STM32 TrustZone 开发调试技巧（3）—— HardFault 调试与处理

关键字：TrustZone，HardFault，SCB，AIRCR，SHCSR

引言

在 STM32 TrustZone 开发调试技巧的前两篇中，我们介绍了内核的 SAU/IDAU，地址的安全属性配置，资源的安全属性配置，内核访问资源的安全规则，以及 TrustZone 环境下外设使用的常见问题等内容。TrustZone 环境开发中还可能经常遇到的一个问题就是软件触发的故障错误。ARM CM33 内核 TrustZone 环境下的异常模型以及 Fault 的处理与不带安全扩展的情况有着很多变化，一旦出现 HardFault，经验不足的开发者可能往会找不到头绪，不知道从哪里着手寻找问题所在。因此，在这一篇的重点将围绕 CM33 TrustZone 环境下的异常模型以及 HardFault 的调试与处理展开，供开发者参考。

1. CM33 TrustZone 架构下的异常模型

在 STM32 TrustZone 开发调试技巧的第二篇中我们介绍过 CM33 带安全扩展的 S 和 NS 侧的中断以及中断向量表，这里不再赘述。表 1 总结了其中的 Fault 异常。

表1. CM33 TrustZone 架构下的 Fault 异常

Exception Number	IRQ Number	Exception Type	优先级	是否 Bank	使能
3	-13	**Secure** HardFault （SCB AIRCR.BFHFNMINS = 1）	-3	否	总是使能 (NS HardFault 是否使能取决于 SCB AIRCR.BFHFNMINS 位)
		Secure HardFault （SCB AIRCR.BFHFNMINS = 0）	-1		
		HardFault （SCB AIRCR.BFHFNMINS = 0）	-1		
4	-12	MemoryManage	可配	是	默认不使能 取决于 SCB_S / SCB_NS SHCSR.MEMFAULTENA
5	-11	BusFault	可配	否	默认不使能 取决于 SCB_S / SCB_NS SHCSR.BUSFAULTENA
6	-10	UsageFault	可配	是	默认不使能 取决于 SCB_S / SCB_NS SHCSR.USGFAULTENA
7	-9	SecureFault	可配	否	默认不使能 取决于 SCB_S SHCSR.SECUREFAULTENA

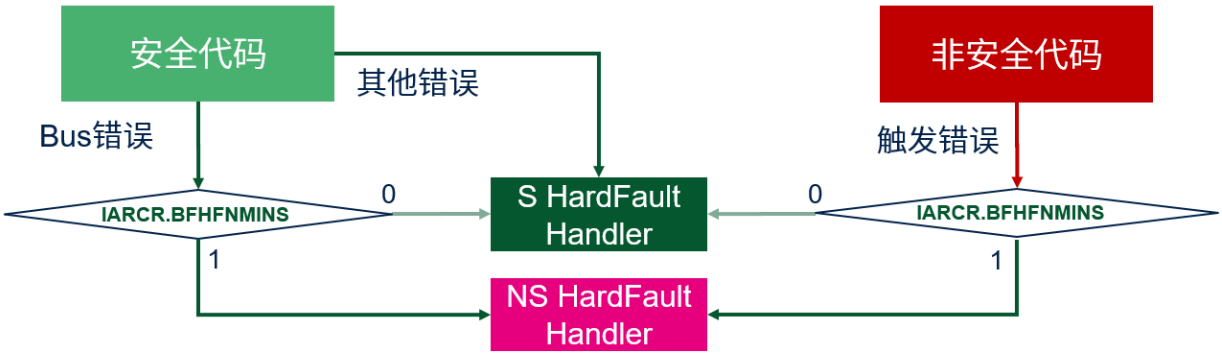
1.1. Fault 异常类型（带安全扩展）

1.1.1. Hard Fault

HardFault 是默认的 Fault 异常，总是使能。触发的原因可能是由于异常处理本身触发了错误，或者某个异常无法被其他机制处理而上升到 HardFault。它的优先级高于所有其他可配置优先级的异常。

在 TrustZone 环境中，HardFault 不是 Bank 的。同一个异常，要么触发 S 侧的 HardFault，要么触发 NS 侧的 HardFault。SCB 的 AIRCR.BFHFNMINS 决定了是否使能 NS 的 BusFault，HardFault 和 NMI。如果 SCB 的 AIRCR.BFHFNMINS=0，HardFault 总是触发 S 侧的 HardFault Hanlder；如果 AIRCR.BFHFNMINS=1，则故障可能触发 NS 侧的 HardFault Handler，也可能触发 S 侧的 HardFault Handler。图 1 给出了在其他 Fault 未使能情况下，HardFault Handler 触发一般情形。

图1. HardFault Handler 的触发



需要注意的是，即使 AIRCR.BFHFNMINS=1，原本 target 到 S 侧并且上升为 HardFault 的异常，将依旧触发 S 侧的 HardFault，他们并不受到 AIRCR.BFHFNMINS 位的影响，例如当安全代码违反 MPU 保护规则，产生 MemManage 错误的时候，即使 AIRCR.BFHFNMINS=1，故障还是会进入 Secure HardFault Handler。而 NS 侧的 HardFault，只有当 AIRCR.BFHFNMINS=1 时才有可能会被触发。

另外还要注意一点，AIRCR 寄存器不能直接修改，需要先写 Key 值才能更改寄存器内容。置位或清除 AIRCR.BFHFNMINS bit 的示例代码如下（只能在安全代码中使用）：

```
void SECURE_SetNMIHFBFTarget(int NS)
{
    uint32_t reg_value;
    uint32_t target = (NS==1)?1:0;

    /* read old register configuration    */
    reg_value  =  SCB->AIRCR;
    /* clear bits to change */
    reg_value &= ~((uint32_t)(SCB_AIRCR_VECTKEY_Msk | SCB_AIRCR_BFHFNMINS_Msk));
    /* insert write key and target bit */
    reg_value  =  (reg_value                                   |
                  ((uint32_t)0x5FAUL << SCB_AIRCR_VECTKEY_Pos) |
                  (target << SCB_AIRCR_BFHFNMINS_Pos)  );
    SCB->AIRCR =  reg_value;
}
```

注意：有的时候，软件可能需要设置 AIRCR.PRIS 位，来整体降低 NS 中断的优先级（例如在 TF-M 的实现中就使用这个机制）。这时候，如果同时设置 AIRCR.PRIS=1，

AIRCR.BFHFNMINS=1，内核的行为将不可预测。因此如果需要设置 AIRCR.PRIS=1，则建议保持 AIRCR.BFHFNMINS=0。

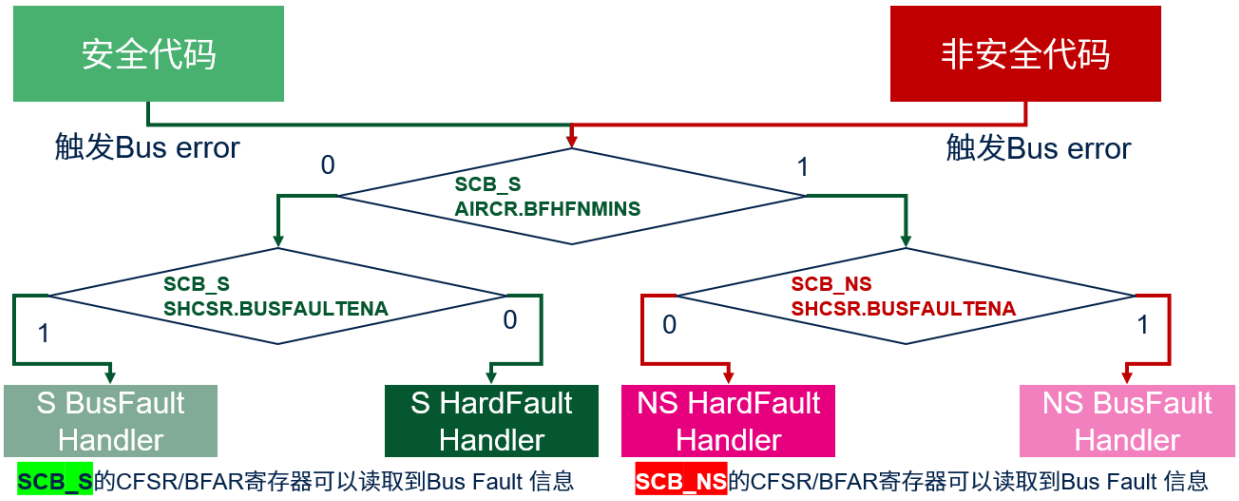
1.1.2. Bus Fault

Bus Fault 通常发生在指令或数据访问时候，可能由于检测到 memory 系统的总线错误而导致。Bus Fault 默认不使能，就是说总线故障默认将触发 HardFault Handler。如果需要单独使能 Bus Fault，可以将 SCB 的 SHCSR.BUSFAULTENA 位设 1。

在 TrustZone 环境中，Bus Fault 也不是 Bank 的。触发 S 还是 NS 侧的 BusFault Handler 与 SCB 的 AIRCR.BFHFNMINS 有关。如果 AIRCR.BFHFNMINS=0，BusFault 总是 target 到 S 安全状态；反之如果 AIRCR.BFHFNMINS=1，则 target 到 NS 非安全状态。

产生 Bus 错误时，实际会触发哪个 Fault Handler，将取决于 AIRCR.BFHFNMINS 和 SCB_S/NS 的 SHCSR.BUSFAULTENA 的设置。图 2 给出了 Bus 错误触发 Fault Handler 的一般情况（例如这里不考虑安全侧 Vector 错误依旧上升到 Secure HardFault 的情况）。

图2. Bus 错误触发 Fault Hanlder



通常情况下，SCB 的 CFSR/BFSR 和 BFAR 寄存器中会标记总线故障信息。在 TrustZone 环境中，SCB 的某些寄存器以及寄存器的某些比特位是 Bank 的。从安全侧和非安全侧都能够看到各自的 SCB 寄存器，但是 CFSR 寄存器的 BFSR 域以及 BFAR 寄存器并不是 Bank 的。而 Bus 故障可能 target 到 S 安全侧也可能 target 到 NS 非安全侧，当发生总线错误的时候，如果分别从 SCB_S、SCB_NS 的相关寄存器中读取 Bus Fault 的信息，可以看到不同的结果。

如果 AIRCR.BFHFNMINS=0，只有安全侧可以看到 BFSR 和 BFAR 的真实数据，非安全侧读取 BFSR、BFAR，或者从安全侧读取 BFSR_NS，BFAR_NS 都只能读到全 0 的值。

如果 AIRCR.BFHFNMINS=1，BFAR_NS 和 BFAR_S 的值一般会读取到相同的值。通常，代码需要处理 BusFault 时，如果使用默认配置，即保持 BusFault target 到 S 侧，AIRCR.BFHFNMINS=0，则 Fault Handler 可以从 SCB_S 的 CFSR.BFSR 和 BFAR 寄存器获取总线故障信息；而如果设置了 AIRCR.BFHFNMINS=1，那么发生 Bus error 的时候，非安全侧的 Fault Handler 可以直接从 SCB_NS 的 CFSR.BFSR 和 BFAR 寄存器获取故障信息。

BusFault 默认没有单独使能，如果需要使能 BusFault，可以将 SHCSR 寄存器的 BUSFAULTENA 位置位。使能或禁止 BusFault 的示例代码如下：

```
void EnableBusFault(int enable)
{
  if( enable == 1)
    SCB->SHCSR |= SCB_SHCSR_BUSFAULTENA_Msk;
  else
    SCB->SHCSR &= (~SCB_SHCSR_BUSFAULTENA_Msk);
}
```

这段代码对安全和非安全侧都是一样的，但是要注意，由于 BusFault 不是 Bank 的，当 AIRCR.BFHFNMINS=0 时，这段代码只能在安全侧使用，也就是修改的是 S 安全侧 SCB SHCSR 的 BusFault，此时写 SCB_NS 的 SHCSR.BUSFAULTENA 位无效。

如果非安全侧应用使用这段代码使能 BusFault，那么前提是安全侧已经设置了 AIRCR.BFHFNMINS=1。

1.1.3. Usage Fault

UsageFault 与指令执行时候的错误有关，包括未定义的指令、非对齐访问、执行指令时的无效状态、中断返回时的错误、除 0 等。

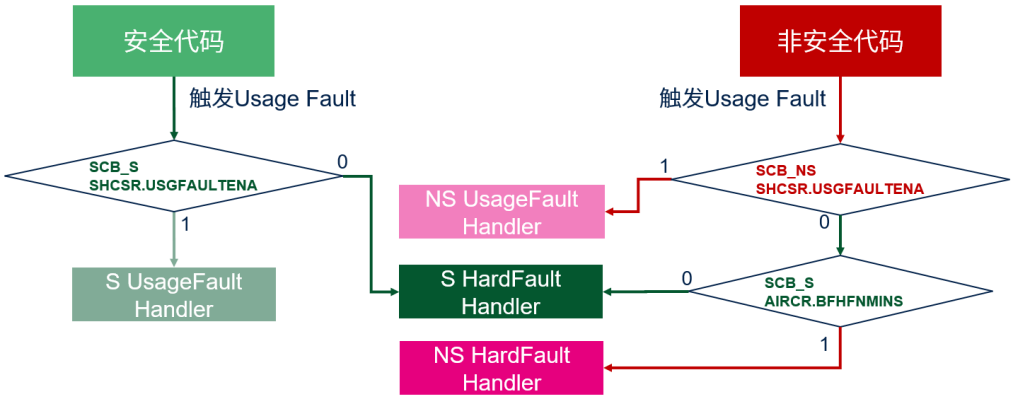
在 TrustZone 环境中，UsageFault 是 Bank 的，因此在 S 和 NS 状态可能产生各自的 UsageFault，并且可能触发各自的 S UsageFault Handler 和 NS UsageFault Handler。UsageFault 默认不使能，因而缺省会上升到 HardFault，是否触发 S 还是 NS 的 HardFault Handler 还要取决于 AIRCR.BFHFNMINS 的值是 0 还是 1。

使能 UsageFault 需要分别设置 SCB_S 和 SCB_NS 的 SHCSR.USGFAULTENA。SCB_S 的 SHCSR.USGFAULTENA=1 用于使能 S 安全侧的 Usage Fault；SCB_NS 的 SHCSR.USGFAULTENA=1 用于使能 NS 非安全侧的 Usage Fault。

另外，通常除 0 操作不会触发 UsageFault，如果希望除 0 操作触发 UsageFault，需要将 SCB_S/NS 对应的 CCR.DIV_0_TRP 比特置 1。

图 3 总结了 Usage 错误触发 Fault Handler 的一般情况。

图3. Usage 错误触发 Fault Handler



只要 SHCSR.USGFAULTENA=1，UsageFault 总是触发软件对应安全状态的 UsageFault Handler，否则上升到 HardFault，安全侧的 UsageFault 总是上升到 Secure HardFault。对于

非安全侧的 UsageFault，如果 AIRCR.BFHFNMINS=0，则上升到 Secure HardFault，否则上升到 Non-Secure HardFault。

使能或禁止 UsageFault 的示例代码如下：

```
void EnableUsageFault(int enable)
{
  if( enable == 1)
  {
    SCB->SHCSR |= SCB_SHCSR_USGFAULTENA_Msk;

    /* Enable divide by 0 trap to trigger usage fault (if necessary) */
    SCB->CCR |= SCB_CCR_DIV_0_TRP_Msk;
  }
  else
    SCB->SHCSR &= (~SCB_SHCSR_USGFAULTENA_Msk);
}
```

如果安全和非安全侧都需要使能 UsageFault，则 S、NS 代码可以分别调用这段代码使能各自的 UsageFault，或者 S 安全侧代码也可以直接控制 NS 非安全侧 UsageFault 的使能，例如可以在 S 安全侧增加下面这段代码来决定 NS 侧的 UsageFault 是否使能。

```
void EnableNSUsageFault(int enable)
{
  if( enable == 1)
    SCB_NS->SHCSR |= SCB_SHCSR_USGFAULTENA_Msk;
  else
    SCB_NS->SHCSR &= (~SCB_SHCSR_USGFAULTENA_Msk);
}
```

1.1.4. MemManage Fault

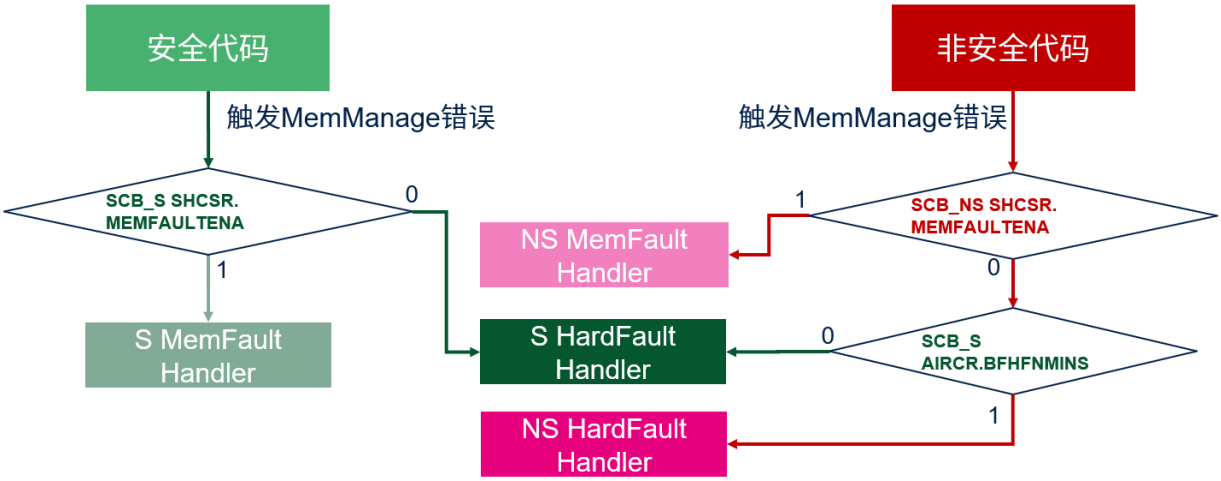
MemManage Fault 是由于 Memory 保护产生的故障异常，例如在取指令或进行数据访问时违反了 MPU region 定义的访问规则，或者违反了默认地址保护规则。

MemManageFault 与 UsageFault 类似，也是默认不使能，如果希望使能 S 或者 NS 侧的 MemManageFault，需要相应将 SCB_S 或者 SCB_NS 的 SHCSR.MEMFAULTENA 比特置位。

另外，也与 UsageFault 类似，MemManageFault 在 S 和 NS 侧也是 Bank 的，也就是 S、NS 有各自的 MemManageFault。由于 MPU 单元本身是 Bank 的，系统中有两套 MPU 寄存器 MPU_S 和 MPU_NS，因而代码在 S 和 NS 侧可以各自定义自己的 MPU region 并使用不同配置，也就是说即使对相同的地址，S/NS 两侧也可以通过各自的 MPU 单元定义不同的访问规则。MPU_S 配置的保护规则只应用于 S 安全侧代码，即控制 CPU 处于安全状态时候的访问，这与 CPU 访问的地址的在 SAU 中定义安全属性无关。而 MPU_NS 配置的保护规则只应用于 NS 非安全侧代码，即 CPU 处于非安全状态时候的访问，二者互不影响。

图 4 给出了 MemManage 故障触发 Fault Handler 的一般情况。如果 S 安全代码违反 memory 访问规则，可能会触发安全侧的 MemManageFault，或者 Secure HardFault。非安全代码违反 memory 访问规则，可能会触发非安全侧的 MemManageFault，或者上升到 HardFault，如果 AIRCR.BFHFNMINS=0 上升到 Secure HardFault，否则上升到 Non-Secure HardFault。

图4. MemManage 错误触发 Fault Handler



使能或禁止 MemManage Fault 的示例代码如下：

```
void EnableMemoryFault(int enable)
{
  if( enable == 1)
    SCB->SHCSR |= SCB_SHCSR_MEMFAULTENA_Msk;
  else
    SCB->SHCSR &= (~SCB_SHCSR_MEMFAULTENA_Msk);
}
```

如果安全和非安全侧都需要使能 MemManage Fault，则 S、NS 应用可以分别调用这段代码使能各自的 MemManage Fault，或者 S 安全侧代码也可以直接使能 NS 非安全侧的 MemManage Fault，例如可以在 S 安全侧增加下面这段代码来控制 NS 侧的 MemManage Fault 使能。

```
void EnableNSMemoryFault (int enable)
{
  if( enable == 1)
    SCB_NS->SHCSR |= SCB_SHCSR_MEMFAULTENA_Msk;
  else
    SCB_NS->SHCSR &= (~SCB_SHCSR_MEMFAULTENA_Msk);
}
```

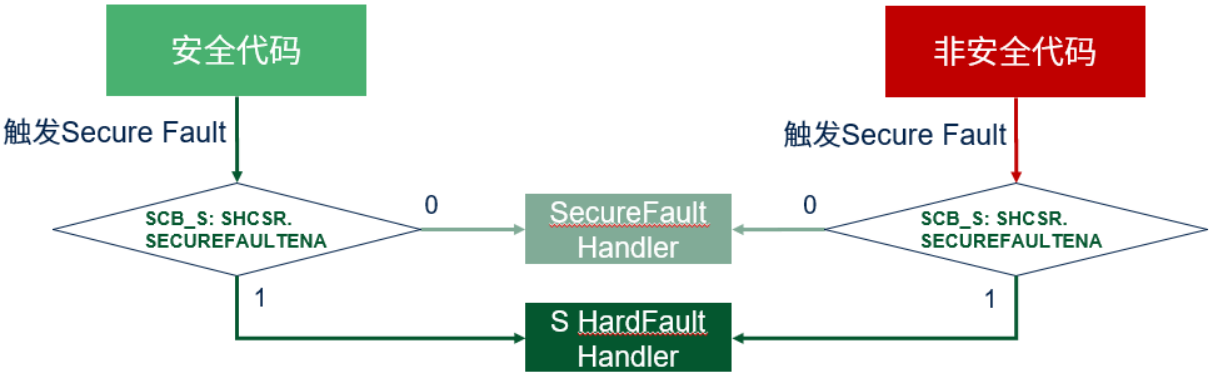
另外，如果代码使用 HAL API 使能 MPU，即调用 HAL_MPU_Enable()，那么 MemManage Fault 在 MPU 使能的函数中会自动被使能，这时候无需额外调用前面提到的代码去单独使能 MemManage Fault。

1.1.5. Secure Fault

Secure Fault 只有在 TrustZone 使能的环境下才存在。SecureFault 可能由于内核中各种各样的安全检查而触发，例如从 NS 跳转至 S 代码时没有从 SG 入口指令进入，或者非安全代码试图访问 SAU/IDAU 规定的安全地址范围等。通常当出现 SecureFault 时，软件的处理可以是直接停止或者复位系统，这样做可以尽可能地避免引入安全漏洞。

SecureFault 不是 Bank 的，总是 target 到 S 侧，因此只有安全代码能够处理 SecureFault。SecureFault 缺省也没有使能，出现 Secure 错误时，默认触发 Secure HardFault。软件可以通过置位 SHCSR.SECUREFAULTENA 来单独使能 SecureFault，使能后 Secure 错误将触发 SecureFault Handler。图 5 给出了 Secure 错误触发 Fault Handler 的一般情况。

图5. Secure 错误触发 Fault Handler



使能或禁止 SecureFault 的示例代码如下：

```
void EnableSecureFault(int enable)
{
  if( enable == 1)
    SCB->SHCSR |= SCB_SHCSR_SECUREFAULTENA_Msk;
  else
    SCB->SHCSR &= (~SCB_SHCSR_SECUREFAULTENA_Msk);
}
```

1.2. 故障升级与 HardFault

除了 HardFault 以外，其他故障类型都具有可配置的优先级。软件可以禁止某个可配优先级的故障异常，但是不能禁止 HardFault。故障异常的优先级和对应的 mask bit 决定了内核是否会进入某个故障的处理程序，以及某个故障是否可以抢占另一个故障。

某些情况下，可配置优先级的故障可能会被当成 HardFault 处理，也就是故障升级或称中断上访，此时，这个具体的 Fault 会升级为 HardFault 故障。

某个 Fault 升级成 HardFault 可能有多种原因，例如：

- 该故障 Fault 没有使能
  - 例如，代码由于执行未定义的指令产生了 UsageFault，但是 UsageFault 没有被使能
- 该故障的 FaultHandler 优先级不够高无法运行
  - 例如，系统配置并使能了 MPU，CPU 正在执行某个中断操作，该操作试图进行地址访问时违反了 MPU 定义的访问规则，进而触发了故障，但是当前执行的中断的优先级高于 MemManage 故障的优先级
- 在故障的 FaultHandler 中产生了同样的故障
  - 例如，在处理 UsageFault 的 handler 中又发生了未定义指令的情况

如果在进入 BusFault Handler 的时候，压栈操作又导致了 BusFault，这种情况下 BusFault 不会升级到 HardFault。这意味着，如果损坏的堆栈导致故障，即使 Fault Handler 压栈失败，故障处理程序还是会执行，但堆栈内容已损坏。

只有 NMI 可以抢占 HardFault，HardFault 可以抢占任何除 Reset、NMI 或者另一个 HardFault 以外的异常。当 BFHFNMINS=1 时，如果 NS 侧的 NMI Handler 产生了安全违规错误，那么它将触发 Secure HardFault，并被其抢占。

在获取异常向量的时候发生的 Bus 错误，总是升级到 HardFault，由 HardFault 处理，而非 BusFault。

1.3. Fault 异常的安全状态

在 TrustZone 使能的环境中，故障异常可能 target 到 S 安全状态或 NS 非安全状态，这会导致 ARMv8-M 内核的行为与以往 ARMv6-M 及 ARMv7-M 内核有很大不同，TrustZone 环境软件开发中对 Fault 的处理要特别注意到这一点。

关于 Fault 异常 target 到 S、NS 的情况在前文中介绍几个 Fault 类型的时候已经有提到，这里在表 2 中再稍加总结。

表2. Fault 异常的安全状态目标

Fault 异常	目标状态
HardFault	默认以 S 安全态为目标 安全代码可以通过置位 BFHFNMINS，将其配置为 NS HardFault。但是原本 Target 到 S 安全状态的 Fault 异常依旧会触发 S HardFault
BusFault	默认以 S 安全态为目标 安全代码可以通过置位 BFHFNMINS，将总线故障导向 NS 非安全侧
UsageFault	目标可以是 S 安全态或者 NS 非安全态，取决于产生错误时内核的安全状态（即 CPU 在执行安全还是非安全代码） S/NS 侧可以各自使能其 UsageFault
MemManageFault	目标可以是 S 安全态或者 NS 非安全态，取决于产生错误时内核的安全状态（即 CPU 在执行安全还是非安全代码） S/NS 侧可以各自使能其 MemManageFault
SecureFault	总是以 S 安全态为目标，无论安全代码还是非安全代码都有可能触发 Secure Fault 只有安全态下才能看到这个异常，因此只有安全代码能够处理 SecureFault

1.4. 异常的进入与返回

1.4.1. 异常的进入与 stack frame

当处理器处于线程模式且系统存在具有足够优先级的 pending 异常时则会进入异常，或者新异常的优先级高于正在处理的异常，这时候新异常将抢占原始异常，即出现异常嵌套。

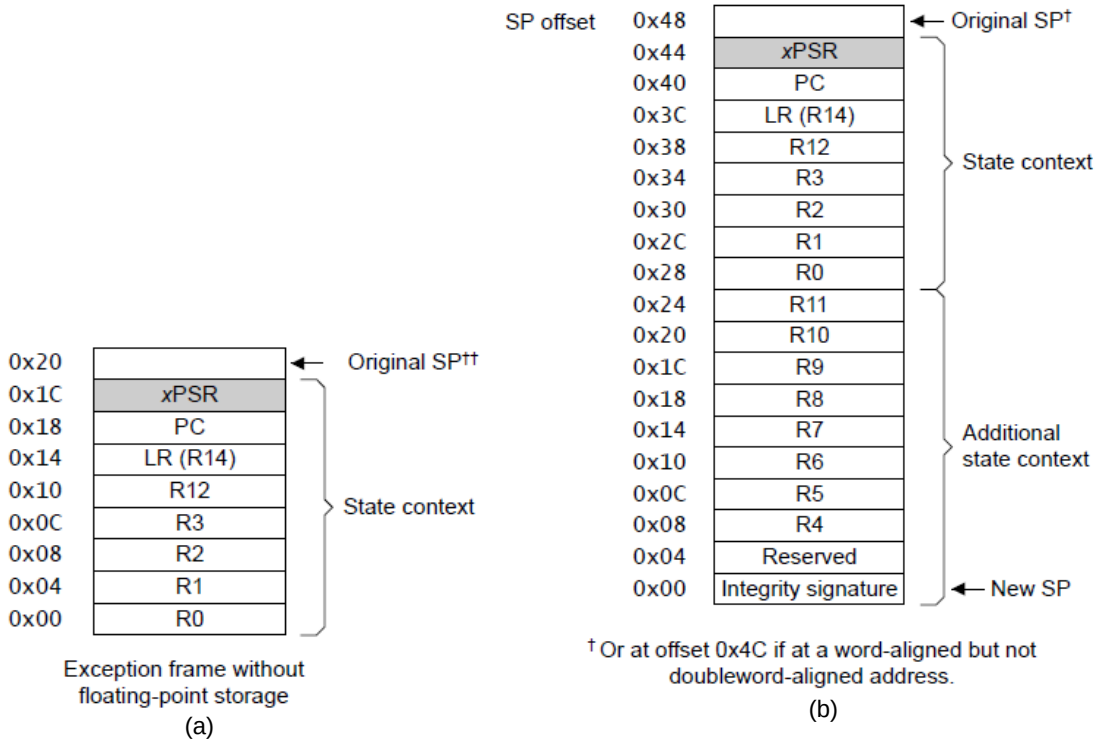
当处理器发生异常时，除非该异常是尾链异常或延迟到达的异常，否则处理器会将上下文信息压入堆栈，压栈的数据结构即 stack frame。

通常 stack frame 的内容如图 6 (a) 所示，包含了 R0 到 R3、R12、LR、PC 和 xPSR 的内容。在 TrustZone 使能的环境中，如果 S 安全代码执行被 NS 非安全异常抢占，那么进入非安全异常前会有更多的信息压栈，如图 6 (b)所示，并且硬件会自动将压栈的寄存器清零，防止任何安全状态下的数据暴露给非安全代码。

如果使用了浮点功能，存在浮点上下文，那么内核也会自动将浮点相关的上下文内容压栈。由于浮点部分数据内容对我们通常的 Fault 调试没有太多帮助，这里不做赘述。

进入异常前的 stack frame 压栈操作使用 MSP 还是 PSP，取决于当时内核的运行状态及其使用的堆栈。如果当时 CPU 运行于 Handler 模式，则使用 MSP 压栈 stack frame；如果 CPU 运行于 Thread 模式，由当时的 CPU CONTROL.SPSEL 位来标记使用的堆栈。

图6. Stack frame



1.4.2. 异常的返回与 EXC_RETURN

Stack frame 中压栈了进入异常前的 PC 值，以便异常返回后从之前的 PC 地址继续执行。同时，内核从 vector table 中获取异常处理函数的地址。当压栈完成之后，内核将开始执行异常处理函数。与此同时，内核将 EXC_RETURN 值写入 LR 寄存器，这个值将在异常处理结束后触发异常返回操作。当内核检测到加载到 PC 的值符合 EXC_RETURN 的模式时，会知道该操作不是正常的分支操作，而是异常处理已经完成，进而启动异常返回流程。

EXC_RETURN 值的[6:0]位标记了返回的堆栈、处理器模式、安全状态和 stack frame 等，带安全扩展的 EXC_RETURN 与不带安全扩展的 EXC_RETURN 相比，增加了 bit6、bit5 和 bit0 的内容，如表 3 所示。

表3. 带安全扩展的 EXC_RETURN 内容

Bit 位	名称	功能
[31:24]	PREFIX	标志 EXC_RETURN，0b11111111
[23 :7]	-	保留位
[6]	S	表示压栈使用 S 还是 NS stack 0：NS stack 1：S stack
[5]	DCRS	指示是否应用默认压栈规则，或者被调用者寄存器是否已在堆栈中 0：跳过压栈操作 1：遵循上下文寄存器压栈的默认规则
[4]	FType	在具有主扩展和浮点扩展的运行环境中： 0：为 FP 上下文在堆栈上分配的空间。 1：未在堆栈上为 FP 上下文分配空间。 在没有浮点扩展的运行环境中，此位是保留保留位
[3]	Mode	指示压栈前的运行模式

		0：Handler 模式 1：Thread 模式
[2]	SPSEL	指示与异常的安全状态相关联的 CONTROL.SPSEL 位的临时值 0：主堆栈指针 MSP 1：进程堆栈指针 PSP
[1]	-	保留位
[0]	ES	指示异常进入的安全状态（即触发 S 还是 NS 侧的异常） 0：非安全 1：安全

说明：

- 关于 bit[2] SPSEL
  SPSEL 位所指示的 MPS/PSP 并不一定标志了触发故障的代码进入故障前使用的堆栈是 MSP 还是 PSP，因为 bit[0] ES 的值决定了这个值来自于 CONTROL_S 还是 CONTROL_NS。也就是说只要异常触发到安全侧，比如默认的 Secure HardFault，那么就一定有 ES=1，此时 SPSEL 位反应的一定是安全侧 CONTROL_S.SPSEL 的状态。而实际触发故障的代码可能来自 S 安全侧，也可能来自 NS 非安全侧，那么在进行故障分析的时候应当首先根据 bit[6]确定故障来源于 S 安全侧还是 NS 非安全侧，进而根据 CONTROL_S.SPEL 或者 CONTROL_NS.SPEC 确定 stack frame 被压栈到了 S 安全或者 NS 非安全侧的 MSP 还是 PSP。
- 关于 bit[5] DCRS
  DCRS 位指示了是否有额外的上下文信息压栈
  DCRS=1：表示使用了标准的压栈数据，没有额外上下文，即类似图 6(a)的 stack frame 数据结构；
  DCRS=0：表示 stack frame 中有额外的上下文压栈，即类似图 6(b)的数据结构

2. Fault 处理

2.1. Fault 出错信息

软件的故障处理程序对 Fault 进行处理时，可以通过读取 Fault 相关寄存器获取 Fault 触发原因的一些相关信息，得知错误触发的具体异常类型。例如 SCB（System Control Block）单元中的 HSFR、CFSR 寄存器，以及 Security Attribution Unit（SAU）单元的 SFSR 寄存器等。CFSR 包含多个位域，分别对应 MemManageFault（MFSR）、BusFault（BFSR）和 UsageFault（UFSR）。具体的错误类型及对应的故障状态寄存器见表 4。

表4. Fault 状态信息

Fault	Handler	Bit 名称	Fault 状态寄存器
Bus error on a vector read	HardFault	VECTTBL	HFSR
Fault escalated to a hard fault		FORCED	
MPU or default memory map mismatch:	MemManage	-	
On instruction access		IACCVIOL	CFSR.MFSR
On data access		DACCVIOL	
During exception stacking		MSTKERR	
During exception unstacking		MUNSKERR	

During lazy floating-point state preservation		MLSPERR	
Bus error:	BusFault	-	
During exception stacking		STKERR	CFSR.BFSR
During exception unstacking		UNSTKERR	
During instruction prefetch		IBUSERR	
During lazy floating-point state preservation		LSPERR	
Precise data bus error		PRECISERR	
Imprecise data bus error		IMPRECISERR	
Attempt to access a coprocessor	UsageFault	NOCP	CFSR.UFSR
Undefined instruction		UNDEFINSTR	
Attempt to enter an invalid instruction set state		INVSTATE	
Invalid EXC_RETURN value		INVPC	
Illegal unaligned load or store		UNALIGNED	
Stack overflow flag		STKOF	
Divide By 0		DIVBYZERO	
Lazy state error flag	SecureFault	LSERR	SFSR
Lazy state preservation error flag		LSPERR	
Invalid transition flag		INVTRAN	
Attribution unit violation flag		AUVIOL	
Invalid exception return flag		INVER	
Invalid integrity signature flag		INVIS	
Invalid entry point		INVEP	

对于 Bus Faut，MemManage 和 SecureFault，除了有其对应的状态寄存器以外，还有相关的错误地址寄存器。当这些 Fault 状态寄存器中的故障地址有效 bit 位（BFSR.BFARVALID，MMFSR.MMARVALID，SFSR.SFARVALID）被置位时，还有额外的 BSAR、MMFAR、SFAR 故障地址寄存器可以参考，他们能够进一步提供触发 Fault 的 Memory 地址信息，帮助软件或者开发者查找出错的位置和原因。

2.1. 代码中处理 Fault

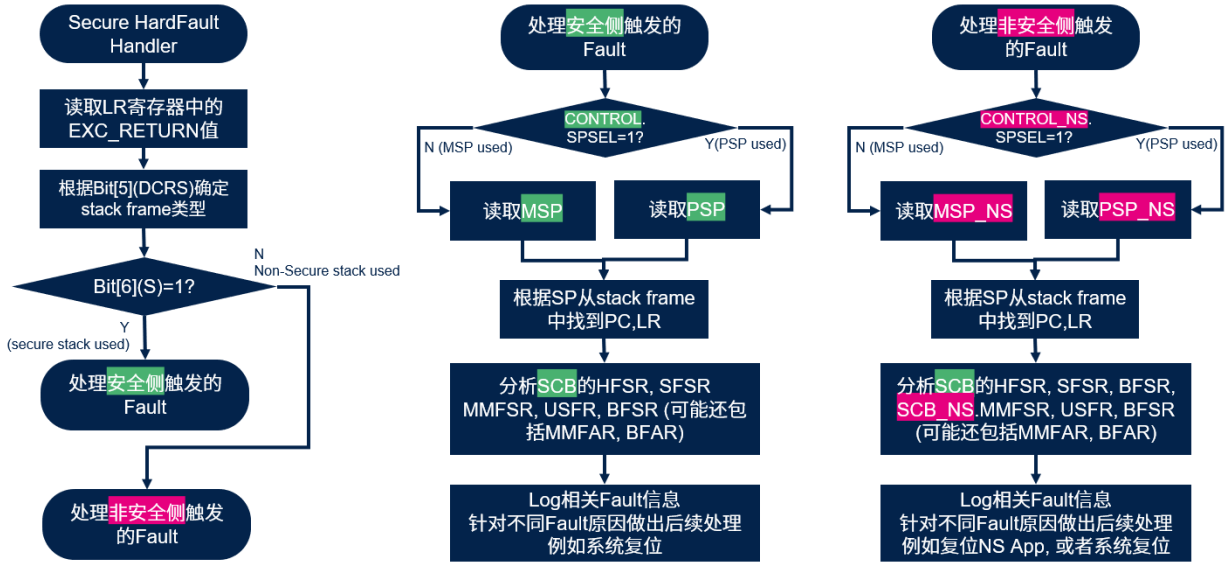
Fault 产生的原因可能来自 S 安全侧，也可能来自 NS 非安全侧，S/NS 代码中也可以有各自的 Fault 处理函数。但是并不是 S/NS 侧的代码就一定触发对应 S /NS 侧的 Fault Handler，关于这一点，我们在前文 2.1 章节的 Fault 类型中有提到过。接下来我们分别讨论两种情况：一种是所有的故障都在 S 安全侧的故障处理程序中处理，这也是默认配置下对 Fault 的处理方式；另一种则是让 S 非安全侧的代码也能够对某些故障进行处理。

2.1.1. S 安全代码处理 Fault

在使能了安全扩展支持 TrustZone 的 CM33 运行环境中，系统默认只有 Secure HardFault 是使能的，这时候安全侧的 HardFault Handler 有能力处理所有的 Fault，因为从安全侧可以看到安全或者非安全侧的 Fault 寄存器，比如 SCB_S，SCB_NS，SAU 单元中的相关寄存器，以及其他一些 Bank 的内核寄存器，例如 MSP_S/PSP_S，MSP_NS/PSP_NS，CONTRO_S，CONTROL_NS 等。所有的 Fault 都在 S 代码的 Secure HardFault Handler 中进行处理是 TrustZone 架构下软件处理 Fault 的一种比较简介而直接的实现方式。

由于错误触发可能来源于 S 或者 NS 软件，因此 Secure HardFault 处理中需要首先知道出错代码的安全状态，进而查看相关的 Fault 状态寄存器和其他 CPU 寄存器等。Secure Fault 处理的一般的流程见图 7。

图7. Secure HardFault 处理的一般流程



如果 S 代码希望有不同的函数单独处理某些 Fuault，则需要在初始化时使能相关的 Fault，例如置位 SHCSR 寄存器的 SECUREFAULTENA、USGFAULTENA、BUSFAULTENA、MEMFAULTENA 位，这部分内容在 2.1 章节有提到，这里不再赘述。

如果单独使能某些 Fault，那么这些 Fault 将具有可配置的优先级。通常对应 Fault 发生时将进入相应的 Fault 处理程序，但是在某些时候，例如 Fault 优先级低于当前执行的异常时，还是有可能触发 HardFault，进入 HardFault 处理程序，所以 HardFault Handler 还是需要做相应处理。

2.1.2. NS 代码处理 Fault

需要 NS 代码处理故障的需求可能来自多方面，例如在某些项目开发中，可能需要非安全侧也能够处理自己产生的一些故障，或者有时候非安全侧需要负责记录故障信息。

有些时候 TrustZone 环境下的项目并不是从头开发的，而是基于已有的一些应用项目，将应用从非 TrustZone 环境迁移到 TrustZone 环境下，而原有的传统项目中已经有对 Fault 进行处理的软件实现。这种场景中通常是将某个传统架构下的 STM32 MCU 的已有应用程序集成到 TrustZone 架构中，作为运行在非安全侧的应用。这时候开发者往往会发现原来应用中的 HardFault 处理程序不起作用了，无论系统发生怎样的故障，都无法触发 NS App 代码中的 HardFault Handler。这是因为系统默认所有的 Fault 都会 target 到安全侧。因此，在 TrustZone 架构中，如果希望非安全侧中运行的应用程序依旧能够处理 Fault，则需要进行一定的修改。通常有几种方法可以参考。

方法一：使能某个具体的 Fault

对于 MemManageFault 和 UsageFault，由于他们都是 bank 的，应用程序可以直接使能 NS 侧的故障，将 SCB 的 SHCSR 寄存器中对应的 bit 位 MEMFAULTENA 和 USGFAULTENA 置位，这样当 NS 代码中出现相关错误时，将进直接入 NS 代码的 MemManageFault 和 UsageFault 处理程序，进而可以在其中对相关错误进行处理。

方法二：使能非安全 Hard Fault

如果 NS 代码还希望能够处理 BusFault，或者直接处理 HardFault，则需要在 S 安全代码中配置 SCB 的 AIRCR.BFHFNMINS=1，让系统有机会进入 NS HardFault Handler。但是要注意 AIRCR.BFHFNMI=1 与 AIRCR.PRIS=1 不能同时设置。另外这样配置不代表 Secure HardFault 不需要做任何处理，因为即使配置了 AIRCR.BFHFNMI=1，原来会 target 到 S 的 Fault（例如 S 代码违反 MPU 保护规则，或者 S 代码产生 UsageFault 等）依旧会触发 Secure HardFault。

方法三：使用 Callback 回调函数

这种方式将保持 SCB 的 AIRCR.BFHFNMINS=0，也就是所有的故障依旧只触发 Secure HardFault Handler，但是 NS App 可以在初始化阶段向 S 侧注册一个 HardFault 回调函数，当 Secure HardFault Handler 发现故障由 NS 侧引起时，可以调用 NS 注册的 HardFault 回调函数，让 NS 的 HardFault Handler 能够对故障做出一些处理。

这里有几点需要注意的地方：

- 当 Fault 默认触发 Secure HardFault 时，NS 侧读取 SCB 的 HFSR 将总是读到 0，因此，如果原有的 NS HardFault Handler 需要读取 HFSR，那么这部分应当改为由 Secure HardFault Handler 读取 HFSR，并将 HFSR 的值作为参数传递给回调函数。
- 类似的，由于 BusFault 也是默认 target 到 S，那么 NS 侧读取 SCB 的 BFSR 和 BFAR 也总是读到 0，因此对于 BFSR/BFAR 也需要由 Secure HardFault Handler 读取，并将读到的值作为参数传递给回调函数。
- 由于 NS 的 HardFault Handler 是作为 callback 函数被 Secure HardFault Handler 调用的，而不是直接作为异常处理句柄进入的，因此在 NS HardFault 处理回调函数中无法直接获取到 EXC_RETURN，所以如果需要处理这个数据，也应当由 Secure HardFault Handler 读取 LR，并将读到的 EXC_RETURN 值作为参数传递给回调函数。

使用第三种方法时，S 侧需要提供一个 API，让 NS 侧注册 Fault 回调函数。在 Secure HardFault Handler 中，通过 EXC_RETURN 值判断产生故障的代码的安全状态，如果是 NS，且 Fault 类型是 NS 可以处理的 Fault（比如不是 Secure Fault），则调用 NS 注册的 HardFault handler 回调函数，并将一些必要的信息作为参数传递给回调函数，参数中包含的信息可能包括：EXC_RETURN 值，BFSR，BFAR 等。

注册回调函数以及在 Secure HardFault Handler 中调用回调函数的示例代码如下：

```
/* Code on SECURE side */

#if defined ( __ICCARM__ )
#  define CMSE_NS_CALL  __cmse_nonsecure_call
#  define CMSE_NS_ENTRY __cmse_nonsecure_entry
#else
```

```
#  define CMSE_NS_CALL  __attribute((cmse_nonsecure_call))
#  define CMSE_NS_ENTRY __attribute((cmse_nonsecure_entry))
#endif

#if defined ( __ICCARM__ )
typedef void (CMSE_NS_CALL *funcptr)(void);
typedef void (CMSE_NS_CALL *hffuncptr)(uint32_t lr, uint32_t *hfsr, uint32_t *bfsr,
uint32_t bfar);
#else
typedef void CMSE_NS_CALL (*funcptr)(void);
typedef void CMSE_NS_CALL (*hffuncptr)(uint32_t lr, uint32_t *hfsr, uint32_t *bfsr,
uint32_t bfar);
#endif

typedef hffuncptr hffuncptr_NS;
void *pHardFaultCallback = NULL;   /* Pointer to hardfault callback in Non-secure */

CMSE_NS_ENTRY void SECURE_RegisterHFCallback(void *func)
{
  if(func != NULL)
  {
      pHardFaultCallback = func;

  }
}

void HardFault_Handler(void)
{
  ...

  /*
   * 读取 EXC_RETURN 值到 lrval,
   * 根据 lrval 值判断触发 Fault 的代码安全状态
   * 如果可以由 NS 侧处理(例如，非 SecureFault)，则调用 NS HardFault 回调函数
   */
  if(pHardFaultCallback != NULL)
  {
    hffuncptr_NS callback_NS; /* non-secure callback function pointer */
    callback_NS = (hffuncptr_NS)cmse_nsfptr_create(pHardFaultCallback);

    /* provide lrval(exc_return), HFSR, BFSR and BFAR info to NS HF handler */
    callback_NS(lrval, SCB->HFSR, (SCB->CFSR & SCB_CFSR_BUSFAULTSR_Msk), SCB->BFAR);
  }

  ...
  /* 其他处理 */
}
```

```
/* Code on NON-SECURE side */

void HardFault_Handler(uint32_t lrval, uint32_t HFSR, uint32_t BFSR, uint32_t BFAR)
{
  ...

  /*
   * 根据需要进行 Fault 处理
   * 例如根据 SPSEL 获取 MPS 或 PSP，从 stack frame 找到 LR, PC 等
   * log Fault 信息，...
   */
  ...
}

int main(void)
{
    HAL_Init();

    /* Register HardFault callback defined in non-secure
     * and to be called by Secure Hardfault handler
     */
    SECURE_RegisterHFCallback((void *)HardFault_Handler);

    ...
}
```

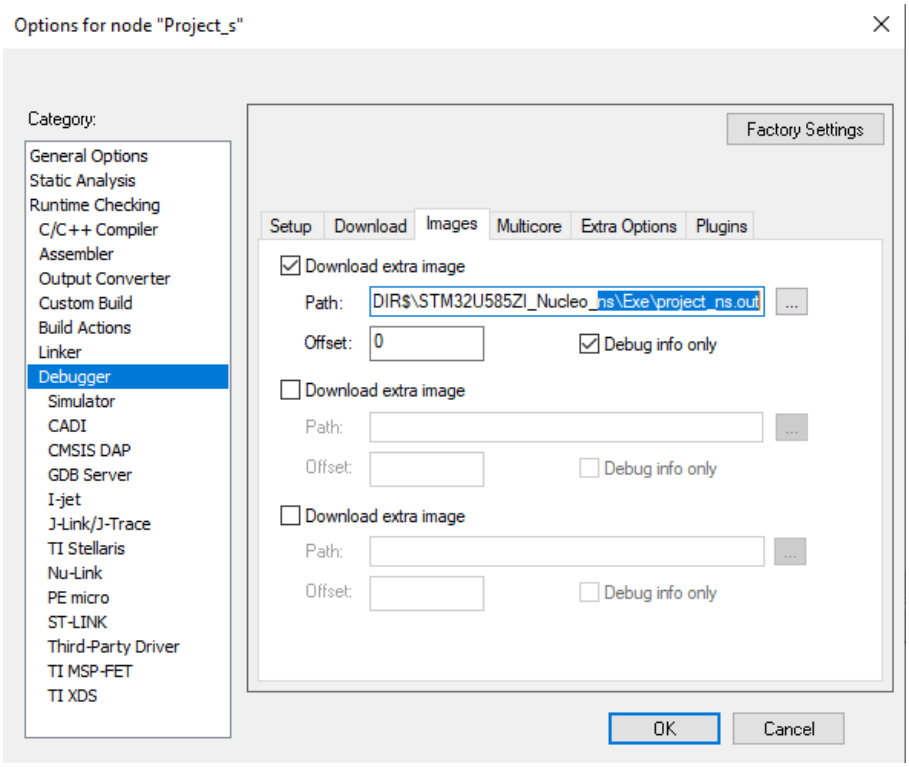
3. Fault 异常的调试

代码的 FaultHandler 处理起到的作用通常是运行期间第一时间提示故障信息以及记录故障日志等，最终采取的措施可能要么是系统复位，要么是进入死循环。但是在开发阶段如果出现 Fault 异常，更有效的方法可能是需要通过在线调试手段找到故障原因，改正代码中的错误。因此本小节我们将讨论 Fault 异常的调试方法。

2.4 节中提到的 EXC_RETURN 以及进入异常前压栈的 stack frame 的内容能够为 Fault 调试提供重要信息，本小结我们将结合一些具体的例子来看如何使用这些信息进行 Fault 异常的调试。例子中我们将以 IAR 作为调试器，并假设所有 Fault 都升级到 Secure HardFault Handler。调试过程主要涉及查看寄存器窗口、Memory 地址以及反汇编窗口的内容，这里以 IAR 为例，如果使用其他的 IDE，也可以找到类似工具窗口查看相关信息。

为了方便，我们一般以安全代码工程为入口进行调试，并且可以在安全工程中将非安全代码的调试信息也加载进来，这样，当我们分析 PC 值的时候可以比较容易地定位到触发故障的源代码。在 S 工程中添加 NS 工程调试信息的方法如图 8 所示。

图8. 在 IAR Secure 工程中添加 Non-Secure 工程的调试信息



对 Fault 进行调试，首先可以在 Secure 代码的 HardFault Handler 入口设置断点，当 Fault 发生时，调试器将停在 HardFault 函数，此时可以开始进一步调查。通常调查的过程可以分为以下几步：

- 定位触发故障的 PC
- 查找故障原因（通过 HFSR, CFSR, SFSR 等）
- 进一步分析其他相关寄存器，找到出错原因

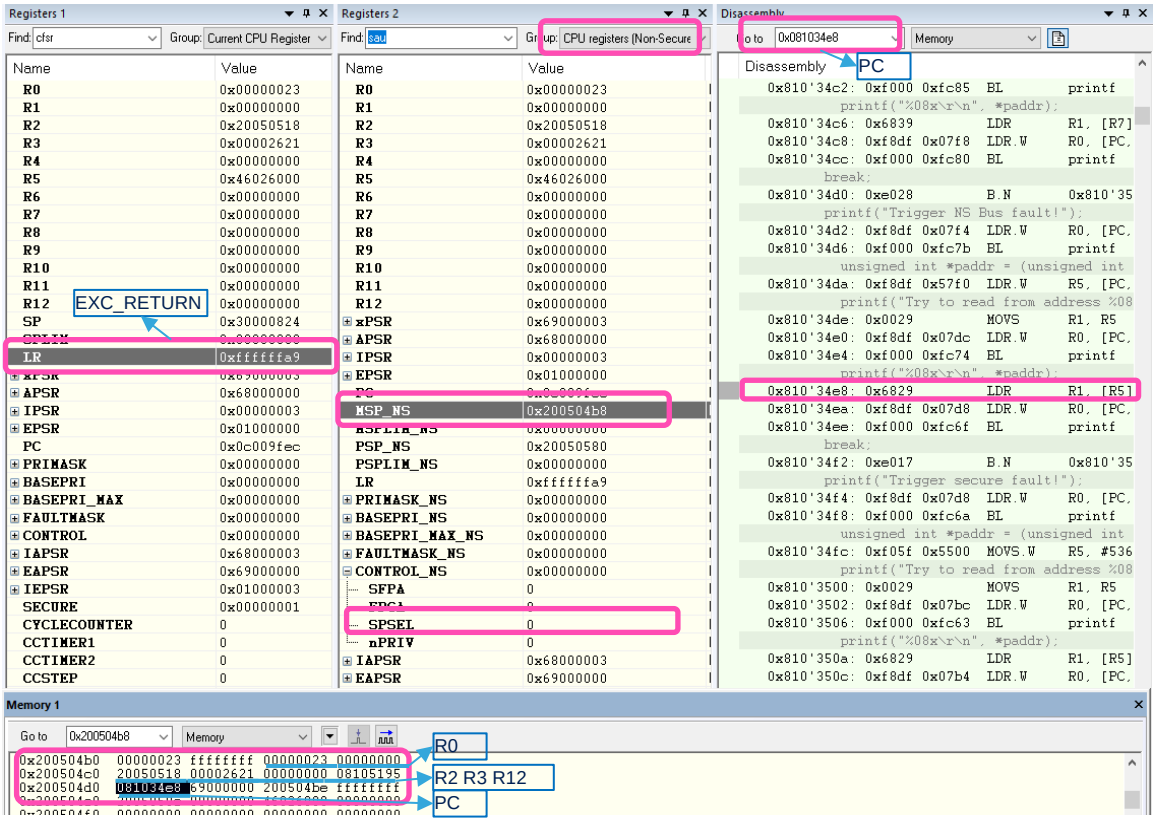
如果使用的 IDE 版本较高，某些情况下调试器可能已经能够直接分析 stack frame 并给出触发故障的函数地址 PC 指针，以及故障原因等。此时下文提到的一些步骤可能可以省去。但是调试器的分析有可能并不能够覆盖所有的情况，比如 NS 触发故障进入 Secure HardFault 等。所以掌握基本的 Fault 分析和调试方法还是会很有帮助。下面具体讲一下几个步骤的分析过程。

3.1. 定位触发故障的 PC

当断点停在 Secure HardFault Handler 时，我们首先观察 LR 寄存器，此时 LR 寄存器中是 EXC_RETURN 值。通过 EXC_RETURN 找到 PC 的过程一般有以下一些步骤：

1) 根据 EXC_RETURN 的 bit[6] S 确定进入 Fault 前压栈使用了 S 还是 NS stack，据此我们可以大概知道是 S 还是 NS 的某个指令触发了 Fault。
2) 通过 bit[5] DCRS 可以知道 stack frame 数据的类型（是否带有额外的上下文内容，但是一般触发安全侧 HardFault 时，DCRS 应当是 1）
3) 根据 Bit[6]提示的安全状态，查看 CONTROL_S.SPSEL 或者 CONTROL_NS.SPSEL 得知压栈使用了 MSP 还是 PSP
4) 找到 MSP/PSP 或者 MSP_NS/PSP_NS 中的 stack frame
5) 根据 stack frame 数据结构，找到 LR、PC

图9. IAR 中观察 CPU 寄存器、memory 和反汇编窗口定位触发故障的 PC



以图 9 的示例为例，此时 EXC_RETURN=0xfffffa9

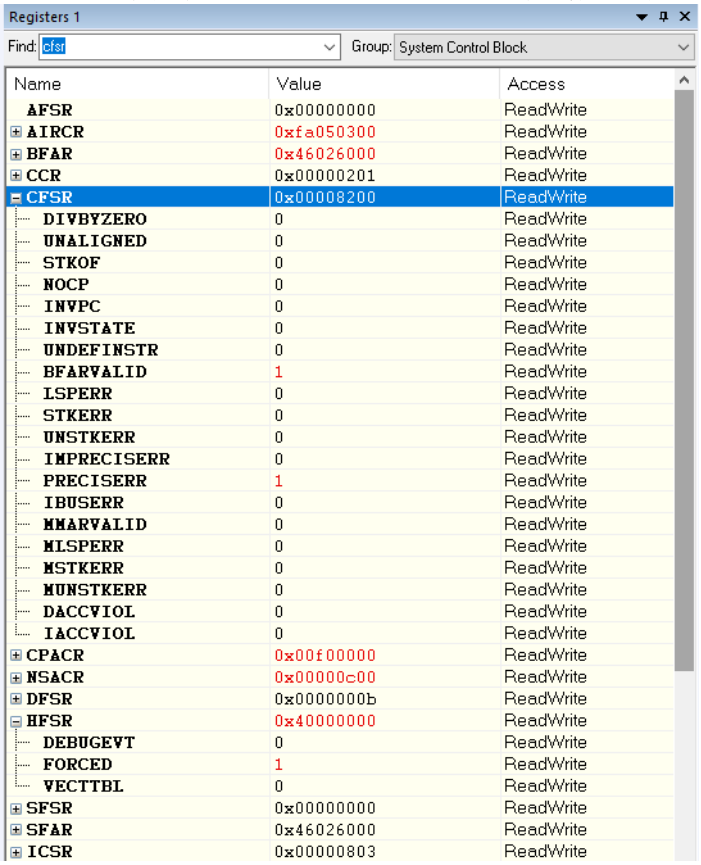
- bit[6] = 0 表示压栈使用了 NS stack，说明是 NS 代码触发了 Fault；

- bit[5] = 1 说明使用默认的 stack frame（即图 6 (a)的数据结构）
- CONTROL_NS.SPSEL = 0 说明 stack frame 在 MSP
- 在寄存器窗口选择“CPU registers [Non-Secure] ”，可以看到 MSP_NS 地址为 0x200504b8
- 在 Memory 窗口查看 0x200504b8 地址的内容，按照 stack frame 的结构可以在 0x200504b8+6*4=0x200504d0 地址找到 PC 的值为 0x081034e8
- 在 Disassembly 窗口输入地址 0x081034e8 则可以看到相关触发故障的函数和指令。

3.2. 查找故障原因

当 Fault 被触发，断点停止在 Fault 处理函数中时，我们通常可以通过观察 HFSR，CFSR 和 SFSR 寄存器的内容知道具体 Fault 的原因。在调试器的寄存器窗口中输入 CFSR，可以看到相关的几个寄存器的内容，进而分析故障原因。

图10. IAR 中观察 CFSR、HFSR、SFSR、BFAR 等寄存器示例



以图 10 给出的几个寄存器值为例，CFSR 的 BFSR 比特域指示了 BusFault（PRECISERR=1），且 BFAR 有效，此时 BFAR 指示 0x46026000，说明 NS 代码由于访问了这个地址触发了总线错误。根据这个分析，结合前一个步骤找到的 PC 位置，可以知道 printf 语句试图打印的 paddr 地址内容时触发了总线故障，因而后续可以检查 paddr 地址赋值是否正确。到此 Fault 的分析基本定位完毕。

3.3. 分析其他相关寄存器找到出错原因

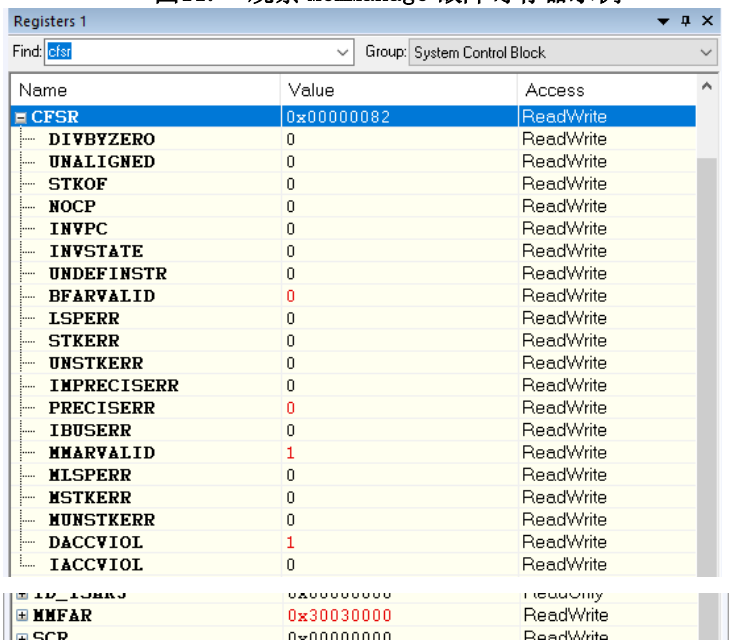
在上一个例子中，触发故障的原因是总线错误，如果在前面的步骤中观察到的 CFSR 指示的是其他故障，例如 MemManage 或者 SecureFault，那么我们可能需要进一步查看其他寄存器分析出错的原因。例如出现违反 MPU 或者 SAU 访问规则的故障时，我们通过 MMFAR、SFAR 我们大概可以知道是那个地址的访问造成了错误，但是具体为什么访问那个地址会触发故障，就可能还需要检查 MPU 和 SAU 的配置才能知道真正的问题所在。

3.3.1. MemManageFault 调试

MemManageFault 是一类比较常见的故障异常，当发生此类错误时，首先可以在 CFSR 中观察到 MemManage 相关位域被置 1，如果 MMARVALID=1 那么还可以参考 MMFAR 指示的地址，得知访问哪个地址触发了 MemManage 错误。

例如图 11 的示例，可以看到 DACCVIOL=1，且 MMARVALID=1，MMFAR=30030000，说明故障是由于代码试图对地址 0x30030000 进行数据访问导致的故障。

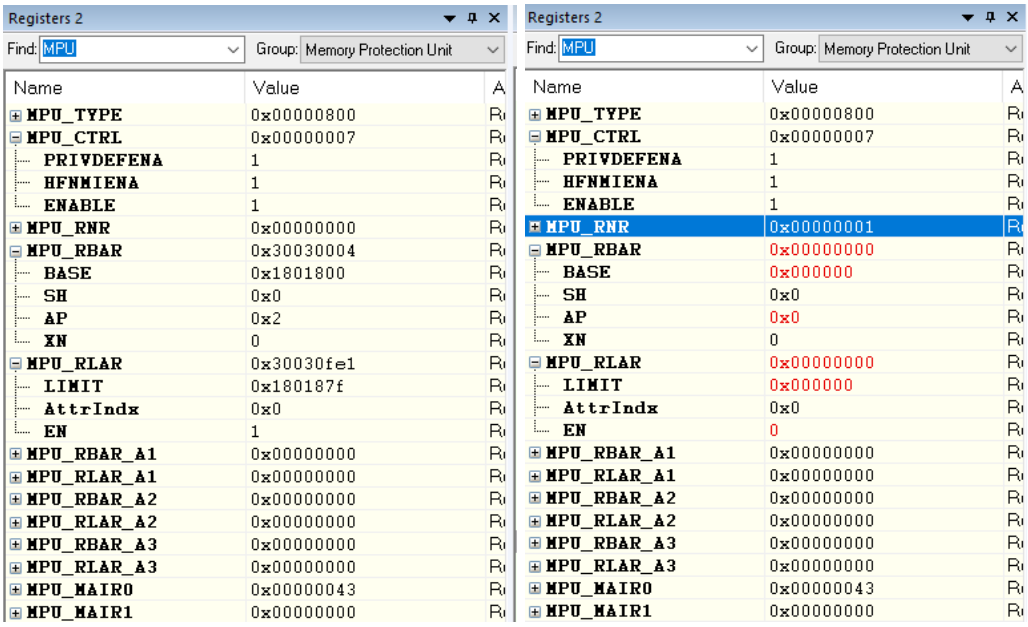
图11. 观察 MemManage 故障寄存器示例



示例中这个导致故障的 0x30030000 地址看起来只是个普通的 RAM 地址，并没有什么特别，如果希望知道为什么访问这个地址导致了故障，我们还可以进一步观察 MPU 寄存器的内容，检查 MPU 的设置是怎样的，确认该地址访问是否违反了 MPU 配置的保护规则。

例如图 12 的示例，可以看到 MPU 定义了一个从 0x30030000 开始大小为 0x1000 的 region，该 region 的访问权限为 AP=2（即 Read-only by privileged code only），所以这个地址范围是只读的区域。按照 4.1 小节提到过的方法，可以找到触发错误的 PC，发现该条指令是向 0x30030000 地址进行写操作，因而导致了故障。

图12. 观察 MPU 寄存器示例



由于 MPU region 是通过先写 RNR 寄存器，再写 RBAR 和 RLAR 寄存器分别配置不同 region 的，因此观察 MPU 寄存器时并不能够一次性看到所有 region 的配置。在调试器中，如果想看到不同 MPU region 的配置，我们可以通过重复以下步骤的方法依次查看不同 region 的配置。

- 向 MPU_RNR 寄存器分别写入 0-7 的值
- 回车后观察 MPU_RBAR 和 MPU_RLAR 寄存器

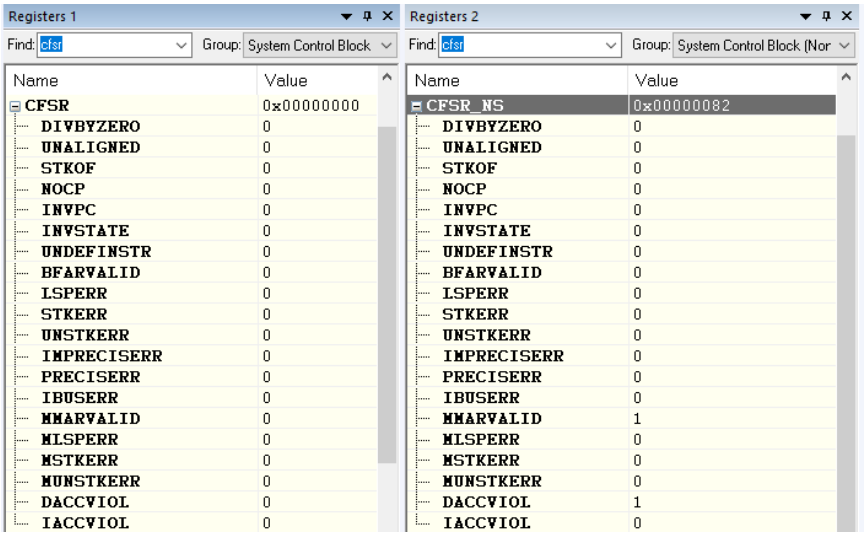
MemManage 故障调试时的几点注意事项：

1） 在某些 STM32 系列中，MPU 配置可以被锁定，在这些系列中，上述方法能够使用的前提是 MPU 配置没有被 lock，如果 MPU 已经处于 lock 状态，则无法通过改变 MPU_RNR 值的方法观察不同 MPU region 的配置。这种情况下建议可以通过断点、单步调试以及修改 PC 的方法，先跳过 MPU lock 的相关代码，让 Fault 发生时 MPU 保持在没有 lock 的状态，方便后续调试观察。

2）支持 security extension 的 CM33 内核中 MPU 寄存器是 bank 的，也就是说 TrustZone 环境中 S 安全侧和 NS 非安全侧有各自的 MPU 寄存器，并且可以有不同的 MPU 配置。在调试 MemManage 故障时，需要根据 EXC_RETURN bit[6]的提示决定应该观察 MPU_S 还是 MPU_NS 寄存器。如果 bit[6]=1，说明故障由 S 代码触发，则应当分析 MPU_S 寄存器的配置；反之，如果故障由 NS 代码触发，则应当分析 MPU_NS 寄存器的配置。

3）支持 security extension 的 CM33 内核中，SCB 的 CFSR 的 MMFSR 和 USFR 位域在 S 和 NS 侧是 bank 的，因此，发生相关 Fault 时，需要参考对应 SCB_S 和 SCB_NS 中的 CFSR 的区域读取故障信息。例如当 NS MemManage 异常使能，且 NS 代码触发了 NS MemManage 故障时，如果读取 SCB_S 的 CFSR 将看到 0，此时应当查看 SCB_NS 的 CFSR，则能够看到实际的故障状态信息。例如图 13 所示。

图13. NS 触发 MemManage 故障时观察 CFSR_S/CFSR_NS 结果示例



这里也列举了一些常见的由于违反 MPU 访问规则导致错误的情况，供大家参考。

- MPU 定义的 region 区域存在重叠，重叠区域为完全不可访问区域，软件访问该区域，无论指令访问还是数据访问都将触发故障
- MPU region 定义了只读数据区，但出现了该区域地址的写访问
- MPU region 定义了不可执行区域，软件试图从不可执行区域取指令
- MPU region 定义某些区域只能由 privilege 特权模式访问，运行在非特权模式的代码尝试访问了该区域（可能是指令访问，也可能是数据访问）
- MPU 配置了 MPU_CTRL.PRIVDEFENA=0，也就是说除了 region 覆盖的区域以外，其他地址都不可访问，而代码尝试访问了 region 覆盖范围以外的地址
- MPU 配置了 MPU_CTRL.HFNMIEN=0，即在 HardFault 和 NMI Handler 中 MPU 不使能，而 HardFault 处理函数中尝试访问了某些必须使能 MPU 才能正常访问的地址（例如尝试执行地址在 0xA0000000 以上的 HSPI 外挂 Memory 上的代码）

3.3.2. SecureFault 调试

SecureFault 是 TrustZone 开发中比较常见的一类错误，可能由多种原因导致，比较典型的有两类：

第一类：指令、数据访问违反了 SAU/IDAU 定义的安全规则。

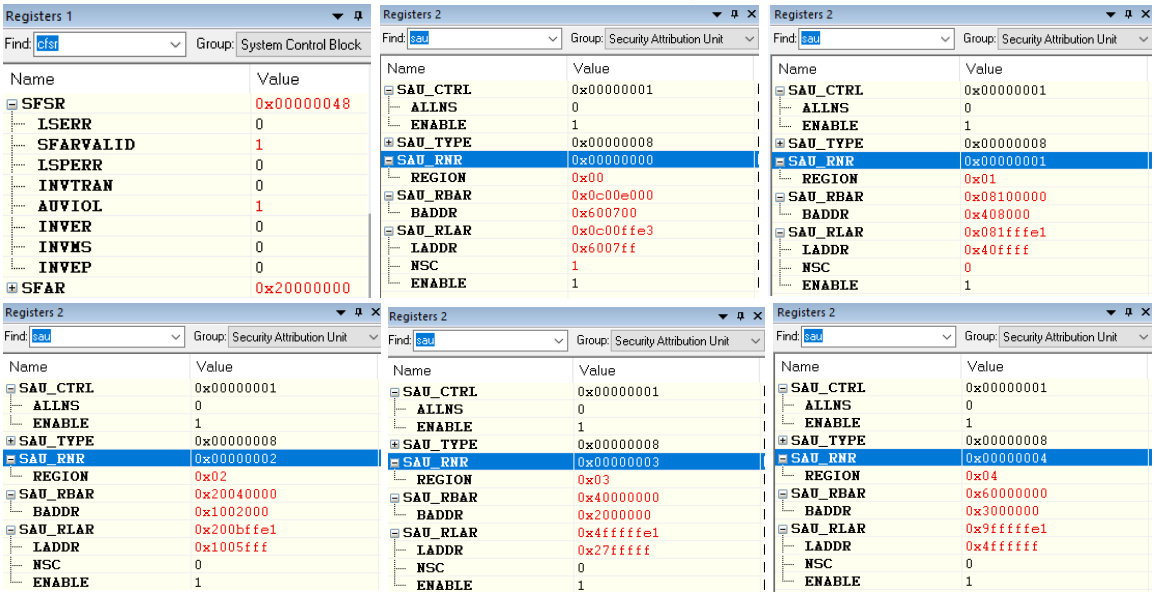
这类问题多见于 NS 代码尝试访问某些地址，而该地址没有在 S 代码中配置相应的 SAU NS 区域的情况，SFSR.AUVIOL 会提示这种类型的 SecureFault。此类问题在 TrustZone 开发初期没有正确配置 SAU 定义安全、非安全区域时尤为多见。

这种情况在调试时，如果看到 SFSR 的一些位被置位，可以通过检查 SAU region 配置的方法确认。例如图 14 的示例所示，这里 SFSR 指示 AUVIOL=1，说明出现了 SAU violation，同时 SFARVALID=1，说明代码是因为访问了 SFAR 提示的 0x20000000 地址触发了 SecureFault。此时可以进一步观察 SAU 寄存器，得知该地址在 SAU 配置中的安全属性。这里可以看到 0x20000000 地址没有在任何一个被配置为 NS 的 SAU region 中，因此 NS 代码尝试访问该地址一定会触发 Secure Fault。

类似 MPU，SAU region 的配置是通过同一组寄存器，先写 RNR 寄存器指定 region number，然后写 RBAR 和 RLAR 寄存器的方法进行 region 配置的，所以在 SAU 寄存器

窗口并不能一次看到所有 region 的配置。这时候可以通过修改 RNR 寄存器值（0 - 7）的方法，依次看到每个 region 的设置，步骤于 MPU region 配置的观察方法相似，见图 14 的示例所示。与 MPU 相似，SAU 配置在某些 STM32 系列中也可以被锁定，一旦 SAU 处于 lock 状态，在调试器中将不能通过修改 RNR 寄存器值的方式观察 SAU region 配置信息，调试阶段可以通过前面提到过的方法先保持 SAU 在非 lock 状态，方便调试。

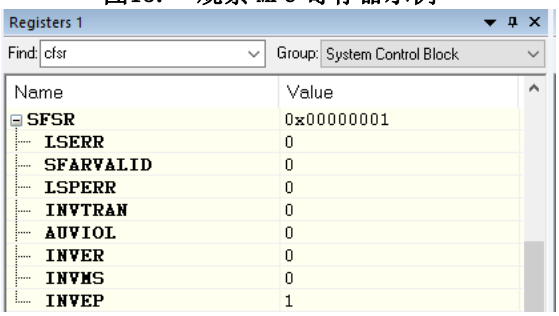
图14.　观察 SFSR 和 SAU 寄存器示例



第二类：NS 调用 S API 时，入口指令不是 SG 指令，或则 S API 入口所在区域没有在 SAU 中定义为 NSC 区域。

通常当 S 代码的 NSC 位置布局发生调整，而 NS 代码又没有重新链接 S 工程输出的.o 时容易发生此类错误。这种情况一般会导致 INVEP 错误，观察 SFSR 可以看到 SFSR.INVEP=1，类似图 15 所示。

图15.　观察 MPU 寄存器示例



当然触发 SecureFault 也可能有其他原因，比如 SFSR 的 INVER 所提示的错误，通常正常的代码应该不会有这些错误，但是如果 stack 内容被意外改写（比如由于 stack overflow 或者其他原因）或者在使用 RTOS 的情况下，RTOS 没有正确处理 TrustZone 的 S/NS 切换，也有可能触发 SecureFault 的这些错误。

3.4. 一些复杂的情况

有些时候，由于软件实现和配置上的不同，某些故障被触发时，有可能已经不是出错的第一现场，真正出问题的地方并没有第一时间触发任何 Fault，而是在错误的基础上运行了一段时间之后才真正触发了 Fault 异常。比如堆栈溢出或者被改写的情况，堆栈被意外修改的时候并不一定触发任何 Fault 异常，但是当堆栈中的数据被再次使用时，可能导致各种各样的错误。这时候 Fault 的分析可能会变得更加复杂，需要深入研究才能找到问题的根源。有时候可能需要借助某些工具（比如堆栈检查等）来分析问题。有时可能还需要借助打开某些其他保护功能，找到错误发生的更早的位置。比如在分析触发故障的 PC 时，如果发现 PC 的位置不是有效的代码段地址，那么可以考虑添加 MPU region 的保护，让非代码段地址范围具有不可取指令访问的属性，这样有可能在执行到当前触发故障的 PC 之前就会触发 MemManage Fault，有助于帮助定位实际出错的代码。

4. 小结

掌握一些 TZ 环境分析 Fault 的技巧，在开发中遇到软件触发 Fault 故障时保持沉着冷静，按照一定的方法认真地进行分析，往往可以快速有效地解决问题。本文针对使用 STM32 MCU 进行 ARM V8M TrustZone 开发中的 Fault 处理与调试做了一些总结，希望对开发者有所帮助。

在后续的篇章中，我们还将讨论 TrustZone 中 App 使用 RTOS 的注意事项等话题，欢迎读者继续关注。

5. 附注

SCB: System Control Block
AIRCR: Application Interrupt and Reset Control Register
SHCSR: System Handler Control and State Register

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

IAR Embedded Workbench for ARM v8.50.9

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。