# 嵌入式系统实验报告



|  |  |
| --- | --- |
| 实验名称： | Lab5 Cortex-M应用编程——异常处理 |
| 姓 名： | 江姝潼 |
| 学 号： | 2019211653 |
| 学 院(系)： | 计算机学院 |
| 专 业： | 网络工程 |
| 指导教师： | 戴志涛、刘健培 |

2021年 12 月 4 日

# 实验环境

* ST-Link 仿真器
* KeiluVision5 MDK集成开发软件
* PC机Window10 (64bit)

# 实验要求

Cortex-M 处理器在发生异常时，会进入异常处理程序。在生产环境下，在无法连接调试器的情况下，可以通过在异常处理程序中实现一个 coredump（核心转储）程序，将发生异常时程序的状态（内核寄存器值、堆栈、关键寄存器与数据结构等）输出到串口或者保存到文件，便于事后分析异常原因。

本实验要求实现一个简单的 coredump 程序，当程序发生异常时，可以在异常程序中将发生异常前 Cortex -M 处理器的核内寄存器的值用 printf 打印出来。

# 实验原理

* 当程序触发一个内存不对齐访问的异常（UNALIGN\_TRP），此异常会进入 HardFault\_Handler 处理，可以通过重写 HardFault\_Handler 以替换默认的汇编实现。
* 当处理器识别出异常时，会通过特定序列的行为来进行自动响应：

1）压栈：将寄存器 R0～R3 和 R12 的内容、返回地址、PSR 和链接寄存器

LR 的内容压栈。

2）在进行压栈操作的同时，处理器识别异常，将其对异常的识别结果作为

PSR 中的异常号记录下来，然后使用该数字从异常向量表（vector table）中定位和拷贝对应异常处理程序的地址到程序计数器（PC）中。

3）LR 寄存器装载除了最低的 4 位外全 1 的序列，最低 4 位记录指向异常的

处理器模式（线程模式或者异常处理模式）。

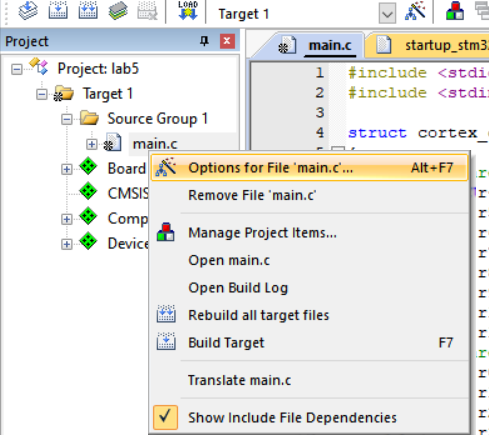
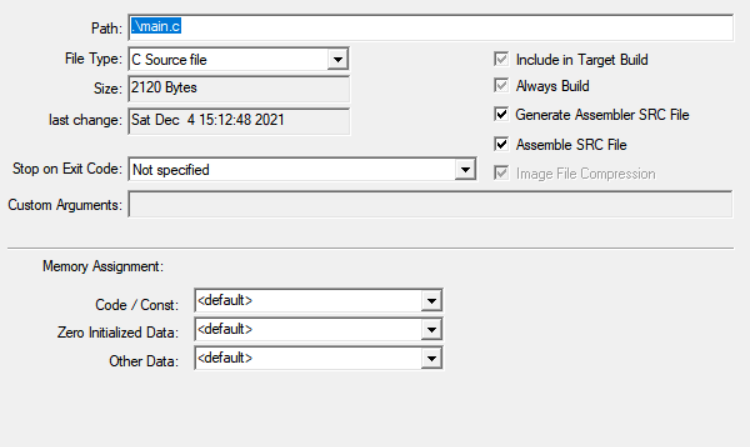
4）异常处理模式使能

1. 异常处理程序开始执行。

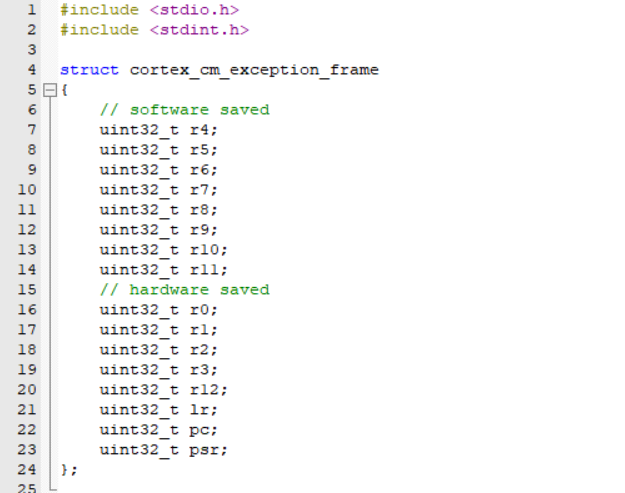
* 程序自动处理异常程序时，没有保存R4-R11寄存器的值，可以通过续保存剩下的 r4-r11 到堆栈，然后将堆栈栈顶地址（SP）传出，再在core\_dump中用 printf 打印堆栈的内容，这些内容即是异常前处理器的寄存器值，实现了保存输出各寄存器的功能，便于异常情况下的检查。

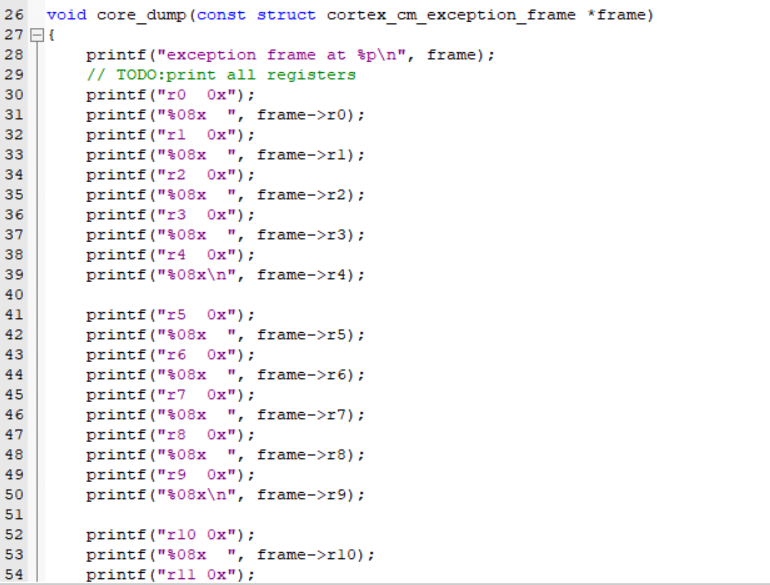
# 实验步骤

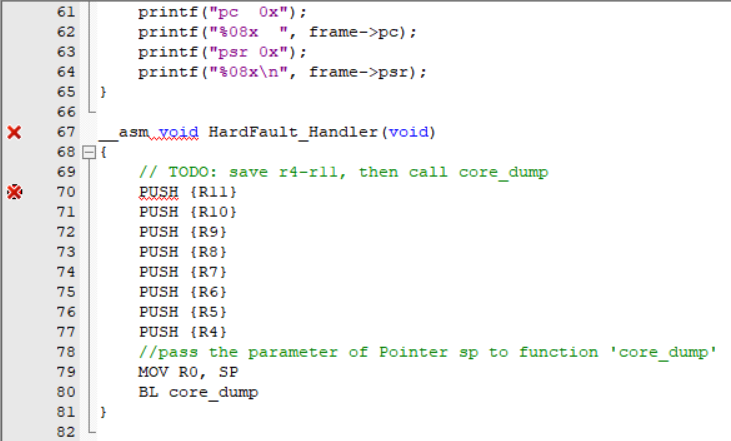
* 为了让程序可以运行汇编代码，首先需要在main.c的options中选择可以生成和运行Assemble的选项。（一开始没有加上这一步，导致程序一直无法正确运行）

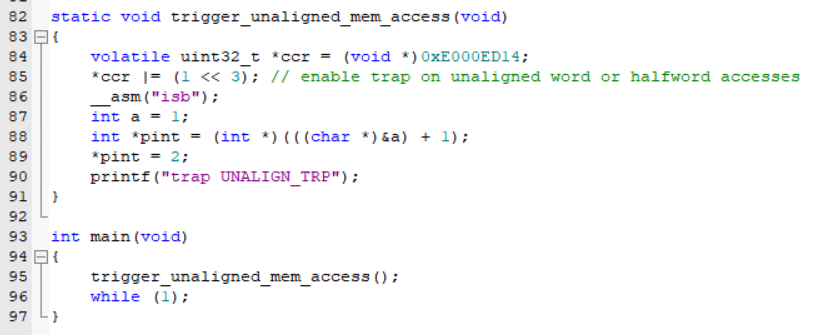
 

* 在main.c文件中加入重写后的代码，并在HardFault\_Handler函数的第一行加上断点：

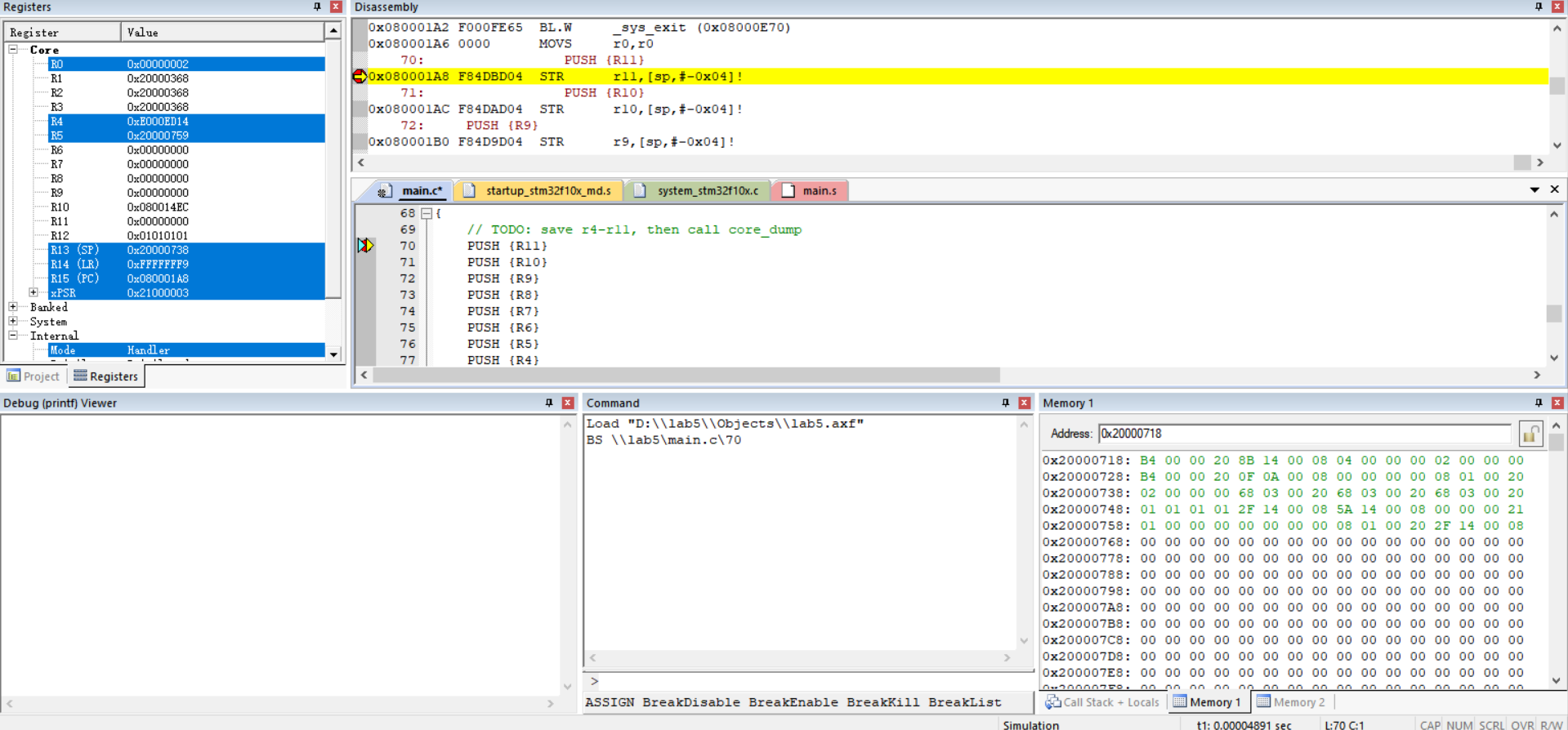








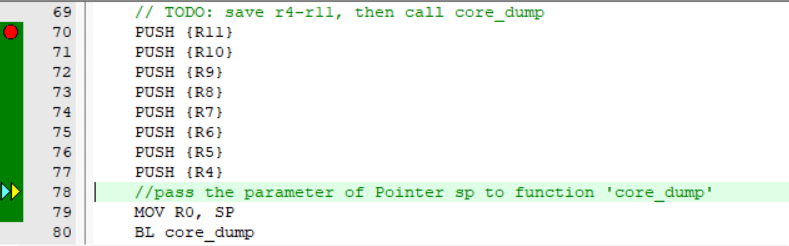
* 启动start Debug Session，开始执行到HardFault\_Handler函数第一行代码处；

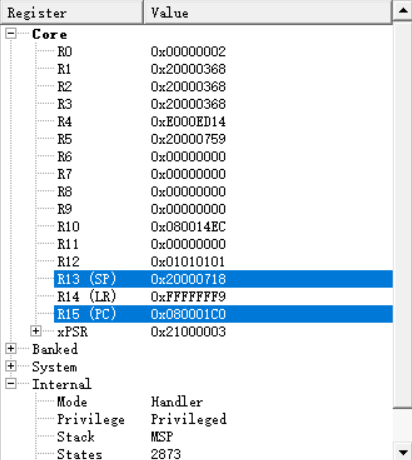


可以看到，此时的栈顶位置为0x20000738，观察栈中内容，从下到上分别保存了PSR、PC、LR、R12、R3、R2、R1、R0。

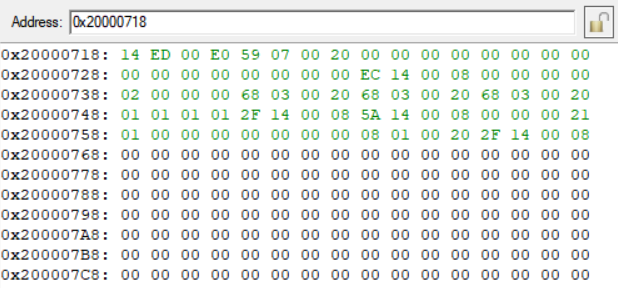


* 继续单步向下执行，直到R4-R11全部进栈：

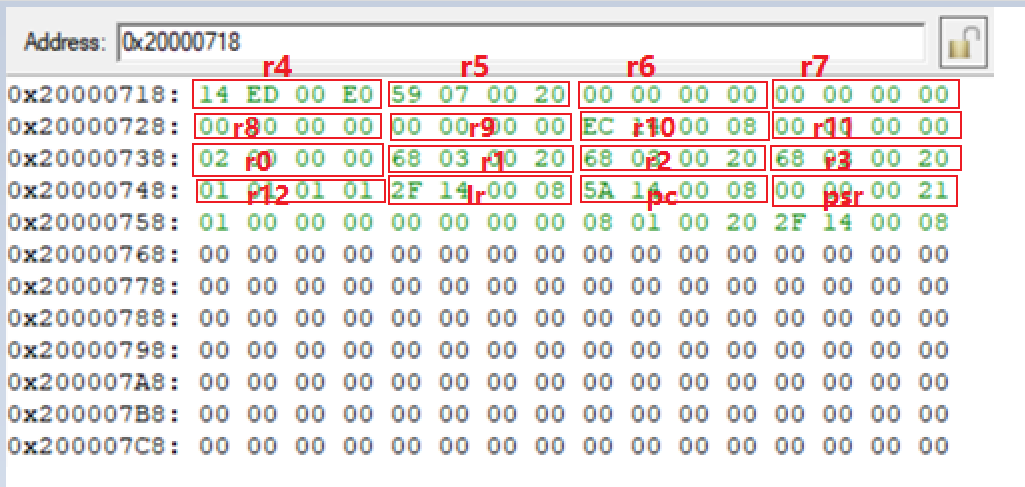




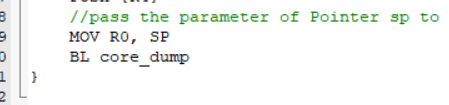
可以看到，栈顶位置从0x20000738变成了0x20000718，可以看出压栈是往低地址方向进行的，在看此时栈中的内容，在查看Memory窗口输入栈顶指针位置，查看如下：



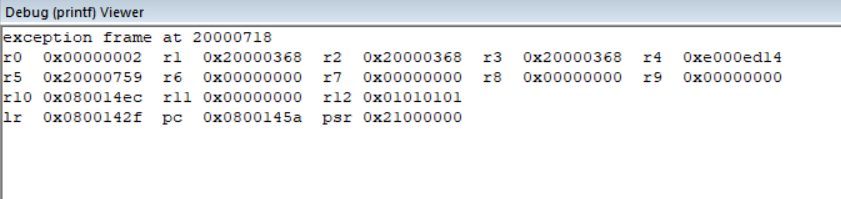
各个位置对应保存的寄存器值对应如下，可以看出和上方显示的寄存器值是一致的：



* 因为core\_dump函数仅有一个参数，将其保存至R0中，并调用该函数：



* 继续运行，得到的输出结果如下，由上可知，该程序正确运行输出了各个寄存器的结果：



# 实验结果与分析

本次实验成功地实现了将R4-R11寄存器中内容保存在栈中，并通过printf实现了输出，但是这个过程并不是一帆风顺的，我遇到了下面几个问题：

首先注意到在函数core\_dump中是需要传递一个参数的。经过查阅资料可知传递函数的参数是保存在R0寄存器中，所以需要把该参数保存传递给R0寄存器。这一步就要做到将汇编函数和将汇编代码和c语言代码结合起来，但在实验的一开始，我不知道怎么将寄存器中保存的值和cortex\_cm\_exception\_frame变量联系起来，一开始我想的解决方法是在汇编函数中先定义一个cortex\_cm\_exception\_frame变量，然后把值保存在里面，后来发现无法实现, 原因是只有汇编代码可以直接对寄存器进行操作，但是cortex\_cm\_exception\_frame变量是定义在c语言中的变量，没法做到两者的传递，之后经探索发现传进core\_dump函数是需要传入的变量是一个指针，指针说白了就是地址，如果我按顺序把它压入栈，然后将栈顶的SP指针直接传给那个函数，那在程序执行的时候，就可以把SP指针后面对应的那一块内存理解为cortex\_cm\_exception\_frame变量，这个问题也就迎刃而解了。变量和函数都是以说白了都是以地址的形式来进行存储的，这个实验也是加深了我对变量和函数保存在内存中的方式。

其次在实验中还要注意压入栈的顺序，因为在定义cortex\_cm\_exception\_frame变量时，注意到它是先将R4~R11定义在前面的，然后再依次保存了R0~R3,R12和其它的指针，故压栈的时候也要按照同样的顺序操作，开始我的实验方式是直接使用一步PUSH R4~R11，但是后来发现顺序存在问题，必须要到倒序压入栈中，从R11依次压到R4，才能正确地实现功能。

# 实验总结

通过这个实验帮助我理解了Cortex-M 处理器在发生异常时，应该如何执行。正常情况下会默认执行HardFault\_Handler函数，但是这个函数不是固定的，通过这个实验也帮助我们意识到，HardFault\_Handler函数是可以通过人为改写，通过不同种的方式来处理各种异常，更为灵活。

这个实验同时也加深了我对汇编语言的理解，因为上一个实验我们是通过MDK环境自动生成汇编代码的，只需要看懂汇编代码之间的过程就可以，不需要自己去手动生成。但是本次实验中的\_\_asm函数中是必须要通过手动来生成汇编代码，这就考验我们对汇编代码的认识和理解。一开始我也采用了其他的方式去实现压栈，比如将栈顶SP指针传给R0，再开辟一片空间，通过在R0减4，R0减8的地址上复制保存寄存器的值的方式来传递，后来发现arm汇编有更简单的处理方式，即“PUSH”操作，就改用了后面的那一种方式，这个探索汇编语言的过程也让我从更底层了解机器的运行机制，加深了我的理解和认识。