**STM32 Cortex-M3 Hard Fault**

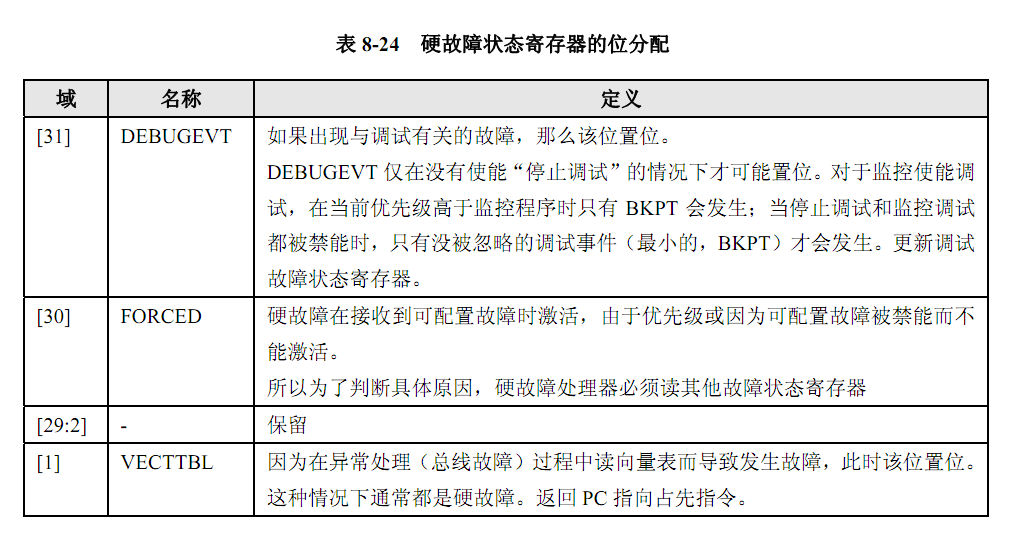
Hard fault (硬错误,也有译为硬件错误的)是在STM32（如无特别说明，这里的STM32指的是Cortex-M3的核）上编写程序中所产生的错误，造成Hard Fault错误的原因也是最为纷繁复杂的。由于能导致该错误的原因很多，所以一但出现，比较难找到其原因。网上有很多类似的这种方法，现在我将其稍加整理，并结合我曾经遇到过的问题，详细说明。

硬fault 是总线fault、存储器管理fault 以及用法fault 上访的结果。如果这些fault 的服务例程无法执行，它们就会成为“硬伤”——上访（escalation）成硬fault。另外，在取向量（异常处理是对异常向量表的读取）时产生的总线fault，也按硬fault 处理。在NVIC 中有一个硬fault 状态寄存器（HFSR），它指出产生硬fault 的原因。如果不是由于取向量造成的，则硬fault 服务例程必须检查其它的fault 状态寄存器，以最终决定是谁上访的。

**1 寄存器描述**

首先查看硬故障寄存器，判别原因。





对于调试故障，有个调试故障寄存器，在0xE000ED30处，有详细介绍，不做探讨；

对于取中断发生的，有两类原因，一是在取向量过程中发生总线 fault，二是向量表偏移量设置有误。

本文重点介绍位30所示的，上访类错误。

这样Fault类异常有了三类，用法错误，存储管理错误，总线错误。



对于这些寄存器详尽的描述，见权威指南。

# 2 确定发生错误的地方

## 2.1 查找出错原因

Cortex-M3有双堆栈功能，在带有操作系统时，一般都会使用。在Keil软件使用JTAG调试为例，系统的启动文件中，将断点打在下面4个地方。

HardFaultException

B HardFaultException

MemManageException

B MemManageException

BusFaultException

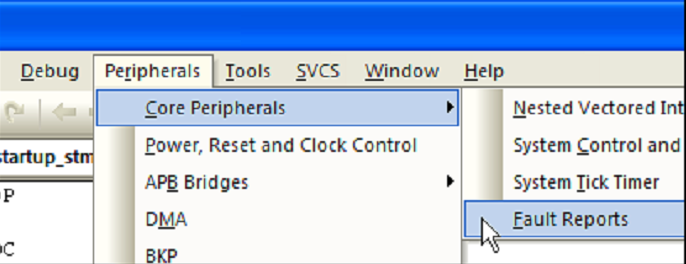
B BusFaultException

UsageFaultException

B UsageFaultException

程序跑飞以后，就会停在上面的4个断点的一个地方。可以通过两种方式查找原因。

第一种，在KEIL软件下，利用软件提供的功能查找故障原因。



在点出的窗口中，可以大体确定是哪个寄存器、什么原因造成了Hard Fault。

第二种，通过在内存观察窗口，直接输入上面那些寄存器的值来确定，通过观看寄存器那个位被置1了，确定出错原因。

## 2.2 确定出错地方

然后查看左侧寄存器栏中Banked确定现在使用的是那个堆栈，MSP或者是PSP，确定以后，在内存查看窗口，输入堆栈的地址，以这个地址开始的8个32位数值，应该依次是R0,R1,R2,R3,R12,R14,R15,XPSR的数值，据此判定你的堆栈地址是不是对的（有时需要考虑堆栈的增长方向）。**R14，R15的地址就是我们出错的代码所在的地址**，需要在这个地址基础上，首先偶数对齐，然后向上减去8个字节。

需要考虑的是，在使用MSP的时候，有出错的地方并不一定在R14，R15处，而是在XPSR往后的第二个地址处，在这个附近查找，排除故障。

# 3 两个例子

下面就我之前碰到过的，举例说明，这两个例子分析出结果后，会觉得很简单，但是查找原因的过程有点费劲。

## 3.1 memcpy内存拷贝函数引发

总线故障寄存器中IMPERCISERR位，标示不精确的数据总线访问错误，权威指南中对此有详尽的说明，“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指令造成的”。

Memcpy函数的原因是这样的void \*memcpy(void \*dest, const void \*src, size\_t n)，其中src是源地址，dest是目的地址，n是要拷贝的字节长度。KEIL自带的函数中并不检查这三个参数是否有效，我所开发的程序中，源地址和目的地址都在外存（外部扩展的内存，本次大小是4M）中，假设size的大小是0xFFFF FFFF，这样的数值非常的大，单纯的拷贝都需要10多秒。程序中定义了很多的变量都在外存，这个拷贝函数所在的任务优先级比较低，可能被中断或者其它的任务打断。

我调试程序的时候，首先是发生在了中断的地方，外存数组地址到了0x21FF 2200,原来定义在6802 1000，加起来立刻超出了外存大小。修改中断，最终确定是传入的参数n太大了，直接是0xFFFF FFFF，这样memcpy函数会在这里陷入死循环，一直到外存耗尽，地址再增加，找不到外存地址了，然后触发Hard Fault。

## 3.2 滥用临界区

程序中的一些关键代码，有时候需要在临界区中执行，但是临界区若使用不当，则也会造成错误。

OS\_ENTER\_CRITICAL();

。。。。。。。。。。。。。。。。。。

。。。。。。。。。。。。。。。。。。

OS\_EXIT\_CRITICAL();

#define OS\_ENTER\_CRITICAL() {cpu\_sr = OS\_CPU\_SR\_Save();}

OS\_CPU\_SR\_Save

MRS R0, PRIMASK ;保存全局中断标志 ;

CPSID I ;关中断

BX LR

将全局中断标志保存到R0中，此时R0是0，CPSID I则执行关中断命令，此时PRIMASK是1。

#define OS\_EXIT\_CRITICAL() {OS\_CPU\_SR\_Restore(cpu\_sr);}

OS\_CPU\_SR\_Restore

MSR PRIMASK, R0 ;恢复全局中断标志

BX LR

将R0放入全部中断寄存器中，则允许所有中断了。

程序中如何保护R0的，细看汇编发现，实际上在执行关中断后，将R0保存到了sp+8处，开中断时再取出来，这样才保证了不会被修改。

STR r0,[sp,#0x08]tPendTimes = 0;

同时，开中断， LDR r0,[sp,#0x08]，则从sp+8处取出来，保存到R0中。

临界区中的代码完成如下内容：

netconn\_write(tradeconn,g\_u8TcpSendBuf,l\_u32CodeSendLen,NETCONN\_COPY);

调用TCPIP\_APIMSG(&msg);，

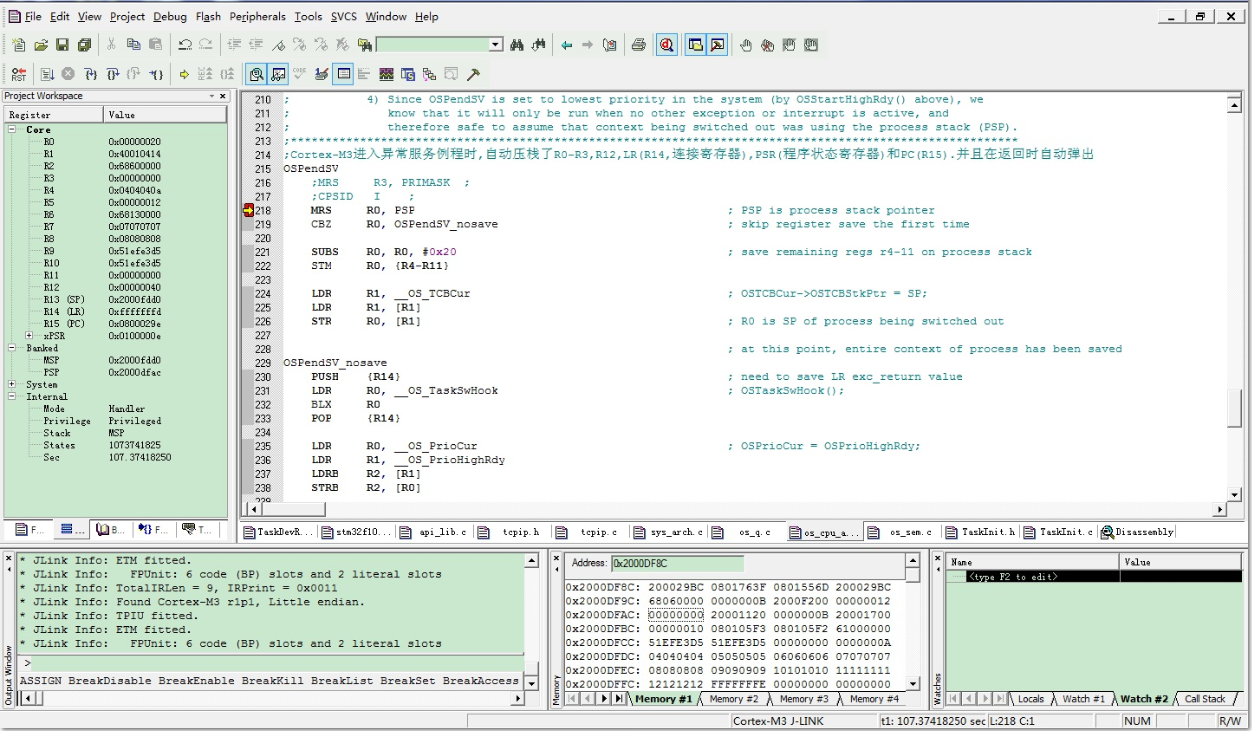
sys\_mbox\_post(mbox, &msg);

OSQPost(mbox->hMBox, msg)发送消息，OS\_EventTaskRdy函数修改线程的状态，使OSTCBStatPend变为等待完毕；

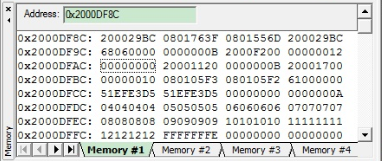
此时**若协议栈线程优先级高于当前任务**，则会触发任务调度，悬起OSPendSV，但是由于关闭了中断，即使在调用OS\_ENTER\_CRITICAL()后，也无法打开中断，故不能执行中断，任务无法切换。

同理，调用sys\_arch\_sem\_wait(apimsg->msg.conn->op\_completed, 0);，也无法阻塞自身，执行任务调度，程序在临界区里面变成了单线程在跑。

一直等待代码执行完毕开中断后，悬起的软中才能执行，本来应该在发送消息和等待消息处执行任务切换的，现在只能等待临界区执行完毕后，才能执行任务切换中断。此刻的PSP是0x2000DFAC，临界区的那段代码我们也有压栈操作，即是0x2000 DFAC后面的内容也是我们需要的，如下图所示。



原来的内容是这样的，如下图所示：



此时在OSPendSV中，执行如下语句

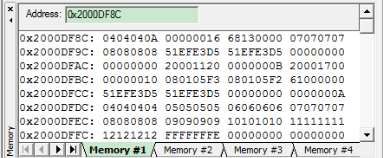
MRS R0, PSP ; PSP is process stack pointer

CBZ R0, OSPendSV\_nosave ;

SUBS R0, R0, #0x20 ; save remaining regs r4-11 on process stack

STM R0, {R4-R11}

从PSP-32个字节处开始，保存R4到R11这8个寄存器32个字节，则原来的内容都被覆盖了，而这些内容正好是我们需要的。被修改后的截图如所示，原来的内容被改成R4到R11这几个寄存器的值。



其中从0801556D变成了68130000，协议栈线程如下执行。

msg->msg.apimsg->function(&(msg->msg.apimsg->msg));

函数的地址变成了6813 0000，而6813 0000，是我们的外存，

在这里执行代码0x68130006 F63A07E1 DCD 0xF63A07E1 ; ? Undefined

最终是这句话，触发了Hard fault。

## 3.3 运行中记录出错位置

以3.2为例子，进行简单的反推。启动文件中的Hard中断处理一般如下所示，即让程序陷入这个死循环。

HardFaultException

; B HardFaultException

现在我们要在记录重要数据，即此刻系统的运行情况，主要包括：此刻堆栈情况、以及R0等8个寄存器的值、相关Hard硬件寄存器的值，若是任务引发的，还要记录任务的ID号，因此修改这个异常处理函数。

HardFaultException

TST LR, #4 ;将LR的值与4按位相与

ITE EQ //若为0则是MSP，否则是PSP

MRSEQ R0, MSP

MRSNE R0, PSP

B hard\_fault\_handler\_c //这个是C语言编写的函数

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

{

unsigned int stacked\_r0,stacked\_r1,stacked\_r2,stacked\_r3;

unsigned int stacked\_r12,stacked\_lr, stacked\_pc, 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]);

sprintf((char\*)g\_cDataBuf,"[Hard fault handler]\n");

Usart232SendStr(g\_cDataBuf);

sprintf((char\*)g\_cDataBuf,"The task pri id = 0x%0.8x\n", OSPrioCur); //任务ID号

Usart232SendStr(g\_cDataBuf);

sprintf((char\*)g\_cDataBuf,"SP = 0x%0.8x\n", hardfault\_args); //堆栈地址

Usart232SendStr(g\_cDataBuf);

sprintf((char\*)g\_cDataBuf,"R0 = 0x%0.8x\n", stacked\_r0);

Usart232SendStr(g\_cDataBuf);

sprintf((char\*)g\_cDataBuf,"R1 = 0x%0.8x\n", stacked\_r1);

Usart232SendStr(g\_cDataBuf);

sprintf((char\*)g\_cDataBuf,"R2 = 0x%0.8x\n", stacked\_r2);

Usart232SendStr(g\_cDataBuf);

sprintf((char\*)g\_cDataBuf,"R3 = 0x%0.8x\n", stacked\_r3);

Usart232SendStr(g\_cDataBuf);

sprintf((char\*)g\_cDataBuf,"R12 = 0x%0.8x\n", stacked\_r12);

Usart232SendStr(g\_cDataBuf);

sprintf((char\*)g\_cDataBuf,"LR = 0x%0.8x\n", stacked\_lr);

Usart232SendStr(g\_cDataBuf);

sprintf((char\*)g\_cDataBuf,"PC = 0x%0.8x\n", stacked\_pc);

Usart232SendStr(g\_cDataBuf);

sprintf((char\*)g\_cDataBuf,"PSR = 0x%0.8x\n", stacked\_psr);

Usart232SendStr(g\_cDataBuf);

exit(0); // terminate

return;

}

以3.2为例，发生异常后，串口的输出入下所示：

[Hard fault handler]

The task pri id = 0x00000014 //任务优先级是**20**

SP = 0x200077d8 //当前任务的堆栈地址是**0x2000 77D8**

R0 = 0x2000dfa0

R1 = 0x68130000

R2 = 0x2000df9c

R3 = 0x20002100

R12 = 0x00000001

**LR = 0x0801c7fb //分析得出，这个地址就是出错的地方**

PC = 0x68130000

PSR = 0x00000000

此时需要借助**map文件**分析，map文件中得出对应的代码和数据位置。

tcpip\_thread **0x0801c7ad** Thumb Code 190 tcpip.o(i.tcpip\_thread)

i.tcpsvr\_accept\_20 **0x0801c874** Section 64 ftpmanage.o(i.tcpsvr\_accept\_20)

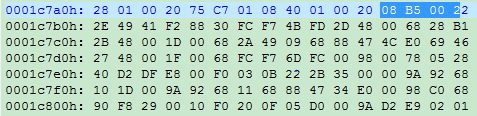
0x0801 c7fb应该在tcpip文件中的tciip\_thread函数里。

T\_LWIP\_THREAD\_STK 0x20007000 Data 2048 sys\_arch.o(.bss)

rsuPib 0x20007800 Data 32 para.o(.bss)

堆栈空间是**0x2000 77D8，是在**T\_LWIP\_THREAD\_STK这个栈空间里，这也是协议栈任务的堆栈空间，证明判断的任务优先级为20是正确的。

从0x0801 C7AD处开始的16进制文件如下图所示，再将汇编文件列出（**需要KeiL生成**）。



tcpip\_thread PROC

;;;232 static void

;;;233 tcpip\_thread(void \*arg)

000000 b508 PUSH {r3,lr} //开始

;;;234 {

;;;235 struct tcpip\_msg \*msg;

;;;236 LWIP\_UNUSED\_ARG(arg);

;;;237

;;;238 #if IP\_REASSEMBLY

;;;239 sys\_timeout(IP\_TMR\_INTERVAL, ip\_reass\_timer, NULL);

;;;240 #endif /\* IP\_REASSEMBLY \*/

;;;241 #if LWIP\_ARP

;;;242 sys\_timeout(ARP\_TMR\_INTERVAL, arp\_timer, NULL);

000002 2200 MOVS r2,#0

000004 492e LDR r1,|L11.192|

000006 f2413088 MOV r0,#0x1388

00000a f7fffffe BL sys\_timeout

;;;243 #endif /\* LWIP\_ARP \*/

;;;244 #if LWIP\_DHCP

;;;245 sys\_timeout(DHCP\_COARSE\_TIMER\_MSECS, dhcp\_timer\_coarse, NULL);

;;;246 sys\_timeout(DHCP\_FINE\_TIMER\_MSECS, dhcp\_timer\_fine, NULL);

;;;247 #endif /\* LWIP\_DHCP \*/

;;;248 #if LWIP\_AUTOIP

;;;249 sys\_timeout(AUTOIP\_TMR\_INTERVAL, autoip\_timer, NULL);

;;;250 #endif /\* LWIP\_AUTOIP \*/

;;;251 #if LWIP\_IGMP

;;;252 sys\_timeout(IGMP\_TMR\_INTERVAL, igmp\_timer, NULL);

;;;253 #endif /\* LWIP\_IGMP \*/

;;;254 #if LWIP\_DNS

;;;255 sys\_timeout(DNS\_TMR\_INTERVAL, dns\_timer, NULL);

;;;256 #endif /\* LWIP\_DNS \*/

;;;257

;;;258 if (tcpip\_init\_done != NULL) {

00000e 482d LDR r0,|L11.196|

000010 6800 LDR r0,[r0,#0] ; tcpip\_init\_done

000012 b128 CBZ r0,|L11.32|

;;;259 tcpip\_init\_done(tcpip\_init\_done\_arg);

000014 482b LDR r0,|L11.196|

000016 1d00 ADDS r0,r0,#4

000018 6800 LDR r0,[r0,#0] ; tcpip\_init\_done\_arg

00001a 492a LDR r1,|L11.196|

00001c 6809 LDR r1,[r1,#0] ; tcpip\_init\_done

00001e 4788 BLX r1

|L11.32|

;;;260 }

;;;261

;;;262 LOCK\_TCPIP\_CORE();

;;;263 while (1) { /\* MAIN Loop \*/

000020 e04c B |L11.188|

|L11.34|

;;;264 sys\_mbox\_fetch(mbox, (void \*)&msg);

000022 4669 MOV r1,sp

000024 4827 LDR r0,|L11.196|

000026 1f00 SUBS r0,r0,#4

000028 6800 LDR r0,[r0,#0] ; mbox

00002a f7fffffe BL sys\_mbox\_fetch

;;;265 switch (msg->type) {

00002e 9800 LDR r0,[sp,#0]

000030 7800 LDRB r0,[r0,#0]

000032 2805 CMP r0,#5

000034 d240 BCS |L11.184|

000036 e8dff000 TBB [pc,r0]

00003a 030b DCB 0x03,0x0b

00003c 222b3500 DCB 0x22,0x2b,0x35,0x00

;;;266 #if LWIP\_NETCONN

;;;267 case TCPIP\_MSG\_API:

;;;268 //if(msg->msg.apimsg->msg.conn == NULL)

;;;269 // break;

;;;270 LWIP\_DEBUGF(TCPIP\_DEBUG, ("tcpip\_thread: API message %p\n", (void \*)msg));

;;;271 msg->msg.apimsg->function(&(msg->msg.apimsg->msg));

000040 9a00 LDR r2,[sp,#0]

000042 6892 LDR r2,[r2,#8]

000044 1d10 ADDS r0,r2,#4

000046 9a00 LDR r2,[sp,#0]

000048 6892 LDR r2,[r2,#8]

00004a 6811 LDR r1,[r2,#0]

00004c 4788 BLX r1

;;;272 break;

00004e e034 B |L11.186| //0x0801 c7fb对应的代码

;;;273 #endif /\* LWIP\_NETCONN \*/

;;;274

;;;275 case TCPIP\_MSG\_INPKT:

;;;276 LWIP\_DEBUGF(TCPIP\_DEBUG, ("tcpip\_thread: PACKET %p\n", (void \*)msg));

;;;277 #if LWIP\_ARP

;;;278 if (msg->msg.inp.netif->flags & NETIF\_FLAG\_ETHARP) {

000050 9800 LDR r0,[sp,#0]

从代码看地址对应是00004e e034 B |L11.186| ，即switch分支的break语句，但是实际应该是上面的那句，BLX r1，而此时R1的值是

R1 = 0x68130000，即跳转到6813 0000处执行，与在3.2的分析是一样的。

这也只能判断出出错的位置，原因还是需要仿真调试，才能找到。

## 3.4 总结

发生Hard Fault以后，意味着程序跑飞了，有的原因是很简单的，但是有的需要仔细分析，以上两个例子应该都算比较简单的。以第一个例子来说，若从首次定位来看是在中断里，中断里变量的数值太大，超出了外存的大小，但是这个值为什么会这么大？正常中断接收数据不会变成有0x21FF 2200这么多字节的，此时就需要考虑是别的地方踩到了此处的内存，导致取出来的数据一下子变成了这么大。然后逐步定位，才能找出真正的原因。

# 4 参考文献

网上有几篇非常不错的文章，可以看看，加深理解。

[1] Cortex-M3 权威指南，Joseph Yiu 著，宋岩 译。（在书的附录E中对Fault类异常有非常详尽的介绍）

[2] Cortex-M3技术参考手册，周立功。（在书的P89到96页对以上所介绍的寄存器有很详细的描述）

[3] Application Note 209,Using Cortex-M3 and Cortex-M4 Fault Exceptions. KEIL Tools by ARM.（这个是KEIL软件下使用的说明，介绍的例子可以一看）

[4] 教你如何找到导致程序跑飞的指令,,http://blog.sina.com.cn/ifreecoding.（博主在文章里一步步讲述，非常清晰，让人一看就明白）