一个 STM32 芯片异常复位之案例分析

前言

本篇主要是介绍一种处理问题的思路，即当我们在做STM32应用开发过程中，遇到芯片异常复位，或者进入了异常处理时，如何通过集成开发环境，如IAR，KEIL等查看相应的ARM内核寄存器，定位出应用软件产生异常的地方！

问题描述

某STM32用户反馈，当使用STM32L4芯片的时候，程序运行一段时间后，会忽然复位。复位后程序继续运行，但是还会继续复位，原因不详！

问题分析

针对于此类问题，我们可以按照一个统一的思路去处理。分析本案例的大致步骤如下：

1、初步确定复位的原因，是硬件复位，如外部NRST被拉低，还是软件复位，包括软件直接调用复位，或者看门狗复位，还是低功耗模式如standby模式被唤醒时产生中断；

2、查看复位状态寄存器了解复位大方向，然后做进一步得拆解分析。

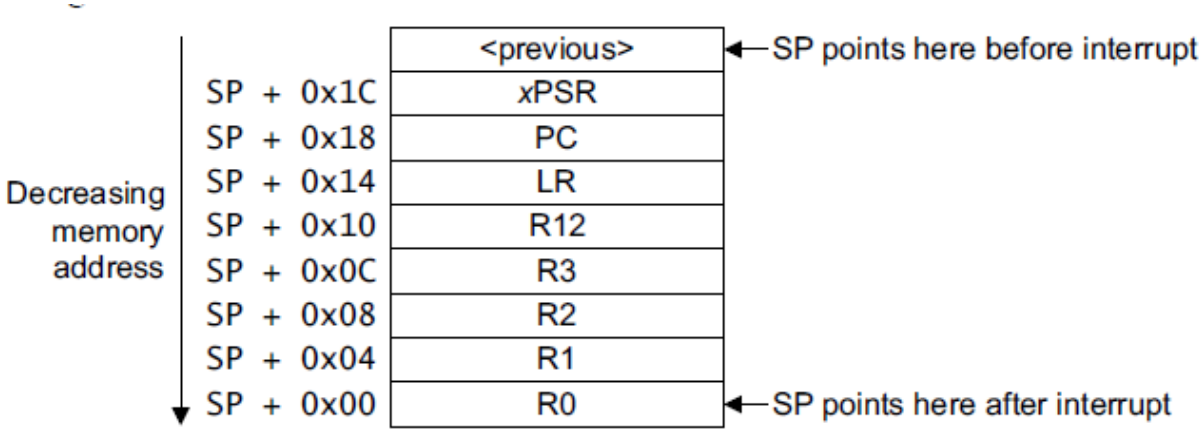
3、目前客户项目的复位原因是因为看门狗复位，即客户使用了IWDG，但由于某种原因没有及时喂狗，导致IWDG超时复位。初步怀疑由于客户软件的问题，程序跑飞，进入异常处理。

因为客户的异常处理函数中并没有做 任何动作，导致独立看门狗IWDG复位。基于此，我们先关闭IWDG，然后在所有的异常处理中，先加入死循环并打上断点，对异常原因进行捕捉。

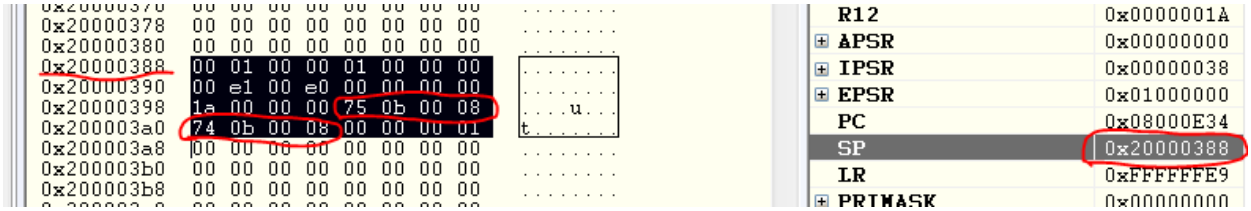
```
void HardFault_Handler(void)
{
  /* Go to infinite loop when Hard Fault exception occurs */
  while (1)
  {
  }
}
void MemManage_Handler(void)
{
  /* Go to infinite loop when Memory Manage exception occurs */
  while (1)
  {
  }
}
```

```
void BusFault_Handler(void)
{
  /* Go to infinite loop when Bus Fault exception occurs */
  while (1)
  {
  }
```

4、正如我们所猜测，的确是由于程序跑飞导致。程序停在了void HardFault_Handler(void) 。
通过查看 SP 以及回溯栈里面的内容，找到了对应的 LR，具体方法如下：



当中断产生时，按照上图所示的顺序进行压栈，同时栈指针SP--，即： R0, R1, R2, R3, R12, LR, PC, xPSR。



如上图所示，当产生异常时，如果call stack窗口显示不出来的话，只能根据core的寄存器手动回溯栈，以找到出错时的指针。根据ARM core的说明，SP+6，即红框的部分，为中断处理后LR和PC，据此可以追溯函数异常时的位置！

5、根据出错时的PC和LR，发现是浮点运算的函数，初步判断是因为浮点运算导致，比如没有对齐导致的Hardfault，但实际检查发现，并不是浮点运算的问题！

6、问题一时陷入了僵局。但有一点是确定的，是因为栈的区域被异常覆盖或者改写导致产生hard fault，

7、由于问题可以稳定复现，采取逐个排除法最终发现了问题的所在：
当把一个局部数组变量改为全局数组时，问题消失！
由于局部数组变量是保存在栈当中，所以怀疑是对这个局部数组变量使用不当导致了栈被覆盖或者改写！
追查这个局部变量数组：

```
void String_Converted_from_FlashIndex ( uint32_t add, uint8_t ypos )
{
 uint8_t i, temp_str[STRING_SIZE];

 Ex_Flash_DataRead(add + Multi_Ling.Spec[Screen_Oper.Multi_Ling_Index].String_Addr , temp_str, STRING_SIZE);

 void Ex_Flash_DataRead(uint32_t address,uint8_t *ReadBuff, uint32_t numberOfBytes)
 {
   uint8_t Address8bit[3];

void SPI_ReceiveData8bit (SpiPort Port, uint8_t* pData, uint16_t length)
{
static uint8_t DummyByte = 0;
switch(Port)
  {
  case Spi1:
    DMA1_Ch2_Set_Source((uint32_t*)pData, length, TRUE); //Set Rx Buffer
    DMA1_Ch3_Set_Source((uint32_t*)(&DummyByte), length, FALSE); //Dum
    SPI_ClearSpi1_HW_RxBuffer();
    SPI_EnableSpi1_DMA_Buffer();
    DMA1_Ch2_Enable();
    DMA1_Ch3_Enable();                    //Send dummy bytes
    break;

  case Spi2:
    DMA1_Ch4_Set_Source((uint32_t*)pData, length, TRUE); //Set Rx Buffer
```

经检查发现，这个原先是8bit的局部变量的数组，在最后被强制转换成了uint32_t *类型的指针，由于是指针， 在对其进行++或--操作时，都是按照4字节宽带操作的，这就相当于扩大了4倍，覆盖了后面的栈的内容， 导致了程序跑飞！

小结

当芯片异常复位或者进入异常处理(如Hard fault, Mem Manage, Bus fault等)时，首先考虑的是，如何快速的复现这个问题，当问题被稳定复现的时候，可以通过调试工具在异常处理的地方打上断点停留，这样就可以获取到栈指针SP，通过SP去看栈里面的内容去回溯栈。当然，如果栈的内容被无端改写时，栈里面的内容，如保存的LR就没有太大的参考意义。不过，可以通过观察栈里面的内容，去估测是哪个模块或者函数异常修改了栈的内容，进而定位最终的问题源！

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。所有其他产品或服务名称均为其各自所有者的财产。

本文档中的信息取代本文档所有早期版本中提供的信息。