HardFault 分析教程

Teach you how to locate HardFault and analyze its causes（教你如何定位HardFault，并分析HardFault的原因）

1. 定位HardFault产生的位置

> **操作方法**:把你的void HardFault_Handler(void) 替换成下摆你这段代码，然后在hardfault处打断点，单步执行就可以找到问题原因处

1.1 先修基础知识

> 参考文档-HardFault定位方法

> 压栈的 xPSR 中的第 9 位表示栈指针的数值是否调整过。在图 8.2 中，栈指针为双字对齐的，因此就不会额外插入一个字，而且 xPSR 的第 9 位也为 0。在双字栈对齐特性被关闭后，栈帧还是会保持这种方式的，只是栈指针的数值有可能未对齐到双字地址。

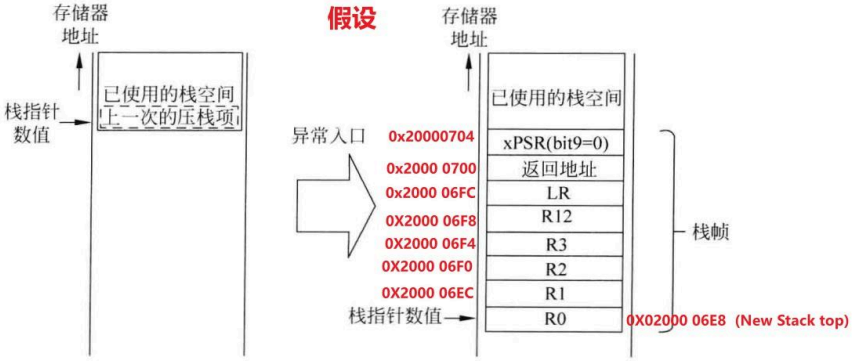


图 8.2　在不需要或禁止双字栈对齐时，Cortex-M3 或 Cortex-M4（无浮点单元）处理器的异常栈帧

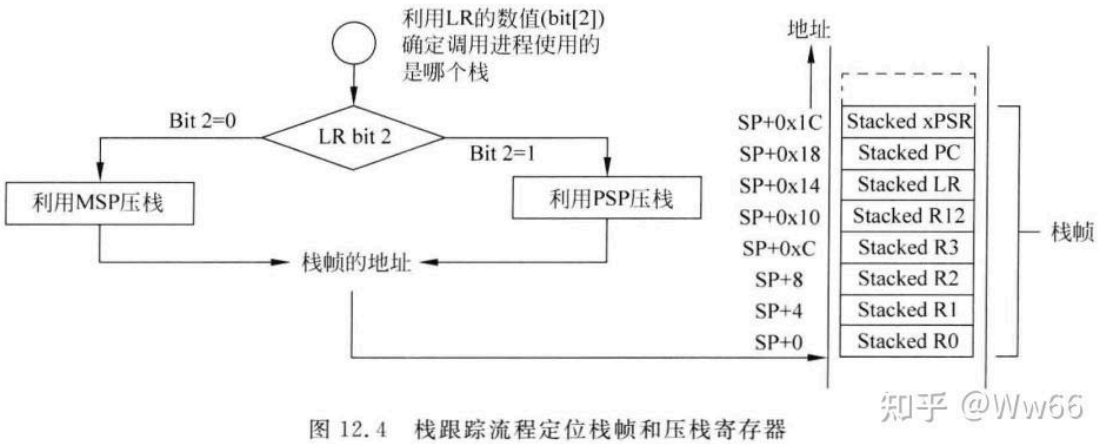


图 12.4　栈跟踪流程定位栈帧和压栈寄存器

> **总结**:根据单片机进中断，寄存器压栈情况，压栈时候把LR寄存器值（存储着返回来要执行的程序地址）也压进去了，那只需要在hardfault ISR里边把LR值取出来赋值给PC就可以定位到引发hardfault的地方，如下面操作，不过这个只是定位到了引发hardfault的位置，下一步还需要结合hradfault状态寄存器分析产生的具体原因

1.2 C语言版本

> 注意！！ 下边的MSP值+0X1C为存 LR的值，这个不同核心单片机有区别，M3核的理论上来说加0X14就可以了，但是M0核得+0x1C，实践得出来的结果

```
// Return the R13(Stack Pointer)
__ASM unsigned int __Get_SP_Value(void)
{
    mov r0, r13 //R13(SP)
    bx lr
}

void (*f1)(void);
unsigned int result;

void HardFault_Handler(void)
{
    //实际操作Arm Coretex-M0核心的是+0x1C，M3核的是+0X14
    result = __Get_SP_Value()+0x14;
    f1 = *( (unsigned int*)result );
    f1();
    while (1);
}
```

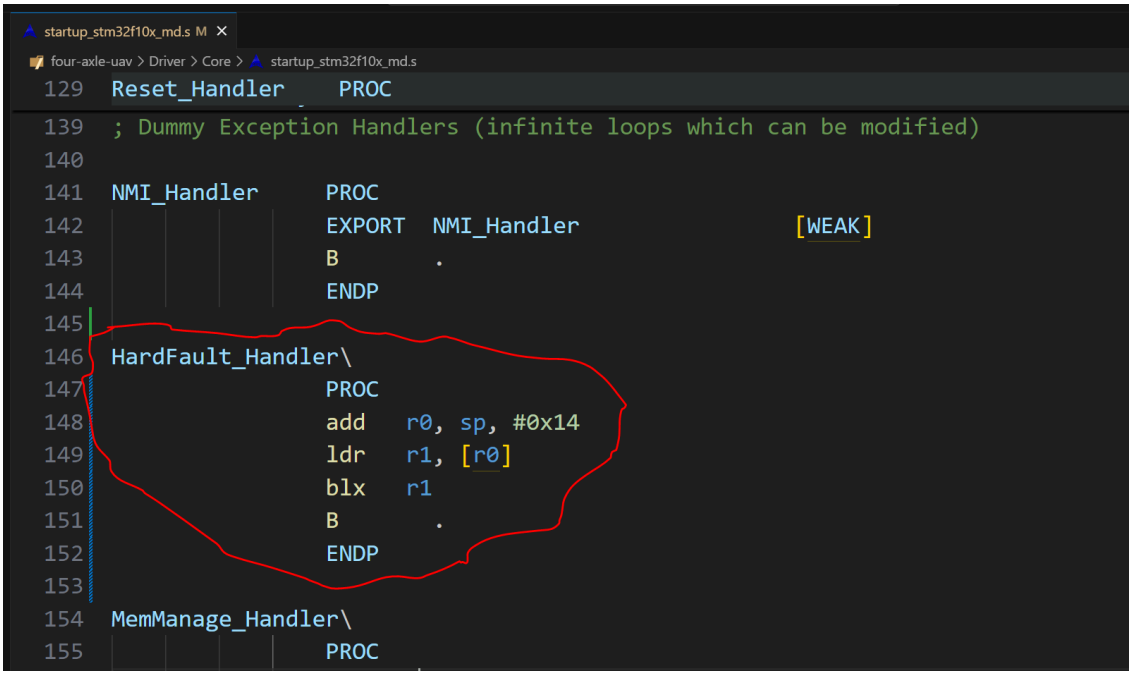
```
52  // Return the R13(Stack Pointer)
53  __ASM unsigned int __Get_SP_Value(void)
54  {
55      mov r0, r13 //R13(SP)
56      bx lr
57  }
58
59  void (*f1)(void);
60  unsigned int result;
61
62  void HardFault_Handler(void)
63  {
64      //实际操作Arm Coretex-M0核心的是+0x1C，M3核的是+0X14
65      result = __Get_SP_Value()+0x14;
66      f1 = *( (unsigned int*)result );
67      f1();
68      while (1);
69  }
```

1.3 汇编语言版本

> 注意！！ 下边的MSP值+0X1C为存 LR的值，这个不同核心单片机有区别，M3核的理论上来说加0X14就可以了，但是M0核得+0x1C，实践得出来的结果

```
；把你的起始.s文件内的HardFault_Handler代码换成下边这个

HardFault_Handler\
                PROC
                add   r0, sp, #0x14
                ldr   r1, [r0]
                blx   r1
                B       .
                ENDP
```



1.4 操作方法

> 当产生了hardfault后，选择上述1.2或者1.3的代码方案后，debug 在hardfult处打断点执行到hardfault ISR处之后再单步执行，就可以找到出问题的地方了

---

2. 分析HardFault原因

> 分析产生HardFault的原因（需要具体代码具体分析仔细崩溃处的代码的问题）
> hardfault产生的原因有很多，例如访问非法地址了，函数指针变量为NULL，然后直接运行了，总共包含以下几种

1. 内存访问错误

- 访问了未映射的内存区域。
- 对只读内存区域执行写操作。
- 访问了无效的内存地址（如NULL指针解引用）。

2. 堆栈溢出：

- 如果任务或中断的堆栈空间不足，可能导致堆栈溢出，进而破坏内存或触发HardFault。

3. 未对齐的内存访问：

- 某些STM32系列要求特定的数据类型在内存中对齐。未对齐的访问可能触发HardFault。

4. 错误的异常处理：

- 如果在异常处理程序中再次发生异常，且没有正确处理，可能会导致HardFault。

5. 中断优先级配置错误：

- 如果配置了错误的中断优先级，可能导致不可屏蔽中断（NMI）或HardFault。

6. 非法指令或状态：

- 执行了未定义的指令。
- 尝试切换到无效的处理器模式。

7. 外设配置错误：

- 错误地配置或使用了外设，如未启用时钟就访问外设寄存器。

8. 电源问题：

- 不稳定的电源可能导致处理器行为异常，从而触发HardFault。

9. 软件错误：

- 逻辑错误、数组越界、类型转换错误等编程错误也可能导致HardFault。

> 但是这些太笼统了，太废话了，具体是哪个呢？？ 那还得看HardFault状态寄存器（HFSR）以及其他相关的故障状态寄存器，如配置管理故障状态寄存器（CFSR）、内存管理故障状态寄存器（MMFSR）、总线故障状态寄存器（BFSR）和使用故障状态寄存器（UFSR）。
> 下图是HardFault状态寄存器，引发hardfault了可以看看这个寄存器值，然后再仔细检查下代码

2.1 Chinese version

Cortex-M3 和 Cortex-M4 处理器具有可用于错误分析的多个寄存器，它们可被错误异常处理代码使用。有些情况下，运行在调试主机上的调试器软件也可以用它们来显示错误状态。表 12.1 对这些寄存器进行了总结，且只能在特权状态访问这些寄存器。

表 12.1　错误状态寄存器

地址	寄存器	CMSIS-Core 符号	功　能
0xE000ED28	可配置错误状态寄存器	SCB->CFSR	可配置错误的状态信息
0xE000ED2C	硬件错误状态寄存器	SCB->HFSR	硬件错误的状态
0xE000ED30	调试错误状态寄存器	SCB->DFSR	调试事件的状态
0xE000ED34	MemManage 错误状态寄存器	SCB->MMFAR	如果存在，表示触发 MemManage 错误时访问的地址
0xE000ED38	总线错误状态寄存器	SCB->BFAR	如果存在，表示触发总线错误时访问的地址
0xE000ED3C	辅助错误状态寄存器	SCB->AFSR	设备定义的错误状态

12.4.5　HardFault 状态寄存器

硬件错误状态寄存器的编程模型如表 12.6 所示。

表 12.6　硬件错误状态寄存器(0xE000ED2C，SCB->HFSR)

位	名称	类型	复位值	描　述
31	DEBUGEVT	R/Wc	0	表明硬件错误由调试事件触发
30	FORCED	R/Wc	0	表明硬件错误由于总线错误、存储器管理错误或使用错误而产生
29:2	—	—	—	—
1	VECTBL	R/Wc	0	表明硬件错误由取向量失败引发
0	—	—	—	—

HardFault 处理可以使用这个寄存器确定 HardFault 是否由可配置错误引起，若 FORCED 置位，则表示 HardFault 是由一个可配置错误引起，并且应该检查 CFSR 的值以确定错误的原因。

与其他的错误状态寄存器类似，当错误产生时每个错误状态指示位都会置位，而且会在将 1 写入这个寄存器前一直保持为高。

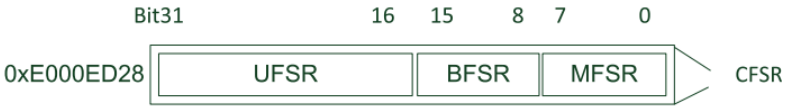
2.1 English version

Table 12.1 Registers for Fault Status and Address Information

Address	Register	CMSIS-Core Symbol	Function
0xE000ED28	Configurable Fault Status Register	SCB->CFSR	Status information for Configurable faults
0xE000ED2C	HardFault Status Register	SCB->HFSR	Status for HardFault
0xE000ED30	Debug Fault Status Register	SCB->DFSR	Status for Debug events
0xE000ED34	MemManage Fault Address Register	SCB->MMFAR	If available, showing accessed address that triggered the MemManage fault
0xE000ED38	BusFault Address Register	SCB->BFAR	If available, showing accessed address that triggered the bus fault
0xE000ED3C	Auxiliary Fault Status Register	SCB->AFSR	Device-specific fault status

Table 12.2 Dividing Configurable Fault Status Register (SCB->CFSR) Into Three Parts

Address	Register	Size	Function
0xE000ED28	MemManage Fault Status Register (MMFSR)	Byte	Status information for MemManage Fault
0xE000ED29	Bus Fault Status Register (BFSR)	Byte	Status for Bus Fault
0xE000ED2A	Usage Fault Status Register (UFSR)	Halfword	Status for Usage Fault



12.4.5 HardFault status register

The programmer's model for the Usage Fault Status Register is shown in Table 12.6.

HardFault handler can use this register to determine whether a HardFault is caused by any of the configurable faults. If the FORCED bit is set, it indicates that the fault has been escalated from one of the configurable faults and it should check the value of CFSR to determine the cause of the fault.

Similar to other fault status registers, each fault indication status bit is set when the fault occurs, and stays high until a value of 1 is written to the register.