动+应用程序头+应用程序的拼接 binary） |
|---|---|---|
| 用途 | 通过 UART 接口使用 SBSFU 或者 UserApp 示例工程中的 Loader 做固件升级 | 可通过烧录器直接写入 Flash，上电后可直接运行，包括安全启动以及供验证的应用程序 |
| SecureBoot | 不包含 | 包含 |
| Header | 其中的 FWState 字段标志应用程序状态为 NEW | 其中的 FWState 字段标志应用程序状态为 Valid |

| | | |
|---|---|---|
| 应用固件 | 如果选择的 Crypto 方案包含 AES CBC/GCM 加密，则固件 binary 部分为加密后的密文固件，需要解密后才能运行 | 无论是否选择了 AES CBC/GCM 加密模式，这里的固件都是未加密的明文固件，可以直接运行 |

# 4. 固件升级功能的集成

如果用户选择了双镜像模式集成 X-Cube-SBSFU 中实现的安全启动和安全更新方案，那么应用程序可以根据其自身需要实现带安全启动的固件升级。新版本固件的下载过程可以在应用程序中完成，而完整的升级过程还需要 SecureBoot 代码的参与，因为对新版本应用程序固件的校验、解密等操作都在 SecureBoot 代码中完成。升级大致过程如图 10 所示。

**图10. 从应用下载新版本进行安全升级的流程**



这里有两点需要注意，一个是从应用下载新版本时，下载内容放置的位置，另一个是升级后新版本固件可能需要额外操作使自己生效。

## 4.1. 通过应用程序下载新版本

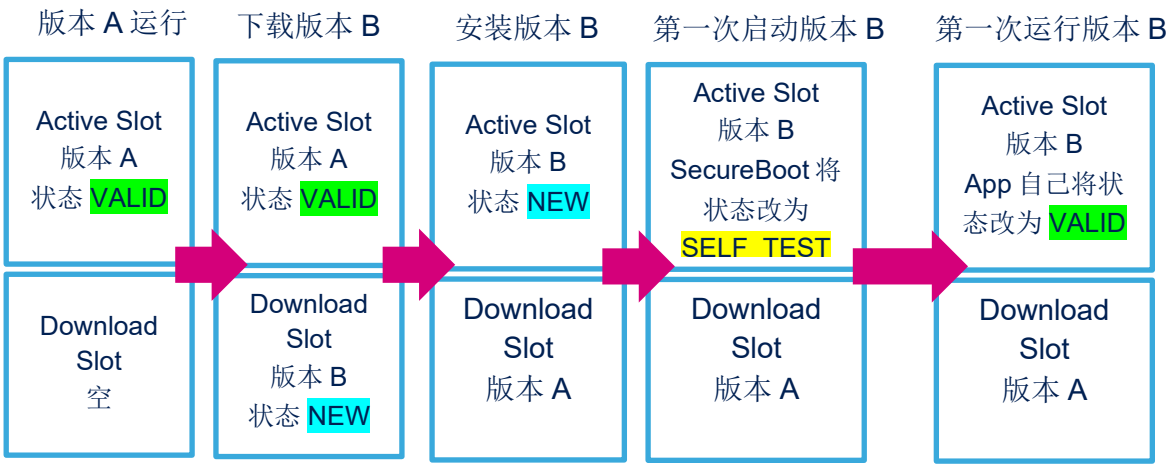
在前面关于 Flash 中的 memory 布局的介绍中我们提到过，双镜像模式包含两个和应用相关的区域，一个为 Active 活动区，也就是当前运行的应用程序所在的区域；一个为 Download 下载区，这个区是升级过程中下载应用程序新版本固件时存放数据的区域。新版本无论是从启动代码中进行的下载还是从应用程序中下载，下载数据存放的地址都必须放在下载区。也就是说应用程序需要知道下载区的起始地址和大小。这个信息在最初定义整个 memory 布局的时候已经确定，即 **SBSFU** 工程中 mapping_fwimg.x 文件中定义的 Download Slot，应用程序可以根据这个 memory 布局获得下载地址和大小信息。具体可以参考 SBSFU 示例 UserApp 中 SFU_APP_GetDownloadAreaInfo() 函数的实现。

新版本固件包为.sfb 文件，文件包含固件头和固件镜像部分，当应用程序完成了新版本固件包所有数据的下载之后，需要将固件头内容写入 Swap 交换区，并触发系统复位，在下一次启动的过程中，SecureBoot 代码将检测到新版本等待安装的状态，完成后续的版本升级操作，见图 9 中蓝色框的 App 部分流程示意。

## 4.2. 验证新版本有效性

前文（3.2 节）中在应用程序固件头（Header）数据结构和内容的部分，我们提到 Header 中有一个 `FwImageState` 字段，该字段用于标志固件当前的的状态。如果 SBSFU 选择了双镜像模式，且编译选项中定义了 `ENABLE_IMAGE_STATE_HANDLING`，那么固件更新过程和每次安全启动过程中都会用到这个状态标志。新下载的应用程序固件处于 NEW 状态，升级完成后第一次启动变成 SELF_TEST 状态，应用程序固件第一次运行的时候需要自己将该状态修改成 VALID，否则下次启动的时候 SecureBoot 会进行 Rollback 操作，回退到之前一个版本。图 11 给出了这个状态转换的示意。

**图11. 固件状态转换**



假设当前运行的应用版本为 A，新版本应用为版本 B，当应用 A 完成新版本 B 的固件下载时，Active Slot 中为版本 A，Download Slot 中为版本 B，此时版本 B 的状态为 NEW。触发复位后，SecureBoot 代码将继续完成后续升级操作，将 Active Slot 与 Download Slot 内容交换，此时 Active Slot 中为版本 B，状态为 NEW，Download Slot 中为版本 A。再次复位后，SecureBoot 会校验 Active Slot 中的版本 B，校验通过后 SecureBoot 会将 B 的状态变为 SELF_TEST，然后跳转版本 B 执行。B 运行起来之后，需要将固件状态修改为 VALID。如果版本 B 第一次运行过程中没有将自己的固件状态修改为 VALID，那么再次复位后，SecureBoot 会检测到 Active Slot 中的版本 B 的状态非 NEW 也非 VALID，则会进行 ROLLBACK 操作，回退到版本 A。因此，这种情况下，应用程序版本 B 中必须包含设置自身状态为 VALID 的代码。这个检查的目的是确保升级的新版本不仅仅能通过校验，而且能够正常运行，如果因为任何原因版本 B 没有能够正常运行至修改固件状态为 VALID 的代码，那么这个版本将被认为无法正常运行，下次启动的时候 SecureBoot 会自动回退到前一个能够运行的版本。

应用程序中设置 Active Slot 的固件状态需要通过 SecureEngine 的接口来完成，代码示例如下，同时应用工程中需要链接来自 SBSFU 工程的输出文件：se_interface_app.o，添加链接文件的具体方法在不同编译器 IDE 中不尽相同，这里不赘述。

```
#include "se_interface_application.h"
...
  SE_StatusTypeDef se_Status = SE_KO;
  SE_FwStateTypeDef fw_state = FWIMG_STATE_INVALID;

  SE_APP_GetActiveFwState(&se_Status, SLOT_ACTIVE_1, &fw_state);
  if (fw_state == FWIMG_STATE_SELFTEST )
  {
    se_retCode = SE_APP_ValidateFw(&se_Status, SLOT_ACTIVE_1);
  }
...
```

# 5. 其他注意事项

## 5.1. 关于 IWDG

如果 SBSFU 的配置中定义了 **SFU_IWDG_PROTECT_ENABLE** 选项，那么 IWDG 应用程序需要定期喂狗（调用类似这样的代码 `WRITE_REG(IWDG->KR, IWDG_KEY_RELOAD);`），否则会导致 IWDG 超时触发系统复位。

## 5.2. 关于调试

SBSFU 安全启动参考实现中缺省会使能多个硬件安全特性，来保护安全启动的代码、关键数据及其执行过程，其中某些安全特性的使能可能给调试带来困难，例如

```
#define SFU_WRP_PROTECT_ENABLE : 写保护，防止 SecureBoot 区被改写
#define SFU_RDP_PROTECT_ENABLE : 读保护，防止调试端口访问
```

```
#define SFU_PCROP_PROTECT_ENABLE：私有代码保护，防止密钥区被读取和改写
#define SFU_DAP_PROTECT_ENABLE：调试端口保护，启动后禁止调试
#define SFU_IWDG_PROTECT_ENABLE：IWDG 保护，SecureBoot 和 UserApp 需要定期喂狗，以保证其执行流
```

在开发阶段，为了方便调试，通常可以考虑将上述几个宏定义在 app_sfu.h 中临时注释掉，当应用集成开发调试完成后，再打开这些宏定义，使能各项保护。

在应用与安全启动 SBSFU 集成的过程中还有可能出现无法正常启动（比如 App 验证失败），或者启动后应用无法正常运行等状况，由于 SecureBoot 本身由 **SECoreBin** 和 **SBSFU** 两个工程组成，校验的实际操作都在 **SECoreBin** 工程中，**UserApp** 也是独立于 SecureBoot 的另外一个工程，调试起来不方便。这时候如果使用 IAR 或者 CubeIDE 开发工程，那么可以通过配置开发工具来联合调试几个工程，比如 SECoreBin，SBSFU 和 App 工程。图 12 和 13 给出了如何配置 IDE 进行多工程联调的示例。

图12.　　IAR 中多工程联调配置

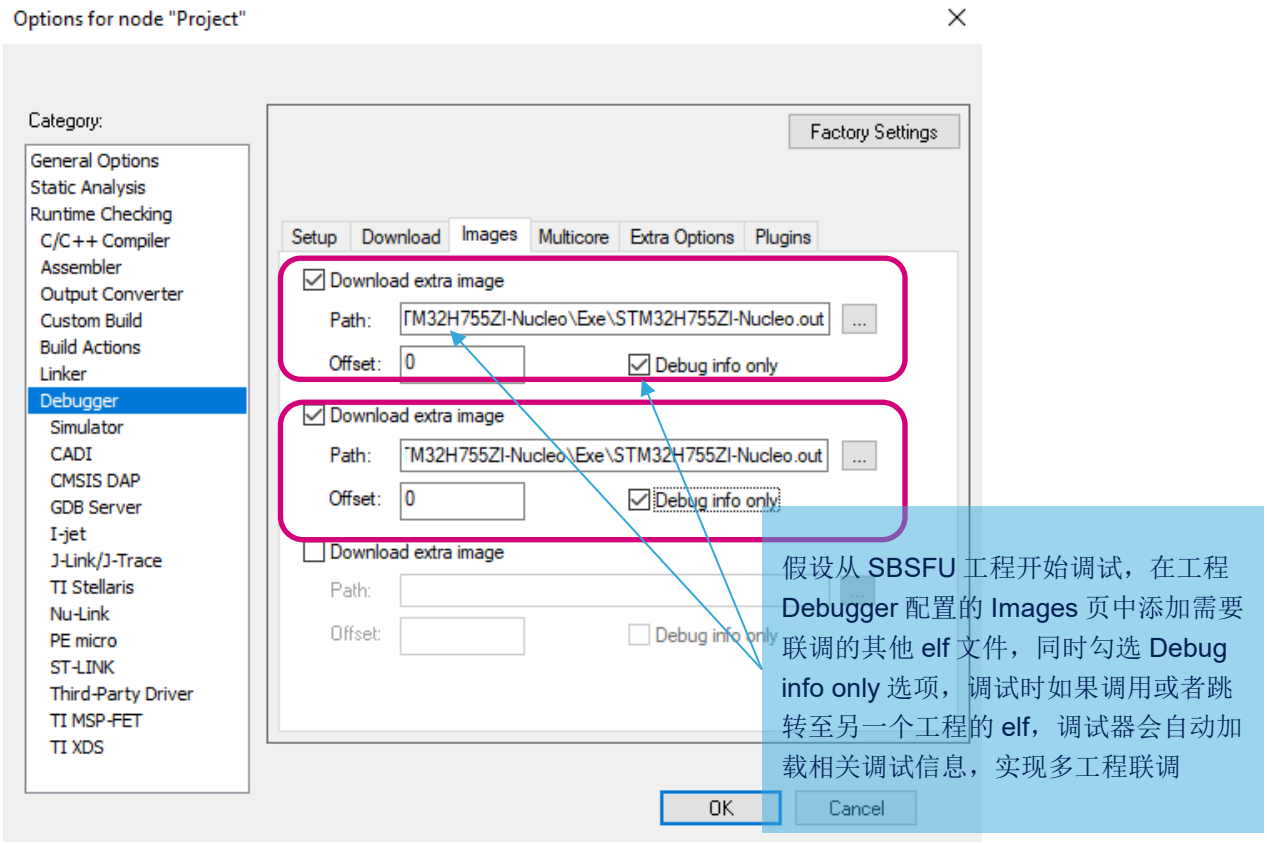
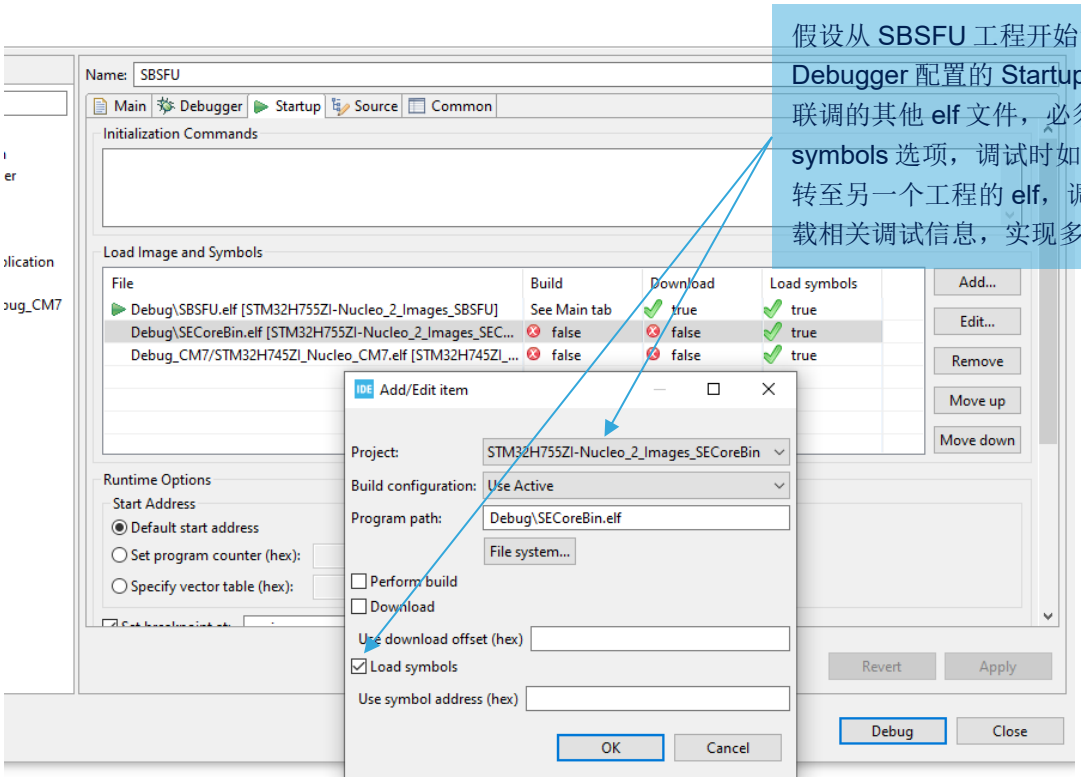



图13.　　CubeIDE 中多工程联调配置




## 6. 小结

X-CUBE-SBSFU 是基于 STM32 MCU 的安全启动和安全更新的参考实现，支持众多 STM32 系列。 X-CUBE-SBSFU 参考实现包含完整的参考代码以及配套的工具，并以源码形式提供给客户，具备非常大的灵活性：用户既可以直接使用该软件包与应用程序进行集成，为其系统快速添加安全启动和安全更新功能；也可以仅以此作为参考，重新编写自己的安全启动和安全更新实现。

本文总结了一些有关如何将 SBSFU 参考实现与客户自己的应用程序进行集成的技巧和注意事项，包括 Flash 和 RAM 的布局，应用链接文件的修改，如何生成用户自己的密钥，如何使用脚本工具对应用固件进行加密、签名和打包处理，如何在应用程序中集成固件升级以及其它一些注意事项等 。

在使用技巧的第三篇，我们将继续讨论有关 X-CUBE-SBSFU 从一个 STM32 硬件平台移植到另一个 STM32 硬件平台的话题。

## 参考文献

【如有， 请注明；否则，请注明：无】

| 文件编号 | 文件标题 | 版本号 | 发布日期 |
|---|---|---|---|
| UM2262 | Getting started with the X-CUBE-SBSFU STM32Cube Expansion Package | Rev9 | June 2021 |
| AN5056 | Integration guide for the X-CUBE-SBSFU STM32Cube Expansion Package | Rev7 | July 2021 |
| | | | |
| | | | |

# 版本历史

| 日期 | 版本 | 撰写/变更 |
|---|---|---|
| 2021 年 11 月 20 日 | 1.0 | 首版发布 |
| | | |
| | | |