文档编号：AN054

上海东软载波微电子有限公司

应用笔记

Cortex-M0 HardFault 诊断

修订历史

版本	修订日期	修改概要
V1.0	2017-07-5	初版发布

地　址：中国上海市龙漕路 299 号天华信息科技园 2A 楼 5 层
邮　编：200235
E-mail：support@essemi.com
电　话：+86-21-60910333
传　真：+86-21-60914991
网　址：http://www.essemi.com/

目 录

内容目录

第1章 HardFault异常原理

1. 1 Cortex-M0 异常类型

复位 — 复位在上电或热复位时启动。异常模型将复位当做一种特殊形式的异常来对待。当复位产生时，处理器的操作停止（可能停止在一条指令的任何一点上）。当复位撤销时，从向量表中复位项提供的地址处重新启动执行。执行在线程模式下重新启动。

NMI — 一个不可屏蔽中断（NMI）可以由外设产生，也可以由软件来触发。这是除复位之外优先级最高的异常。NMI 永远使能，优先级固定为-2。NMI 不能：

1. 被屏蔽，它的执行也不能被其他任何异常中止；
2. 被除复位之外的任何异常抢占。

HardFault — HardFault 是由于在正常操作过程中或在异常处理过程中出错而出现的一个异常。HardFault 的优先级固定为-1，表明它的优先级要高于任何优先级可配置的异常。

SVCall — 管理程序调用（SVC）异常是一个由 SVC 指令触发的异常。在 OS 环境下，应用程序可以使用 SVC 指令来访问 OS 内核函数和器件驱动。

PendSV — PendSV 是一个中断驱动的系统级服务请求。在 OS 环境下，当没有其它异常有效时，使用 PendSV 来进行任务切换。

SysTick — SysTick 是一个系统定时器到达零时产生的异常。软件也可以产生一个 SysTick 异常。在 OS 环境下，处理器可以将这个异常用作系统节拍。

中断（IRQ） — 中断（或 IRQ）是外设发出的一个异常，或者是由软件请求产生的一个异常。所有中断都与指令执行不同步。在系统中，外设使用中断来与处理器通信。

异常编号	IRQ编号	异常类型	优先级	向量地址
1	—	复位	-3，优先级最高	0x00000004
2	-14	NMI	-2	0x00000008
3	-13	HardFault	-1	0x0000000C
4-10	—	保留	—	
11	-5	SVCall	可配置	0x0000002C
12-13	—	保留	—	
14	-2	PendSV	可配置	0x00000038
15	-1	SysTick	可配置	0x0000003C
16和大于16的值	0和大于0的值	中断（IRQ）	可配置	0x00000040 以及更高的地址

表 1-1 Cortex-M0 异常类型特性表

1. 2 异常的进入和返回

描述异常处理时使用了下列术语：

抢占 — 当处理器正在执行一个异常处理程序时，如果一个异常的优先级比正在处理的异常的优先级更高，那么低优先级的异常就被抢占。

当一个异常抢占另一个异常时，这些异常就被称为嵌套异常。详见本节 1.2.1“异常进入”。

返回 — 当异常处理程序结束，并且满足以下条件时，异常就返回。

- 没有优先级足够高的挂起异常要处理
- 已结束的异常处理程序没有在处理一个迟来的异常

处理器弹出堆栈，处理器状态恢复到中断出现之前的状态，详见本节 1.2.2“异常返回”。

末尾连锁 — 这个机制加速了异常的处理。当一个异常处理程序结束时，如果一个挂起的异常满足异常进入的要求，就跳过堆栈弹出，控制权移交给新的异常处理程序。

迟来（Late-arriving） — 这个机制加速了抢占的处理。如果一个高优先级的异常在前一个异常正在保存状态的过程中出现，处理器就转去处理更高优先级的异常，开始提取这个异常的向量。状态保存不受迟来异常的影响，因为两个异常保存的状态相同。从迟来异常的异常处理程序返回时，要遵守正常的末尾连锁规则。

1. 2. 1 异常进入

当有一个优先级足够高的挂起异常存在，并且满足下面的任何一个条件，就进入异常处理：

- 处理器处于线程模式
- 新异常的优先级高于正在处理的异常，这时，新异常就抢占了正在处理的异常。

当一个异常抢占了另一个异常时，异常就被嵌套。

当处理器处理异常时，除非异常是一个末尾连锁异常或迟来的异常，否则，处理器把信息都压入到当前堆栈中**入栈（stacking）**，8 个数据字的结构被称为**栈帧（stack frame）**。

栈帧包含以下信息：

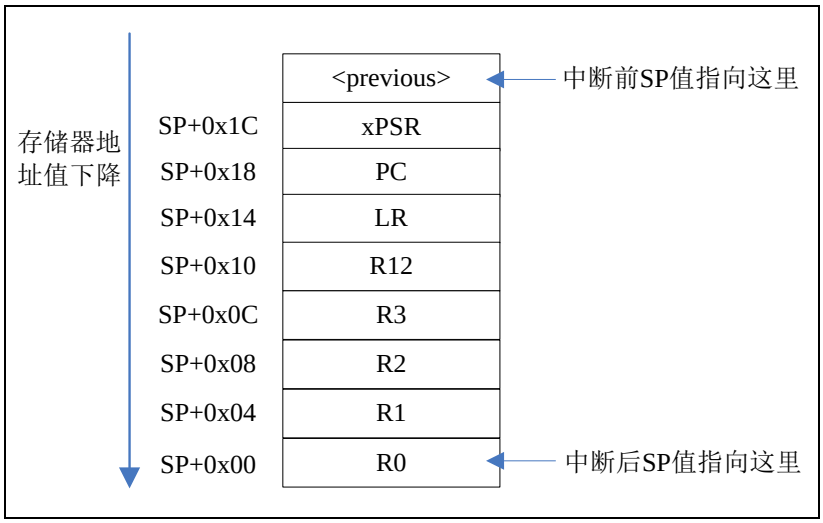

图 1-1 异常进入时堆栈的内容

入栈后，堆栈指针立刻指向栈帧的最低地址单元。栈帧按照双字地址对齐。

栈帧包含返回地址。这是被中止的程序中下条指令的地址。这个值在异常返回时返还给 PC，使被中止的程序恢复执行。

处理器执行一次向量提取，从向量表中读出异常处理程序的起始地址。当入栈结束时，处理器开始执行异常处理程序。同时，处理器向 LR 写入一个 EXC_RETURN 值。这个值指示了栈帧对应哪个堆栈指针以及在异常出现之前处理器处于什么工作模式。

如果在异常进入的过程中没有更高优先级的异常出现，处理器就开始执行异常处理程序，并自动将相应的挂起中断的状态变为有效。

如果在异常进入的过程中有另一个优先级更高的异常出现，处理器就开始执行这个高优先级异常的异常处理程序，不改变前一个异常的挂起状态。这是一个迟来异常的情况。

1. 2. 2 异常返回

当处理器处于处理模式（Handler），且执行下面其中一条指令尝试将 PC 设为 EXC_RETURN 值时，出现异常返回：

- POP 指令，用来加载 PC
- BX 指令，用来使用任意的寄存器

在异常进入时，处理器将一个 EXC_RETURN 值保存到 LR 中。异常机制依靠这个值来检测处理器何时执行完一个异常处理程序。EXC_RETURN 值的 bit[31:4]为 0xFFFFFFF。当处理器将一个相应的这种形式的值加载到 PC 时，它将检测到这个操作并不是一个正常的分支操作，而是异常已经结束。因此，处理器启动异常返回序列。EXC_RETURN 的 bit[3:0]指出了所需的返回堆栈和处理器模式，如表 1-2 所示。

EXC_RETURN	描述
0xFFFFFFF1	返回到处理器模式 异常返回获得主堆栈的状态 返回后执行使用MSP
0xFFFFFFF9	返回到线程模式 异常返回获得MSP的状态 返回后执行使用MSP
0xFFFFFFFD	返回到线程模式 异常返回获得PSP的状态 返回后执行使用PSP
所有其他值	保留

表 1-2 异常返回行为

1. 3 HardFault 异常

1. 3. 1 硬件错误原因

硬件错误是异常的一个子集。

在 ARM 处理器中，如果一个程序产生了错误并且被处理器检查到，这时就会产生错误异常，Cortex-M0 处理器只有一种异常用来处理错误：硬件错误处理（HardFault）。

所有的故障都导致 HardFault 异常被处理，还有如果故障在 NMI 或 HardFault 处理程序中出现，会导致处理器进入锁定状态。

对于 Cortex-M0 处理器，主要有以下情况会导致出现硬件错误异常：

- 在一个优先级等于或高于 SVCall 的地方执行 SVC 指令；
- 在没有调试器的情况下执行 BKPT 指令；
- 在加载或存储时出现一个系统产生的总线错误；
- 执行一个 XN 存储器地址中的指令；
- 从系统产生了一个总线故障的地址单元中执行指令；
- 在提取向量时出现了一个系统产生的总线错误；
- 执行一个未定义的指令；
- 由于 T 位之前被清零而导致不再处于 Thumb 状态的情况下执行一条指令；
- 尝试对一个不对齐的地址执行加载或存储操作。

注意：只有复位和 NMI 可以抢占优先级固定的 HardFault 处理程序。HardFault 可以抢占除复位、NMI 或其他硬故障之外的任何异常。

1. 3. 2 锁定

如果在执行 NMI 或 HardFault 处理程序时出现故障，或者在一个使用 MSP 的异常返回时出栈的却是 PSR 的时候系统产生一个总线错误，处理器进入一个锁定状态。当处理器处于锁定状态时，它不执行任何指令。处理器保持处于锁定状态，直到下面任何一种情况出现：

- 出现复位；
- 调试器将锁定状态终止；
- 出现一个 NMI，以及当前的锁定处于 HardFault 处理程序中。

注意：如果锁定状态出现在 NMI 处理程序中，后面的 NMI 就无法使处理器离开锁定状态。

第2章 HardFault异常分析

对于 HardFault 异常的原因，需要进一步确定每次发生硬件错误时的问题根源。例如，总线错误可以有很多情况引发，例如错误的指针操作、栈空间损坏、内存溢出、非法存储器映射等。

2. 1 分析错误

根据错误类型的不同，通常能够直接确定引起硬件错误异常的指令位置。要实现这一目的，就需要知道进入硬件错误异常时的寄存器内容，以及异常处理前压入栈中的寄存器的内容。这些值中包含了程序返回地址，通过它也能知道引起错误的指令地址。

用户如果使用调试器调试，那么可以通过在硬件错误开始位置添加暂停处理器的断点指令，比较方便的观察 HardFault 异常，当硬件错误发生时，处理器就会在此自动暂停。在处理器由于硬件错误暂停后，我们就可以尝试着按照图 2-1 所示的流程对错误进行定位。

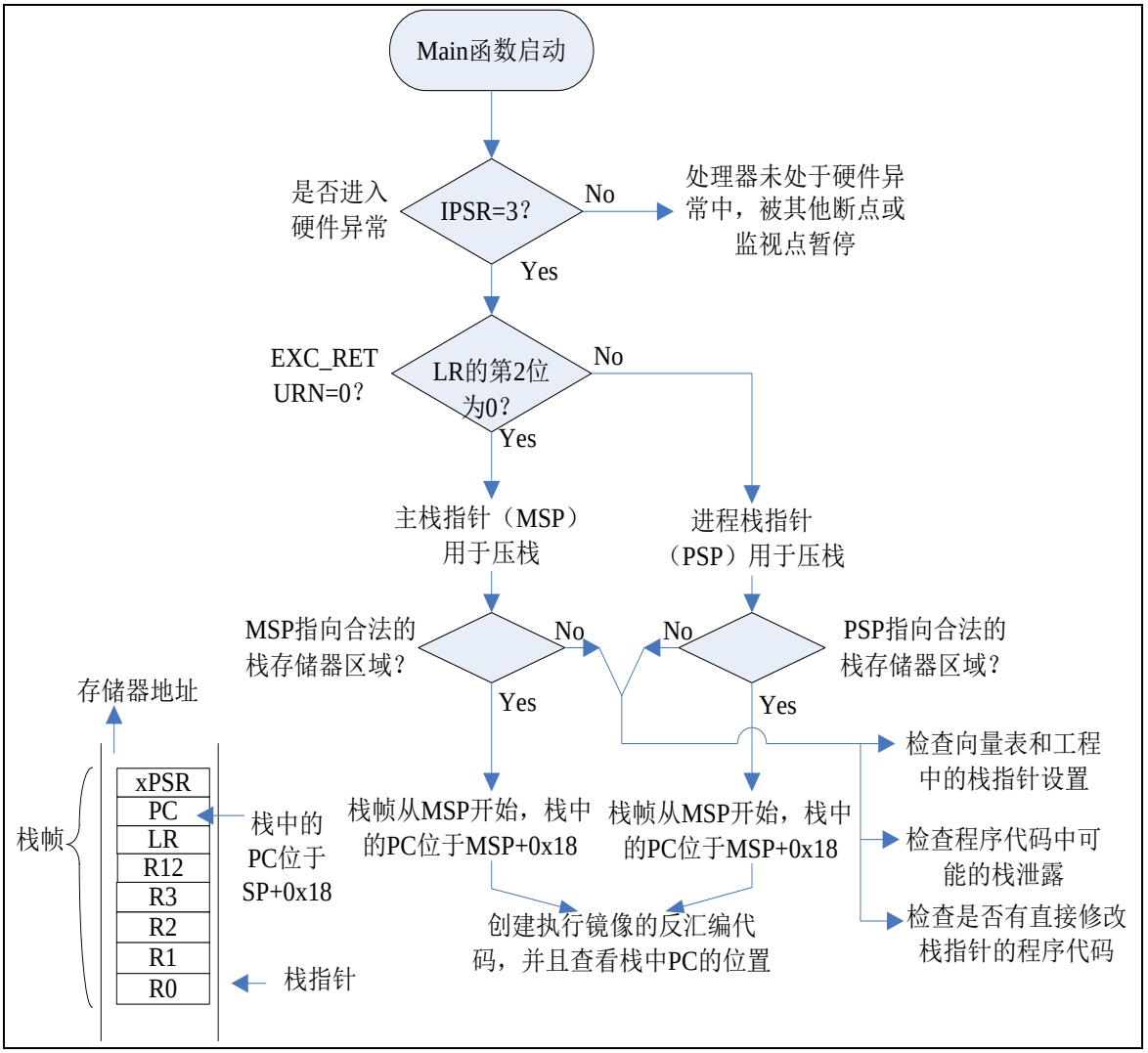

图 2-1 HardFault 定位错误流程

为了给异常分析提供更多的信息，可以生成程序映射的汇编代码，并且利用在栈帧中找到的 PC 值确定错误的位置。如果错误的地址为存储器访问指令，就应该检查寄存器的值确定存储器访问的地址是否合法，除了检查地址范围，也应该确认存储器的地址是否正确的对齐。

除了压入栈中的 PC 值，栈帧中也包含了其他有助于调试的寄存器，例如，压入栈的 IPSR 能够反映处理器是否在进行异常处理；EPSR 则代表了处理器状态：EPSR 的 T 位为 0，则代表错误由意外切换至 ARM 状态引起。栈中的 LR 也可能提供一些信息，例如发生错误的函数的返回地址、错误是否发生在异常处理中，以及 EXC_RERURN 的值是否被异常破坏等。

另外，当前的寄存器也可以提供有助于定位错误原因的各种信息，除了当前栈指针的值，当前的链接寄存器（R14）的值也可能有帮助。如果 LR 中为非法的 EXC_RERURN 值，这就意味着它在前面异常处理中被错误地修改了。

CONTROL 寄存器也可以提供帮助，在没有 OS 的简单应用程序中，进程栈指针（PSP）不能被用到，并且 CONTROL 寄存器会一直保持 0，如果 CONTROL 寄存器被设置为 0x02（PSP 用于线程），这就意味着 LR 在之前的异常处理中被错误地修改了，或者栈内容被破坏导致了 RERURN 的值错误。

在实际用户系统开发中，系统运行不可能实时连调试器，更不可能在硬件异常入口打断点，所以用户没办法直接从调试界面知道 HardFault 异常时压栈和各寄存器的值。但是，用户可以编写程序，当硬件错误发生时处理器主动上传硬件错误报告，并同时上传异常发生时压到栈里的 PC 和相关寄存器的值，以帮助分析。

2. 2 硬件错误异常举例

根据 ARM HardFault 异常机理和基本的诊断方法，具体分析例程见附录 1。

在程序中定义一个超出芯片实际 RAM 物理地址的全局变量，然后在 main 函数中操作这个变量。由于实际中，定义的变量地址 MCU 是没有的，故处理器在执行这个变量时就会产生硬件错误，进入 HardFault 异常中断，我们在 HardFault 异常中断中创建了硬件错误报告通过串口打印出来。

该段 C 程序需要由汇编程序封装，放在 startup.s 文件中。它主要用于发生异常时提取栈帧的起始地址，封装的函数代码如下：

```
HardFault_IRQHandler\
                PROC
                IMPORT  HardFault_IRQHandler_c      ;函数声明
                movs    r0, #4                      ;判断主栈指针还是进程栈指针
                mov     r1, lr
                tst     r0, r1
                beq     hf_used_msp                 ;如果是主栈指针
                mrs     r0, psp                     ;否则是进程栈指针,把进程栈指针地址付给 R0
                ldr     r1, =HardFault_IRQHandler_c ;跳转到 HardFault 中断程序
                bx      r1
hf_used_msp
                mrs     r0, msp                     ;把主栈指针地址赋给 R0
                ldr     r1, =HardFault_IRQHandler_c
                bx      r1
                ENDP
```

执行程序，处理器按照我们的设想进入了 HardFault 异常，并打印出了如下报告：

```
this is HardFault text

[Hard fault handler-all number in hex]
R0=ffff0001
R1=00000100
R2=40006400
R3=00000df9
R12=ffffffff
LR[R14]=00000437
PC[R15]=00000dc6
PSR=61000000
SCB_SHCSR=0000000
```

图 2-2　硬件错误打印报告

根据 2.1 节的分析方法，我们得到发生硬件错误时 PC 值为 0x00000DC6，根据这个 PC 值我们可以通过生成程序映像的反汇编找到当时程序执行的位置：

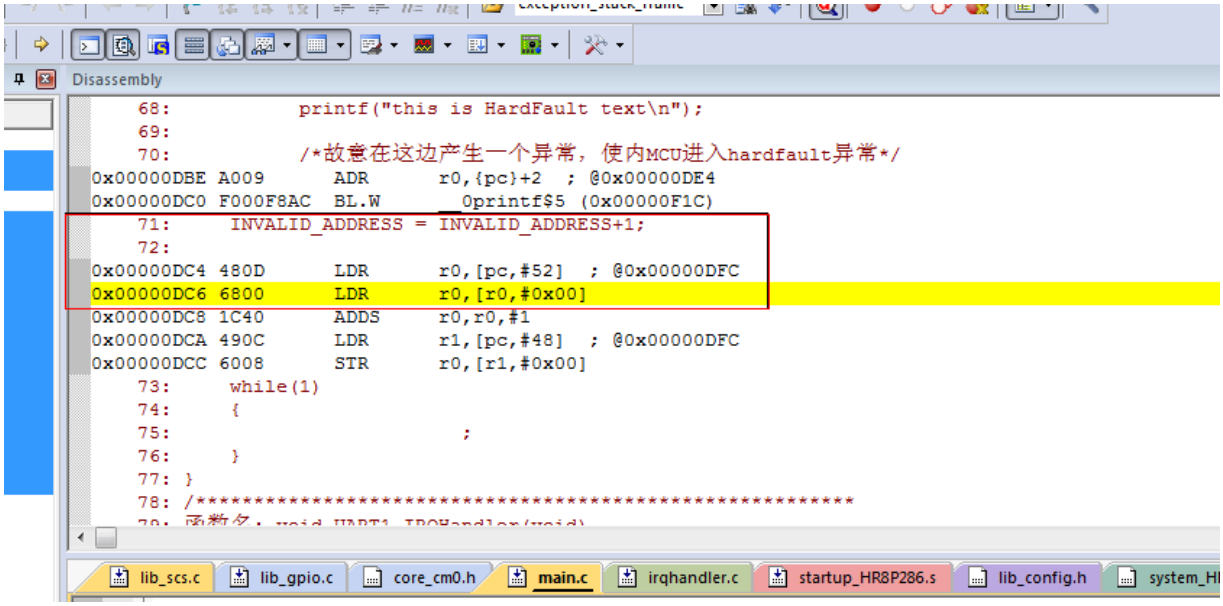

图 2-3　程序反汇编代码

从图 2-3 很容易看出，程序是在执行了 INVALID_ADDRESS 这个变量操作时进了 HardFault 异常，查看这个变量的定义：

```
#define INVALID_ADDRESS (*((volatile unsigned long *)(0xFFFF0001)))
```

显然，INVALID_ADDRESS 变量在 RAM 中的物理地址映射已经超出了芯片的实际 RAM 物理地址范围，故可以确认处理器进入 HardFault 异常正是由于访问存储器越界，违反 MPU 设定的存储器访问规则造成。

附录1 HardFault demo程序

```
#define INVALID_ADDRESS (*((volatile unsigned long *)(0xFFFF0001)))
/************************************************************
函数名: int main(void)
描　述: 主函数
输入值: 无
输出值: 无
返回值: 无
************************************************************/
int main(void)
{
    SystemClockConfig();                                    //配置时钟
    DeviceClockAllEnable();                                 //打开所有外设时钟
    UARTInit();                                             //UART 初始化
    printf("this is HardFault text\n");
    /*故意在这边产生一个异常，使 MCU 进入 HardFault 异常*/
    INVALID_ADDRESS = INVALID_ADDRESS+1;

    while(1)
    {
        ;
    }
}
/************************************************************
函数名: void HardFault_IRQHandler_c(unsigned int * HardFault_args)
描　述: HardFault 异常中断子程序
输入值: 异常压栈指针
输出值: 无
返回值: 无
************************************************************/
void HardFault_IRQHandler_c(unsigned int * HardFault_args)
{
      /*栈帧里面内容：*/
    unsigned int stack_r0;                                  //压栈的 R0
    unsigned int stack_r1;                                  //压栈的 R1
    unsigned int stack_r2;                                  //压栈的 R2
    unsigned int stack_r3;                                  //压栈的 R3
    unsigned int stack_r12;                                 //压栈的 R12
    unsigned int stack_lr;                                  //压栈的 lr
    unsigned int stack_pc;                                  //压栈的 pc
    unsigned int stack_psr;                                 //压栈的 psr
    stack_r0 = ((unsigned int)HardFault_args[0]);
    stack_r1 = ((unsigned int)HardFault_args[1]);
```

```
    stack_r2 = ((unsigned int)HardFault_args[2]);
    stack_r3 = ((unsigned int)HardFault_args[3]);
    stack_r12 = ((unsigned int)HardFault_args[4]);
    stack_lr = ((unsigned int)HardFault_args[5]);
    stack_pc = ((unsigned int)HardFault_args[6]);
    stack_psr = ((unsigned int)HardFault_args[7]);

    printf("R0=%x\n",stack_r0);
    printf("R1=%x\n",stack_r1);
    printf("R2=%x\n",stack_r2);
    printf("R3=%x\n",stack_r3);
    printf("R12=%x\n",stack_r12);
    printf("LR[R14]=%x\n",stack_lr);
    printf("PC[R15]=%x\n",stack_pc);
    printf("PSR=%x\n",stack_psr);
    printf("SCB_SHCSR=%x\n",SCB->SHCSR);
    while(1);
}
```