附录E

# Cortex-M3疑难解答

## E.1 简介

使用CM3，一大令人闹心之处就是会有那么多的faults，不知什么时候就会来一个。来的是哪个，为什么来，如何避免？这连珠炮一般的问题简直成了学习的拦路虎了。

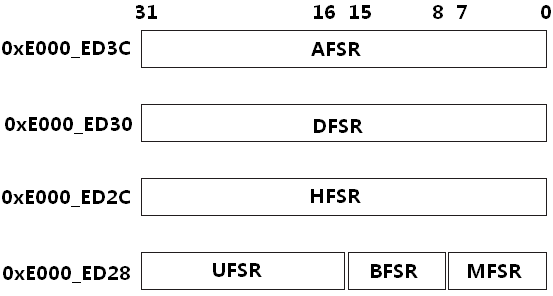
其实，只要我们掌握和理解得深入，不粗心，规范设计和编程，这些faults就成了纸老虎。如果善加利用，甚至还能变害为利——在系统出现故障时由它们为我们提供有用的诊断信息——这才是CM3设计这些faults的初衷。

当fault发生时，首先要弄清楚的就是fault源，下表列出了相关的寄存器

表E.1 CM3中的fault状态寄存器组

|  |  |  |  |
| --- | --- | --- | --- |
| 地址 | 寄存器 | 全名 | 尺寸 |
| 0xE000\_ED28 | MMSR | MemManage fault状态寄存器 | 字节 |
| 0xE000\_ED29 | BFSR | 总线fault状态寄存器 | 字节 |
| 0xE000\_ED2A | UFSR | 用法fault状态寄存器 | 半字 |
| 0xE000\_ED2C | HFSR | 硬fault状态寄存器 | 字 |
| 0xE000\_ED30 | DFSR | 调试fault状态寄存器 | 字 |
| 0xE000\_ED3C | AFSR | 辅助fault状态寄存器 | 字 |

上表的另一可视化视图如下图所示：



图E.1 各fulat状态寄存器的地址组织

因为MMSR, BFSR和UFSR的地址是相连的，所以可以使用按字加载指令一次性地全部读进来。在这种情况下，这个三合一的fault状态亦有一个名字：可配置fault状态寄存器(CFSR)。

另一个提供重要信息的寄存器是什么？它远在天边，近在眼前——人尽皆知的程序计数器PC！进入fault服务例程后，当时的PC值在(SP-0x24)处。因为CM3中有两个堆栈指针，fault服务例程还要判定发生fault时使用的是哪一个堆栈——MSP还是PSP。

进一步地，对于总线fault和存储器管理fault，有时还能精确定位肇事指令的地址——当MMAVALID/BFARVALID位被置位时，即是精确fault。此时，存储器管理fault的地址存储在MMAR中，总线fault的地址则存储在BFAR中。在物理实现上，MMAR与BFAR其实是同一个寄存器，因此同一时刻只能用一个——这是因为同一时刻只能出现一个fault（但是访问地址还是不同的），如表E.2所示。

表E.2 CM3中的fault地址寄存器

|  |  |  |  |
| --- | --- | --- | --- |
| 地址 | 寄存器 | 全名 | 尺寸 |
| 0xE000\_ED34 | MMAR | MemManage fault地址寄存器 | 字 |
| 0xE000\_ED38 | BFAR | 总线fault地址寄存器 | 字 |

最后，在执行fault服务例程时，LR寄存器的值也常常是一个线索，间接反映了发生fault时的情景。如果fault是由无效的EXC\_RETURN值导致的，则进入fault时，LR的值则是上次异常返回时使用的EXC\_RETURN值。Fault服务例程可以据此上报有问题的LR值，从而使开发人员可以检查为何会使用非法的EXC\_RETURN（常常是粗心造成的）。

## E.2 设计Fault服务例程

用于开发阶段的fault服务例程，与用于实际系统中的服务例程，在绝大多数场合下是截然不同的。对于软件开发，fault服务例程应关注于准确及时地上报发生fault时上下文；而实际系统中的fault服务例程则要把这当作是危急关头来处理，它要尽可能地想办法来恢复系统，实在不可救要时可能只有重启。这里我们主要讨论前者，因为后者是比较有技术含量的，而且不同的应用需要不同的策略。

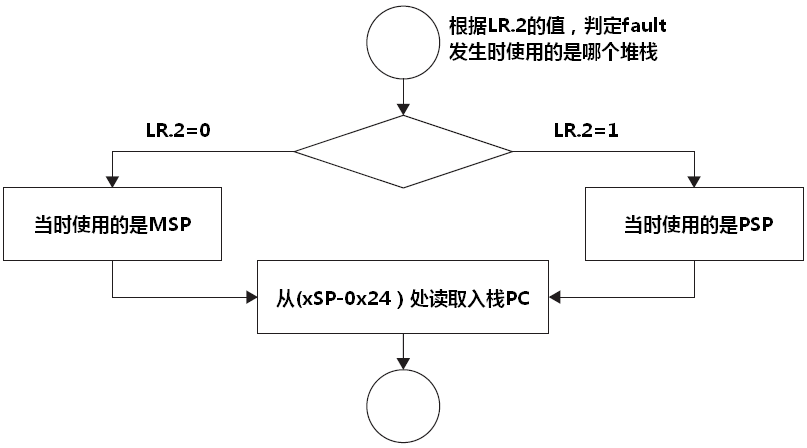
对于比较复杂的软件，通常不直接在fault服务例程中报告与fault相关的状态，而是把它倾倒（专业术语：dump）到一块专用的内存中（主要包括fault状态寄存器，通用寄存器，自动入栈的内容等），接着悬起PendSV。等到PendSV服务例程执行后，再上报问题。这是因为上报过程执行的工作可能比较多，夜长梦多——有可能在上报过程中又不小心触发了其它的fault，使得处理器被锁死。因此先暂记下来，稍后转交PendSV处理。如果软件很简单，则可以斟情简化fault的处理过程，甚至直接在服务例程中上报。

### E.2.1 上报fault状态寄存器

Fault服务例程最基本的工作就是上报fault状态寄存器的值，即上图所列出的那些。

### E.2.2 上报入栈的PC

定位入栈PC的流程如下图所示



图E.2 定位入栈PC的流程图

上图所示的工作流程可以由如下代码来演示：

TST LR, #0x4 ; EXC\_RETURN.2=0?

ITTEE EQ ; 如是为零，则

MRSEQ R0, MSP ; 把MSP加载入R0中

LDREQ R0,[R0,#24] ; 从MSP中获取入栈的PC

MRSNE R0, PSP ; 否则, 就把PSP加载入R0中

LDRNE R0,[R0,#24] ; 从PSP中获取入栈的PC

为了辅助调试，应该同时创建一个反汇编的指令列表（如果使用RVMDK，则自动创建），从而可以容易地定位问题所在。

### E.2.3 上报fault地址寄存器

如果MMARVALID或BFARVALID为1，则可以提供出事时的地址。但要注意的是，当MMARVALID/BFAVRALID被清除后，fault地址寄存器中的值可能被擦除。因此，必须先读BFAR/MMAR，再读BFARVALID/MMARVALID。如果后者为零，则丢弃读出的地址值。最后一步再清除BFARVALID/MMARVALID。

如果先读取了状态位，则有可能在下一步操作前被其它fault异常抢占，这也是一种紊乱危象，有可能导致下述的错误处理序列：

1. 读取BFARVALID/MMARVALID
2. 发现VALID位有效，于是准备读BFAR/MMAR
3. 高优先级异常抢占了fault服务例程，而且它又触发了另一个fault，导致另一个fault服务例程被执行
4. 高优先级fault服务例程清除了BFARVALID/MMARVALID，导致BFAR/MMAR被擦除。
5. 回到先前的fault服务例程，此时再读取BFAR/MMAR时，内容已经丢失了。

可见，后读取VALID位，可以减少这种紊乱危象出现的概率（但如果还没来得及读取BFAR/MMAR就出现了被抢占的情况，则只能看人品了，所以最好第一件事就是读取BFAR/MMAR——译者注）。

清除fault状态位

在fault上报完毕后，一定不要忘记清除FSR中的fault状态位。否则下次再发生fault时，就分不清FSR中的状态位是反映新来的fault，还是反映以前的fault了。而且，如果fault地址有效位没有清除，下次发生fault时，BFAR/MMAR的值就无法更新。

### E.2.4 其它注意事项

我们经常需要在fault服务例程的开始处保存LR的值。然而，如果fault是由于堆栈操作错误导致的，此时再把LR压栈就更添乱了。但我们已经知道，R3-R0以及R12的值已经被保存，因此我们可以在呼叫其它函数之前先把LR的值拷贝到它们中去（事实上在出现堆栈错误时，是无法保证寄存器已经正常入栈了的。此时的问题比较棘手。可能行得通的作法是，在SRAM中专门开出一个块服务于fault服务例程，并把通用寄存器的值保存到那里，但这种办法无法用于嵌套的情况——这已经很钻牛角尖了）。

## E.3 在C中上报入栈的寄存器和各fault状态寄存器

大多数的CM3项目还是以C语言为主的。然而，在C中不方便定位和直接访问堆栈帧（入栈的寄存器）。因为在标准C语言中是不能获取SP指针的。因此，如果使用C来写fault服务例程，最好配合一小段汇编码来获取SP的值，再把该值以一个参数传送给fault上报函数。

译注：在使用MDK自带的ARM编译器时，可以使用\_\_builtin\_frame\_address()函数来获取堆栈帧的地址。在GNU工具中也可以这样做。此法方便，并且可取代上文的通用作法，但降低了编译器间的可移植性。

这个机制与第12章讲的SVC范例相同（“在C中使用SVC”）。下例就以嵌入式汇编的方式来演示。这个例子可以在RealView MDK中编译。

示例程序的第一部分是个汇编封皮。在使用前，要在向量表中的硬fault入口地址项中，填写好该封皮的入口地址。这个封皮代码把正确的堆栈指针值拷贝到R0中，以作为参数来传送给C函数。

// 使用汇编写就的硬fault服务例程

// 该例程提取堆栈帧的位置并且把它传递

// 给C程序

\_\_asm void hard\_fault\_handler\_asm(void)   
{   
 IMPORT hard\_fault\_handler\_c   
 TST LR, #4   
 ITE EQ   
 MRSEQ R0, MSP   
 MRSNE R0, PSP   
 B hard\_fault\_handler\_c   
}

示例程序的第二部分，也是主体部分，使用C语言来写。在这里，我们主要是演示如何访问入栈的寄存器和fault状态寄存器。

// 使用C写就的硬fault服务例程  
// 第一个参数即堆栈帧的位置

void hard\_fault\_handler\_c(unsigned int \* hardfault\_args)

{

unsigned int stacked\_r0;

unsigned int stacked\_r1;

unsigned int stacked\_r2;

unsigned int stacked\_r3;

unsigned int stacked\_r12;

unsigned int stacked\_lr;

unsigned int stacked\_pc;

unsigned int stacked\_psr;

stacked\_r0 = ((unsigned long) hardfault\_args[0]);

stacked\_r1 = ((unsigned long) hardfault\_args[1]);

stacked\_r2 = ((unsigned long) hardfault\_args[2]);

stacked\_r3 = ((unsigned long) hardfault\_args[3]);

stacked\_r12 = ((unsigned long) hardfault\_args[4]);

stacked\_lr = ((unsigned long) hardfault\_args[5]);

stacked\_pc = ((unsigned long) hardfault\_args[6]);

stacked\_psr = ((unsigned long) hardfault\_args[7]);

printf ("[Hard fault handler]\n");

printf ("R0 = %x\n", stacked\_r0);

printf ("R1 = %x\n", stacked\_r1);

printf ("R2 = %x\n", stacked\_r2);

printf ("R3 = %x\n", stacked\_r3);

printf ("R12 = %x\n", stacked\_r12);

printf ("LR = %x\n", stacked\_lr);

printf ("PC = %x\n", stacked\_pc);

printf ("PSR = %x\n", stacked\_psr);

printf ("BFAR = %x\n", (\*((volatile unsigned long \*)(0xE000ED38))));

printf ("CFSR = %x\n", (\*((volatile unsigned long \*)(0xE000ED28))));

printf ("HFSR = %x\n", (\*((volatile unsigned long \*)(0xE000ED2C))));

printf ("DFSR = %x\n", (\*((volatile unsigned long \*)(0xE000ED30))));

printf ("AFSR = %x\n", (\*((volatile unsigned long \*)(0xE000ED3C))));

exit(0); // terminate

return;

}

请注意：如果发生了堆栈溢出或其它错误，使SP指向了无效的存储空对空区域，则上段代码会失能。在大多数情况下，这种错误都会影响C代码，因为所有C代码都需要堆栈。

## E.4 理解发生fault的原因

在收集到所需的信息后，就需要分析问题了。表E.3-表E.7列出了导致faults的典型原因。

表E.3 MemManage fault状态寄存器提供的讯息

|  |  |
| --- | --- |
| 位 | 可能的原因 |
| MSTKERR | 入栈时发生错误（异常响应序列开始时）   1. 堆栈指针的值被破坏 2. 堆栈容易过大，已经超出MPU允许的region范围 |
| MUNSTKERR | 出栈时发生错误（异常响应序列终止时）。入栈时没有发生错误，出栈时却出错，总令人有些匪夷所思，可能的原因是   1. 异常服务例程破坏了堆栈指针 2. 异常服务例程更改了MPU配置 |
| DACCVIOL | 内存访问保护违例。这是MPU发挥作用的体现。常常是用户应用程序企图访问特权级region所致 |
| IACCVIOL | 1. 内存访问保护违例。常常是用户应用程序企图访问特权级region。在这种情况下，入栈的PC给出的地址，就是产生问题的代码之所在 2. 跳转到不可执行指令的 regions 3. 异常返回时，使用了无效的EXC\_RETURN值 4. 向量表中有无效的向量。例如，异常在向量建立之前就发生了，或者加载的是用于传统ARM内核的可执行映像 5. 在异常处理期间，入栈的PC值被破坏了 |
|  |  |

表E.4 总线fault状态寄存器提供的讯息

|  |  |
| --- | --- |
| 位 | 可能的原因 |
| STKERR | （自动）入栈期间出错   1. 堆栈指针的值被破坏 2. 堆栈用量太大，到达了未定义存储器的区域 3. PSP未经初始化就使用 |
| UNSTKERR | （自动）出栈期间出错。如果没有发生过STKERR，则最可能的就是在异常处理期间把SP的值破坏了 |
| IMPRECISERR | 与设备之间传送数据的过程中发生总线错误。可能是因为设备未经初始化而引起；或者在用户级访问了特权级的设备，或者传送的数据单位尺寸不能为设备所接受。此时，有可能是LDM/STM指令造成了非精确总线fault。 |
| PRECISERR | 在数据访问期间的总线错误。通过BFAR可以获取具体的地址。发生fault的原因同上。 |
| IBUSERR | 同MemManage fault中的IACCVIOL |

表E.5 用法fault状态寄存器提供的讯息

|  |  |
| --- | --- |
| 位 | 可能的原因 |
| DIVBYZERO | 当DIV\_0\_TRP置位时则发生了除数为零的情况。引发此fault的指令可以从入栈的PC读取 |
| UNALIGNED | 当UNALIGN\_TRP置位时发生未对齐访问。引发此fault的指令可以从入栈的PC读取 |
| NOCP | 企图执行一个协处理器指令。引发此fault的指令可以从入栈的PC读取 |
| INVPC | 1. 异常返回时使用了无效的EXC\_RETURN，例如 2. 当EXC\_RETURN=0xFFFF\_FFF1时却要返回线程模式 3. 当EXC\_RETURN=0xFFFF\_FFF9时却要返回handler模式 4. 无效的异常活动状态，例如 5. 当前异常的活动状态已经清除了，却在此时执行异常返回。往往是因为滥用VECTCLRACTIVE或清除了SHCSR中活动状态所致 6. 在尚有异常的活动位置位时，却要返回线程模式 7. 由于堆栈指针错误导致了IPSR的值不正确。对于INVPC fault，入栈的PC指出了该fault服务例程在何处抢占了其它代码。这个问题往往是由比较隐晦的程序错误造成的，欲详细调查该问题的原因，最好使用ITM的跟踪功能。 8. ICI/IT位对当前指令无效。当LDM/STM指令被异常打断后，在异常服务例程中又更改了入栈的PC。结果在中断返回时，非零的ICI位段作用到了不使用ICI位段的指令上。如果是其它原因破坏了PSR的值，也可能导致此fault。 |
| INVSTATE | 1. 加载到PC中的跳转地址值是偶数（LSB=0）。通过检查入栈PC的值，一下子就可以查出该问题。 2. 向量地址的LSB=0，诊断方法同上。 3. 入栈的PSR在异常处理过程中被破坏，使得在返回时内核尝试进入ARM状态。 |
| UNDEFINSTR | 1. 使用了CM3不支持的指令 2. 代码段中的数据被破坏 3. 连接时加载了ARM目标码。请检查编译阶段的设置 4. 指令对齐的问题。例如，在使用GNU工具链时，忘记了在.ascii后使用.align，就有可能导致下一条指令没有对齐 |

表E.6 硬fault状态寄存器提供的讯息

|  |  |
| --- | --- |
| 位 | 可能的原因 |
| DEBUGEVF | 因调试事件导致的fault   1. 断点/观察点事件 2. 在硬fault服务例程的执行过程中，没有使能监视器异常（MON\_EN=0）也没有使能停机调试（C\_DEBUGEN=0），却执行了BKPT指令。缺省时，有些C编译器可能会在半主机代码中使用BKPT指令。 |
| FORCED | 这是fault“上访”的情况   1. 试图在SVC/监视器服务例程中执行SVC/BKPT，或者在其它拥有相同或更高优先级的服务例程中执行SVC/BKPT。 2. 发生了fault，但是它的服务例程被除能 3. 发生了fault，但是当前处理器在响应同级或更高优先级的异常 4. 发生了fault，但是它被掩蔽了 |
| VECTBL | 取向量失败，   1. 在取向量过程中发生总线fault 2. 向量表偏移量设置有误 |

表E.7 调试fault状态寄存器提供的讯息

|  |  |
| --- | --- |
| 位 | 可能的原因 |
| EXTERNAL | EDBGREQ信号置为有效 |
| VCATCH | 发生了向量抓捕事件 |
| DWTTRAP | 发生了DWT观察点事件 |
| BKPT | 1. 执行了BKPT指令 2. FPB单元产生了断点事件 |