移植 SBSFU 到 STM32G070 的过程

关键字：SBSFU， 移植

1. 前言

客户使用 STM32G070RBT6 给海外用户开发产品，由于当地新需求，产品需要增加安全启动的功能。 但是由于 X-Cube-SBSFU 包提供的示例中，只有基于 STM32G071 的示例。客户因此询问该怎么移植。本文将讲解这个移植过程。

2. 基于 STM32G070 和 STM32G071 的 SBSFU 实现差异

在正式讲解之前，我们首先来看一看 STM32G070 和 STM32G071 的 SBSFU 实现差异。

STM32G070 是一个 value line 产品，首先，我们要意识到，有一些安全特性，相比于 STM32G071，它是没有的，比如：PCROP，BOOT_LOCK 和 Secure User Memory。那么，缺少了这些安全特性的 STM32G070，是否还能实现安全启动的功能呢？答案是肯定的。我们先来看 PCROP，BOOT_LOCK，以及 Secure User Memory 在 STM32G071 上的 SBSFU 实现中所扮演的角色是什么?

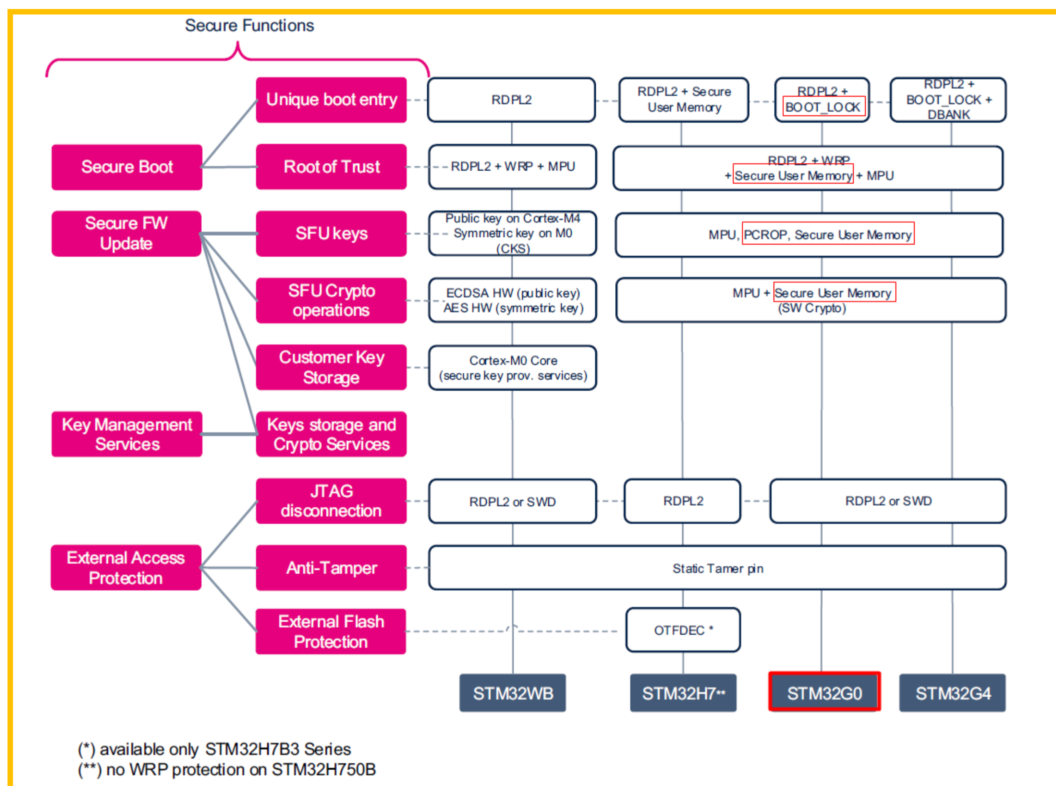

Figure 1 STM32G0 的 SBSFU 安全实现

如上图，在 STM32G071 中，在安全启动的实现中，BOOT_LOCK 用来参与实现唯一启动入口，Secure User Memory 则用来参与实现信任根。PCROP 在安全固件升级实现中用来与

MPU 配合实现密钥的安全存储，同时在安全升级过程中涉及到一些密钥的加解密操作，借助于 Secure User Memory 和 MPU 的功能， 将 App 与 SBSFU 本身实现完美隔离。

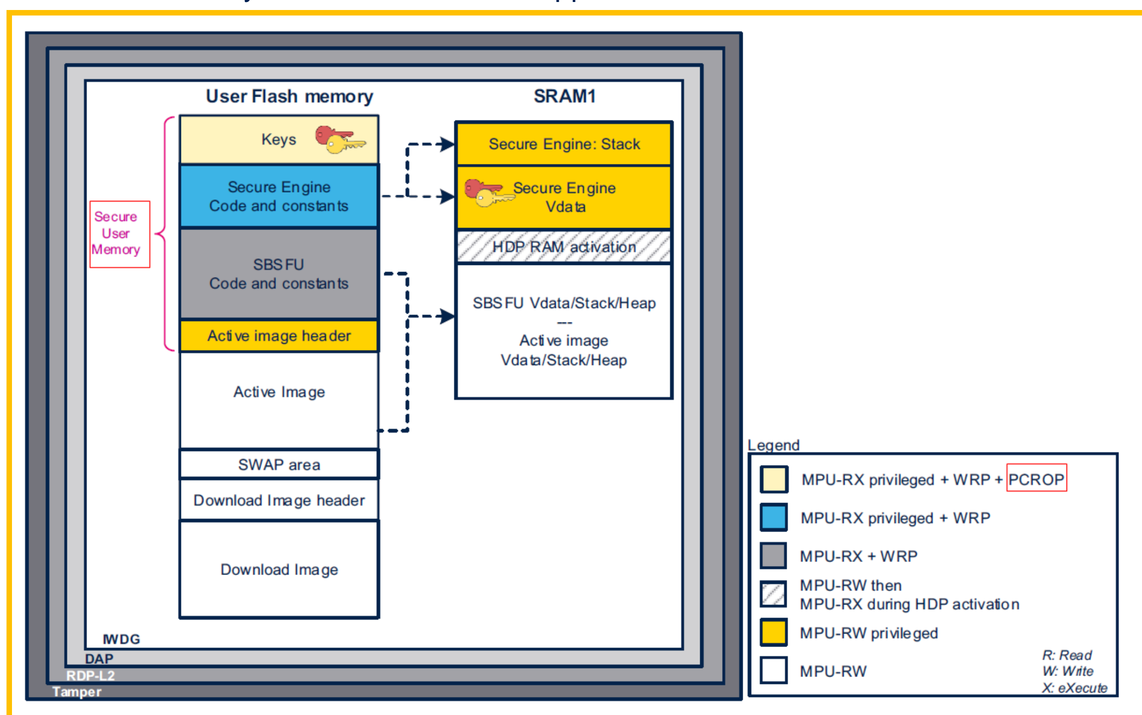

Figure 2 STM32G0 内存安全映射(运行 SBSFU 时)

回到当前问题，一旦 BOOT_LOCK，PCROP，以及 Secure User Memory 缺少的情况下，这些功能还能实现吗?

我们再来看下对于安全启动而言， 它需要实现哪些基本功能?

1> 不可更改不可绕过的一段启动代码

2> 每次复位必先执行安全启动代码

3> 验证系统配置的完整性

- 时钟配置
- 寄存器配置
- 存储器保护设置， ....

4> 启动信任根服务

- 通过密码学算法与密钥，校验 App 的完整性与合法性（来源可信，未经篡改）

这里需要注意地是，上面提到的某一项安全属性也只是参与实现某一项功能，比如 BOOT_LOCK，它只是"参与"实现了唯一启动入口这个功能。从图 1 可知， 除了 BOOT_LOCK，还有 RDP，那么在缺少 BOOT_LOCK 的情况下，RDP 是否也可以实现唯一入口启动的功能。很明显，在 RDP2 时，MCU 的入口是唯一的。也就是说，没有 BOOT_LOCK 的参与下，RDP2 一样可以实现安全启动对唯一入口启动的需求。RDP2+WRP 就可以实现安全启动的前两条基本要求。而第三条基本要求，完全是 SBSFU 内的纯软件实现。第四条要求，通过 RDP2+WRP+MPU 也可以实现。其实， 在缺少了 PCROP，BOOT_LOCK 和 Secure User Memory 后，STM32G070 的安全特性其实跟 STM32F4 差不多，我们不妨来看下 STM32F4 是如何来实现 SBSFU 功能的。

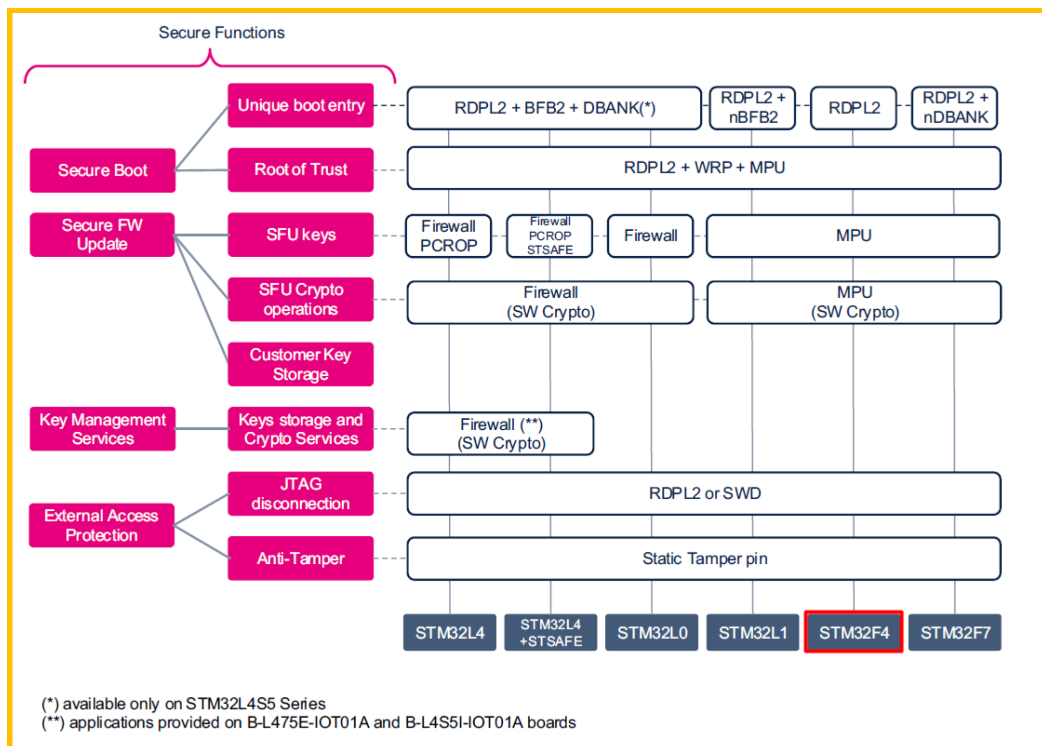

Figure 3 STM32F4 的 SBSFU 安全实现

如上图所示，在 STM32F4 中，借助于 RDPL2，WRP，MPU 就实现了 SBSFU 的全部功能。

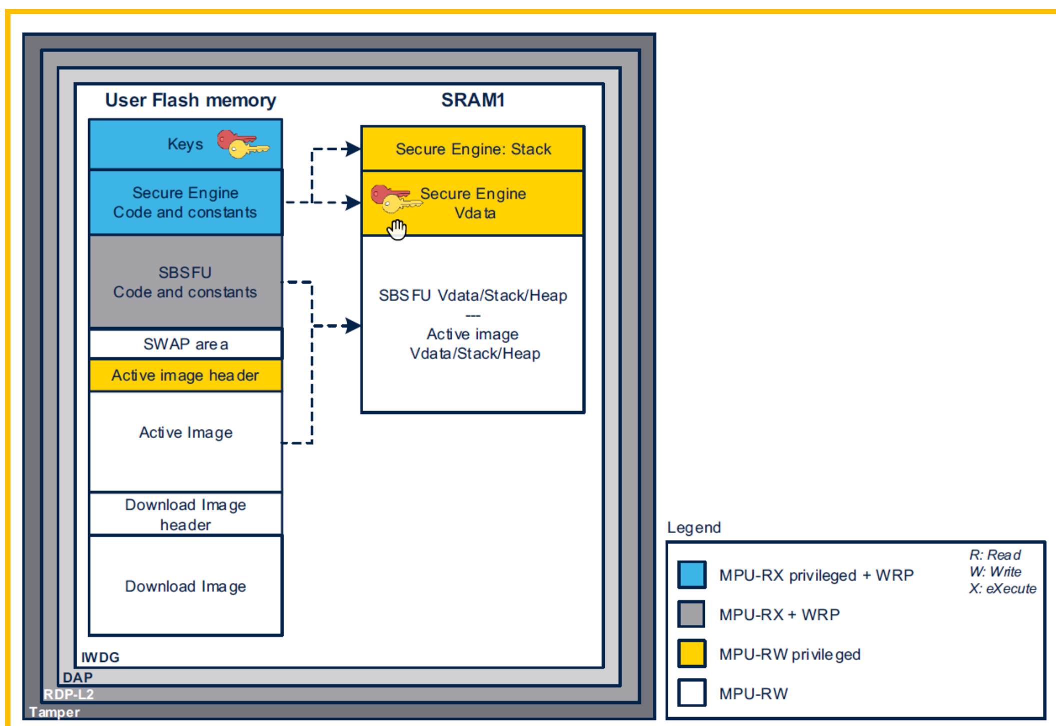

Figure 4 STM32F4 的 SBSFU 内存映射

到这里，我们完全可以确信，在缺少了 BOOT_LOCK，PCROP 和 Secure User Memory 这些安全特性之后，STM32G070 完全可以按照 STM32F4 实现 SBSFU 的方式来进行!

在确立了大方向后， 我们接下来看具体如何实现。

3. 开始移植

第一步：确保原始工程运行正常

从 ST 官网上下载最新了 SBSFU 包(v2.6.1)，打开 STM32CubeExpansion_SBSFU_V2.6.1\Projects\NUCLEO-G071RB\Applications\2_Images 目录，其下有三个工程，2_Images_SECoreBin(后续简称 SECoreBin 工程)，2_Images_SBSFU(后续简称 SBSFU 工程)，2_Images_UserApp(后续简称 UserApp 工程)。使用对应 IDE 按顺序依次编译， 然后将 SBSFU 工程生成的 bin 文件烧录到 NUCLEO-G071RB 板内，打开 Tera Term 串口终端， 通过 Tera Term 烧录 APP 进去。目的是首先确认原始工程一切运行正常。接下来就开始修改了。

第二步：将与 BOOT_LOCK， PCROP， Secure User Memory 相关的宏全部关闭

打开 SBSFU 工程的 app_sfu.h 头文件，找到并关闭下面三个宏：

```
//#define SFU_PCROP_PROTECT_ENABLE
//#define SFU_SECURE_USER_PROTECT_ENABLE /*!< Only accessible in Secure access mode,
//                                         the Secure user software is stored in
the secure user memory, a configurable
//                                         protected area which is part of the user
main memory. */
//#define SFU_UNIQUE_BOOT_ENTRY                        /*!< Force to boot from main
flash memory */
```

重新依次编译 SBCoreBin，SBSFU，UserApp 三个工程，并重新测试通过。

至此， NUCLEO-G071RB 板上运行的是移除了 BOOT_LOCK， PCROP，Secure User Memory 三个安全特性后的 SBSFU 程序，这个原理上与 STM32G070 上原则上是一致的。接下来就是要移植到 NUCLEO-G070RB 板上了，剩下的就只有 STM32G070 与 STM32G071 的非安全特性方面的差异了。

第三步：移植到 STM32G070RB

首先得准备下一块 NUCLEO-G070RB 板。接着将三个工程 SBCoreBin，SBSFU，UserApp 的 device 修改成目标 MCU STM32G070RB，然后将三个工程的 C++预定义宏 STM32G071xx 修改成 STM32G070xx。

以 Keil 为例：

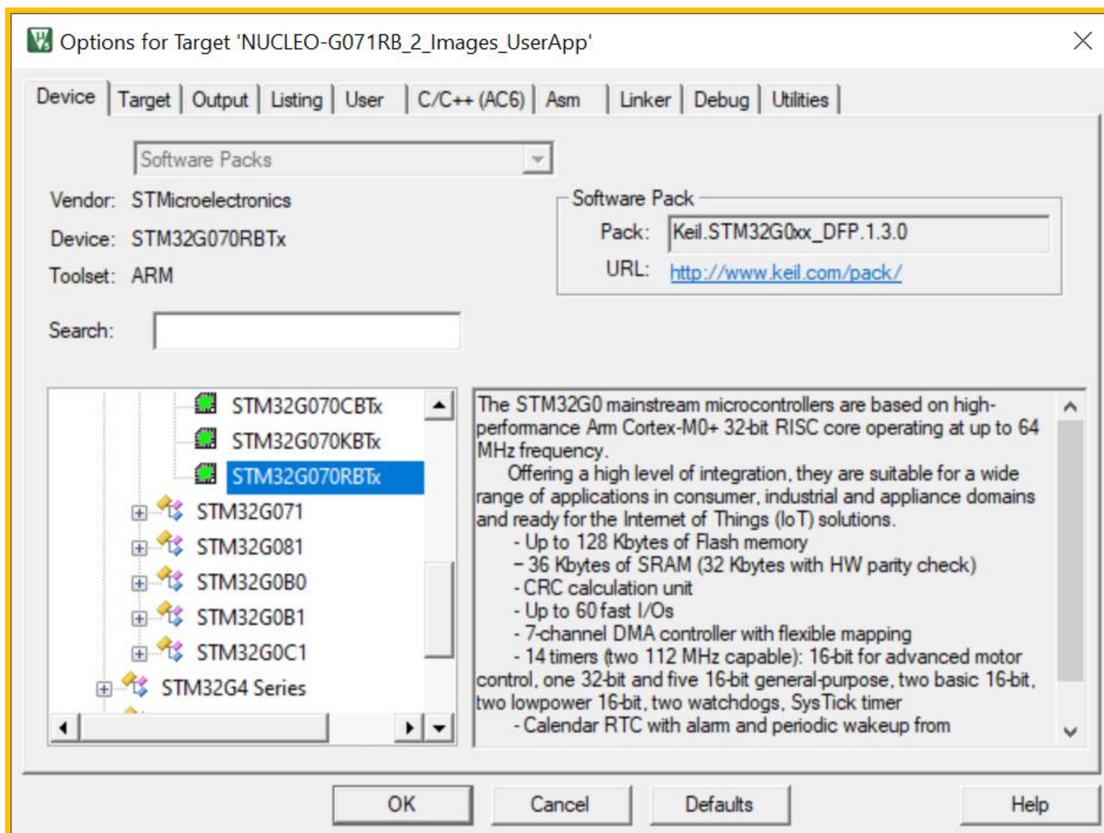

Figure 5 device 选择 STM32G070RBTx

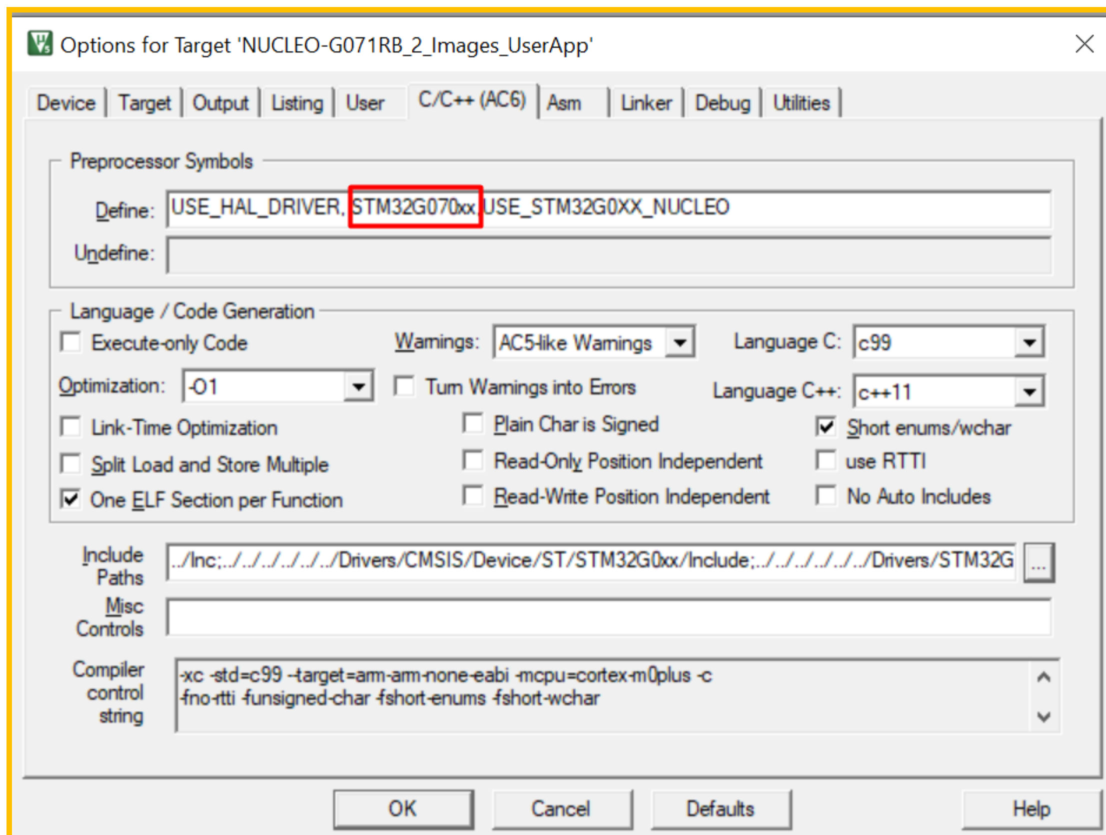

Figure 6 C++编译宏修改

STM32CubeIDE 工程的 device 配置比较难修改，因为它原本是灰色的，不允许修改。但我们使用 UE 打开.cproject 一样可以强制修改：

打开对应的.cproject 文件，搜索关键字 G071，将以下几处替换成 G070(修改两处)：

```
="org.eclipse.cdt.build.core.buildArtefactType=org.eclipse.cdt.build.core.buildArtefactType.exe,org.eclipse.cdt.bui
u.gnu.managedbuild.toolchain.exe.debug">
edbuild.option.target_mcu" useByScannerDiscovery="true" value="STM32G070RBTx" valueType="string"/>
managedbuild.option.target_board" useByScannerDiscovery="false" value="NUCLEO-G070RB" valueType="string"/>
ide.mcu.gnu.managedbuild.option.instructionset" useByScannerDiscovery="true" value="com.st.stm32cube.ide.mcu.gnu.mar
mcu.gnu.managedbuild.option.floatabi" useByScannerDiscovery="true" value="com.st.stm32cube.ide.mcu.gnu.managedbuild
.gnu.managedbuild.option.fpu" useByScannerDiscovery="true" value="com.st.stm32cube.ide.mcu.gnu.managedbuild.option.
```

Figure 7 STM32CubeIDE 下的 device 修改

修改完后，打开 STM32CubeIDE 工程，在其 MCU 和 Board 的配置也会有相应的变化:

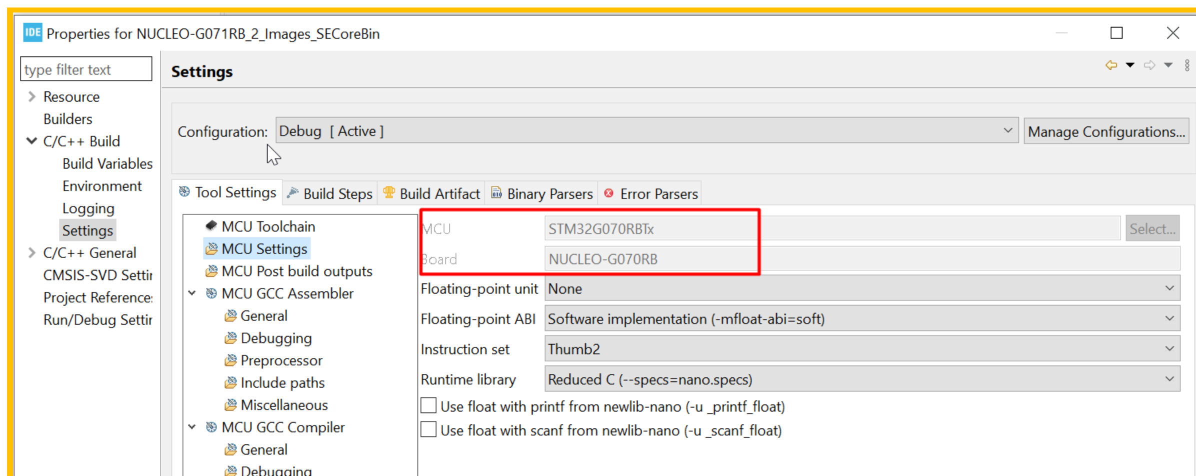

Figure 8 STM32CubeIDE 的 MCU 配置

由此可见，在 STM32CubeIDE 工程的 MCU 配置也可以做相应的修改了。这是一个小技巧。

至此，三个工程的工程配置都做完了相应修改。接下来的就是代码方面的修改了。

首先 STM32G070 的时钟树是没有 PLLQ 输出的，因此，在 SBSFU 和 UserApp 这两个工程内找到 SystemClock_Config()函数，注释掉 PLLQ 的设置，如下所示：

```
//  RCC_OscInitStruct.PLL.PLLQ        = RCC_PLLQ_DIV5;
```

原先 STM32G071 的工程中的打印信息是通过 LPUART1 对应的 PA2，PA3 打印的，换成 NUCLEO-G070RB 板后，其引脚虽仍然是 PA2，PA3 引脚用来串口打印，但是 STM32G070 中是没有 LPUART1 这个外设的，需要换成 USART2，因此，其对应的代码修改如下：

在 SBSFU 工程中， 打开 sfu_low_level.h 头文件， 和 UserApp 工程的 com.h 头文件中:

```
#define SFU_UART                          USART2
#define SFU_UART_CLK_ENABLE()             __HAL_RCC_USART2_CLK_ENABLE()
#define SFU_UART_CLK_DISABLE()            __HAL_RCC_USART2_CLK_DISABLE()

#define SFU_UART_TX_AF                    GPIO_AF1_USART2
#define SFU_UART_TX_GPIO_PORT             GPIOA
#define SFU_UART_TX_PIN                   GPIO_PIN_2
#define SFU_UART_TX_GPIO_CLK_ENABLE()     __HAL_RCC_GPIOA_CLK_ENABLE()
#define SFU_UART_TX_GPIO_CLK_DISABLE()    __HAL_RCC_GPIOA_CLK_DISABLE()

#define SFU_UART_RX_AF                    GPIO_AF1_USART2
```

```
#define SFU_UART_RX_GPIO_PORT                         GPIOA
#define SFU_UART_RX_PIN                               GPIO_PIN_3
#define SFU_UART_RX_GPIO_CLK_ENABLE()                 __HAL_RCC_GPIOA_CLK_ENABLE()
#define SFU_UART_RX_GPIO_CLK_DISABLE()                __HAL_RCC_GPIOA_CLK_DISABLE()
```

如上红色部分即为修改处。

至此，代码部分全部修改完成。重新按顺序依次编译 SBCoreBin，SBSFU，UserApp 三个工程，将 SBSFU 工程首先烧录到 NUCLEO-G070RB 板，然后通过串口终端，按提示，用 Y-Modern 协议将 UserApp 对应的.sfu 文件烧录进去。整个流程都可以正常运行的。这说明软件框架基本已经 OK。接下来运行下 APP 中各种安全测试。

4. 测试安全保护特性

当程序跳入到 APP 后，显示如下界面：

```
=================== Test Menu =============================

  Test : CORRUPT ACTIVE IMAGE --------------------------- 1

  Test Protection: Secure User memory ------------------- 2

  Test Protection: IWDG --------------------------------- 3

  Test Protection: TAMPER ------------------------------- 4

  Previous Menu ----------------------------------------- x

  Selection :
```

Figure 9 测试主界面

当选择 2 Test Protection :Secure User Memory 时，结果会出错：

```
====== Test Protection: Secure User Memory ==================

If the Secure User Memory is enabled you should not be able to read the key and
get stuck.

  -- Reading Key

Press the RESET button to restart the device (or wait until IWDG expires if enab
led).

  -- Key: OEM_KEY_COMPANY1

  -- !! Secure User Memory protection is NOT ENABLED !!
```

Figure 10 测试保护

很明显，这里需要修改下，因为 STM32G070 是没有 Secure User Memory 的。查看其对应代码：

```
/**
```

```
  * @brief  TEST Run Secure User Memory
  * @param  None.
  * @retval None.
  */
static void TEST_PROTECTIONS_RunSecUserMem_CODE(void)
{
  /* 128 bit key + 1 char for NULL-terminated string */
  unsigned char key[17U];
  unsigned char pattern[17U];

  printf("\r\n====== Test Protection: Secure User Memory =================\r\n\n");
  printf("If the Secure User Memory is enabled you should not be able to read the
key and get stuck.\r\n\n");
  printf("  -- Reading Key\r\n\n");
  printf("Press the RESET button to restart the device (or wait until IWDG expires
if enabled).\r\n\n");

  memset(key, 0xFF, 16);
  memset(pattern, 0xFF, 16);
  SE_ReadKey = (void (*)(unsigned char *))((unsigned char
*)(TEST_PROTECTIONS_SE_ISOLATED_CODE_READKEY_ADDRESS) + 1U);

  /* Executing Read Key Code which is located in Secure User memory */
  SE_ReadKey(&(key[0U]));

  /* When activated secure user memory access will return 0x00 (NOP)
     Most of the time we will enter while(1) loop when starting code execution part
of secure user memory
     but as NOP are executed this is not 100% guarantee */

  if (memcmp(key, pattern, 16) != 0U)
  {
    /* Add the string termination to have a proper display */
    key[16] = '\0';

    /* Should not get here if Secure User Memory is available and enabled  */
    printf("  -- Key: %s \r\n\n", key);
    printf("  -- !! Secure User Memory protection is NOT ENABLED !!\r\n\n");
  }
  else
  {
    while (1)
    {
    }
  }
}
```

原来 UserApp 是测试直接读取保存在 SECoreBin 内的密钥数据， 测试是否能读出。结果发现是可以的，因此，保护效果是出问题了。

首先我们修改下此函数，由于 STM32G070 中 Secure User Memory 是不存在的， 此函数叫 TEST_PROTECTIONS_RunSecUserMem_CODE()已经不再合适， 依照 STM32F4 的 SBSFU 实现，将此函数换成 TEST_PROTECTIONS_RunSE_CODE()函数:

```
/**
  * @brief  TEST Run SE_CODE
  * @param  None.
  * @retval None.
  */
static void TEST_PROTECTIONS_RunSE_CODE(void)
{

  /* 128 bit key + 1 char for NULL-terminated string */
  unsigned char key[17U];
  printf("\r\n====== Test Protection: MPU privileged - CODE ===========\r\n\n");
  printf("  -- Reading Key\r\n\n");

  SE_ReadKey = (void (*)(unsigned char *))((unsigned char
*)(TEST_PROTECTIONS_SE_ISOLATED_CODE_READKEY_ADDRESS) + 1U);

  /* Executing Read Key Code into isolated enclave */
  SE_ReadKey(&(key[0U]));
  /* Add the string termination to have a proper display */
  key[16U] = '\0';

  /* Should not get here if isolated enclave is enabled  */
  printf("  -- Key: %s \r\n\n", key);
  printf("  -- !! MPU privileged CODE protection is NOT ENABLED !!\r\n\n");
}
```

同样的尝试读取密钥：

```
====== Test Protection: MPU privileged - CODE ===========

  -- Reading Key

  -- Key: OEM_KEY_COMPANY1

  -- !! MPU privileged CODE protection is NOT ENABLED !!
```

Figure 11 UserApp 尝试读取密钥

测试结果可想而知， 肯定是可以读取的。于是得查看下为什么可以。

仔细查看图 2 的 STM32G071 的 SBSFU 原来实现中，SE Key 是通过 PCROP+Secure User Memory 来保护的，现在换成 STM32G070，原本 Secure User Memory 用来作隔离机制， 现在换成了 MPU， 而原本保护 SE Key 的 PCROP 在 G070 中压根就不存在，于是 SE Key 只剩下 MPU 来保护，因此，我们接下来得着重分析 MPU 对 SE Key 的保护。

为了方便调试，我们首先得将除 MPU 以外的所有保护通通关闭，只剩下 MPU 保护。 于是在 SBSFU 工程中，在 app_sfu.h 头文件中，将以下宏通通注释掉:

```
//#define SFU_WRP_PROTECT_ENABLE
//#define SFU_RDP_PROTECT_ENABLE
```

```
//#define SFU_PCROP_PROTECT_ENABLE      //PCROP 在 STM32G070 中不存在
//#define SFU_TAMPER_PROTECT_ENABLE
//#define SFU_DAP_PROTECT_ENABLE
//#define SFU_DMA_PROTECT_ENABLE
//#define SFU_IWDG_PROTECT_ENABLE
//#define SFU_SECURE_USER_PROTECT_ENABLE   //Secure User Memory 在 STM32G070 中不存在
//#define SFU_UNIQUE_BOOT_ENTRY         //BOOT_LOCK 在 STM32G070 中不存在
```

然后开始调试。在调试过程中，发现程序在从 SBSFU 跳转到 UserApp 之前，在代码中特意将 MPU 关闭：

如在 sfu_low_level_security.c 源文件中的内存函数 SFU_LL_SECU_ActivateSecUser()中有这么一行代码：

```
/* Disable the MPU and clear the control register*/
MPU->CTRL  = 0U; //关闭 MPU
```

这就是为什么 UserApp 中仍然可以直接读取密钥的原因了。很明显，接下来我们需要将 SE Key 用 MPU 保护起来。但在这之前，我们得首先弄清楚，当程序跳转到 UserApp 后，Flash 和 Ram 该如何设置 MPU 保护?

在 UM2262 中， 有提到当程序跳转到 UserApp 后 flash 和 RAM 的状态：

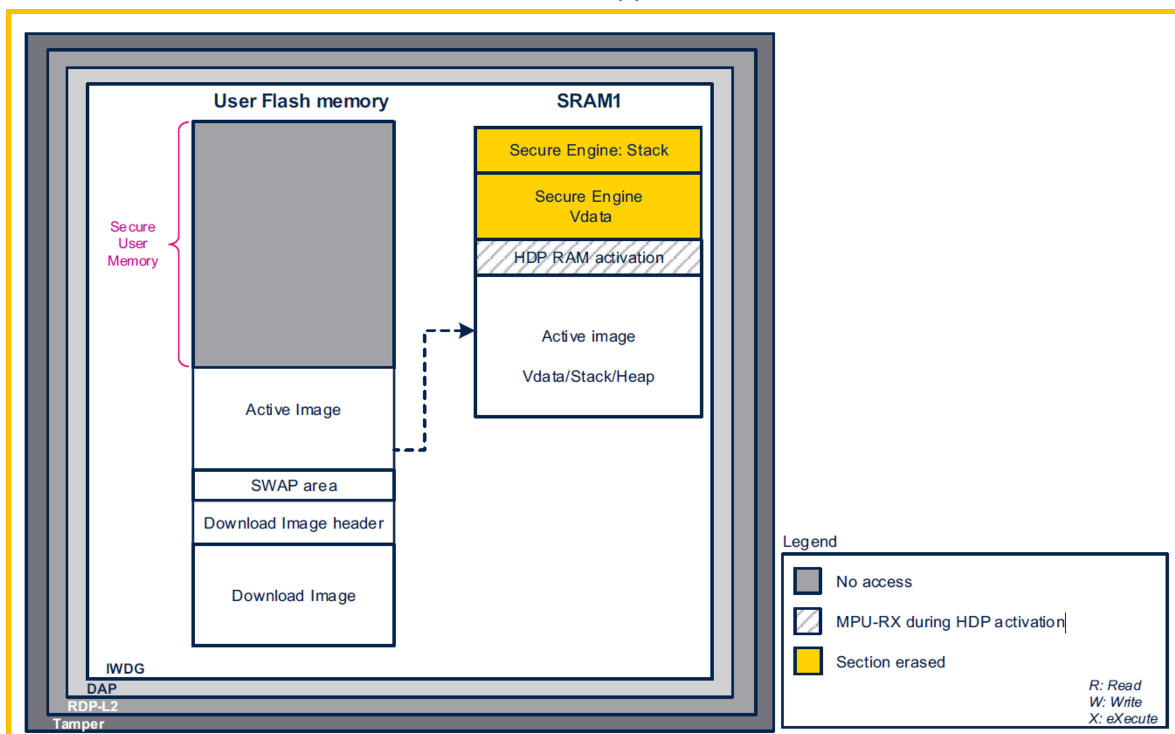

Figure 12 UserApp 运行时的 flash 和 RAM 状态

从上图可以看出，原本 Secure User Memory 保护的区域，我们得使用 MPU 来替代实现相应功能，包含 SBSFU 整个代码和 Slot#1 内的 header 信息。这也就是在代码跳转到 UserApp 之前需要做的事情。而黄色对应的 SRAM 区别已经擦除，当程序跳转到 UserApp 后其实已经没必要再保护。

于是在跳转到 UserApp 的内存函数中配置 MPU：

```
/**
  * @brief  Apply MPU protection before executing UserApp
```

```
  * @param  None
  * @retval SFU_ErrorStatus SFU_SUCCESS if successful, SFU_ERROR otherwise.
  */
__RAM_FUNC SFU_ErrorStatus SFU_LL_SECU_SetProtectionMPU_UserApp(void)
{

  MPU_Region_InitTypeDef MPU_InitStruct;
  uint32_t i;

  /* Make sure outstanding transfers are done */
  __DMB();

  /* Disable the MPU and clear the control register*/
  MPU->CTRL  = 0;

  for(i=0; i<8; i++)
  {
        MPU->RNR =i;
        MPU->RASR &= 0xFFFFFFFE;
  }
  /* Ensure MPU setting take effects */
    __DSB();
    __ISB();

  //confige the whole of SBSFU flash area no access
  MPU_InitStruct.Enable                = MPU_REGION_ENABLE;
  MPU_InitStruct.Number                = 0;
  MPU_InitStruct.BaseAddress           = FLASH_BASE;            //0x0800 0000
  MPU_InitStruct.Size                  = MPU_REGION_SIZE_64KB;   //64K total for
SBSFU
  MPU_InitStruct.SubRegionDisable      = 0;
  MPU_InitStruct.AccessPermission      = MPU_REGION_NO_ACCESS;
  MPU_InitStruct.DisableExec           = MPU_INSTRUCTION_ACCESS_DISABLE;
  MPU_InitStruct.IsShareable           = MPU_ACCESS_NOT_SHAREABLE;
  MPU_InitStruct.IsBufferable          = MPU_ACCESS_NOT_BUFFERABLE ;
  MPU_InitStruct.IsCacheable           = MPU_ACCESS_CACHEABLE;
  MPU_InitStruct.TypeExtField          = MPU_TEX_LEVEL0;
  MPU_ConfigRegion(&MPU_InitStruct);

    //confige the header of active slot#1 no access
  MPU_InitStruct.Enable                = MPU_REGION_ENABLE;
  MPU_InitStruct.Number                = 2;
  MPU_InitStruct.BaseAddress           = SLOT_ACTIVE_1_HEADER;   //0x0801 0000
  MPU_InitStruct.Size                  = MPU_REGION_SIZE_2KB;    //2K for header
  MPU_InitStruct.SubRegionDisable      = 0;
  MPU_InitStruct.AccessPermission      = MPU_REGION_NO_ACCESS;
  MPU_InitStruct.DisableExec           = MPU_INSTRUCTION_ACCESS_DISABLE;
  MPU_InitStruct.IsShareable           = MPU_ACCESS_NOT_SHAREABLE;
  MPU_InitStruct.IsBufferable          = MPU_ACCESS_NOT_BUFFERABLE ;
  MPU_InitStruct.IsCacheable           = MPU_ACCESS_CACHEABLE;
  MPU_InitStruct.TypeExtField          = MPU_TEX_LEVEL0;
  MPU_ConfigRegion(&MPU_InitStruct);
```

```
    /* Enable the MPU */
  MPU->CTRL = (MPU_HFNMI_PRIVDEF | MPU_CTRL_ENABLE_Msk);

//   /* Ensure MPU setting take effects */
//   __DSB();
//   __ISB();

  return SFU_SUCCESS;
}
#endif /* SFU_MPU_PROTECT_ENABLE */
```

注意这里是一个内存函数，它所调用的所有子函数也都是内存函数。在这个函数中我们将 SBSFU 所在的 64K Flash，再加上 2K 的 Active Slot 的 header 信息保护起来。内存 RAM 我们并没有配置保护，因为跳转到 UserApp 后它就完全开放，且内容已经清空。

对应的， 我们在 UserApp 中增加对 Header 的测试函数 ：

```
/**
  * @brief  TEST Access the header of activated app
  * @param  None.
  * @retval None.
  */
static void TEST_PROTECTIONS_Access_Header(void)
{
  uint32_t u_read_header_data = 0;
  printf("\r\n====== Test Protection: MPU no access - header  ==========\r\n\n");
  printf("  -- Reading address: 0x%x\r\n\n", TEST_PROTECTIONS_HEADER_ADDRESS);

  /* Try to read a 32-bit data from the SRAM1 protected by the Firewall */
  u_read_header_data = *((uint32_t *)TEST_PROTECTIONS_HEADER_ADDRESS);

  /* Should not get here if firewall is available and enabled  */
  printf("  -- Address: 0x%x = %d\r\n\n", TEST_PROTECTIONS_HEADER_ADDRESS,
u_read_header_data);
  printf("  -- !! MPU no access for header protection is NOT ENABLED !!\r\n\n");
}
```

重新烧录程序，当程序运行后，选择 2 进行 protection 测试，然后再选择 2，测试访问 SBSFU 所有的 64K 区域：

Figure 13 测试 SBSFU 的 64K 代码区

发现会一直卡住， 这说明 MPU 对整个 SBSFU 64K 区域的保护已经生效。

同样的，选择'3'测试 header 所在区域：

```
====== Test Protection: MPU no access - header  ==========

  -- Reading address: 0x8010000
```

Figure 14 测试 Header 对应的 2K 区域

发现程序读取 0x0801 0000 地址时会一直卡住，这说明 MPU 对 header 的保护也已经生效。然后再测试了下其它选项，包含下载新固件，还有其它默认的一些安全特性，基本都是 OK 的。这说明整个程序基本已经 OK。

最后再恢复之前注释掉的除 MPU 之后的保护。

5. 后述

本文旨在通过一个相对容易的移植， 让读者对 SBSFU 的移植过程有一个大概了解以起到参考和示范作用。

参考文献

文件编号	文件标题	版本号	发布日期
1	UM2262 Getting started with the X-CUBE-SBSFU STM32Cube Expansion Package	v9	2021-06-22
2	AN5056 Integration guide for the X-CUBE-SBSFU STM32Cube Expansion Package	v7	2021-07-22
3	PM0223 Cortex®-M0+ programming manual for STM32L0, STM32G0, STM32WL and STM32WB Series	Rev 5	2019-10-10

文档中所用到的工具及版本

STM32CubeProgrammer v2.11.0

STM32CubeMx v6.6.1

STM32CubeIDE v1.7.0

IAR v9.20.4

Keil v5.35.0

LAT 中的附件

SBSFU_NUCLEO-G070RB.zip

此附件解压后为一个 NUCLEO-G070RB 目录，需要放置于 SBSFU v2.6.1 包下，具体路径为 ：STM32CubeExpansion_SBSFU_V2.6.1\Projects\ 。

使用 IAR 工程编译时最好能放在 C 盘根目录下编译，确保目录无空格，无中文字符。

附件工程采用的路径为：

C:\workspace\en.STM32CubeExpansion_SBSFU_V2.6.1\STM32CubeExpansion_SBSFU_V2.6.1\Projects。

下载测试时请使用 STM32CubeProgrammer 进行。

版本历史

日期	版本	变更
2023 年 06 月 13 日	1.0	首版发布

重要通知 － 请仔细阅读

意法半导体公司及其子公司 （“ST”）保留随时对 ST 产品和 / 或本文档进行变更的权利，恕不另行通知。买方在订货之前应获取关于 ST 产品的最新信息。 ST 产品的销售依照订单确认时的相关 ST 销售条款。

买方自行负责对 ST 产品的选择和使用， ST 概不承担与应用协助或买方产品设计相关的任何责任。

ST 不对任何知识产权进行任何明示或默示的授权或许可。

转售的 ST 产品如有不同于此处提供的信息的规定，将导致 ST 针对该产品授予的任何保证失效。

ST 和 ST 徽标是 ST 的商标。若需 ST 商标的更多信息，请参考 *www.st.com/trademarks*。所有其他产品或服务名称均为其 各自所有者的财产。

本文档是 ST 中国本地团队的技术性文章，旨在交流与分享，并期望借此给予客户产品应用上足够的帮助或提醒。若文中内容存有局限或与 ST 官网资料不一致，请以实际应用验证结果和 ST 官网最新发布的内容为准。您拥有完全自主权是否采纳本文档（包括代码，电路图等）信息，我们也不承担因使用或采纳本文档内容而导致的任何风险。

本文档中的信息取代本文档所有早期版本中提供的信息。

© 2020 STMicroelectronics - 保留所有权利