STM32 TrustZone 开发调试技巧（1）—— 地址安全区及资源安全属性配置

关键字：ARM Cortex V8M，CM33 内核，TrustZone，SAU(Security Attribution Unit)，IDAU(Implementation Defined Attribution Unit)，GTZC（Global TrustZone Controller），Secure(S)，NonSecure(NS)，Secure NonSecure Callable(NSC)

1. 引言

STM32 MCU 中较新的产品系列例如 STM32L5、STM32U5 采用了 ARM Cortex V8M 的 CM33 内核，并引入了 TrustZone 概念。在此基础上，从内核到存储器再到外设等设计了完整的支持 TrustZone 架构的系统隔离机制。

在新的 TrustZone 架构下，软件的开发由原来的单一工程，变成了 Secure 安全和 NonSecure 非安全两个工程的联合开发，系统上电首先从 Secure 工程开始运行，完成必要初始化之后跳转至 NonSecure 工程运行，之后 NonSecure 工程还可以通过调用 Secure 工程经由 NSC（Secure NonSecure Callable）区提供的 API 调用 Secure 工程的函数。

由于此时系统的所有资源，包括 Memory、外设等等都区分了安全和非安全属性，而 CPU 也区分安全和非安全运行状态，其对应的安全或非安全代码及其使用的数据都需要有相应的存储空间存放，且该存储空间需要具有相同的安全或者非安全属性，否则代码无法正常运行。因此，在 TrustZone 架构下首先需要针对不同物理区间做地址安排及相应安全属性的正确配置。对于 V8M 内核来说，这涉及到 SAU/IDAU 的配置，且取指令与取数据可能有不同的访问规则，同时指令以及数据是否能够从 memory 中正确取得，除了和 SAU/IDAU 的配置有关以外，还与 Memory 自身的安全属性配置有关。

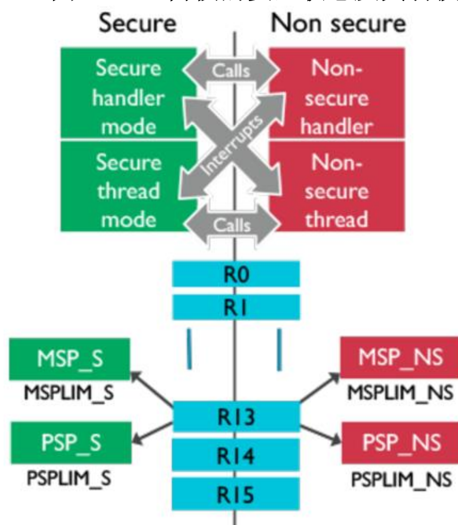
本文将对 SAU/IDAU 配置，Memory 的自身安全属性配置，以及内核访问指令与数据时的安全访问规则加以阐述，希望可以帮助相关开发者更好地理解 V8M TrustZone 的架构以及在 STM32 中的实现，同时，还会列举一些与 memory 的 TrustZone 安全配置相关的常见问题及分析方法，给开发者做参考。

2. CM33 内核的安全扩展

CM33 内核使用 AHB5 总线，具有可选安全扩展(Security Extension)功能。使能安全扩展的内核则支持 V8M TrustZone 架构，此时 AHB5 总线上将携带安全访问标记信号(HNONSEC)，这个标记将指示当前访问的 Transaction 是安全访问还是非安全访问，总线上的从设备需要识别这个标记信号，根据 Transaction 携带的安全、非安全权限信息，决定是否允许最终的物理访问。

内核的安全扩展除了总线携带的安全标记信号以外，还包括 CPU 本身的安全/非安全运行状态，有对应安全/非安全状态的 CPU 寄存器（例如各自的 stack pointer MSP 和 PSP），中断等等。CPU 可以在安全和非安全状态之间切换，这个切换可能来自于函数调用，也可能由中断触发，如图 1 所示。

图1. CM33 内核的安全状态及其切换



本文的内容将主要集中在内核在不同安全状态下对资源的访问权限和规则方面。

3. 内核的 SAU 与 IDAU

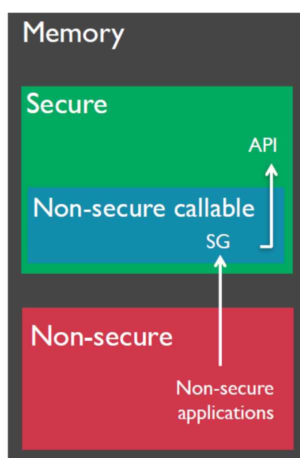
SAU 是 CM33 内核的单元，负责内核对地址的安全访问控制。IDAU 同样作为内核安全访问控制的一部分，与 SAU 共同作用，只不过 IDAU 是芯片厂商实现的独立接口，其行为由芯片厂商的设计决定。

CM33 带 TrustZone 的内核所访问的任何地址在 SAU 和 IDAU 看来都具有其安全属性，地址安全属性可能是下面三种中的一种：

- S - Secure: 纯粹 Secure 地址，只能由安全代码访问，不应放置 SG（Secure Gate）入口指令，不允许 NS 代码直接函数调用。
- NSC - Secure NonSecure Callable：特殊的 Secure 地址，依旧是 Secure 属性，但是可以放置 SG 指令，作为 NS 调用 S 函数的入口地址。
- NS - NonSecure：NonSecure 地址。

图 2 显示了这三种类型的的地址在出现 NS 到 S 的函数调用时的作用。

图2. S，NS，NSC 三种类型地址使用示例



3.1. IDAU

IDAU 是 SOC 对地址安全属性的缺省静态配置，无论是否使能 SAU，IDAU 上电都生效。表 1 是一个 IDAU 缺省配置下地址安全属性的示例（具体的定义请参考对应 STM32 MCU 型号参考手册的说明）。表中给出了 IDAU 对不同地址段的缺省属性配置，这里标成淡蓝色的三类区域存在**地址别名**，也就是说相同的物理资源可以通过两套不同的地址进行访问，例如片上 Flash Bank，既可以通过 0x08xxxxxx 地址访问，也可以通过 0x0Cxxxxxx 的地址访问，但是访问效果可能不同，比如内核访问 0x08000000 地址时，IDAU 会认定访问的是 NS 地址，而当内核访问 0x0C000000 时，IDAU 认为访问的是安全地址。类似的对于片上 SRAM 和外设也一样，从 0x2xxxxxxx 访问 SRAM，IDAU 会认为是安全地址，同样的物理区域，从 0x3xxxxxxx 就会被认为访问的是非安全地址；从 0x4xxxxxxx 访问外设寄存器，IDAU 会认为是安全地址，同样的外设寄存器，如果从 0x5xxxxxxx 访问，就会被 IDAU 认定访问的是非安全地址。对于外部 Memory 映射的地址，不存在地址别名，且 IDAU 默认该地址范围具为非安全地址。

表1. IDAU 缺省配置地址安全属性示例

区域类型	地址范围	IDAU 缺省配置地址安全属性
Code: 外部 Memory	0x0000 0000 - 0x07FF FFFF	NS
Code：Flash 和 SRAM	0x0800 0000 - 0x0BFF FFFF	NS
	0x0C00 0000 - 0x0FFF FFFF	NSC
Code：外部 Memory	0x1000 0000 - 0x17FF FFFF	NS
	0x1800 0000 - 0x1FFF FFFF	
片上 SRAM	0x2000 0000 - 0x2FFF FFFF	NS
	0x3000 0000 - 0x3FFF FFFF	NSC
片上外设	0x4000 0000 - 0x4FFF FFFF	NS
	0x5000 0000 - 0x5FFF FFFF	NSC
外部 Memory	0x6000 0000 - 0xDFFF FFFF	NS

3.2. SAU

SAU 是内核中用来动态配置某段地址安全属性的单元，类似于 MPU，最多可以定义 8 个 region，region 及其配置可以动态修改（除非设置了 lock），但是 SAU 寄存器只能由 Secure Privilege code 进行修改。SAU 上电默认的状态是“未使能”，此时 SAU 单元默认所有地址都具有安全属性，因此只有安全状态的 CPU 能够进行地址访问，当然上电默认 CPU 也处于安全状态，所以上电复位时一定只能运行安全代码。需要通过安全代码进行相关配置，在系统中划分出非安全属性的地址区域后，才能开始运行非安全代码。

3.2.1. SAU 的配置

在 STM32 CubeFW 软件包的 TrustZone Template 工程中通常我们会看到一个与 TrustZone 有关的配置文件，文件名类似 partition_xxxx.h，这个文件里面会包含 SAU 的配置，中断的 Target 配置等等。先来说说 SAU 的配置的几种用例。

- Case1：上电缺省效果，即不使能 SAU，且 SAU 默认所有地址为 S。

这种配置下，SAU 认为所有地址都是 S 属性的，所以 NS 代码一定**无法运行**

```
1 #define SAU_INIT_CTRL          1
2 #define SAU_INIT_CTRL_ENABLE   0 // 0 表示 <<不使能>> SAU
3 #define SAU_INIT_CTRL_ALLNS    0 // 0 表示不使能SAU的情况下，所有地址SAU认为S（这其实就是上电默认的状态）
4     SAU->CTRL = ((SAU_INIT_CTRL_ENABLE << SAU_CTRL_ENABLE_Pos) & SAU_CTRL_ENABLE_Msk) |
5                 ((SAU_INIT_CTRL_ALLNS  << SAU_CTRL_ALLNS_Pos)  & SAU_CTRL_ALLNS_Msk)   ;
```

- Case2 不使能 SAU，且 SAU 默认所有地址为 NS

这种配置下，SAU 相当于不做任何检查，最终的效果则遵循 IDAU 的缺省定义

```
1 #define SAU_INIT_CTRL          1
2 #define SAU_INIT_CTRL_ENABLE   0 // 0 表示 <<不使能>> SAU
3 #define SAU_INIT_CTRL_ALLNS    1 // 1 表示不使能SAU的情况下，所有地址SAU认为 <<NS>>
4     SAU->CTRL = ((SAU_INIT_CTRL_ENABLE << SAU_CTRL_ENABLE_Pos) & SAU_CTRL_ENABLE_Msk) |
5                 ((SAU_INIT_CTRL_ALLNS  << SAU_CTRL_ALLNS_Pos)  & SAU_CTRL_ALLNS_Msk)   ;
```

- Case3: 使能 SAU，并根据需要配置不同属性的 region

```
1 #define SAU_INIT_CTRL          1
2 #define SAU_INIT_CTRL_ENABLE   1 // 0 表示 <<使能>> SAU
3 #define SAU_INIT_CTRL_ALLNS    0 // 当SAU ENABLE的情况下，这个配置被忽略
4
5 #define SAU_REGIONS_MAX   8                 /* Max. number of SAU regions，最多8个region */
```

SAU 最多可以配置 8 个 region 的安全属性，8 个 region 都没有覆盖到的地址范围，SAU 默认都是 S 安全的，无需额外配置，所以配置 region 的时候只能设置某个 region 的属性为 NSC 或者 NS。如果使能了 SAU，那么某个地址的安全属性就由 IDAU 和 SAU 共同决定，已经被 IDAU 配置为 S 属性的地址，不能通过 SAU region 定义为 NS，这样的 SAU region 是无效的，所以简单地说，有效的 SAU region 定义只有两种：

- **IDAU** 认为是 NS 的地址范围：SAU 可以从中定义出 NS 或者 NSC 区，其余没被 SAU region 覆盖的地址都是 S 地址。
- **IDAU** 认为是 NSC 的地址范围：SAU 可以从中定义 NSC 区域，确认 NSC 的实际范围，其余没被 SAU region 覆盖的地址都是 S 地址。

为了让系统能够运行非安全代码，通常我们至少需要配置下面几个 SAU region，来分别指定 flash 中的 NSC secure API 入口区、NS code 区、SRAM 中的 NS rw data 区、外设寄存器 NS 地址别名区等。

- Flash NSC 地址范围，例如

```
1 /* Region 0 配置定义 */
2 #define SAU_INIT_REGION0    1 /* 需要使能的region number，这里就写 1， 否则，如果某个region 不用，则写 0*/
3 #define SAU_INIT_START0     0x0C0FE000      /* start address of SAU region 0 */
4 #define SAU_INIT_END0       0x0C0FFFFF      /* end address of SAU region 0 */
5 #define SAU_INIT_NSC0       1 /* 配置该region属性为NSC，这里要和linker file里面定义的 NSC区域对应 */
```

- Flash NS 地址范围，例如

```
1 /* Region 1 配置定义 */
2 #define SAU_INIT_REGION1    1 /* 需要使能的region number，这里就写 1， 否则，如果某个region 不用，则写 0*/
3 #define SAU_INIT_START1     0x08100000      /* start address of SAU region 1 e.g. NS uses 2nd 1MB Flash */
4 #define SAU_INIT_END1       0x081FFFFF      /* end address of SAU region 1 e.g. NS uses 2nd 1MB Flash */
5 #define SAU_INIT_NSC1       0 /* 配置该region属性为NS，这个region的地址和大小要和NS项目的linker file使用的 */
6                               /* Flash地址区域对应，同时要和Flash SECWM定义匹配 */
```

- SRAM NS 地址范围，例如

```
1 /* Region 2 配置定义 */
2 #define SAU_INIT_REGION2    1 /* 需要使能的region number，这里就写 1， 否则，如果某个region 不用，则写 0*/
3 #define SAU_INIT_START2     0x200D0000      /* start address of SAU region 2 : e.g. NS code uses SRAM3*/
4 #define SAU_INIT_END2       0x2019FFFF      /* end address of SAU region 2: e.g. NS code uses SRAM3 */
5 #define SAU_INIT_NSC2       0 /* 配置该region属性为NS，这个region的地址和大小要和NS项目的linker file使用 */
6                               /* 的地址区域对应，同时要和Flash SECWM定义匹配 */
```

- 外设寄存器 NS 地址别名范围，例如

```
1 /* Region 3 配置定义 */
2 #define SAU_INIT_REGION3    1  /* 需要使能的region number，这里就写 1， 否则，如果某个region 不用，则写 0*/
3 #define SAU_INIT_START3     0x40000000      /* start address of SAU region 3 */
4 #define SAU_INIT_END3       0x4FFFFFFF      /* end address of SAU region 3 */
5 #define SAU_INIT_NSC3       0  /* 配置该region属性为NS，这个region的地址和大小要和NS项目的linker file使用 */
6                                /* 的地址区域对应，同时要和Flash SECWM定义匹配 */
```

- 外部 Memory NS 地址范围 (如果需要用到外部 memory 以地址映射方式访问)，例如

```
1 /* Region 3 配置定义 */
2 #define SAU_INIT_REGION4    1 /* 需要使能的region number，这里就写 1， 否则，如果某个region 不用，则写 0*/
3 #define SAU_INIT_START4     0x60000000      /* start address of SAU region 4 */
4 #define SAU_INIT_END4       0xAFFFFFFF      /* end address of SAU region 4 */
5 #define SAU_INIT_NSC4       0 /* 配置该region属性为NS，可根据实际需要调整 */
```

另外可能还需要将 0x0BF90000 到 0x0BFA08FFF 的区域设为 NS（具体地址范围请参考实际 STM32 型号的参考手册），其中可能包括类似 OTP 以及 device 的 flash information block 等内容，flash driver 可能需要读取该区域。

其他为可选，可根据实际需要进行配置，例如 NS system Bootloader 区。不使用的 region，只需要在该头文件中保持对应的 SAU_INIT_REGIONx 定义为 0 即可

```
#define SAU_INIT_REGION5    1
#define SAU_INIT_START5     0x0BF90000      /* start address of SAU region 5 */
#define SAU_INIT_END5       0x0BFA8FFF      /* end address of SAU region 5 */
#define SAU_INIT_NSC5       0

#define SAU_INIT_REGION6    0 // Region 6 is not used
#define SAU_INIT_START6     0x00000000      /* start address of SAU region 6 */
#define SAU_INIT_END6       0x00000000      /* end address of SAU region 6 */
#define SAU_INIT_NSC6       0

#define SAU_INIT_REGION7    0 // Region 7 is not used
#define SAU_INIT_START7     0x00000000      /* start address of SAU region 7 */
#define SAU_INIT_END7       0x00000000      /* end address of SAU region 7 */
#define SAU_INIT_NSC7       0
```

SAU 的配置可由下面这段代码完成

```
#define SAU_INIT_REGION(n) \
    SAU->RNR  =  (n                                     & SAU_RNR_REGION_Msk); \
    SAU->RBAR =  (SAU_INIT_START##n                     & SAU_RBAR_BADDR_Msk); \
    SAU->RLAR =  (SAU_INIT_END##n                       & SAU_RLAR_LADDR_Msk) | \
                ((SAU_INIT_NSC##n << SAU_RLAR_NSC_Pos)  & SAU_RLAR_NSC_Msk)   | 1U
**
  \brief   Setup a SAU Region
  \details Writes the region information contained in SAU_Region to the
           registers SAU_RNR, SAU_RBAR, and SAU_RLAR
 */
__STATIC_INLINE void TZ_SAU_Setup (void)
{
#if defined (__SAUREGION_PRESENT) && (__SAUREGION_PRESENT == 1U)
  #if defined (SAU_INIT_REGION0) && (SAU_INIT_REGION0 == 1U)
    SAU_INIT_REGION(0);
  #endif
  #if defined (SAU_INIT_REGION1) && (SAU_INIT_REGION1 == 1U)
    SAU_INIT_REGION(1);
  #endif
  #if defined (SAU_INIT_REGION2) && (SAU_INIT_REGION2 == 1U)
    SAU_INIT_REGION(2);
  #endif
  #if defined (SAU_INIT_REGION3) && (SAU_INIT_REGION3 == 1U)
    SAU_INIT_REGION(3);
  #endif
  #if defined (SAU_INIT_REGION4) && (SAU_INIT_REGION4 == 1U)
    SAU_INIT_REGION(4);
  #endif
```

```
29   #if defined (SAU_INIT_REGION5) && (SAU_INIT_REGION5 == 1U)
30     SAU_INIT_REGION(5);
31   #endif
32   #if defined (SAU_INIT_REGION6) && (SAU_INIT_REGION6 == 1U)
33     SAU_INIT_REGION(6);
34   #endif
35   #if defined (SAU_INIT_REGION7) && (SAU_INIT_REGION7 == 1U)
36     SAU_INIT_REGION(7);
37   #endif
38   /* repeat this for all possible SAU regions */
39 #endif /* defined (__SAUREGION_PRESENT) && (__SAUREGION_PRESENT == 1U) */
40   #if defined (SAU_INIT_CTRL) && (SAU_INIT_CTRL == 1U)
41     SAU->CTRL = ((SAU_INIT_CTRL_ENABLE << SAU_CTRL_ENABLE_Pos) & SAU_CTRL_ENABLE_Msk) |
42                 ((SAU_INIT_CTRL_ALLNS  << SAU_CTRL_ALLNS_Pos)  & SAU_CTRL_ALLNS_Msk)   ;
43   #endif
44 ...
45 }
```

4. 资源的安全属性及其配置

4.1. 片上 Flash

Flash 控制器属于 TZ aware 外设，直接支持 AHB5 总线，能够识别总线上面携带的 HNONSEC 信号（标记是否为 Secure 访问），Flash 控制器自身有相关寄存器用于配置 Flash 区域的 S 安全与 NS 非安全属性。具有 S 安全访问属性的 Flash 区域只能接受安全访问（即 Transaction 的 HNONSEC=0），具有 NS 非安全访问属性的 Flash 区域只能接受非安全访问（即 Transaction 的 HNONSEC=1）。这里注意不要将安全/非安全访问与 CPU 的安全/非安全状态混为一谈。这二者未必是一致的，后面讲 CPU 运行状态与访问规则的时候会进行解释。

Flash 控制器对安全属性的配置通常有两个来源，

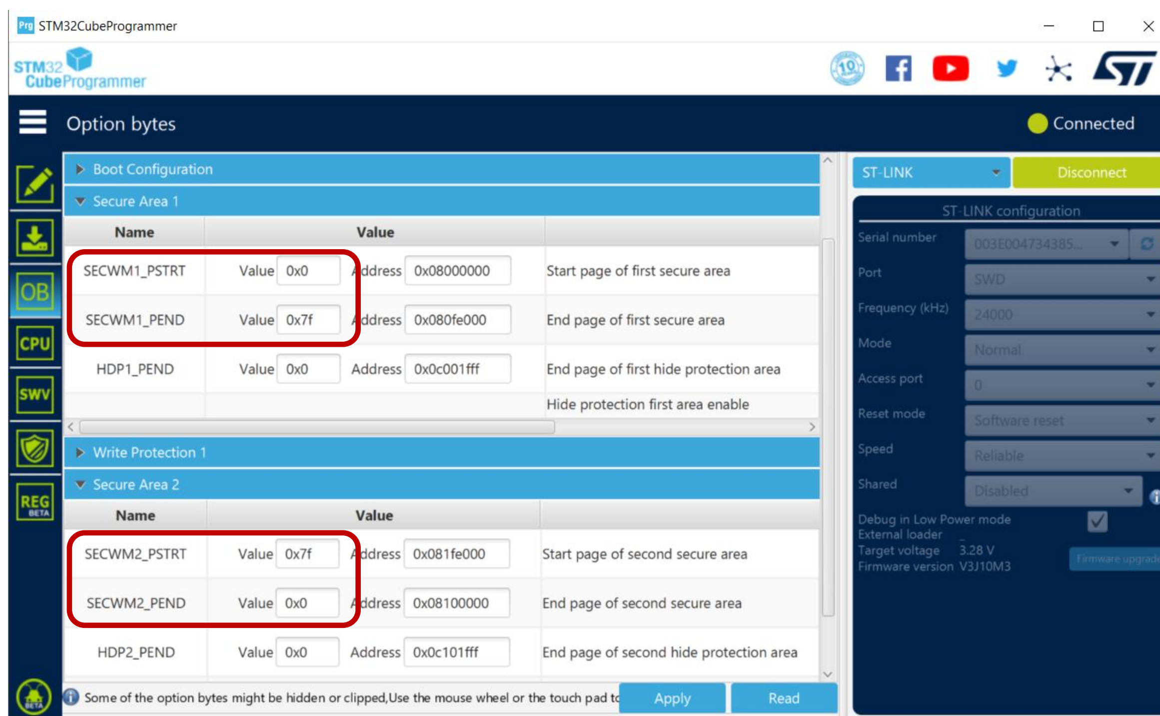
- OptionByte 中的 SECWM（Secure Watermark）配置

  OB 的 SECWM 属于静态配置，只需要设置一次，上电复位时系统首先做 OptionByte 加载，OB 加载后，Flash 安全区配置即生效，无需软件进行任何操作。

  SECWM 可在每个 Flash bank 中划分出一个区域做为 S 安全区，由 SECWM 的 START 和 END 选项来标记安全区的起始和结束 page/section，该 Bank 中其余没有被覆盖到的部分则为 NS 非安全区。

  图 3 是 STM32CubeProgrammer 配置 SECWM OB 的示例，其中 Bank1 的 SECWM1_PEND > SECWM1_START，因此，有效的 Secure 区范围起始页为 0，结束页为 0x7F，即整个 Bank1 的 1MB 都配置为安全属性，而 Bank2 的 SECWM1_PEND < SECWM1_START，表示没有定义有效的安全区，因此第二个 Bank 的 1MB 为 NS 非安全属性。

图3. SecureWatermark OptionByte 配置示例



- Flash 寄存器中的 SECBB 配置

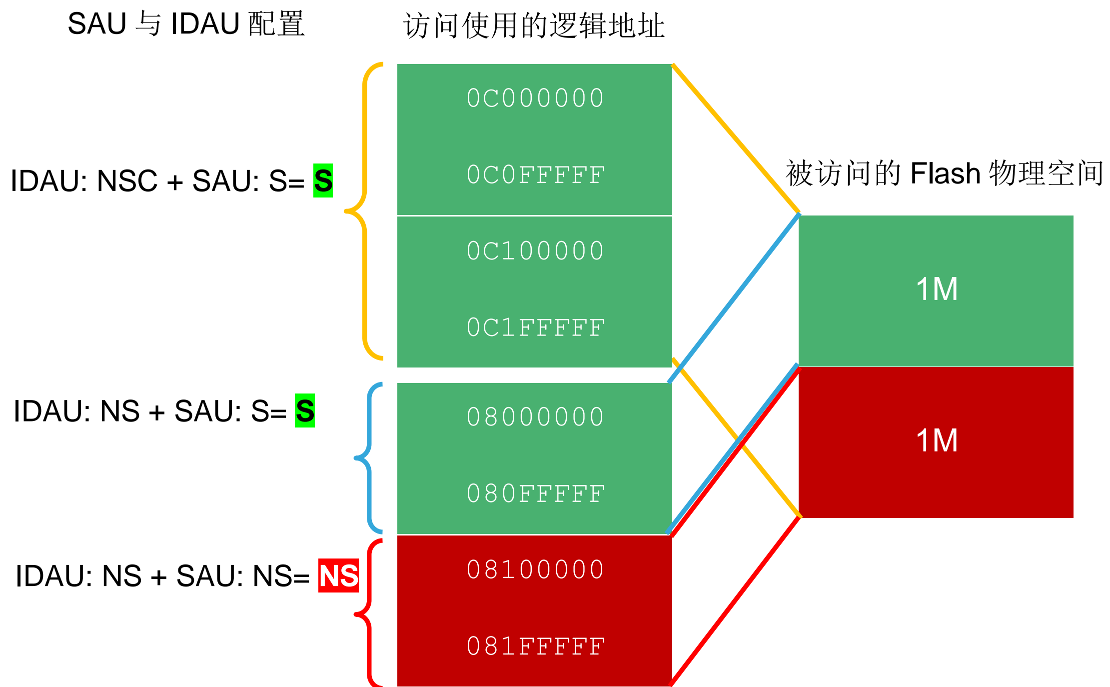
SECBB 的设置则是通过软件写相关寄存器来完成，并不是上电即生效，所以也可以看作是动态配置。SECBB 配置通常以 page/sector 为单位，可以分别设置每个 page/sector 的安全属性，但是它只能将 OB 的 SECWM 定义为 NS 非安全属性的 flash page/sector 配置为 S 安全的，而不能反过来，也就是说，只要某个 Flash page/sector 在 SECWM 或者 SECBB 中配置为 S 安全了，那么它就一定具有 S 安全属性，只有某个 page/sector 在 SECWM 和 SECBB 中同时配置为 NS 非安全时，它才具有 NS 非安全属性。SECBB 的具体配置方法请参考相关 STM32 参考手册的 Flash 章节，或者 HAL Flash driver 的相关 API。

这里需要明确的一点是，Flash 控制器配置的属性指的是物理存储器的某个部分允许安全还是非安全访问，这与访问到该空间的时候使用的逻辑地址无关。举例说明，以 STM32U575 2MB Flash 为例，假设 SECWM 配置了前 1MB 为 S 安全区，后 1MB 为 NS 非安全区。由于存在地址别名机制，访问前 1MB 既可以从 0x08000000 开始，也可以从 0x0C000000 开始。但是无论从那个地址访问，前 1MB 空间都只能允许安全访问，这与是否从 0x0C000000 访问还是从 0x08000000 访问本身无关。Flash 控制器只看总线上 transaction 携带的安全访问标记是什么。只有当访问的安全标记与被访问的 Flash 物理地址自身配置的安全属性一致的时候，访问才能真正成功，否则效果会是读为 0 （RAZ——Read as Zero）和写忽略（WI——Write Ignore）。

图 4 结合前面 OB 配置 SECWM 的例子，给出了从不同地址访问 Flash Bank1（Secure 区）和 Bank2（NonSecure 区）时的情况的示例。在图 4 的例子中，能够正常访问 Flash 的地址范围为 0C000000 到 0C0FFFFF，08000000 到 080FFFFF，以及 08100000 到

081FFFFF，前两个地址段可以访问 Flash 前 1MB 的 S 安全区，第三个地址段可以访问 Flash 后 1MB 的 NS 非安全区。

图4. 从不同逻辑地址访问 Flash 物理地址的效果



4.2. SRAM

SRAM 上电默认为 S 安全访问属性，只有安全代码能够读写 SRAM。如果需要 NS 非安全代码能够使用 SRAM，就必须对 SRAM 的安全属性进行配置，划分出 NS 非安全 SRAM 区。

SRAM 自身并不能直接支持 AHB5 总线，因此，它的安全属性配置是通过 GTZC（Global TrustZone Controller）来支持的，具体说是由其中的 MPCBB 子模块来进行配置。MPCBB 的配置通常以 SRAM 的 block 为单位，block 的大小视具体的 STM32 型号而定，请参考对应型号的参考手册说明。通过 MPCBB 可以配置 SRAM 的安全访问属性，使之允许 Secure 或 NonSecure 访问。同时 MPCBB 还可以配置 SRAM 的 PRIV/NPRIV 访问权限，配置也是以 Block 为单位，具有很高的灵活性。

通过 GTZC MPCBB 配置 SRAM 安全属性的示例如下：

- 直接操作寄存器进行 SRAM block 的安全属性配置

```
1  __HAL_RCC_GTZC1_CLK_ENABLE();
2  /*  unsecure only one block in SRAM1 */
3  GTZC_MPCBB1_S->SECCFGR[0] = 0xfffffffe;
```

这个例子仅仅将 SRAM1 的第一个 block 设置为 NS 非安全属性。

- 通过 HAL API 进行配置

```
1  /* MPCBB SRAM Config Example */
2  #define MPCBB_BLOCK_SIZE GTZC_MPCBB_BLOCK_SIZE
3  #define MPCBB_SUPER_BLOCK_SIZE (GTZC_MPCBB_SUPERBLOCK_SIZE)
4  #define FLAG_NPRIV (1)
5  #define FLAG_NSEC (1<<1)
6
7  static void  gtzc_config_sram(uint32_t base, uint32_t max_size, uint32_t off_start, uint32_t off_end, uint32_t flag)
8  {
9    /* by default SRAM is privileged secure */
10   MPCBB_ConfigTypeDef MPCBB_desc;
11   uint32_t secure_regwrite = 0xffffffff;
12   uint32_t privilege_regwrite = 0xffffffff;
13   uint32_t index;
14   uint32_t block_start = (off_start) / (MPCBB_BLOCK_SIZE);
15   uint32_t block_end = (off_end + 1) / (MPCBB_BLOCK_SIZE);
16
17   /*  Check alignment to avoid further problem  */
18   if ((off_start & (MPCBB_BLOCK_SIZE - 1)) ||
19      ((off_end & (MPCBB_BLOCK_SIZE - 1))!=(MPCBB_BLOCK_SIZE - 1)))
20     Error_Handler();
21   if (off_end >  (max_size-1))
22     Error_Handler();
23
24   if (HAL_GTZC_MPCBB_GetConfigMem(base, &MPCBB_desc) != HAL_OK)
25   {
26     Error_Handler();
27   }
31   for (index = 0; index < (max_size/MPCBB_BLOCK_SIZE); index++)
32   {
33     /* clean register on index aligned */
34     if (!(index & 0x1f))
35     {
36       /* at index 0 */
37       secure_regwrite = MPCBB_desc.AttributeConfig.MPCBB_SecConfig_array[index >> 5];
38       privilege_regwrite = MPCBB_desc.AttributeConfig.MPCBB_PrivConfig_array[index >> 5];
39     }
40     if ((index >= block_start) && (index < block_end))
41     {
42       if (flag & FLAG_NSEC)
43         /* clear bit to set as non secure */
44         secure_regwrite &= ~(1 << (index & 0x1f));
45       else
46         /* set bit to set secure */
47         secure_regwrite |= (1 << (index & 0x1f));
48
49       if (flag & FLAG_NPRIV)
50         /* clear bit to allow non privileged access */
51         privilege_regwrite &=  ~(1 << (index & 0x1f));
52       else
53         /* set bit to set secure */
54         privilege_regwrite |=  (1 << (index & 0x1f));
55
56     }
```

```
57     /* write register when 32 sub block are set  */
58     if ((index & 0x1f) == 0x1f)
59     {
60       if (uFlowStage == FLOW_STAGE_CFG)
61       {
62         MPCBB_desc.AttributeConfig.MPCBB_SecConfig_array[index >> 5] = secure_regwrite;
63         MPCBB_desc.AttributeConfig.MPCBB_PrivConfig_array[index >> 5] = privilege_regwrite;
64       }
65       else
66       {
67         if (MPCBB_desc.AttributeConfig.MPCBB_SecConfig_array[index >> 5] != secure_regwrite )
68             Error_Handler();
69         if (MPCBB_desc.AttributeConfig.MPCBB_PrivConfig_array[index >> 5] != privilege_regwrite)
70             Error_Handler();
71       }
72     }
73   }
74
75   if ((uFlowStage == FLOW_STAGE_CFG) && (HAL_GTZC_MPCBB_ConfigMem(base, &MPCBB_desc) != HAL_OK))
76     /* FIX ME */
77     Error_Handler();
78
79 }
```

```
80
81 void gtzc_init_cfg(void)
82 {
83     /* Enable GTZC clock */
84     __HAL_RCC_GTZC1_CLK_ENABLE();
85     __HAL_RCC_GTZC2_CLK_ENABLE();
86     /* assume secure ram in SRAM 2 , the rest is configured as non-secure */
87     gtzc_config_sram(SRAM1_BASE, SRAM1_SIZE, 0, SRAM1_SIZE -1, FLAG_NPRIV | FLAG_NSEC);
88     gtzc_config_sram(SRAM2_BASE, SRAM2_SIZE, 0, SRAM2_SIZE -1, FLAG_NPRIV); // SRAM2 is still SEC, but config as NPRIV
89     gtzc_config_sram(SRAM3_BASE, SRAM3_SIZE, 0, SRAM3_SIZE -1, FLAG_NPRIV | FLAG_NSEC);
90     gtzc_config_sram(SRAM4_BASE, SRAM4_SIZE, 0, SRAM4_SIZE -1, FLAG_NPRIV | FLAG_NSEC);
91 #if defined(STM32U595xx) || defined(STM32U599xx) || defined(STM32U5A5xx) || defined(STM32U5A9xx)
92     gtzc_config_sram(SRAM5_BASE, SRAM5_SIZE, 0, SRAM5_SIZE -1, FLAG_NPRIV | FLAG_NSEC);
93 #endif /* defined(STM32U595xx) || defined(STM32U599xx) || defined(STM32U5A5xx) || defined(STM32U5A9xx) */
94
95     /* Lock MPCBB Config */
96     GTZC_MPCBB1_S->CFGLOCKR1=MPCBB_LOCK(SRAM1_SIZE);
97     GTZC_MPCBB2_S->CFGLOCKR1=MPCBB_LOCK(SRAM2_SIZE);
98     GTZC_MPCBB3_S->CFGLOCKR1=MPCBB_LOCK(SRAM3_SIZE);
99     GTZC_MPCBB4_S->CFGLOCKR1=MPCBB_LOCK(SRAM4_SIZE);
100 #if defined(STM32U595xx) || defined(STM32U599xx) || defined(STM32U5A5xx) || defined(STM32U5A9xx)
101     GTZC_MPCBB5_S->CFGLOCKR1=MPCBB_LOCK(SRAM5_SIZE);
102 #endif
103 }
```

类似于 Flash，SRAM 的正常访问也需要满足 memory 的自身安全属性与访问信号携带的安全属性一致的条件，也就是说 S 安全访问只能读写 S 安全属性的 SRAM 区，NS 非安全访问只能读写 NS 非安全属性的 SRAM 区，只要二者不一致，访问则不能成功， 此时效果会是读为 0 （RAZ）和写忽略（WI）。

SRAM 同样可以通过不同的地址别名进行访问，20000000 或者 30000000 开始的地址都映射到相同的 SRAM 物理地址，从不同地址访问的效果与该逻辑地址在 SAU/IDAU 中的配置，以及实际对应的 SRAM 物理区域在 GTZC MPCBB 中所具有的安全属性配置有关。

为了能够正常读写 SRAM，需要注意所访问的逻辑地址在 SAU/IDAU 中的配置与 GTZC MPCBB 配置的一致性。

与 Flash 有区别的是，SRAM 有一个额外的控制位，通过使能 GTZC MPCBB CR 寄存器的 SRWILADIS 位，可以允许 S 安全访问读写 NS 非安全 SRAM 区，这是个特例。

- **SRWILADIS** bit = 1 时，允许 S 访问 NS SRAM

这个配置上电缺省状态为 0，即禁止 S 访问 NS SRAM，如果保持该 bit 位为 0，则 MPCBB 总是对 SRAM 的访问进行严格检查，transaction 的安全标记必须和 SRAM block 的安全属性一致，当发生不一致的访问时，将触发 Illegal Access 非法访问事件）

4.3. 外部 Memory 和 Backup SRAM

当应用程序需要通过某些端口使用外部扩展 memory, 而且将通过 Memory Map 的方式直接通过地址访问外部存储，或者需要使用 Backup SRAM 时，同样需要配置 memory 地址的安全属性。

这部分的配置是由 GTZC 的 MPCWM 子模块来控制的。外部 memory 以及 Backup SRAM，上电缺省都具有 S 安全属性，只有 S 安全代码能够操作这些 memory 地址。如果运行在 NS 非安全侧的应用程序代码需要访问这些资源，则需要指定这些 memory 对应地址的 NS 非安全区。通常每个外部 Memory 接口以及 Backup SRAM 都有对应的 MPCWM 单元，用来控制其访问权限。如在 STM32U5 上有如表 2 列出的多个 MPCWM 对每个外部接口对应的地址的安全属性进行管理。 STM32U5 的 MPCWM 除了管理接口对应外部 memory 地址范围的安全/非安全属性，还控制其 PRIV/NPIRV 访问属性。

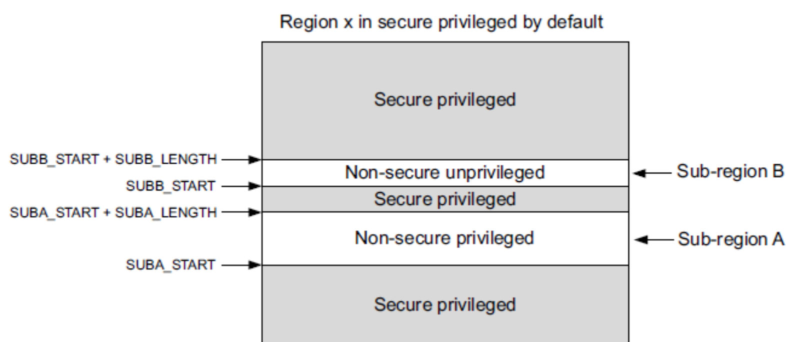
表2. MPCWM 管理的资源

MPCWM	目标存储器接口	Sub-region 个数	Memory 映射地址
MPCWM1	OCTOSPI1	2	0x90000000
MPCWM2	FSMC_NOR bank	2	0x60000000
MPCWM3	FSMC_NAND bank	1	0x80000000
MPCWM4	Backup SRAM	1	0x50036400
MPCWM5	OCTOSPI2	2	0x70000000
MPCWM6	HSPI*	2	0xA0000000

*STM32U599/5A9

某些接口可以指定两个 sub-region, A 和 B，如图 5 所示，有些接口只能指定一个 sub-region。

图5. MPCWM sub-region 配置外部 Memory 地址安全属性



当某个接口对应的地址范围上指定的 sub-region A 和 B 出现重叠时，将遵循如下规则（见表 3）：

- 只要 sub-region A、B 中有一个指定该区域为 NS，则重叠区域属性为 NS
- 只要 sub-region A、B 中有一个指定该区域为 NPRIV，则重叠区域属性为 NPRIV

表3. MPCWM sub-region 重叠时的规则

Sub-region A	Sub-region B	重叠区域的属性
NS	NS	NS
NS	S	NS
S	NS	NS
S	S	S
Sub-region A	**Sub-region B**	**重叠区域的属性**
NPRIV	NPRIV	NPRIV
NPRIV	PRIV	NPRIV
PRIV	NPRIV	NPRIV
PRIV	PRIV	PRIV

配置 MPCWM sub-region 可以通过 HAL API 完成，例如

```
uint32_t MemBaseAddress;
MPCWM_ConfigTypeDef MPCWM_Desc;
...
HAL_GTZC_TZSC_MPCWM_ConfigMemAttributes(MemBaseAddress,&MPCWM_Desc);
```

其中 MemBaseAddress 为某个 region 的基地址，例如 HSPI1_BASE，MPCWM_Desc 数据结构则包括以下一些内容：

- uint32_t AreaId; sub-region ID，GTZC_TZSC_MPCWM_ID1 或 GTZC_TZSC_MPCWM_ID2 (对应 sub-region A, B)
- uint32_t Offset; sub-region 起始地址相对 BASE 地址的偏移量，FMC/OSPI 必须为 128KB 对齐
- uint32_t Length; sub-region 长度（与 offset 相同的对齐要求）
- uint32_t Attribute; sub-region 安全属性 S/NS, PRIV/NPRIV
- uint32_t Lock; 是否 Lock 该 sub-region 的属性配置，如果 Lock，则设置后不能修改，状态保持到下次复位
- uint32_t AreaStatus; 是否使能该 sub-region， ENABLE 或 DISABLE

示例代码如下，该示例配置 HSPI 对应的 0xA000000 开始的地址段中的一个 sub-region A 为 NS+NPRIV 属性，偏移量为 0，大小为 8MB，配置后锁定 lock，不允许进一步修改。

```
1  void GTZC_Config_MPCWM (void)
2  {
3    MPCWM_ConfigTypeDef MPCWM_Desc;
4
5    __HAL_RCC_GTZC1_CLK_ENABLE();
6    __HAL_RCC_GTZC2_CLK_ENABLE();
7    MPCWM_Desc.AreaId = GTZC_TZSC_MPCWM_ID1; /* Area 1 */
8    MPCWM_Desc.Offset = 0; /* Start from HSPI base address */
9    MPCWM_Desc.Length = 0x800000; /* Total size 8MB */
10   MPCWM_Desc.Attribute = GTZC_TZSC_MPCWM_REGION_NSEC | GTZC_TZSC_MPCWM_REGION_NPRIV; /* Set area attribute to NS + NPRIV */
11   MPCWM_Desc.Lock = GTZC_TZSC_MPCWM_LOCK_ON; /* Lock the configuration */
12   MPCWM_Desc.AreaStatus = ENABLE; /* Enable the region */
13
14   HAL_GTZC_TZSC_MPCWM_ConfigMemAttributes(0xA0000000, &MPCWM_Desc);
15 }
```

类似于 Flash 以及 SRAM，外部 Memory 地址的正常访问也需要满足 memory 的自身安全属性与访问信号携带的安全属性一致的条件，也就是说 S 安全访问只能读写 S 安全属性的外部 memory 地址，NS 非安全访问只能读写 NS 非安全属性的外部 memory 地址，只要二者不一致，访问则不能成功， 此时效果会是读为 0 （RAZ）和写忽略（WI）。

由于外部 memory 地址不存在地址别名，因此只有一组地址能够用于访问外部 Memory，而该地址要么位于 MPCWM 标记的 sub-region 中，其安全属性取决于 sub-region 的设置，要么位于 sub-region 之外，只能允许安全访问。那么一旦某个地址范围在 sub-region 中被设定为 NS 非安全属性，对应的 SAU 的配置中应当存在对应该地址范围的 region，且也配置为 NS 非安全属性。这样，无论是安全代码还是非安全代码，都将能够从该地址正常访问对应的外部 Memory 区域。对于 MPCWM 中标记为 S 安全访问的地址范围，只有安全代码能够以 S 安全访问的方式读写该地址范围的外部 Memory。

BackupSRAM 不属于外部 Memory，但是其访问权限也通过 MPCWM 来指定。而 BackupSRAM 的地址也存在地址别名，以 STM32U5 为例，16KB 的 BKPSRAM 的起始地址对应为 0x50036400 和 0x40036400，而 IDAU 默认 0x5xxxxxxx 地址为安全地址（NSC），这时候如果通过 MPCWM 配置 BKPSRAM 非安全区时，sub-region 指定的一定是 0x40036400 开始的 16KB 以内的某个区域。一旦这个 NS 非安全区域配置生效，该范围内的 BKPSRAM 只能通过 0x4003xxxx 地址进行访问（安全和非安全代码都如此），因为 0x5003xxxx 地址将会被 IDAU 默认为安全访问地址，会与 BKPSRAM 的非安全属性冲突，产生 RAZ/WI 效果，不能正常操作 BKPSRAM。

4.4. 外设

所有的外设寄存器都有地址别名，可以从 0x4xxxxxxx 或者 0x5xxxxxxx 地址访问相同的 IP。但是 IDAU 默认 0x5xxxxxxx 为安全地址，通过该地址只能产生安全访问，因此非安全代码永远都是使用 0x4xxxxxxx 的地址对外设寄存器进行读写。安全代码则可以使用 0x5xxxxxxx 地址对外设寄存器进行操作。

从 TrustZone 的架构看，外设有两大类型：

- TurstZone aware IP :

  - 这类 IP 自身直接支持 TrustZone 架构，能够识别 AHB5 总线上的 HNONSEC 信号，包括总线主设备（例如 DMA）和总线从设备（例如 GPIO，Flash 控制器，TAMP 等），他们都被称为 TZ aware 外设。
  - 他们自身有相关的寄存器能够配置其自身的安全属性，例如前面提到的 Flash Controller，它有 SECWM，SECBB 寄存器用于设置 Flash sector/page 的安全属性，再例如 GPIO 也有相关的安全控制寄存器，可以配置每一个 GPIO 管脚的安全属性。
- Securable IP：
  - 这类 IP 自身不能直接支持 TrustZone 架构，也包括总线主设备（例如 SDMMC）和总线从设备（例如 UART, I2C, AES 等），这类外设被称为 Securable 外设。
  - 他们的安全属性需要通过 GTZC 的 TZSC 进行配置和管理。以 STM32U5 为例， TZ aware 外设包括以下这些，除此之外的其他 IP 都属于 Securable IP：
    - GPIOA 到 GPIOJ
    - GTZCx_MPCBB, GTZCx_TZIC 和 GTZCx_TZSC (GTZC blocks)
    - OTFDEC1/2 当 TZ 使能时，只有安全访问能够写 OTFDEC 寄存器
    - EXTI
    - Flash memory
    - RCC 和 PWR
    - GPDMA 和 LPDMA
    - SYSCFG registers
    - RTC 和 TAMP
    - MCU debug 单元 DBGMCU
    - ICACHE 和 DCACHE

GTZC TZSC 配置寄存器可用来配置 Securable IP 的 S/NS 和 PRIV/NPRIV 访问属性，通过 HAL API 配置 Securable IP 安全属性的示例如下

```
HAL_GTZC_TZSC_ConfigPeriphAttributes(uint32_t PeriphId, uint32_t PeriphAttributes);
举例：
  • 将 AES 配置为 SEC+PRIV 访问
  HAL_GTZC_TZSC_ConfigPeriphAttributes(GTZC_PERIPH_AES, GTZC_TZSC_PERIPH_SEC | GTZC_TZSC_PERIPH_PRIV);
  • 将 LPUART1 配置为 NSEC+NPRIV 访问
  HAL_GTZC_TZSC_ConfigPeriphAttributes(GTZC_PERIPH_LPUART1, GTZC_TZSC_PERIPH_NSEC | GTZC_TZSC_PERIPH_NPRIV);
```

外设的安全访问规则比 Memory 要宽松，具有 S 安全属性的 IP 当然能只能被安全代码控制，但是具有 NS 非安全属性的 IP 则可以由安全或者非安全代码控制，即当某个外设或者其中的一部分具有 NS 非安全属性时，这个外设既允许非安全访问，也允许安全访问，具体规则如下：

- 上电缺省所有 Securable IP 属性都为 NS, NPRIV
- 具有 NS 属性的 IP，S 和 NS 代码都可以控制，具有 S 属性的 IP 只能由 S 代码控制
- 具有 NPRIV 属性的 IP，PRIV 和 NPRIV 代码都可以控制，具有 PRIV 属性的 IP 只能由 PRIV 代码控制

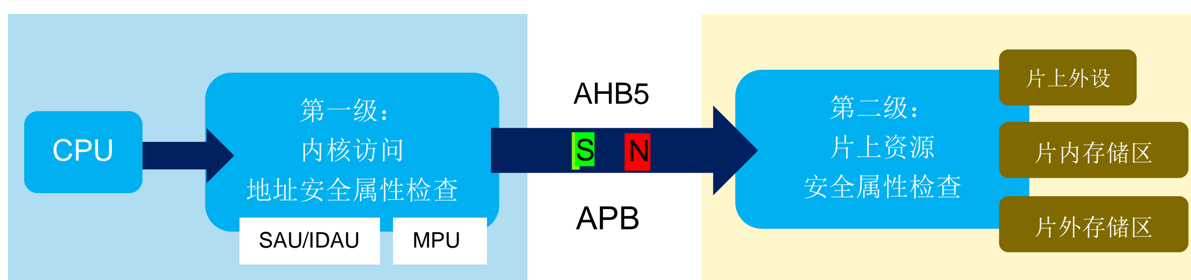
5. CPU 访问资源的安全规则

5.1. CPU 对资源访问的一般规则

CPU 的运行分为安全和非安全两种状态，CPU 处于哪种状态取决于当前 PC 的指令来自于安全地址还是非安全地址。简单地说，如果 CPU 执行的指令来自安全地址（以 SAU/IDAU 角度看），则 CPU 正在执行安全代码，处于安全状态；当 CPU 执行的指令来自于非安全地址（同样从 SAU/IDAU 角度看），则 CPU 正在执行非安全代码，处于非安全状态。

内核产生的任何访问都经过两级检查，如图 6 所示

图6. 内核访问的两级安全规则检查



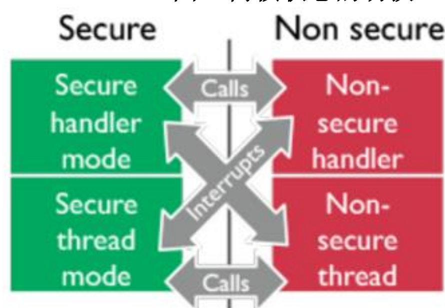
- 第一级检查发生在内核里：
  - SAU+IDAU 从被访问地址的安全/非安全配置的角度，检查 CPU 发起的某个访问是否被允许。如果该访问由 SAU/IDAU 判断需要被禁止（例如处于 NS 状态的 CPU 试图访问安全地址上的数据），则访问无法来到总线，同时会触发 SecureFault。
  - 如果 SAU+IDAU 的检查通过，而系统中还使能了 MPU 的情况下，MPU_S 和 MPU_NS 还将进一步从 MPU 的配置进行检查，判断某个访问是否被允许，例如访问的特权级别（PRIV）和非特权级别（NPRIV），某段地址是否允许读写，是否允许取指令等。如果出现违反 MPU 访问规则的情况（例如 NPRIV 代码试图读取 MPU 划分的 PRIV 地址），则访问无法来到总线，且会触发 Memory Fault。内核中有两套 MPU 寄存器，CPU 执行安全代码时，使用 MPU_S 的配置进行检查，CPU 执行非安全代码，使用 MPU_NS 的配置进行检查，二者互不相干。
- 第二级检查发生在资源侧：
  - Flash 控制器会根据片上 Flash 的 SECWM/SECBB 配置决定访问是否允许
  - GTZC 会根据 MPCBB 设置决定某个 SRAM block 是否允许访问
  - GTZC 会根据 MPCWM sub-region 设置决定外部 Memory 地址或者 Backup SRAM 是否允许访问
  - GTZC 会根据 TZSC PPC 设置决定某个外设寄存器是否允许访问
  - 其他 TZ aware 外设也会根据自身的相关配置决定访问是否被允许

CPU 的安全状态与其发出访问的安全、非安全标记并不一定一致。这里要区分取指令访问和数据读写访问两种情况。

- 取指令访问

- Cortex V8M TrustZone 架构 CPU 的安全、非安全状态由硬件进行切换，因此 CPU 可以在 Secure Handler，Secure Thread，NonSecure Handler 和 NonSecure Thread 四种模式间任意切换，如图 7 所示。

图7. 内核状态的切换

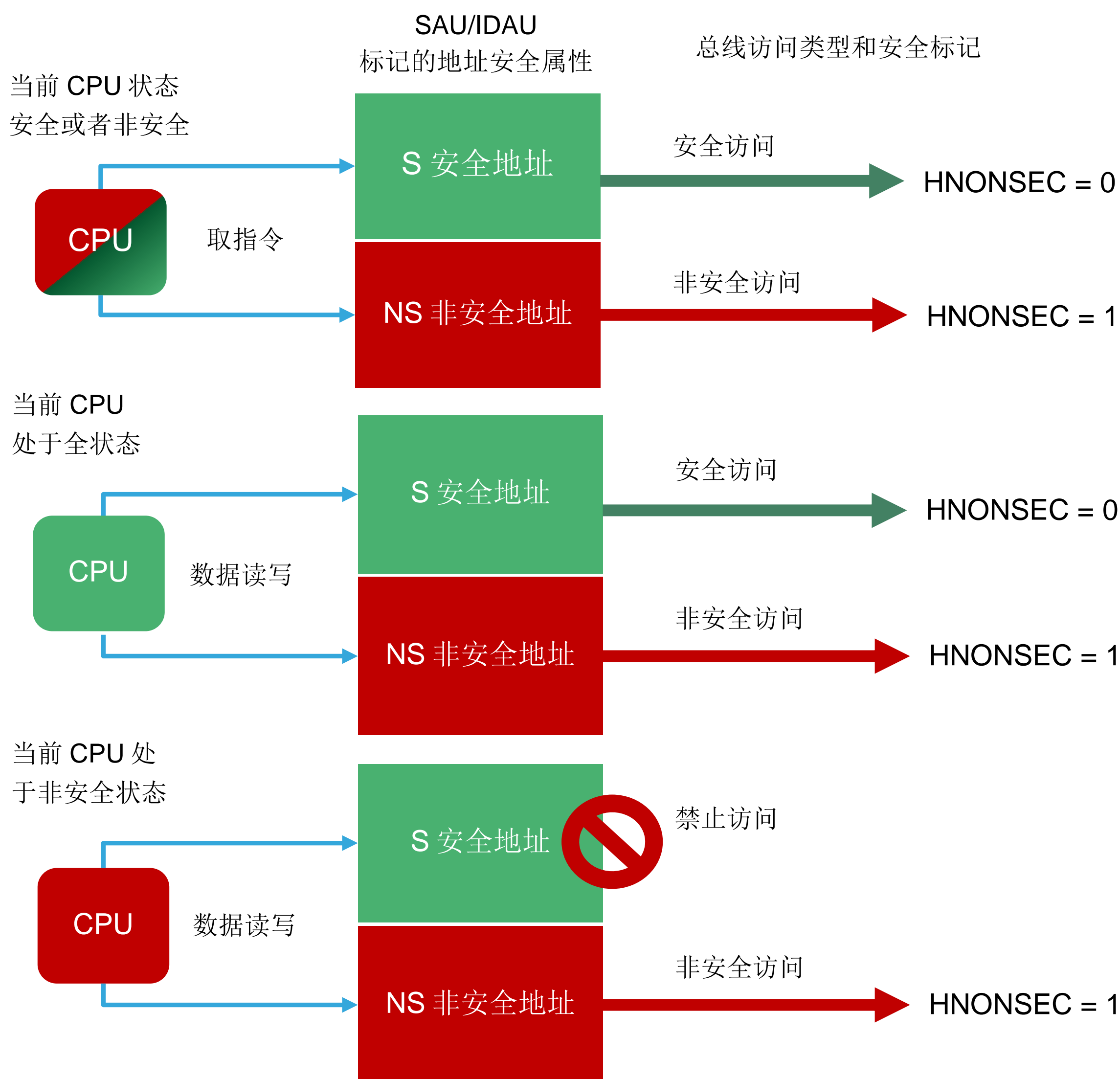


- 切换可能来自于函数调用（非安全代码调用安全代码 API，或者安全代码调用非安全注册的回调函数等）；也可能来自中断，例如 CPU 执行安全代码时可能被非安全中断打断，进入非安全中断处理程序，CPU 执行非安全代码时，也有可能被安全中断打断，直接进入安全中断处理程序。
- 这就意味着，无论 CPU 处于安全还是非安全状态，下一条将要取得的指令都有可能是安全或者非安全指令，因此，SAU/IDAU 对于取指令访问通常不做限制，总是允许 CPU 进行取指令操作，无论目标地址是安全地址还是非安全地址。这里唯一的特例是从 NS 非安全代码进行 S 安全函数调用的情况，这时候 SAU 将检查，处于非安全状态的 CPU 将要读取的安全函数的入口指令的地址是否在 NSC 地址范围内，如果不是，则会产生 SecureFault。
- 当 CPU 处于安全状态时，如果取指令操作的目标地址为 NS 非安全地址（以 SAU+IDAU 角度看），则经过 SAU/IDAU 后，来到总线上的访问将成为 NS 非安全取指令访问
- 当 CPU 处于非安全状态时，如果取指令操作的目标地址为 S 安全地址（以 SAU+IDAU 角度看），则经过 SAU/IDAU 后，来到总线上的访问将成为 S 安全取指令访问

- 数据读写访问
  - 当 CPU 处于 NS 非安全状态时
    - 如果数据访问的目标地址在 SAU/IDAU 中标记为 S 安全地址，则 SAU/IDAU 将禁止该数据访问，并且触发 SecureFault。
    - 如果数据访问的目标地址在 SAU/IDAU 中标记为 NS 非安全地址，则 SAU/IDAU 也会允许访问，且保持来到总线上的 transaction 的 NS 非安全访问标记(HNONSEC=1)
  - 当 CPU 处于 S 安全状态时
    - 如果数据访问的目标地址在 SAU/IDAU 中标记为 S 安全地址，则 SAU/IDAU 允许访问，且保持来到总线上的 transaction 的 S 安全访问标记 (HNONSEC=0)
    - 如果数据访问的目标地址在 SAU/IDAU 中标记为 NS 非安全地址，则 SAU/IDAU 也会允许访问，但是来到总线上的 transaction 的安全访问标记会指示当前访问为 NS 非安全访问 (HNONSEC=1)

图 8 总结了不同 CPU 状态下取指令和数据读写时产生的访问及其携带的安全标记的规律

图8. CPU 当前安全状态 vs. 访问目标地址安全属性 vs. 来到总线的访问携带的安全标记



5.2. SAU 配置效果对 CPU 访问的影响

接下来，以片上 FLASH 为例，看一下第三节中提到的 SAU 的三种配置情况，结合 IDAU 以及 FLASH SECWM 的选项字节的设置，分别会产生什么样的不同效果。

在下面的示例中，依旧假设 FLASH SECWM 设置 Flash 的第一个 Bank 的 1MB 为 S（offset 0 - 0xF FFFF），第二个 Bank 的 1MB 为 NS （offset 0x10 0000 - 0x1F FFFF），则 CPU 对不同地址进行**数据访问**时，我们分为对几种情况加以讨论。为了方便理解，这里使用一个表格的方式来分别列出以下一些内容

- IDAU、SAU 对某段地址的属性配置（a, b）
- IDAU+SAU 配置的最终组合效果 （c）
- CPU 在不同状态下，通过某个地址进行数据访问时的 transaction 的访问类型（即携带的 HNONSEC 信号标志为 S 还是 NS 访问）（d）
- 通过逻辑地址实际访问的 Flash 物理区域（第一个 1MB 还是第二个 1MB）

- Flash SECWM 对某个物理区域配置的安全属性（e）
- 最终访问是否成功的结果

5.2.1. Case1：不使能 SAU，且 SAU 默认所有地址为 S

这也是上电复位后 SAU 的缺省效果，不对 SAU 进行任何配置就是这个默认行为，即 SAU_CTRL.ENABLE=0，SAU_CTRL.ALLNS=0。

表4. CPU 运行安全代码时的数据访问情况

	a	b	c	d	e		
访问地址范围	IDAU 安全属性	SAU 安全属性	IDAU+SAU 最终组合的安全属性	CPU 运行 S 代码时访问该地址的 Transaction 安全类型	实际访问的 Flash 物理区域	Flash SECWM 安全属性	是否允许访问（取决于 d 与 e 是否匹配）
1：0x08000000 - 0x080FFFFF	NS	S	S	S	第一个 1MB	S	是
2：0x08100000 - 0x081FFFFF	NS	S	S	S	第二个 1MB	NS	否
3：0x0C000000 - 0x0C0FFFFF	NSC	S	S	S	第一个 1MB	S	是
4：0x0C100000 - 0x0C1FFFFF	NSC	S	S	S	第二个 1MB	NS	否

当 SAU 不使能，保持默认状态的时候，所有地址从 SAU 角度看都是安全地址，因此，这种配置下，一定只有安全代码可以运行。所以表格中仅仅列出了当 CPU 为安全状态时的访问效果，如果 CPU 为非安全状态，任何访问都将被 SAU 禁止。

从表 4 中可以看到，此时，只能够从 1、3 两个地址范围访问 Flash 的前 1MB 的内容，这两个逻辑地址都对应 Flash 的前 1MB 区域。而由于 Flash 的第二个 1MB 具有 NS 非安全属性，只能允许 NS 非安全访问，因此，无论从 0x08xxxxxx 还是 0x0Cxxxxxx 都不能正常读取 Flash Bank2 第二个 1MB 空间的内容。

5.2.2. Case2：不使能 SAU，且 SAU 默认所有地址为 NS

这是一个最简单的 SAU 配置，也就是说不使能 SAU，但是让 SAU 默认所有地址空间都是非安全属性，即 SAU_CTRL.ENABLE=0，SAU_CTRL.ALLNS=1，那么这个时候 SAU 可以看作无效，所有地址的安全属性遵循 IDAU 的默认配置。

这种情况下，就需要区分两种情况来看，一个是 CPU 运行 S 代码（即 CPU 处于安全状态）时候的数据访问效果，另一个是 CPU 运行 NS 代码（即 CPU 处于非安全状态）时候的数据访问效果。

下面两个表格分别列出了 CPU 处于 S /NS 状态时，对不同地址进行**数据**访问时的情况及其结果。

表5. CPU 运行安全代码时的数据访问情况

	a	b	c	d	e		是否允许访问（取决于 d 与 e 是否匹配）
访问地址范围	IDAU 安全属性	SAU 安全属性	IDAU+SAU 最终组合的安全属性	**CPU 运行 S 代码时访问该地址的 Transaction 安全类型**	实际访问的 Flash 物理区域	**Flash SECWM 安全属性**	
1 : 0x08000000 - 0x080FFFFF	NS	NS	NS	NS	第一个 1MB	S	否
2 : 0x08100000 - 0x081FFFFF	NS	NS	NS	NS	第二个 1MB	NS	是
3 : 0x0C000000 - 0x0C0FFFFF	NSC	NS	NSC	S	第一个 1MB	S	是
4 : 0x0C100000 - 0x0C1FFFFF	NSC	NS	NSC	S	第二个 1MB	NS	否

从表 5 可以看出，S 代码可以从 2、3 两个地址范围分别访问 Flash 的 NS（第二个 1MB）和 S 区域（第一个 1MB）

表6. CPU 运行非安全代码时的数据访问情况

	a	b	c	d	e		是否允许访问(取决于访问是否被 SAU/IDAU 禁止以及 d 与 e 是否匹配)
访问地址范围	IDAU 安全属性	SAU 安全属性	IDAU+SAU 最终组合的安全属性	**CPU 运行 NS 代码时访问该地址的 Transaction 安全类型**	实际访问的 Flash 物理区域	**Flash SECWM 安全属性**	
1 : 0x08000000 - 0x080FFFFF	NS	NS	NS	NS	第一个 1MB	S	否
2 : 0x08100000 - 0x081FFFFF	NS	NS	NS	NS	第二个 1MB	NS	是
3 : 0x0C000000 - 0x0C0FFFFF	NSC	NS	NSC	数据访问被 IDAU 禁止	第一个 1MB	S	否
4 : 0x0C100000 - 0x0C1FFFFF	NSC	NS	NSC	数据访问被 IDAU 禁止	第二个 1MB	NS	否

从表 6 可以看出 NS 代码只能从第 2 个地址范围访问 Flash 的 NS 区（第二个 1MB），通过其他的地址段的数据访问，要么会被 SAU+IDAU 禁止，要么会被 Flash 控制器阻止。

总结这种配置下的访问效果：

- Flash 的第一个 1MB（SECWM 中的 S 属性区域）

- ○ S 代码可以通过 0x0C000000 - 0x0C0FFFFF 地址访问
- ○ NS 无论通过什么地址都无法访问这个部分，要么被 SAU/IDAU 拦截，要么被 Flash controller 禁止

• Flash 的第二个 1MB（SECWM 中的 NS 属性）

- ○ S 代码和 NS 代码均可以通过 0x08100000 - 0x081FFFFF 地址进行访问
- ○ 如果 S 代码想要通过 0x0C100000 - 0x0C1FFFFF 地址进行访问，则需要通过配置 FLASH SECBB 将对应的 FLASH 区域临时动态设置为 S 属性

在有 S/NS 项目的时候，保证 S 和 NS 代码各自放置于对应的 0x0Cxxxxxx 和 0x08xxxxxx 地址范围就可以了，NSC 区域只需要位于 0x0Cxxxxxxxx 且 offset 不超过第一个 1MB 即可，没有具体地址的限制。

5.2.3. Case3：使能 SAU，并根据需要配置不同的 region

这是实际应用中通常会使用的方式，根据实际的安全和非安全地址划分来配置 SAU。这里假设 SAU 对内部 Flash 地址配置了 NS 和 NSC 两个 region，NS region 为 0x08100000 - 0x081FFFFF，用于运行非安全代码，NSC region 为 0x0C0FE000 - 0x0C0FFFFF，作为非安全代码调用安全代码 API 的入口地址，其余没有配置的地址 SAU 默认依旧为 S 安全地址。

下面两个表格分别列出了 CPU 处于 S/NS 状态时对不同地址进行**数据**访问时的效果。

表7. CPU 运行安全代码时的数据访问情况

	a	b	c	d	e		
访问地址范围	IDAU 安全属性	SAU 安全属性	IDAU+SAU 最终组合的安全属性	**CPU 运行 S 代码时访问该地址的 Transaction 安全类型**	实际访问的 Flash 物理区域	**Flash SECWM 安全属性**	是否允许访问取决于 d 与 e 是否匹配
1：0x08000000 - 0x080FFFFF	NS	S	S	S	第一个 1MB	S	是
2：0x08100000 - 0x081FFFFF	NS	NS	NS	NS	第二个 1MB	NS	是
3：0x0C000000 - 0x0C0FDFFF	NSC	S	S	S	第一个 1MB	S	是
4：0x0C0FE000 - 0x0C0FFFFF		NSC	NSC	S		S	是
5：0x0C100000 - 0x0C1FFFFF	NSC	S	S	S	第二个 1MB	NS	否

从表 7 中可以看出，这种配置下，安全代码可以从 1、2、3、4 几个地址范围访问 Flash 的不同区域，包括 S 安全和 NS 非安全区，唯一无法正常访问数据的只有地址范围 5，因为不能通过安全地址访问非安全属性的 Flash 区域。

表8. CPU 运行非安全代码时的数据访问情况

	a	b	c	d	e	

访问地址范围	IDAU 安全属性	SAU 安全属性	IDAU+SAU 最终组合的安全属性	**CPU 运行 NS 代码时访问该地址的 Transaction 安全类型**	实际访问的 Flash 物理区域	**Flash SECWM 安全属性**	是否允许访问取决于访问是否被 SAU/IDAU 禁止以及 4 与 5 是否匹配
1：0x08000000 - 0x080FFFFF	NS	S	S	数据访问被 SAU 禁止	第一个 1MB	S	否
2：0x08100000 - 0x081FFFFF	NS	NS	NS	NS	第二个 1MB	NS	是
3：0x0C000000 - 0x0C0FDFFF	NSC	S	S	数据访问被 SAU/IDAU 禁止	第一个 1MB	S	否
4：0x0C0FE000 - 0x0C0FFFFF		NSC	NSC	数据访问被 SAU/IDAU 禁止		S	否
5：0x0C100000 - 0x0C1FFFFF	NSC	S	S	数据访问被 SAU/IDAU 禁止	第二个 1MB	NS	否

从表 8 中可以看出，这种配置下，NS 非安全代码只能够从地址范围 2 对 Flash 的 NS 非安全区（第二个 1MB）进行数据访问，访问其他的地址范围都将被 SAU/IDAU 禁止。

这种配置的效果和 Case2 类似，但也有些区别，总结如下：

- Flash 的第一个 1MB（SECWM 标记位 S 属性的区域）
  - S 代码可以通过 0x0C000000 - 0x0C0FFFFF 地址访问，
  - S 代码也可以通过 0x08000000 - 0x080FFFFF 地址访问
  - NS 无论通过什么地址都无法访问这个部分
- Flash 的第二个 1MB（SECWM 标记位 NS 属性的区域）
  - S 代码和 NS 代码均可以通过 0x08100000 - 0x081FFFFF 地址访问
  - 如果 S 代码想要通过 0x0C100000 - 0x0C1FFFFF 地址进行访问，则需要通过配置 FLASH SECBB 将对应的 FLASH 区域临时动态设置为 S 属性
- 特别地，Flash 的第一个 1MB 中 0x0C0FE000 - 0x0C0FFFFF 为 NSC 区域，只有这个区域可以放置 S API 的入口 SG 指令，其他地址均为 S 属性（也就是说其他地址不能放置 SG 指令，不能被 NS 作为 S 函数调用入口进行函数调用使用）

有 S/NS 项目的时候，需要保证 S/NS 代码各自放置于对应的 0x0Cxxxxxx 和 0x08xxxxxx 地址范围，同时 S 项目 linker file 的 NSC 区域必须在 SAU 定义的 NSC 区域范围内（linker 文件中的定义要和 SAU 配置保持一致）。

这部分 SAU 的配置效果是以 Flash 进行举例的，其实对于 SRAM 和其他的 Memory 是类似的，比如对于 SRAM 的访问，根据哪些地址允许 S/NS 访问，需要使得 SAU/IDAU 的配置与 MPCBB 对 SRAM 的安全属性配置匹配，对于外部 Memory 的访问，根据哪些地址允许 S/NS 访问，需要使得 SAU/IDAU 的配置与 MPCWM 对外部 Memory 的安全属性配置匹配，等等。

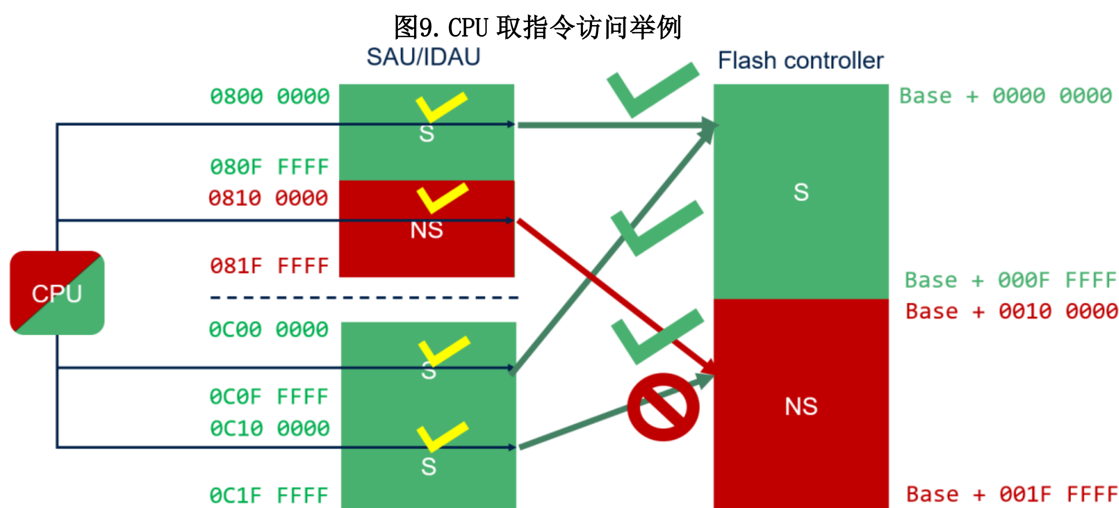
注意：

前面讨论的情况都是以 CPU 进行数据访问为例。如果是指令访问，那么 SAU/IDAU 通常都会允许访问（当然从 NS 调用 S API 时，会有入口地址是否为 NSC 属性的检查，除此之外不会禁止取指令的访问），而且，所访问的目标地址在 SAU/IDAU 中的配置是 S 还是 NS 属性，将会决定该取指令访问的 transaction 的 S/NS 属性，这与当前 CPU 的状态无关，所以一般不会出现指令访问被 SAU/IDAU 禁止的情况，但是取指令操作是否能够成功完成，还要看目标地址对应的物理存储器的 S/NS 属性，如果属性不一致也会产生取指令失败的结果，也就是前面表中提到的 transaction 的安全标记与被访问物理存储区的安全属性要匹配，否则会出现 RAZ/WI。

下面几个图片以 Flash 和外设为例，给出了 CPU 进行取指令访问、取数据访问以及进行外设寄存器访问时候的资源访问规则。

黄色对勾表示 SAU/IDAU 允许访问，绿色对勾表示资源（例如 Flash 控制器）允许改访问，红色园杠表示访问 SAU/IDAU 或者资源禁止。

图9. CPU 取指令访问举例



取指令访问对 CPU 状态无特别限制，除了从 NS 调用 S API 的情况。当 NS 代码通过非 NSC 区域进行函数调用，从 S 区直接作为入口取指令，则会被 SAU 禁止，这种情况见图 10 的示例。

图10. CPU 取指令访问举例（NS 调用 S API）

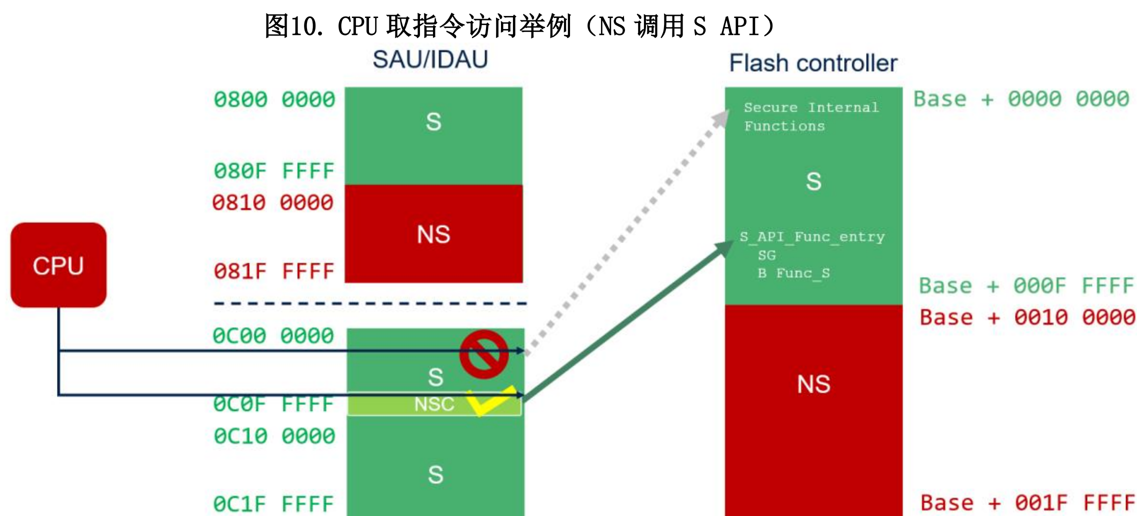


图11. 安全状态 CPU 取数据访问举例

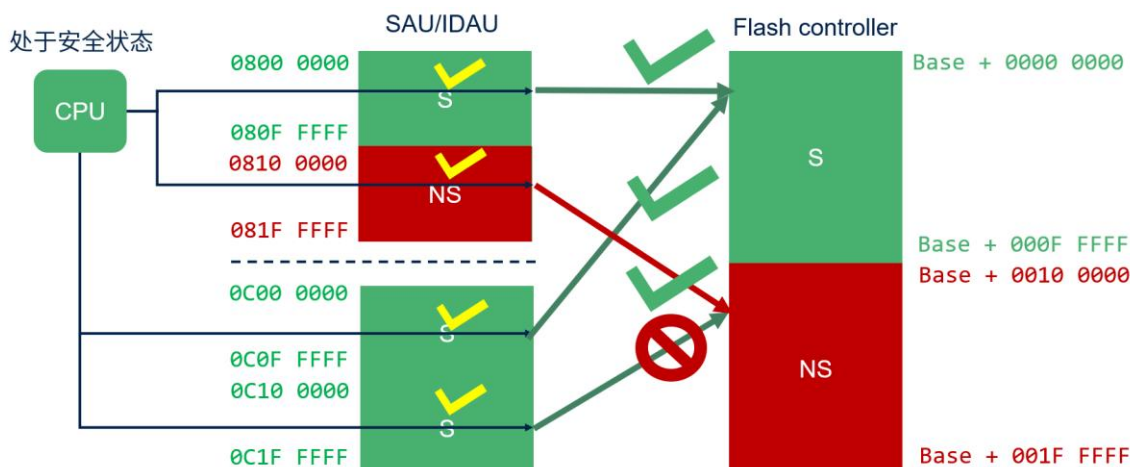
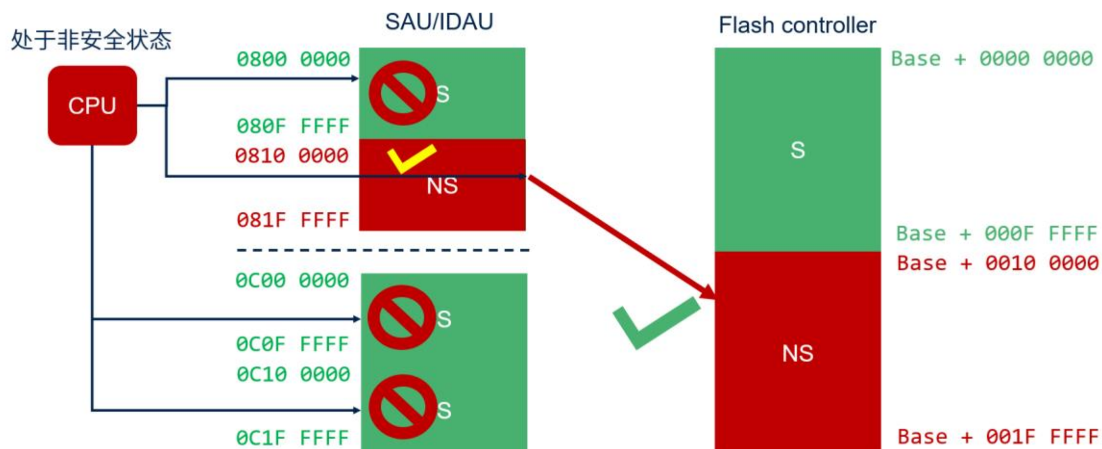


图 11 的示例说明，只要 SAU 配置正确，安全状态 CPU 可以从不同地址访问到所有 Memory 资源。

图12. 非安全状态 CPU 取数据访问举例



由图 12 可见，非安全状态的 CPU 想要访问 Memory 需要 SAU 配置对应的 NS region，配合 Memory 资源中的 NS 区域配置，同时只能从非安全地址访问非安全属性的 Memory 资源。

图13. 安全状态 CPU 外设寄存器访问举例

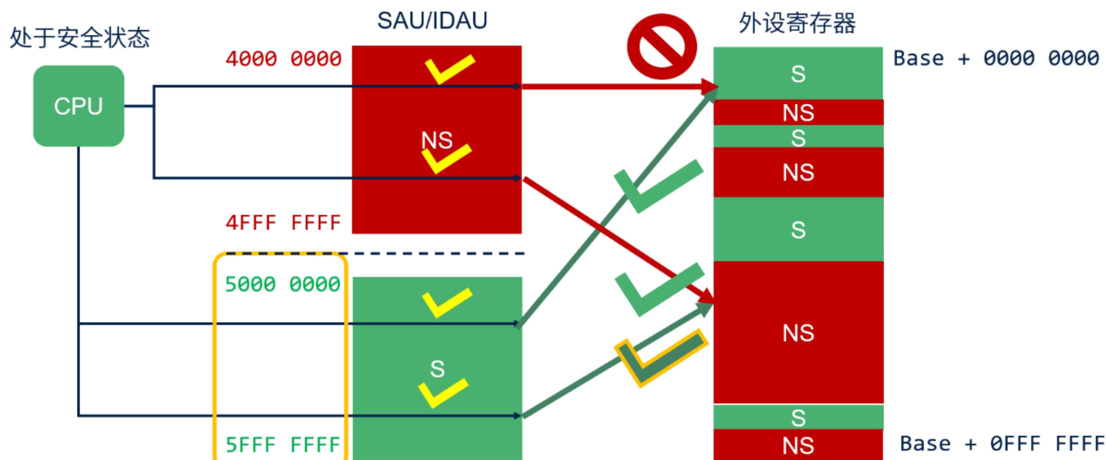


图 13 给出了安全状态 CPU 从不同地址访问外设的情况，可以看到，安全状态的 CPU 可以从安全或者非安全资源访问所有非安全属性的外设，可以从安全地址访问安全属性的外设。

图14. 非安全状态 CPU 外设寄存器访问举例

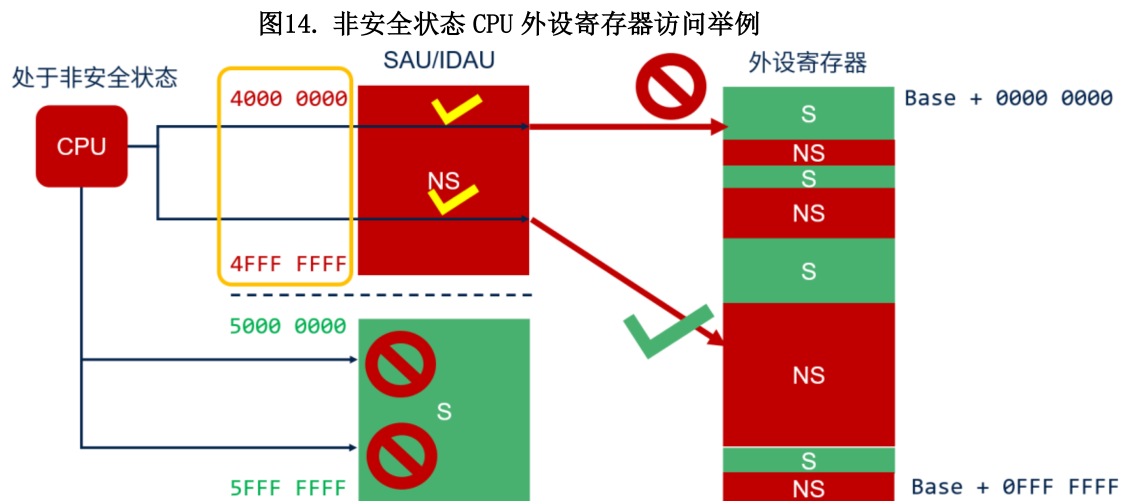


图 14 的示例是非安全状态 CPU 访问外设寄存器的情况，可以看出，非安全状态的 CPU 只能从非安全地址访问非安全属性的外设。

6. TrustZone Memory 安全配置常见问题及分析方法

6.1. 上电 Secure 代码无法运行

正常情况下，安全代码能够运行的基本条件包括以下几点：

- CPU 处于安全状态
- SAU 将安全代码 code 区（通常为 Flash）和 data 区（通常为内部 SRAM）配置为 S 安全属性
- 安全代码使用的 SRAM 区具有 S 安全访问属性
- 安全代码 code 所在的 Flash 区配置为 S 安全访问属性
- 启动地址与代码链接文件指定的启动地址一致

上电的默认状态：

- CPU 自动处于安全状态，只能运行安全代码；
- SAU 默认所有地址都是 S 安全属性，因此不需要配置 SAU 也可以运行安全代码；
- SRAM 上电默认具有 S 安全访问属性（GTZC MPCBB 默认状态），因此不需要额外配置也应该可以运行安全代码
- Flash 上电的默认配置取决于 Flash controller 的 SECWM 配置
- 启动地址可能取决于 Option Byte 中 SecBootAdd 的配置（需要参考具体 STM32 型号的参考手册）

由此可见，如果上电安全代码无法运行，那么问题应该可能出现在两个地方：Flash SECWM 的配置，以及 SECBOOTADD 地址。此时首先应当检查芯片的 OptionByte 中的 SECBOOTADD 配置是否与代码的启动地址一致，可以通过 STM32CubeProgrammer 读取 OptionByte，如图 15 所示。

图15. OptionByte 中的 SECBOOTADDR 设置

Boot Configuration

Name	Value	Description
NSBOOTADD0	Value 0x1000 Address 0x08000000	Non-secure Boot base address 0
NSBOOTADD1	Value 0x17f2(Address 0x0bf90000	Non-secure Boot base address 1
SECBOOTADD0	Value 0x1800 Address 0x0c000000	Secure boot base address 0

如果 SECBOOTADD 正确，则需要检查 Flash SECWM 的配置，确认安全 Flash 区已经在 SECWM 对应的 OptionByte 中正确配置，例如图 16 所示。

图16. Option Byte 中的 Flash SECWM 安全区设置

Secure Area 1

Name	Value	Description
SECWM1_PSTRT	Value 0x0 Address 0x08000000	Start page of first secure area
SECWM1_PEND	Value 0x7f Address 0x080fe000	End page of first secure area

如果 Flash SECWM 没有设置 Secure 安全区，那么上电运行安全代码会立即触发 Hardfault，此时如果在调试器中观察 SCB（System Control Block）的 CFSR 寄存器，会发现 STKERR 和 IBUSERR 被置位，如果进一步从 memory 窗口观察代码启动地址的 flash 内容，会发现读取到的数据都为 0，图 17 为 IAR 调试窗口观察结果示例。产生这样效果是因为 Flash SECWM 中如果没有设置 Secure 区，那么 CPU 以安全方式访问 Flash 的结果就是 RAZ，此时一定无法获取正确的 stack pointer 和 reset handler PC，因而会触发 Hardfault。

图17. Flash SECWM 未设置安全区时 IAR 调试窗口观察 CFSR 和 Flash 启动地址内容

Registers 1 — Find: cfsr — Group: System Control Blo

Name	Value
CFSR	0x00001100
DIVBYZERO	0
UNALIGNED	0
STKOF	0
NOCP	0
INVPC	0
INVSTATE	0
UNDEFINSTR	0
BFARVALID	0
LSPERR	0
STKERR	1
UNSTKERR	0
IMPRECISERR	0
PRECISERR	0
IBUSERR	1
MMARVALID	0
MLSPERR	0
MSTKERR	0
MUNSTKERR	0
DACCVIOL	0
IACCVIOL	0

Memory 1 — Go to 0x0c000000 — Memory

```
0x0c000000  00000000  00000000
0x0c000008  00000000  00000000
0x0c000010  00000000  00000000
0x0c000018  00000000  00000000
0x0c000020  00000000  00000000
0x0c000028  00000000  00000000
0x0c000030  00000000  00000000
0x0c000038  00000000  00000000
0x0c000040  00000000  00000000
0x0c000048  00000000  00000000
0x0c000050  00000000  00000000
0x0c000058  00000000  00000000
0x0c000060  00000000  00000000
0x0c000068  00000000  00000000
0x0c000070  00000000  00000000
0x0c000078  00000000  00000000
0x0c000080  00000000  00000000
0x0c000088  00000000  00000000
0x0c000090  00000000  00000000
0x0c000098  00000000  00000000
0x0c0000a0  00000000  00000000
0x0c0000a8  00000000  00000000
0x0c0000b0  00000000  00000000
0x0c0000b8  00000000  00000000
0x0c0000c0  00000000  00000000
0x0c0000c8  00000000  00000000
0x0c0000d0  00000000  00000000
0x0c0000d8  00000000  00000000
0x0c0000e0  00000000  00000000
```

6.2. 上电后 NonSecure 代码无法运行

通常在 TrustZone 架构中，上电首先运行安全代码，安全代码需要完成一系列的 TrustZone 安全配置，然后跳转至非安全代码部分执行，非安全代码能够执行的最基本条件包括以下几点：

- CPU 由安全状态切换至非安全状态，通常由安全代码中的跳转指令完成，编译器会产生类似 BLXNS 的指令，完成跳转和状态切换。跳转前已经正确设置 NS 的 MSP，并取得 NS reset handler 的 PC 值。
- SAU 将非安全代码 code 区所在地址（通常为 Flash）和 data 区所在地址（通常为内部 SRAM）配置为 NS 非安全属性，通常还需要配置非安全代码调用安全 API 的 NSC 区，以及访问 driver 寄存器地址的 NS 非安全区
- 非安全代码使用的 SRAM 区具有 NS 非安全访问属性（MPCBB 配置）
- 非安全代码 code 所在的 Flash 区配置为 NS 非安全访问属性（SECWM 配置）
- 非安全代码的 linker 文件指定的 code 和 data 地址范围应当与 SAU 的配置以及 Flash SECWM 和 GTZC MPCBB 的配置一致

如果由于 Flash/SRAM 等非安全区未正确配置导致的非安全代码无法执行，这种情况下，如果 GTZC 的 TZIC 中断以及 Flash/MPCBB 对应的 illegal access event 使能，则可能会触发 GTZC 中断，通过读取 TZIC 的 SRx 寄存器可以大概知道是那个模块产生了非法访问，导致了中断的触发。在图 18 中，我们可以看到，当 SAU 没有配置，且 GTZC 中断使能的情况下，跳转 NS 代码触发了 GTZC 中断，此时 TZIC 给出的指示是 FLASH_REGF，表示 Flash 寄存器出现了非法访问。这时候，其实我们能看到 MPS_NS 为 0，本应存放跳转 PC 值的 R0 也是 0，并不是正确的值，这说明无法从 NS 代码区取得正确的 stack pointer 和 reset handler 地址。

图18. GTZC 中断触发时，通过 IAR 调试窗口观察 SEC_TZIC 寄存器内容

Registers 1 — Find: tzic — Group: SEC_GTZC1_1

Name	Value
IER1	0x001fffff
IER2	0x000001ff
IER3	0x007fffff
IER4	0x3f0fc01f
SR1	0x00000000
SR2	0x00000000
SR3	0x00000000
SR4	0x00000004
MPCBB3_REGF	0
SRAM3F	0
MPCBB2_REGF	0
SRAM2F	0
MPCBB1_REGF	0
SRAM1F	0
OCTOSPI2_MEMF	0
BKPSRAMF	0
FSMC_MEMF	0
OCTOSPI1_MEMF	0
TZIC1F	0
TZSC1F	0
OTFDEC2F	0
OTFDEC1F	0
FLASH_REGF	1
FLASHF	0
GPDMA1F	0

Registers 2 — Find: — Group: CPU registers (Non-

Name	Value
R0	0x00000000
R1	0xe002ed08
R2	0x00000000
R3	0x02000000
R4	0x56020c88
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0xffffffff
R9	0xffffffff
R10	0xffffffff
R11	0xffffffff
R12	0x00000000
xPSR	0x41000018
APSR	0x40000000
IPSR	0x00000018
EPSR	0x01000000
PC	0x0c00de12
MSP_NS	0x00000000
MSPLIM_NS	0x00000000
PSP_NS	0xfffffffc
PSPLIM_NS	0x00000000
LR	0xfffffffe9
PRIMASK_NS	0x00000000
BASEPRI_NS	0x00000000

Disassembly — Go to 0x0c00bce3 — Memory

```
Disassembly
}
                    SVC_Handler:
   0xc00'de08: 0x4770          BX
 while (1)
                    DebugMon_Handler:
   0xc00'de0a: 0xe7fe          B.N
 while (1)
                    PendSV_Handler:
   0xc00'de0c: 0xe7fe          B.N
 HAL_IncTick();
                    SysTick_Handler:
   0xc00'de0e: 0xf000 0xb803   B.W
 HAL_GTZC_IRQHandler();
                    GTZC_IRQHandler:
   0xc00'de12: 0xf7fd 0xbf25   B.W
   0xc00'de16: 0x0000          MOVS
 uwTick += (uint32_t)uwTickFreq;
                    HAL_IncTick:
   0xc00'de18: 0x4902          LDR.N
   0xc00'de1a: 0x684a          LDR
   0xc00'de1c: 0x7808          LDRB
   0xc00'de1e: 0x1882          ADDS
   0xc00'de20: 0x604a          STR
}
   0xc00'de22: 0x4770          BX
```

注意：在 IAR 调试中如果打开某些寄存器显示窗口以及 memory 显示窗口，都有可能导致 GTZC 中断的触发，原因是 debugger 在这些窗口中试图访问某些寄存器或者 memory 地址的

时候也有可能与外设寄存器或者 memory 的安全属性冲突，也可能触发 GTZC 中断。所以，在调试器中尝试查看 GTZC event 的时候，建议关闭不必要的寄存器以及 memory 窗口，避免额外的 GTZC TZIC event 被触发。

如果没有使能 GTZC 中断，这种情况也可能触发 SecureFault，此时可以在调试器中查看 SFSR 以及 SFAR 寄存器，它们可能会给出一些指示，比如错误的原因和产生 Fault 的地址（如果 SFSR.SFARVALID=1），可以有针对性的检查相关配置是否正确。

图 19 是 SAU 没有配置，跳转 NS 触发 SecureFalut 时，在 IAR 中观察 CPU_NS 寄存器和 SFSR/SFAR 寄存器的示例。

这里 SFSR 显示 AUVIOL 置位，说明出现了 NS transaction 试图访问 S 地址的情况，这里 SFAR 给出的地址是 0xfffffff8，原因可能是由于跳转前 NS Flash 区访问失败，导致 MSP_NS 被置为 0；同时 INVEP 置位，表示出现了非法 entry point 的错误，原因也是跳转的 PC 指针不正确。

图19. SecureFault 触发时，通过 IAR 调试窗口观察 SFSR/SFAR 寄存器内容

Registers 1 — Find: sfsr — Group: System Control

Name	Value
AFSR	0x00000000
AIRCR	0xfa050300
BFAR	0xfffffff8
CCR	0x00000201
CFSR	0x00000000
CPACR	0x00f00000
NSACR	0x00000c00
DFSR	0x0000000b
HFSR	0x00000000
SFSR	0x00000049
LSERR	0
SFARVALID	1
LSPERR	0
INVTRAN	0
AUVIOL	1
INVER	0
INVMS	0
INVEP	1
SFAR	0xfffffff8
ICSR	0x00000807
CPUID	0x410fd214
ID_PFR0	0x00000030
ID_PFR1	0x00000210
ID_DFR0	0x00200000
ID_AFR0	0x00000000
ID_MMFR0	0x00101f40
ID_MMFR1	0x00000000
ID_MMFR2	0x01000000
ID_MMFR3	0x00000000
ID_ISAR0	0x01101110
ID_ISAR1	0x02212000

Registers 2 — Find: — Group: CPU registers (Non-

Name	Value
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
xPSR	0x01000007
APSR	0x00000000
IPSR	0x00000007
EPSR	0x01000000
PC	0x0c00dca8
MSP_NS	0xffffffe0
MSPLIM_NS	0x00000000
PSP_NS	0xfffffffc
PSPLIM_NS	0x00000000
LR	0xfffffffb9
PRIMASK_NS	0x00000000
BASEPRI_NS	0x00000000
BASEPRI_MAX_NS	0x00000000
FAULTMASK_NS	0x00000000
CONTROL_NS	0x00000000
IAPSR	0x00000007
EAPSR	0x01000000
IEPSR	0x01000007

Disassembly — Go to: 0x0c00bce3 — Memory

```
                NMI_Handler:
  0xc00'dc9c: 0x4770          BX
  while (1)
                HardFault_Handler:
  0xc00'dc9e: 0xe7fe          B.N
  while (1)
                MemManage_Handler:
  0xc00'dca0: 0xe7fe          B.N
  while (1)
                BusFault_Handler:
  0xc00'dca2: 0xe7fe          B.N
  while (1)
                UsageFault_Handler:
  0xc00'dca4: 0xe7fe          B.N
void SecureFault_Handler(void)
{
                SecureFault_Handler:
  0xc00'dca6: 0xb580          PUSH
  if(pSecureFaultCallback != (funcptr_
  0xc00'dca8: 0x480f          LDR.N
  0xc00'dcaa: 0x6800          LDR
  0xc00'dcac: 0xb1d8          CBZ
  callback_NS();
  0xc00'dcae: 0xb0a2          SUB
  0xc00'dcb0: 0xec2d 0x0a00   VLSTM
  0xc00'dcb4: 0x0840          LSRS
  0xc00'dcb6: 0xe92d 0x0ff0   PUSH.W
  0xc00'dcba: 0x2100          MOVS
  0xc00'dcbc: 0x0040          LSLS
  0xc00'dcbe: 0x2200          MOVS
  0xc00'dcc0: 0x2300          MOVS
```

通常出现此类 SecureFault 或者 GTZC 中断时，可以首先检查 SAU 配置：

- 是否正确配置了 NS 代码需要的 Flash NS 非安全区（注意地址别名，NS Flash 区的地址应该以 0x08xxxxxx 开头，且地址范围应当与 NS code linker 文件的配置一致）
- 是否正确配置了 NS 代码的 SRAM NS 非安全区（注意地址别名，NS SRAM 区的地址应该以 0x2xxxxxxx 开头，且地址范围应当与 NS code linker 文件的配置一致）

- 是否正确配置了 Secure NSC 区，用于 NS 代码调用 S API（注意地址别名，Flash NSC 区的地址应该以 0x0Cxxxxxx 开头，且地址范围与 S 代码的 linker 文件中指定的 NSC 区一致）

如果确认 SAU 配置正确，但是调试发现依旧无法取得 NS 代码的指令，或者发现无法正常操作 NS SRAM，这时候应当怀疑 Flash SECWM 或者 SRAM 的安全属性在 GTZC 中的配置有有问题。此时，可以做以下一些调试和检查：

- 在 S 跳转 NS 的指令设置断点，在调试器的 memory 窗口输入 NS 代码启动地址，观察其内容，如果发现都是 0，说明 Flash SECWM 没有正确配置 Flash NS 区（或者与 SAU 配置不匹配）。
- IAR 中观察 NS 代码地址内容可能如图 20 所示，R0（即跳转地址）和 MSP_NS 都是 0，Flash 对应地址的内容也是 0。然而，如果从 0x0C000000 开始读取相同偏移量位置的 Flash 内容，则可以读取到正确的结果。这说明该段 Flash 区域只能允许安全方式访问，具有安全属性。原因可能是 Flash SECWM 的 NS 非安全区配置错误，导致从 flash 非安全区地址读取内容时，transaction 携带的安全信号与实际 Flash 地址的安全属性不一致，导致 RAZ。

图20. 通过 IAR 调试窗口在跳转 NS 之前观察 CPU 寄存器和 Flash 地址内容

- Flash SECWM 没有配置 NS 非安全区可能在 OptionByte 中看到的是图 21 这样的设置，这是使能 TZEN 后的缺省配置：

图21. OptionByte 中 SECWM 没有配置 Flash NS 区

Secure Area 1

Name	Value		Description
SECWM1_PSTRT	Value 0x0	Address 0x08000000	Start page of first secure area
SECWM1_PEND	Value 0x7f	Address 0x080fe000	End page of first secure area

Secure Area 2

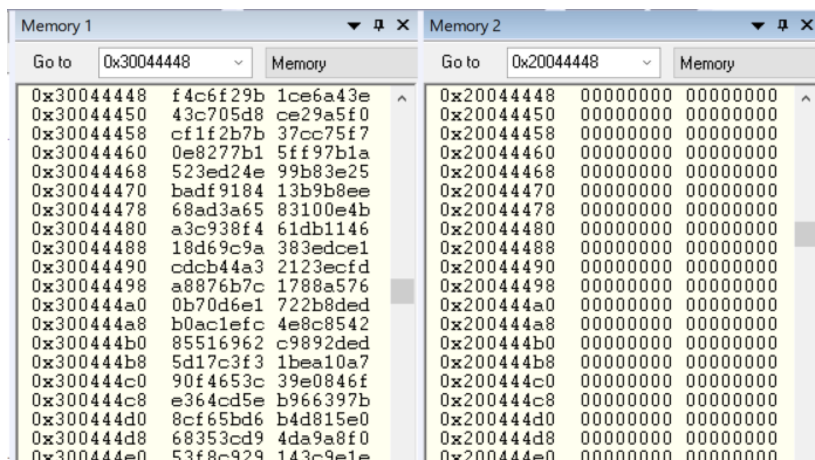
Name	Value		
SECWM2_PSTRT	Value 0x0	Address 0x08100000	Start page of second secure area
SECWM2_PEND	Value 0x7f	Address 0x081fe000	End page of second secure area

- 如果 NS App 指令的地址内容是正常的，可以继续在 memory 窗口观察 NS SRAM 区内容，此时可以将 MSP_NS 指向的 SRAM 地址输入到 memory 窗口，查看其内容，结果可能都是 0。图 22 中所示为 IAR 中调试观察示例，R0 中是即将跳转的 PC 地址，MSP_NS 是从 NS 启动地址读取到的 stack pointer，由此可见内容都是合理的，说明 Flash NS 区读取没有问题。

图22. 通过 IAR 调试窗口在跳转 NS 之前观察 CPU 寄存器和 SRAM 地址内容

- 但是 memory 窗口中看到的 MSP_NS 对应的 SRAM 地址内容全部是 0，如图 23 所示，说明 GTZC MPCBB 没有正确配置 SRAM NS 区（或者与 SAU 配置不匹配）。而如果从 0x30000000 开始以同样的偏移量查看 SRAM 内容却能够读取到正确的值，说明该段 SRAM 区域具有安全访问属性，只能通过安全方式访问，也就是说 MPCBB 没有正确地进行 SRAM NS 区域配置。

图23. IAR 调试窗口中观察 SRAM 地址内容



- 这时候应当按照 NS 代码 linker 文件使用的 RAM 地址对 GTZC MPCBB 进行正确配置，这部分配置代码需要在 S code 的初始化代码中完成。

6.3. NonSecure 代码访问外部 memory 失败

在一些应用当中，由于内部 memory 空间大小的限制，运行在 NS 侧的应用程序可能会使用到外部 Memory，用于代码执行以及数据读写，这时候在开发阶段往往也会出现 NS App 无法正确访问外部 Memory 的情况。与上一小节提到的情况类似，NS 代码以地址映射的方式直接通过地址访问外部 Memory 时候，也需要满足以下几个条件：

- SAU 中有对应外部 memory 访问的地址范围的 region 且设置为 NS 非安全属性
- 或者 SAU 不使能，且 SAU_CTRL.ALLNS=1，此时 SAU 默认所有地址为 NS 属性，IDAU 默认外部 memory 映射的地址也具有 NS 属性
- GTZC MPCWM 配置了该外部 memory 映射地址 sub-region，且属性设置为 NS（请参考 4.3 章节的说明）

类似上一小节提到的方法，通过调试的手段也比较容易发现问题的所在，当访问外部 memory 地址出现问题的时候，可以在调试器的 memory 窗口查看相应地址是否能够正常地读取数值，如果是 RAM，应当能够正常读写。这里需要注意一点，有些 STM32 系列带有 DCACHE 单元，能够对某些 IP 外挂的 RAM 做读写 cache。如果在这些系列上已经使能了 DCACHE，这时候在调试器的 memory 窗口尝试读写外部 memory，有可能操作的是 DCACHE 里面的内容，而不是真正的外部 memory，因此，做此类调试的时候，可以考虑暂时先讲 DCACHE 功能关闭，调试访问正常后，再使能 DCACHE。

7. 小结

本文简单介绍了 ARM CM33 内核 TrustZone 架构下内核中 SAU/IDAU 如何区分安全非安全地址，存储器以及外设资源如何设置自身的安全访问属性，以及 CPU 访问不同资源时的访问规则等内容。最后列举了一些使用 STM32 进行 V8M TZ 开发中常见的 memory 配置相关的问题以及调试方法，希望对广大开发者有所帮助。

在后续的篇章中，我们还将总结一些其他的 TrustZone 开发中的常见问题及调试技巧，欢迎读者继续关注。

参考文献

文件编号	文件标题	版本号	发布日期
RM0456	Reference manual STM32U575/585 Arm®-based 32-bit MCUs	V2	September 2021
100235_0100_01_en	Arm® Cortex®-M33 Devices Generic User Guide	r1p0	19 June 2020
DDI0553B.t	Arm®v8-M Architecture Reference Manual	ID30062022	2022/Jun/30

文档中所用到的工具及版本

STM32CubeProgrammer v2.11.0

IAR Embedded Workbench 8.50.9 33462

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