



**《微机原理》实验报告**

姓名：曹建钬

学号：20375177

# 实验内容

**（一）C程序调用汇编程序**

1. 运行Keil uVision5，按照实验一的项目创建方式建立一个新的项目，并在其中添加两个汇编文件（start.s和Mymain.s）和一个.c文件（main.c），其中 start.s 文件作为启动文件，Mymain.s文件保存编写的汇编程序，main.c文件中写入main函数并在main函数中调用汇编程序，将通过C调用汇编实现字符串的拼接功能，项目结构如图1所示：

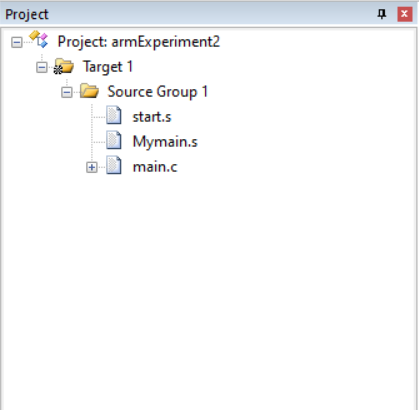


图 1：C程序调用汇编程序项目结构图

2. 补充后的start.s文件内容如图2所示：

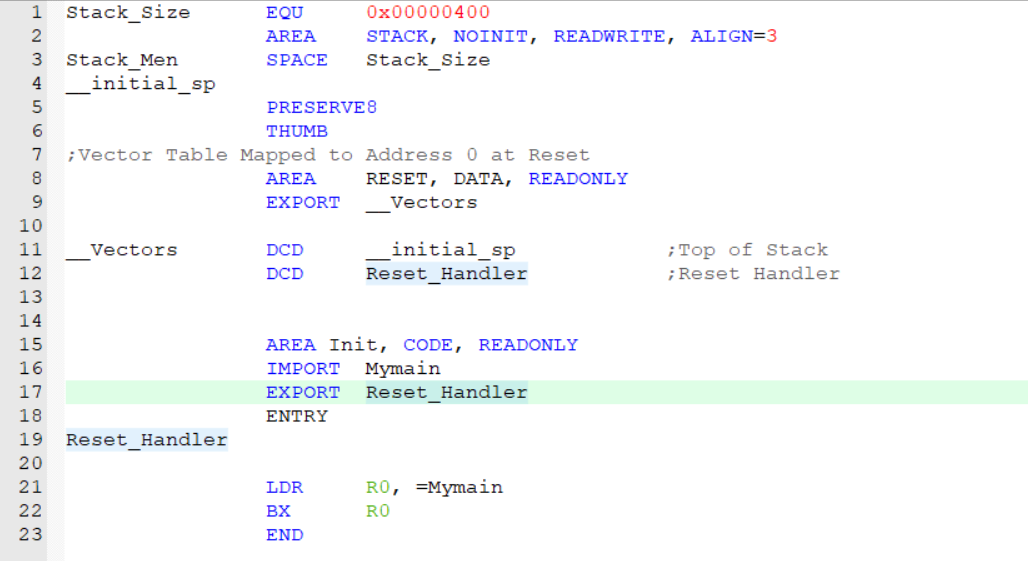
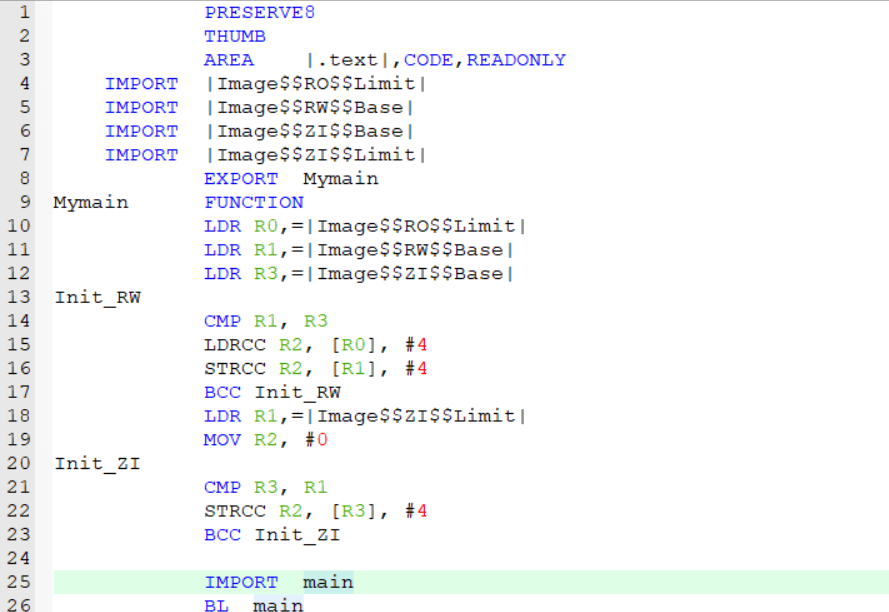


图 2：start.s文件内容

3. Mymain.s文件的内容如图3所示，my\_strcat代码段通过汇编程序实现字符串拼接功能。需要注意的是C程序调用汇编函数时，父函数为C函数，子函数为汇编子程序my\_strcat，根据AAPCS标准，父函数将第一个参数first\_sentence存入到R0，将第二个参数second\_sentence存入到R1，同时这个参数都为存放数据的地址，注意到second\_sentence定义为const，则尾字符串存放于Flash ROM，而first\_sentence定义在main函数内且为字符数组，则首字符串存放于RAM的栈区中。



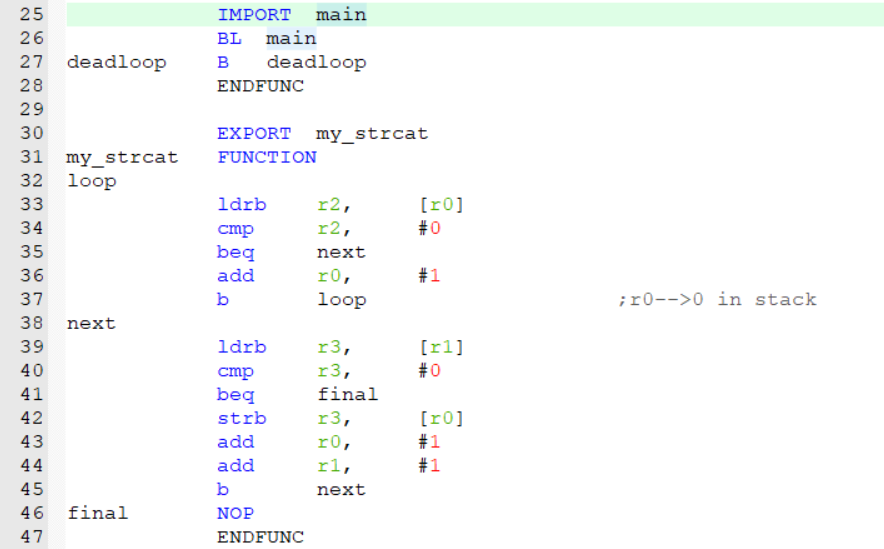


图 3：Mymain.s 文件内容

4. main.c 文件的内容如图4所示，first\_sentence 字符数组中保留一句话的前半句，second\_sentence是指向后半句字符串常量的指针，通过调用my\_stract汇编程序将second\_sentence指向的字符串添加在first\_sentence字符串数组的末尾，从而实现字符串的拼接。

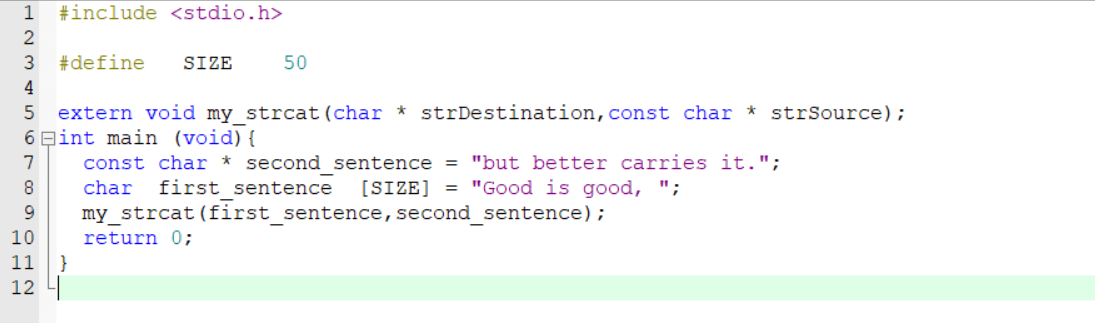


图 4：main.c文件内容

5. 观察寄存器和相关存储器在运行过程中和调用函数时的变化，以及完成拼接后first\_sentence字符数组的内容

调试过程如下：

* 复位后全速执行至Mymain.s第26行（如图5），此时已完成RW段和ZI段的初始化。

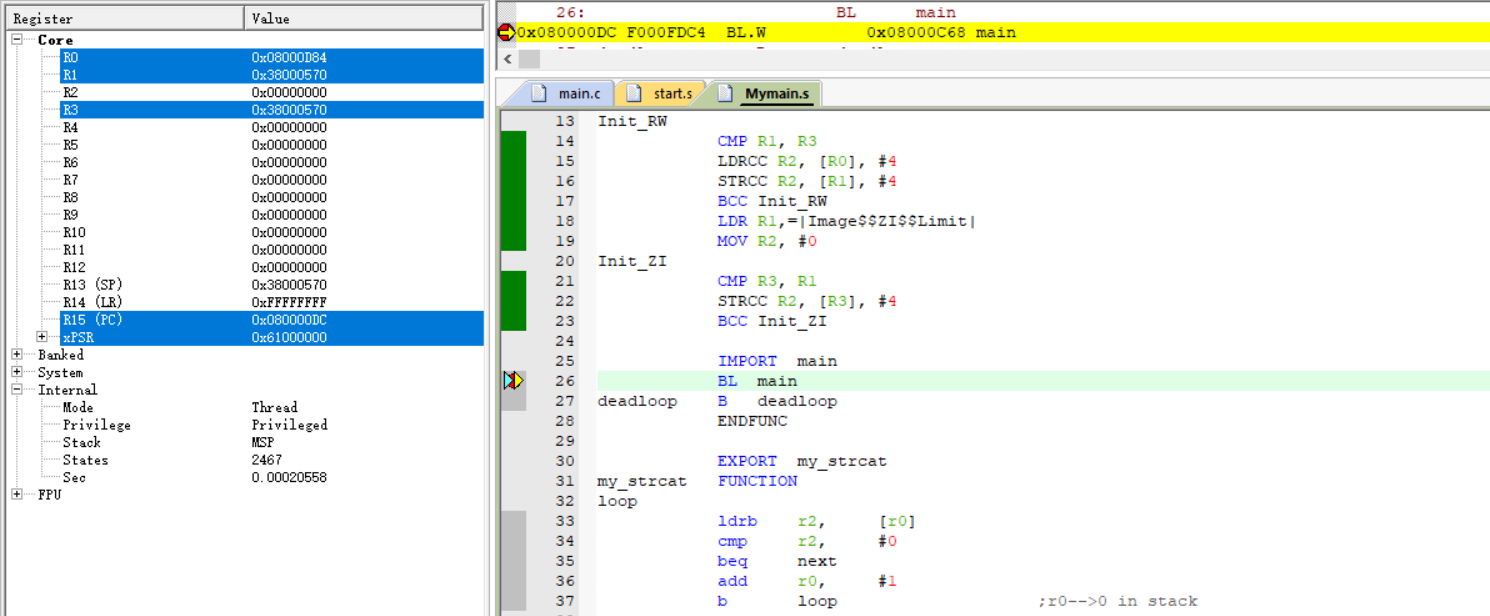


图 5：全速执行至Mymain.s第26行

* 单步执行Mymain.s第26行指令，转移到main函数并且把转移前的下条指令地址保存到LR，如图6所示，查看Register窗口可发现连接寄存器的值为0x080000E1，全速执行至Mymain.s第27指令（死循环），查看Disassembly反汇编窗口可看出转移到main函数前下一条指令的地址为0x080000E0，与LR中存储的数一致（0x008000E1作为一条指令的地址必须是偶地址，其最低位1仅用于标识该指令为Thumb指令，而实际指令地址为偶地址0x008000E0）。

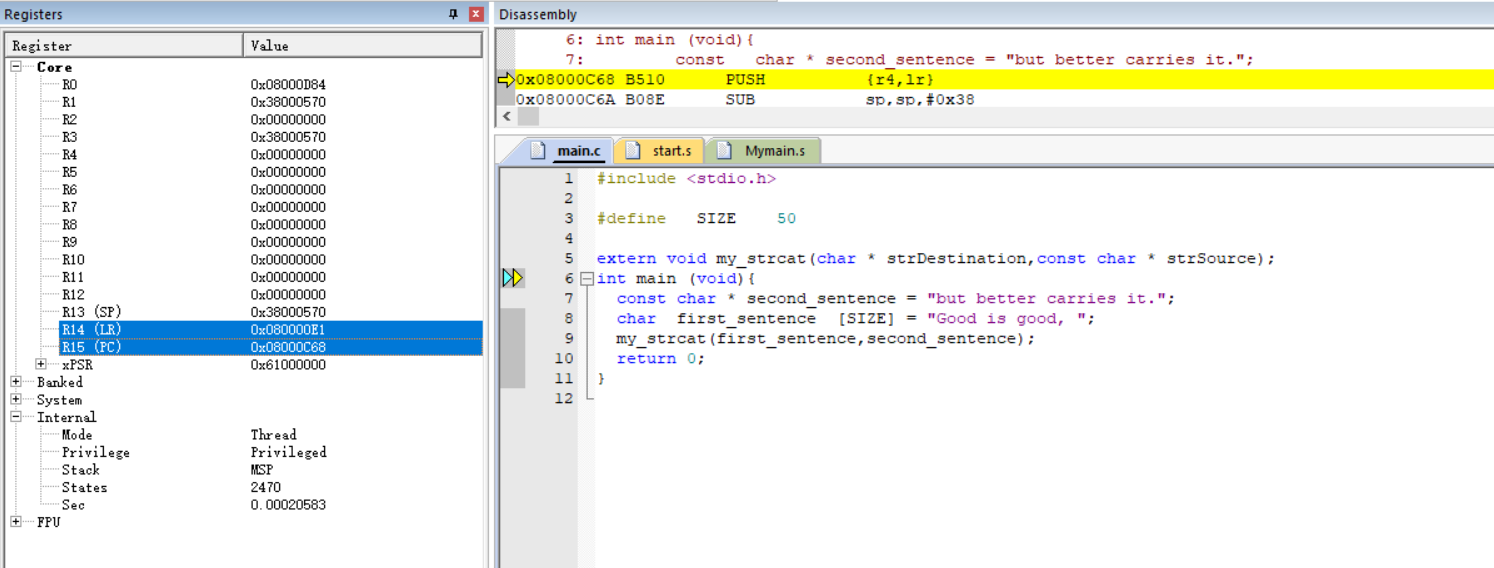


图 6：单步执行Mymain.s第26行指令进入到main函数

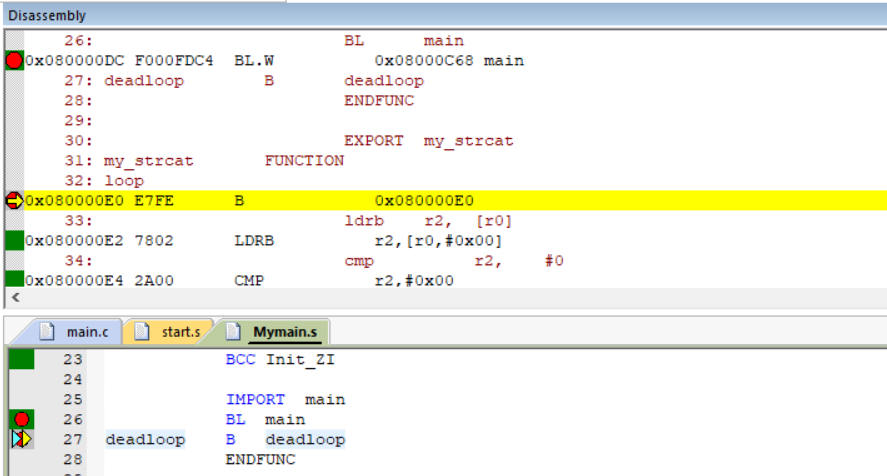


图 7：全速执行至Mymain.s第27行指令

* 重新开始调试，执行至main函数（如图6）。执行main.c中第8、9行代码，定义了一个常量指针second\_sentence指向一句话的后半句，且定义了一个字符数组first\_sentence存放一句话的前半句。

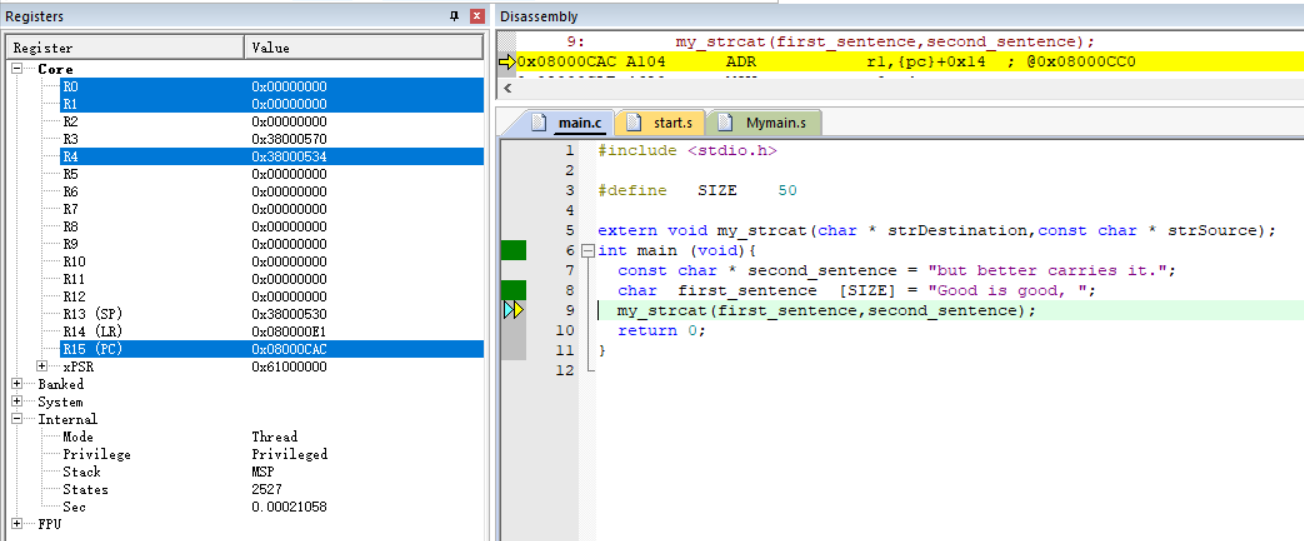


图 8：执行main.c第7、8行代码

* 执行main.c第9行代码后如图9所示。父函数通过R0和R1传递参数，查看Registers窗口，R0中的值为0x38000534，为RAM中的定义的栈区；R1中的值为0x08000CCD，为ROM中的一片区域；LR中的值为0x08000CB5，为main.c文件中第10行代码：return 0的地址。在Memory窗口查看地址0x38000534（如图10所示），可以发现“Good is good, ”确实已经存入栈区；在Memory窗口查看地址0x08000CC0（如图11所示），可以发现“but better carries it.”的确已经存入ROM中。

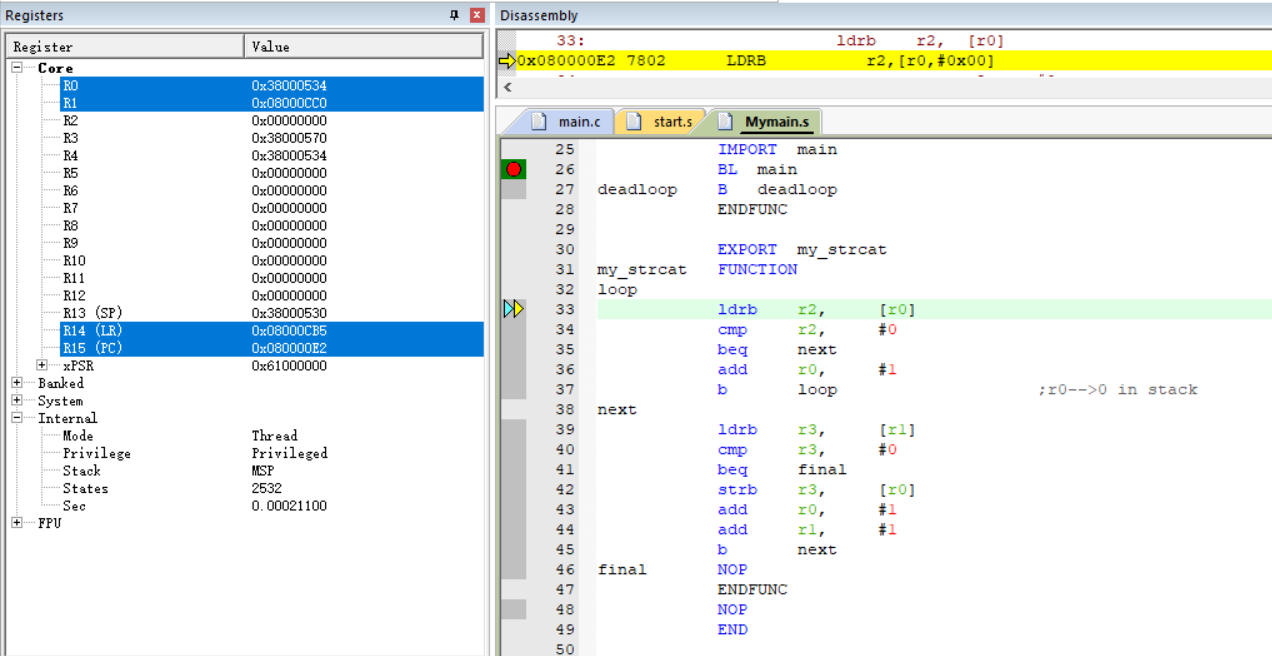


图 9：执行main.c第9行代码

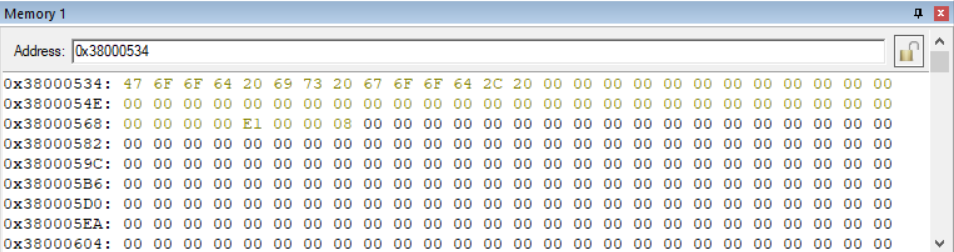


图 10：Memory窗口查看地址0x38000534

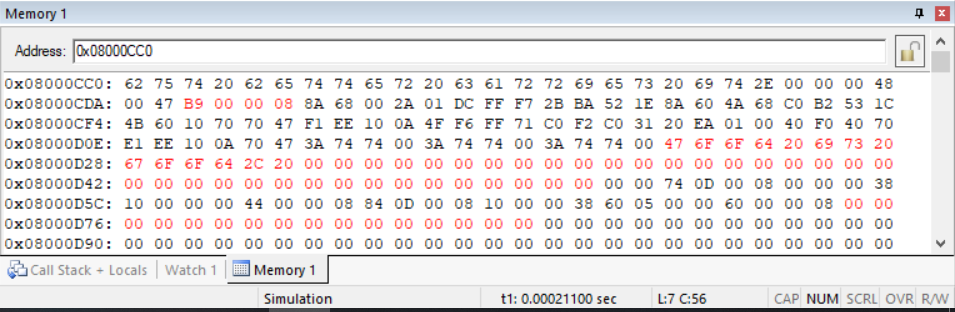


图 11：Memory窗口查看地址0x08000CC0

* Mymain.s中第32行到37行为循环结构，目的是使r0存储前半句话末尾后一个字节的地址。执行完第一次循环（如图12所示）后r0存储第二个字符o（6F）的位置；执行完循环（如图13所示）后r0已经存储第一个空字符的位置。

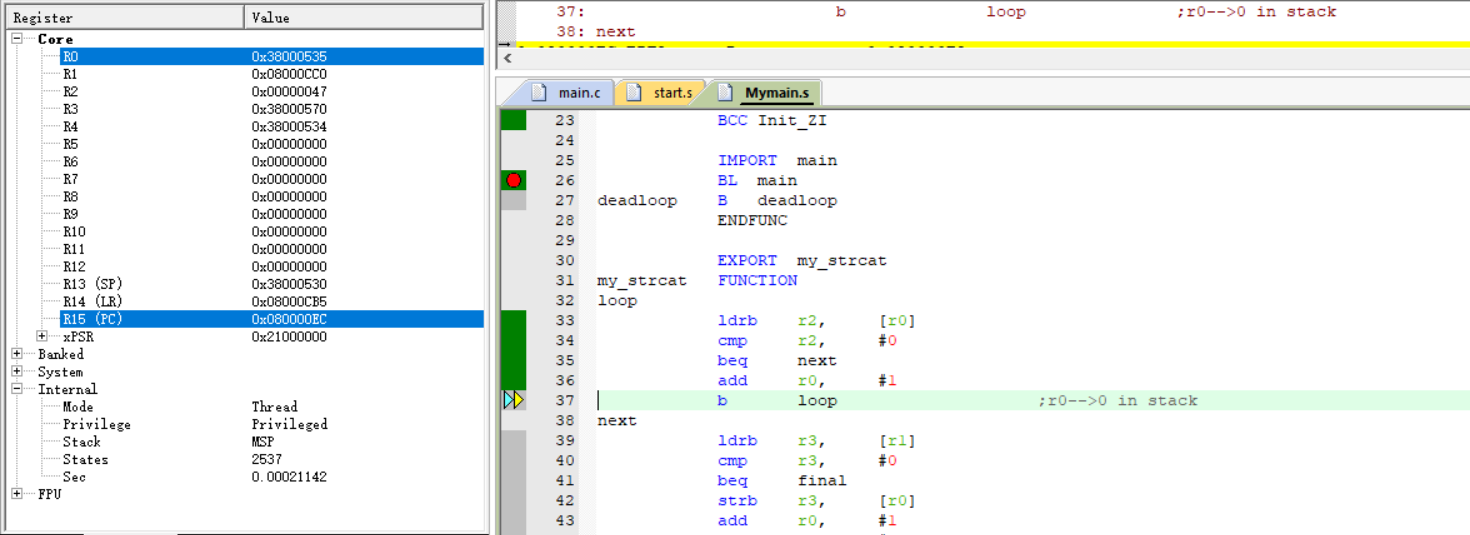


图 12：执行第一次循环

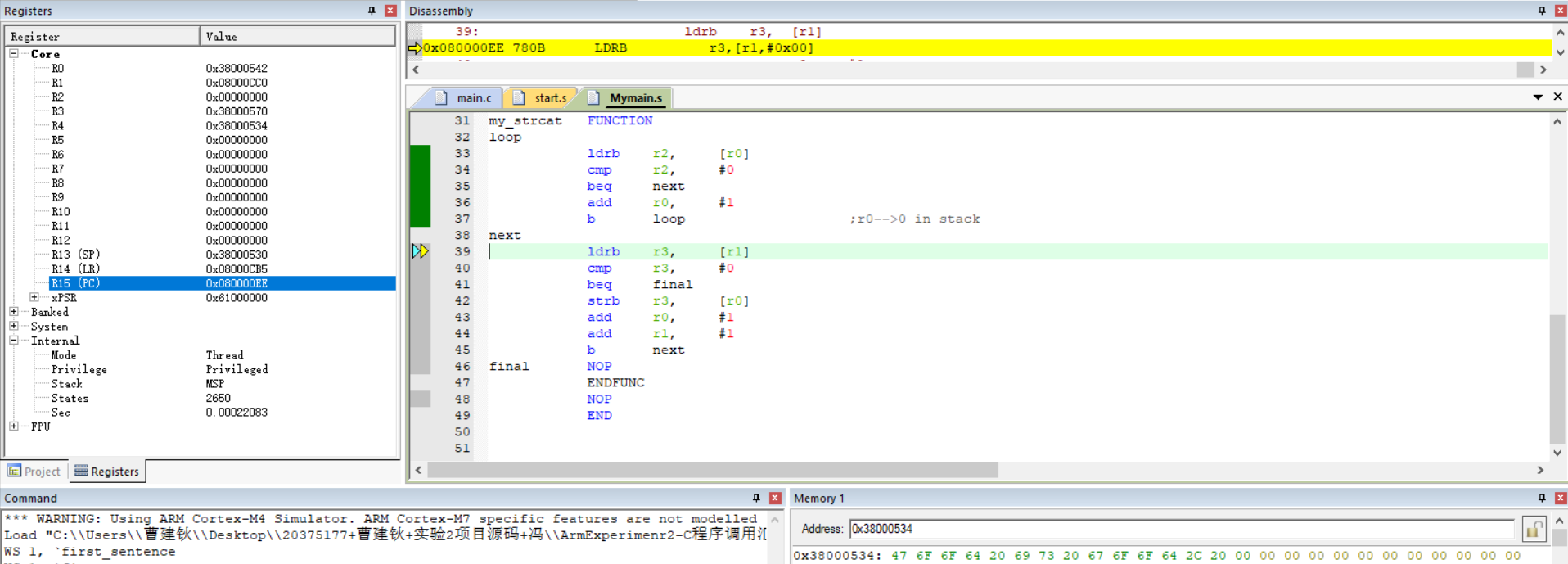


图 13：执行完循环

* 第39行至第45行同样为循环结构，实现将后半句中的字符挨个取出并存入栈中的操作（即拼接功能）。第39行指令为：从地址为r1中的值处读取一个字节给r3寄存器；第40行指令为：比较r3和0；第41行指令为：如果r3等于0则跳转至final；第42行指令为：将r3中的值按字节存入r0值对应的地址中；第43行、44行指令为：r0和r1自增1；第45行指令为：跳回至标号next处。每次循环始r0存的值都指向末尾后的第一个空字符，r1存的值都指向新的字符，r3作为中转站将字符从ROM移到栈中，循环结束的条件为r1也指向空字符，即后半句的所有字符已经遍历完毕。执行第一次循环后，后半句的第一个字符（“b”）连上了前半句，如图14所示。执行完循环后（如图15所示），查看Memory窗口，可发现后半句已经连上前半句（如图16所示）

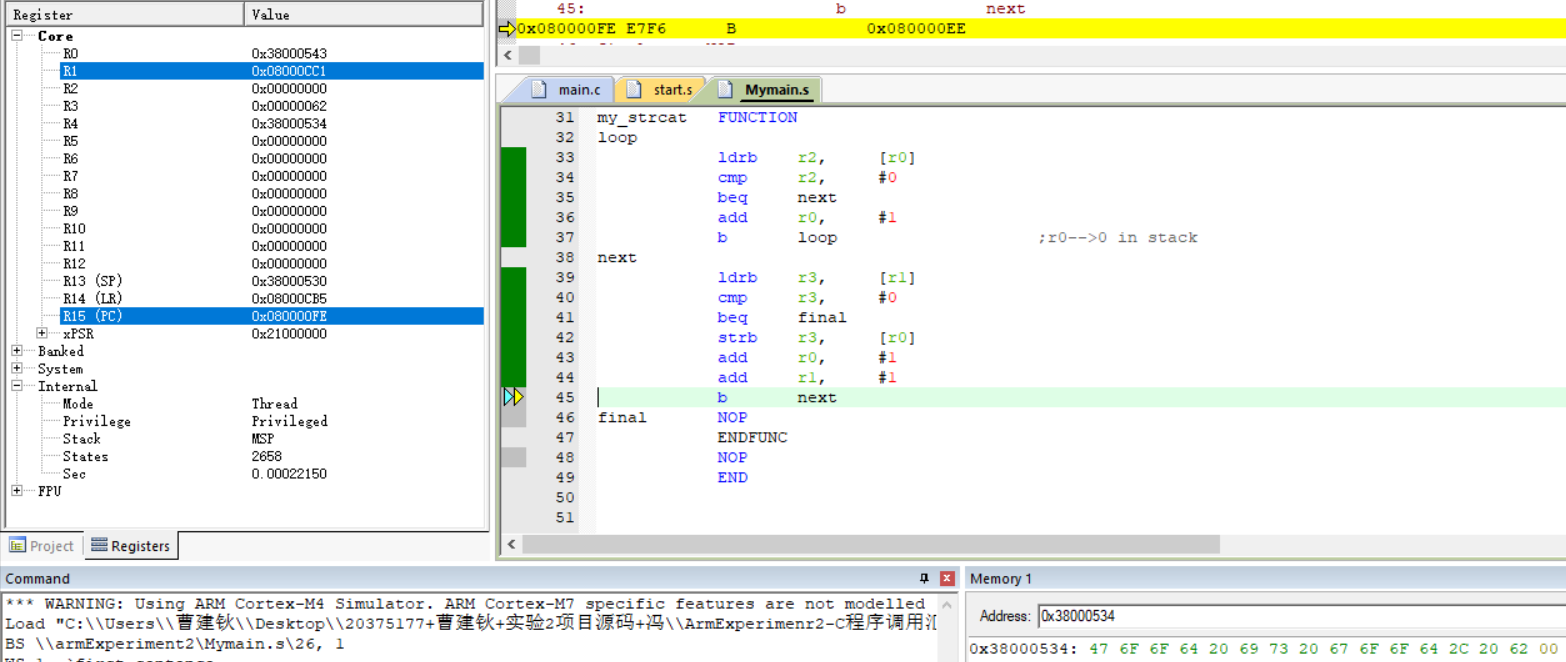


图 14：执行一次循环

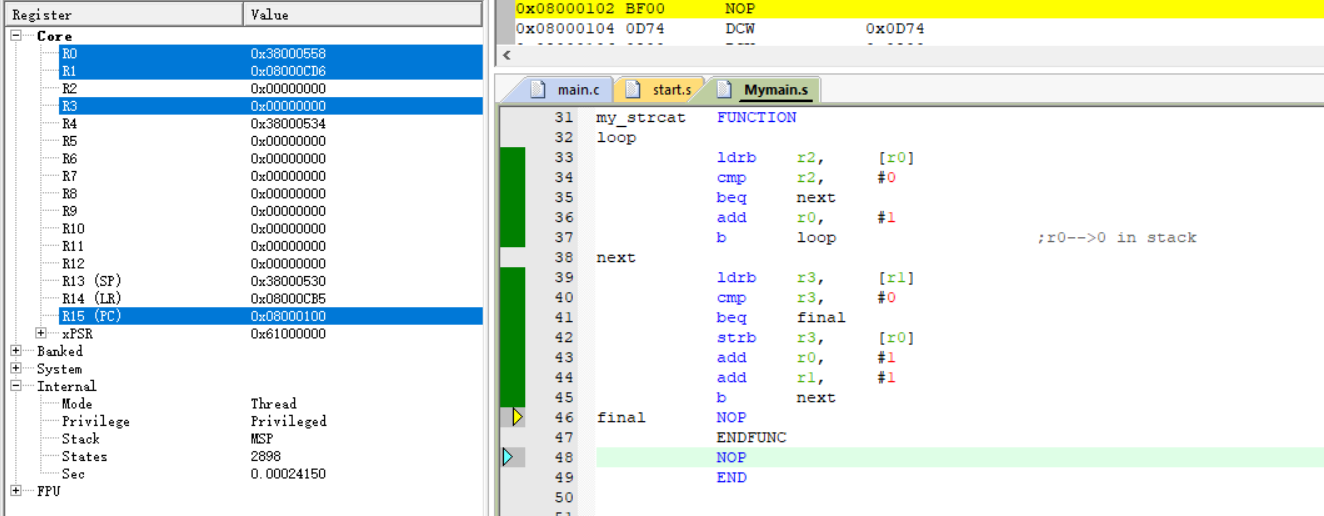


图 15：执行完最后一次循环

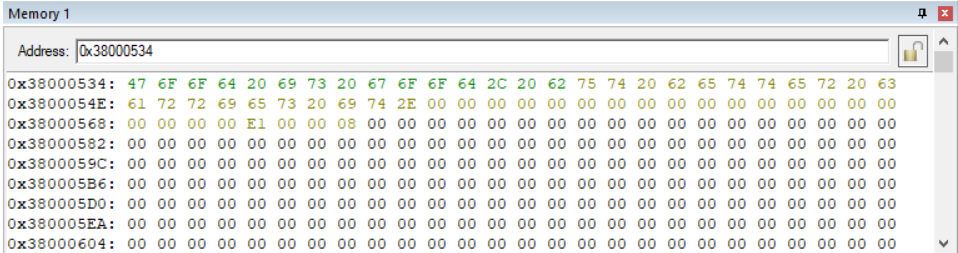


图 16：查看Memory窗口

* 结束子程序my\_strcat后如图17所示，下一行代码return 0的地址的确为0x08000CB4。查看watch窗口（如图18、19），字符数组first\_sentence内的值已经变为“Good is good, but better carries it.”，字符串的拼接任务圆满完成。

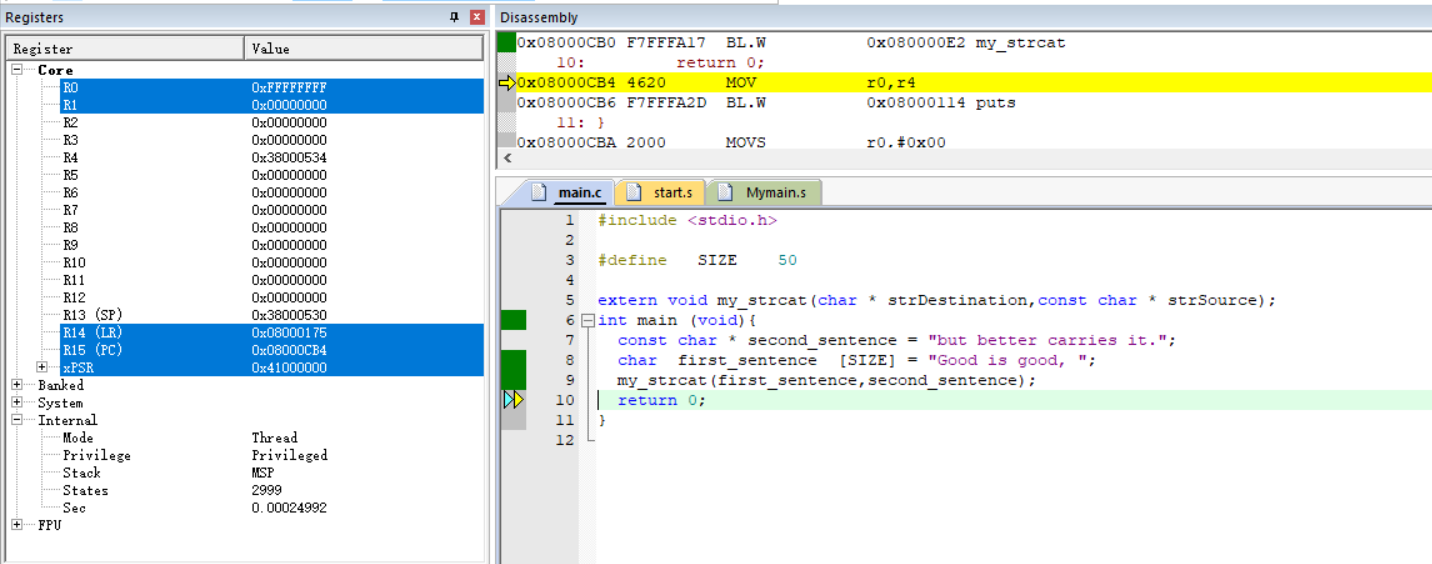


图 17：结束汇编子程序my\_strcat的调用

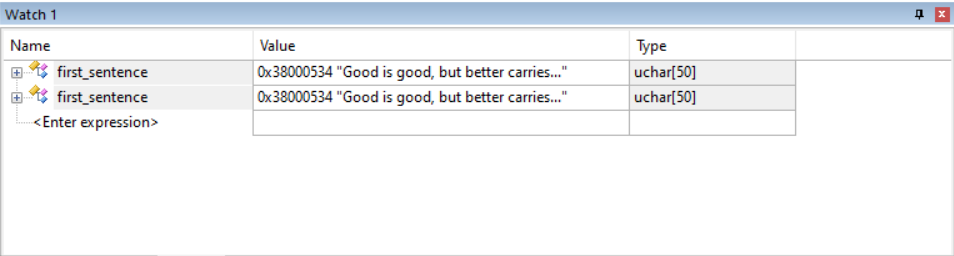


图 18：watch窗口查看first\_sentence字符数组内容

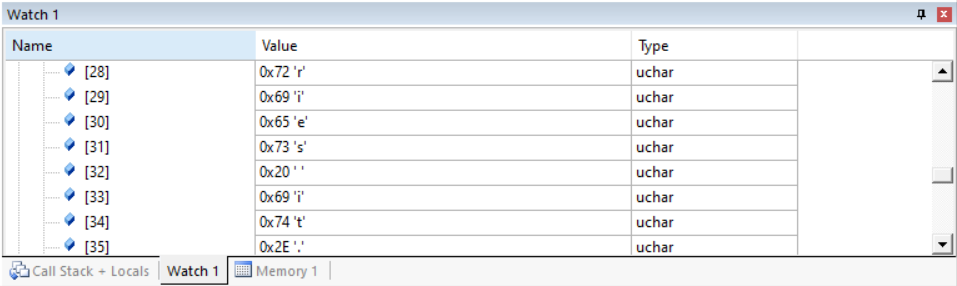


图 19：watch窗口查看first\_sentence字符数组内容

**（二）汇编程序调用C程序**

1. 运行Keil uVision5，按照实验一的项目创建方式建立一个新的项目，如图20所示，并在其中添加两个汇编文件start.s和Mymain.s和一个.c 文件my\_func.c，其中start.s 文件作为启动文件，Mymain.s文件保存编写的汇编程序，my\_func.c文件定义 my\_func函数，并在Mymain.s中的汇编程序调用C程序。

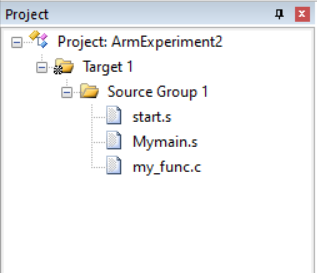


图 20：汇编程序调用C程序项目结构图

2. start.s文件的内容如图所示：

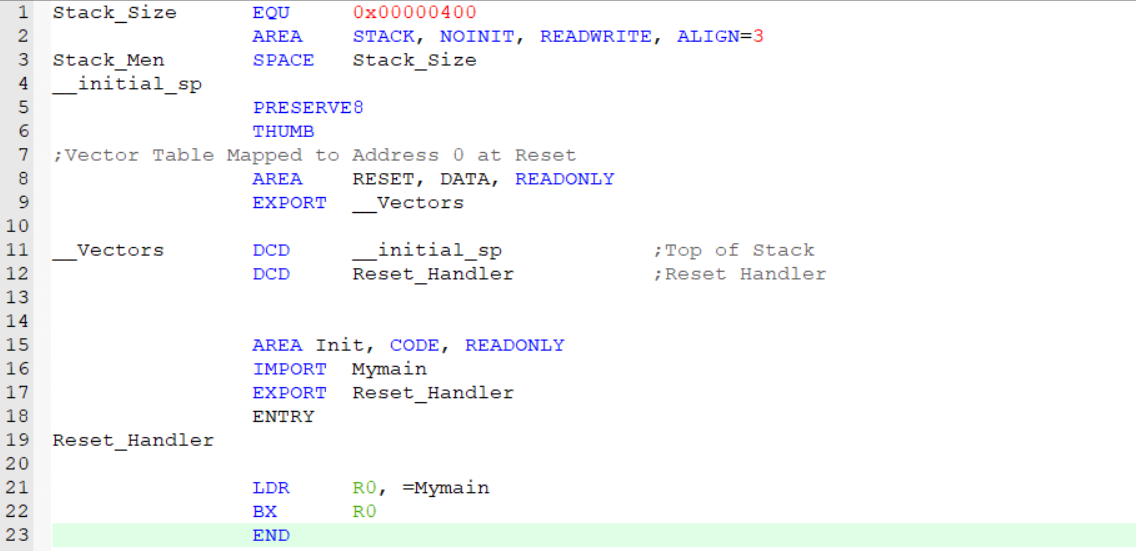
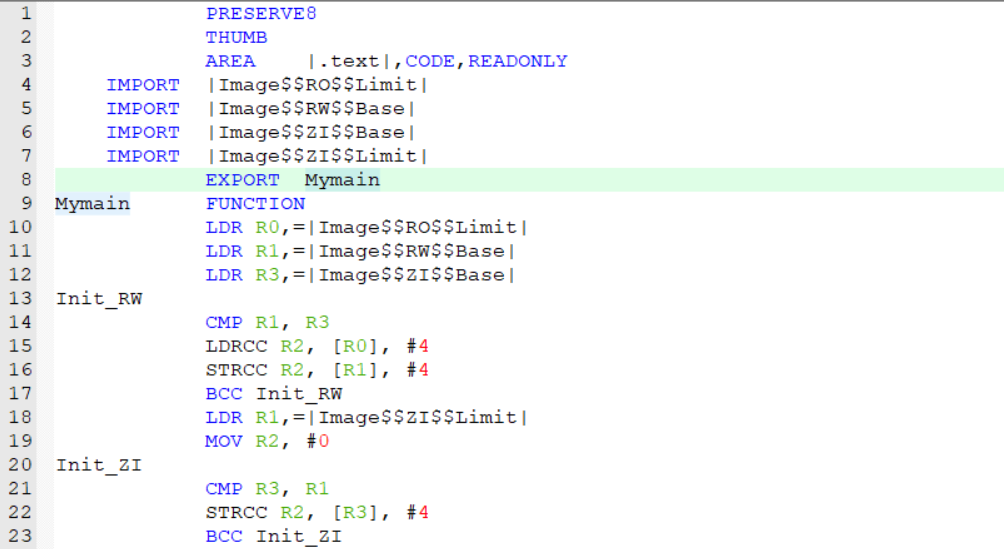


图 21：start.s文件内容

3. Mymain.s 文件的内容如图所示：



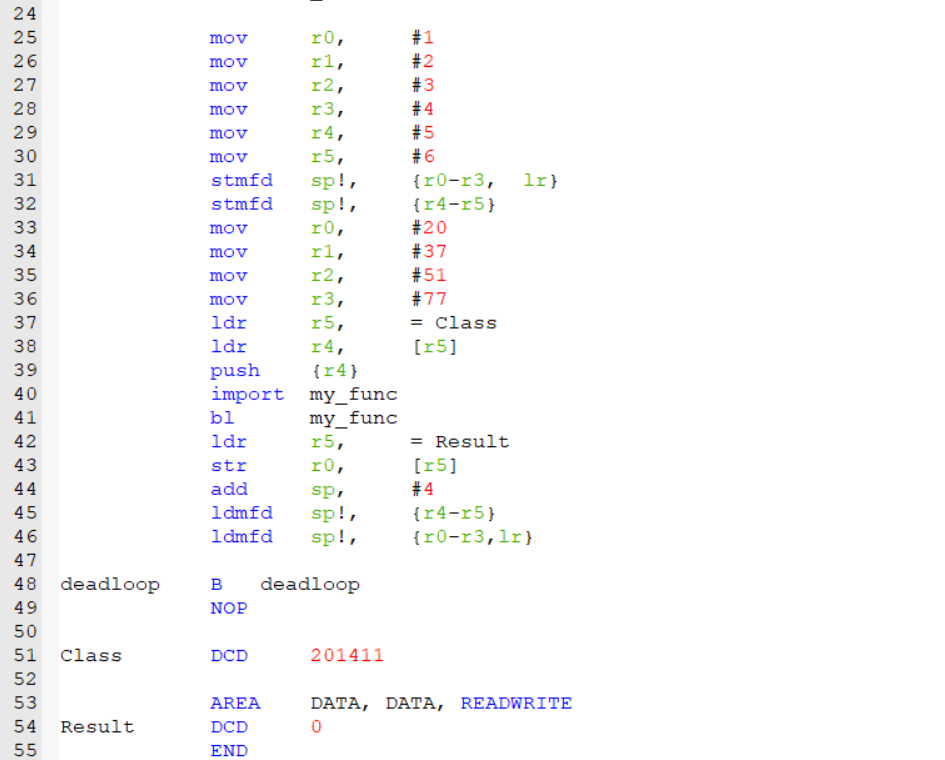


图 22：Mymain.s文件内容

4. my\_func.c文件的内容如图所示，将会返回表达式的结果

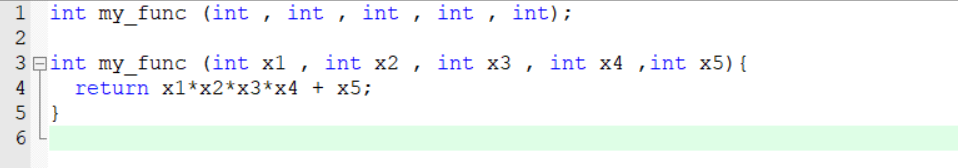


图 23：my\_func.c文件内容

5. 观察寄存器在运行过程中和调用函数时的变化，以及保存在Result中的结果是否正确，进行详细分析，计算理论结果并验证实际输出结果是否正确

学号：20375177 专业班号：201411

理论结果为：

调试过程如下：

* 复位后全速执行至Mymain.s第25行指令，如图24所示。

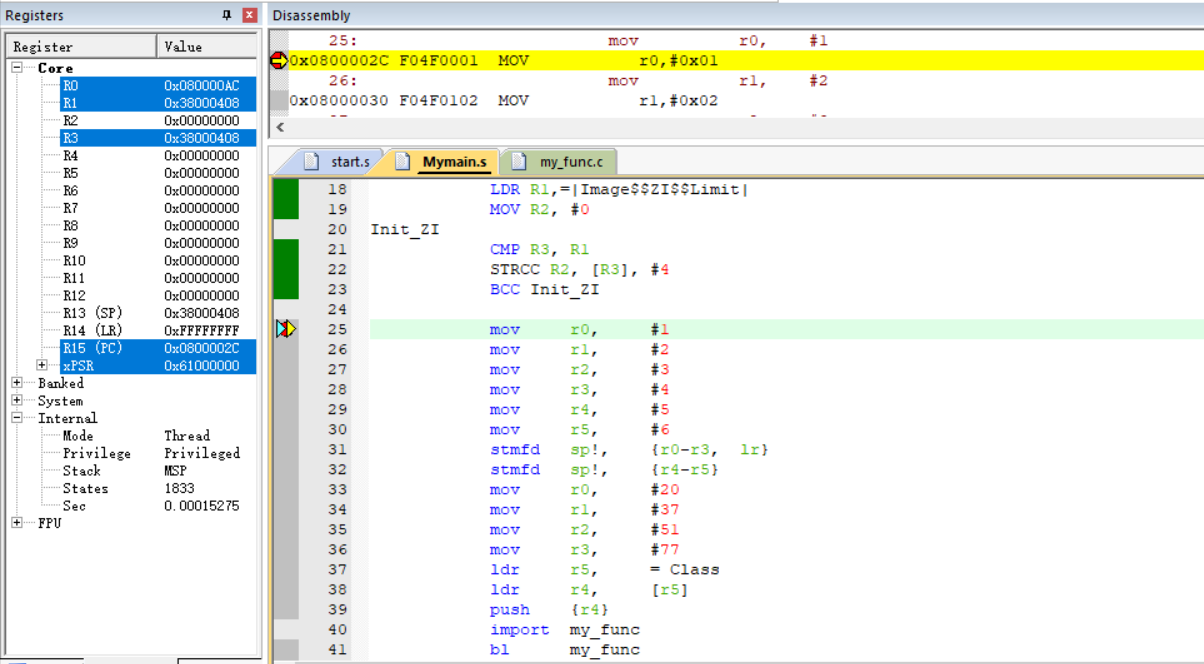


图 24：复位后全速执行至Mymain.s第25行指令

* 第25行至第30行指令将寄存器赋初值，R0~R5的初始值分别为1~6，执行完第30行指令的结果如图25所示

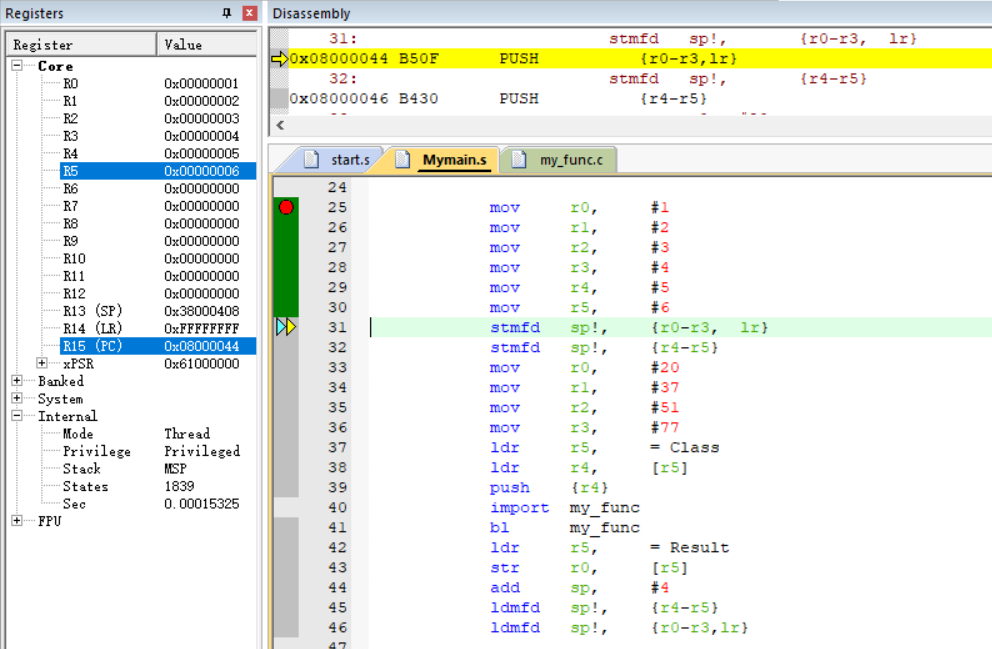


图 25：寄存器赋初值

* 根据AAPCS标准，调用者需要压栈保存寄存器R0~R3以及LR旧值，第31行指令即实现此要求；而实验过程用到了R4~R5寄存器，因此也需要压栈保存R4~R5旧值；第32行指令即实现此要求。执行完第32行指令后各寄存器的值如图26所示；查看栈，可以发现R0~R5和LR已经成功入栈，如图27所示。

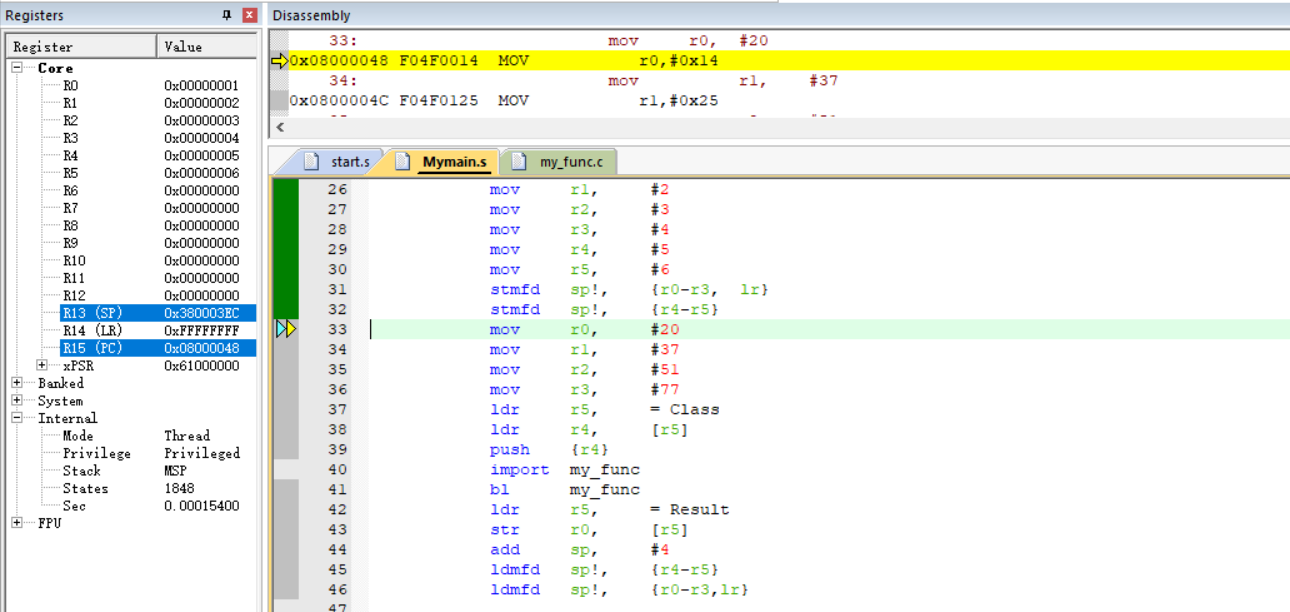


图 26：执行完31-32行压栈过程

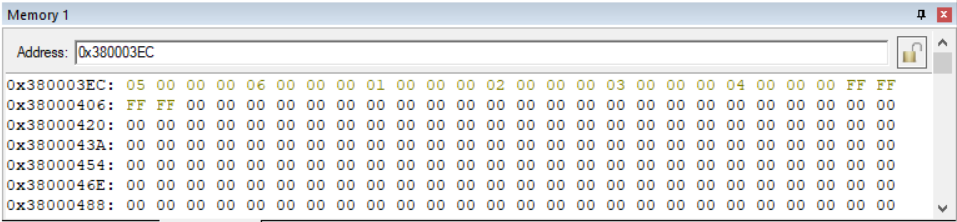


图 27：查看栈区

* 第33到36行指令将四部分学号20-37-51-77分别保存于R0~R3，执行完第36行指令后结果如图28所示，可以发现R0~R3已经成功赋值。

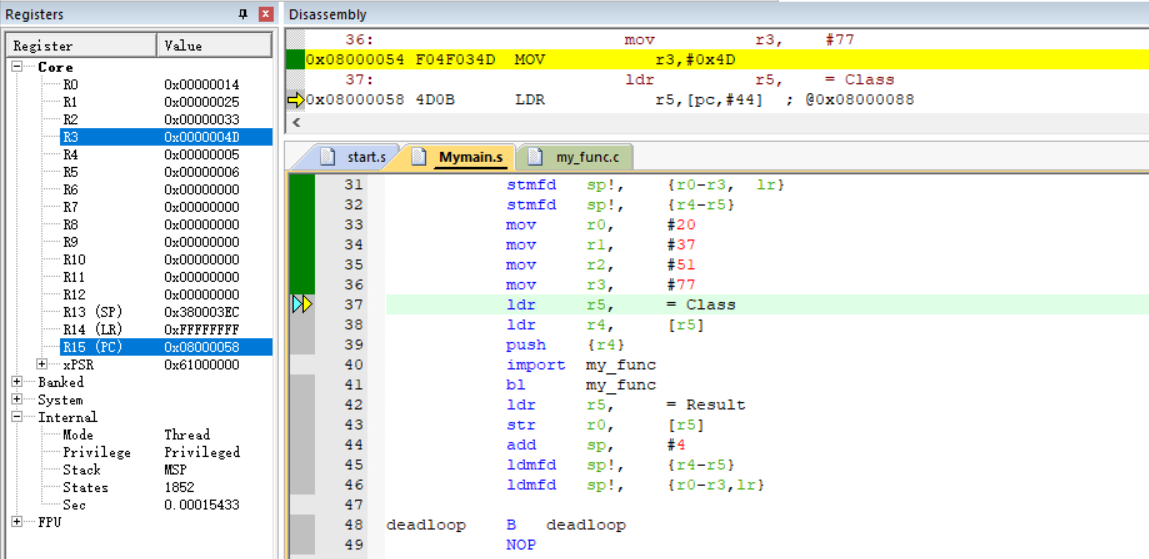


图 28：执行完第36行指令

* 第37到38行指令将R5作为中转站取出存于ROM的专业标号并将其赋值给R4，执行完第38行指令后的结果如图29所示。

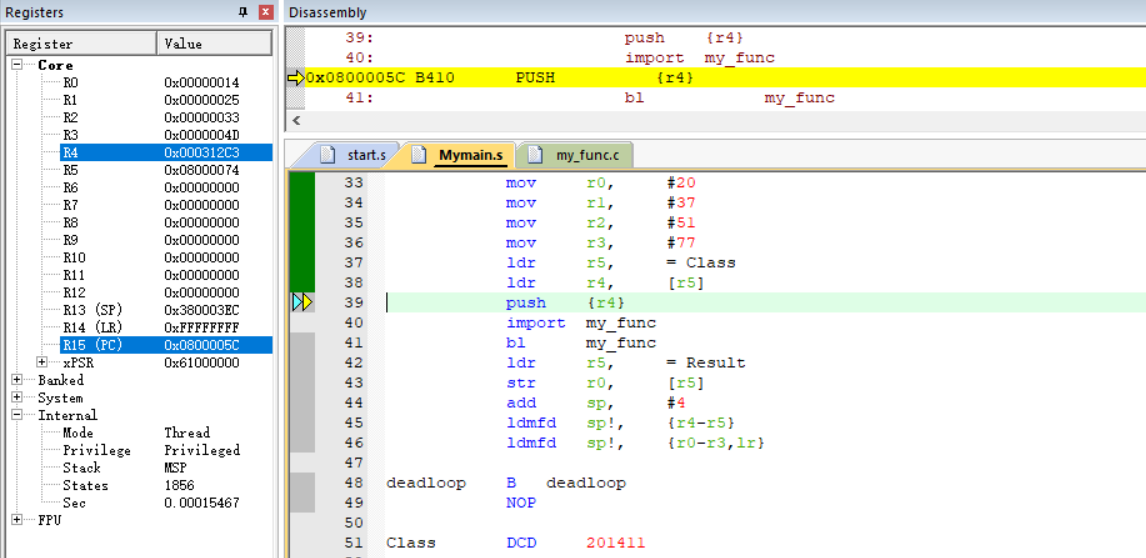


图 29：执行完第38行指令

* 注意到需要传递5个参数，因此第5个参数需要通过栈传递，故第39行指令将R4的值压入栈中，结果如图30所示。

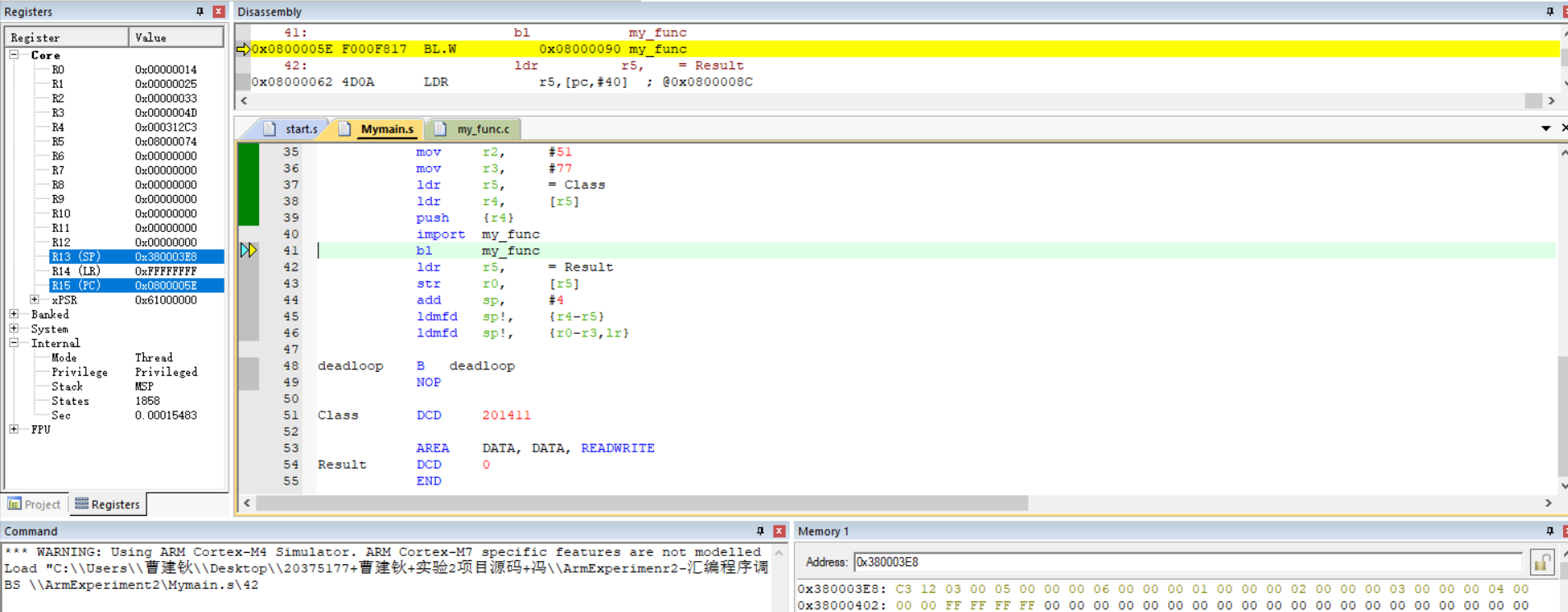


图 30：将R4的值压入栈中

* 执行第41行指令跳转至C程序中的my\_func函数，并将转移前的下条指令地址保存到LR，如图31所示。

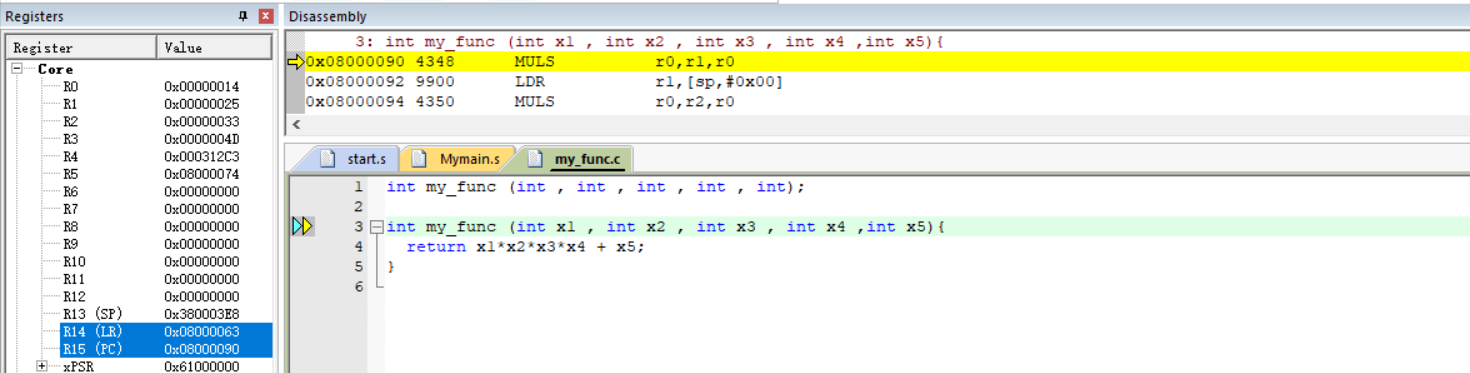


图 31：执行第41行指令

* 执行完my\_func函数后，结果保存在R0，如图32所示，与理论计算得到的结果一致。

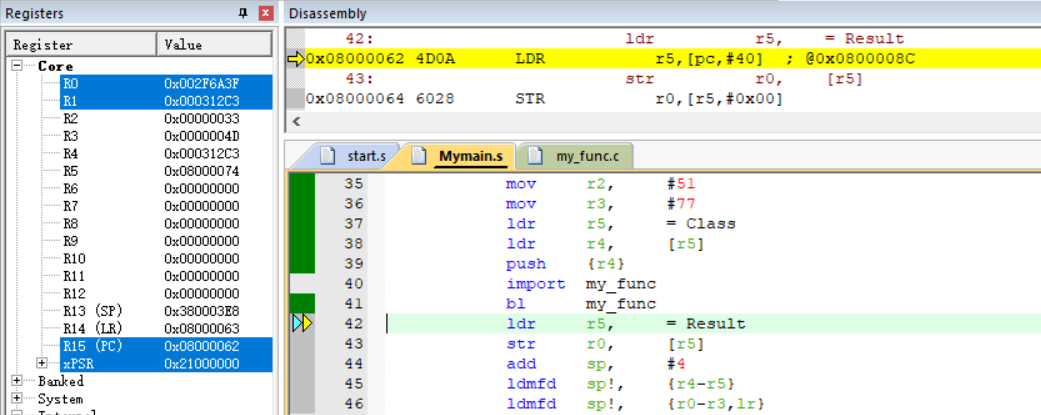


图 32：执行完my\_func函数

* 执行第42和43行指令将C函数的输出结果放入Result中保存，如图33所示。查看Register窗口可知道Result标号所在的地址为0x38000000，在Memory窗口中查找此地址，如图34所示，可验证保存在Result中的结果正确。

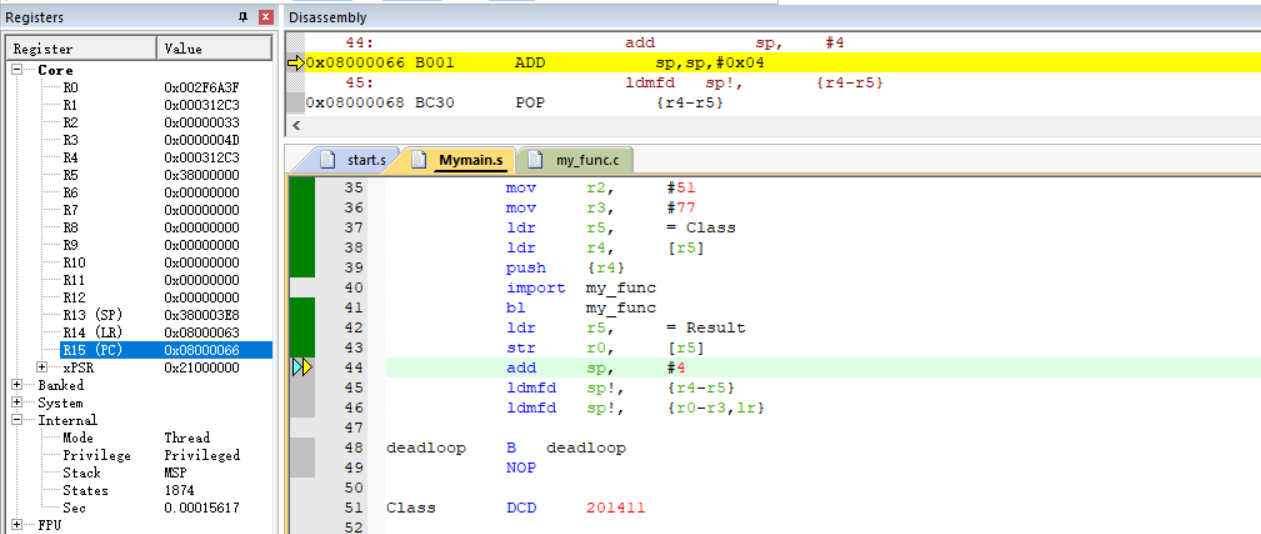


图 33：执行完第43行指令



图 34：Memory窗口查找Result地址

* 第44行指令消除栈中参数x5，本语句执行完后SP指向栈中R4旧址，如图35所示。

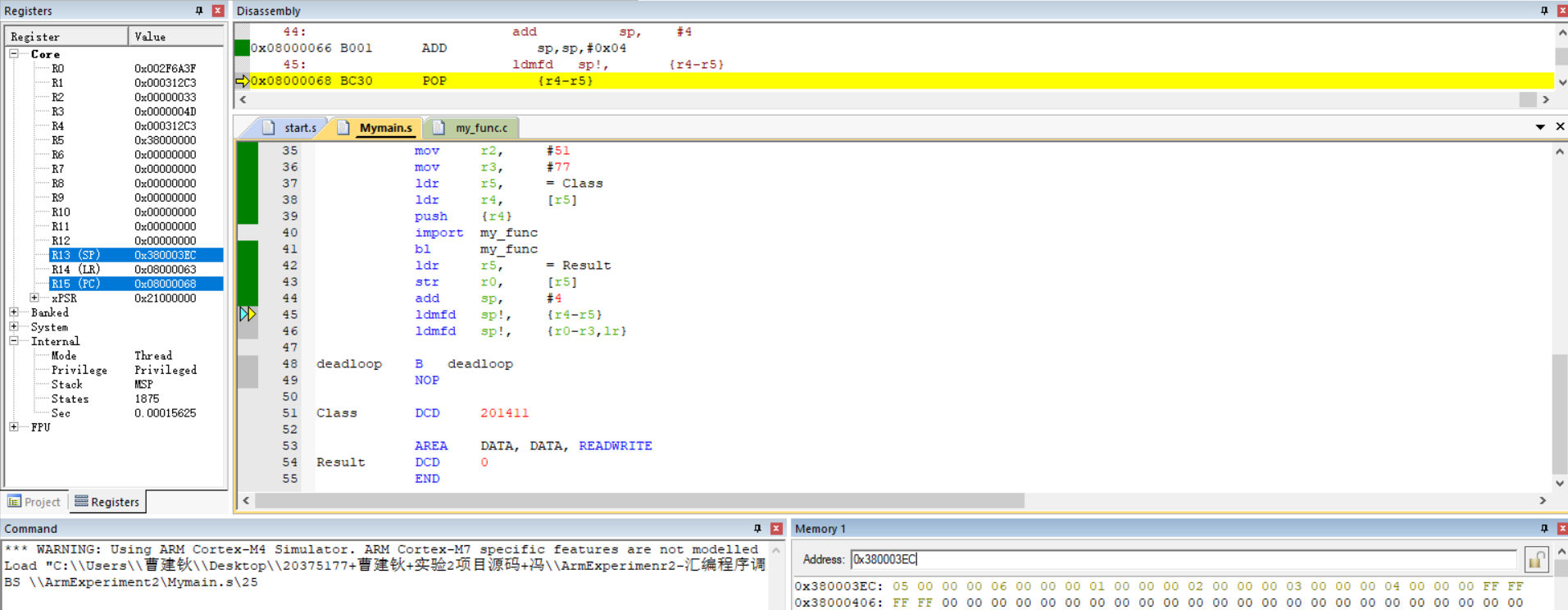


图 35：执行完第44行指令

* 第45行指令弹栈恢复R4~R5旧值，第46行指令弹栈恢复R0~R3，LR旧值。最终结果如图36所示。寄存器初始值在程序开始时和结束时未发生变化。

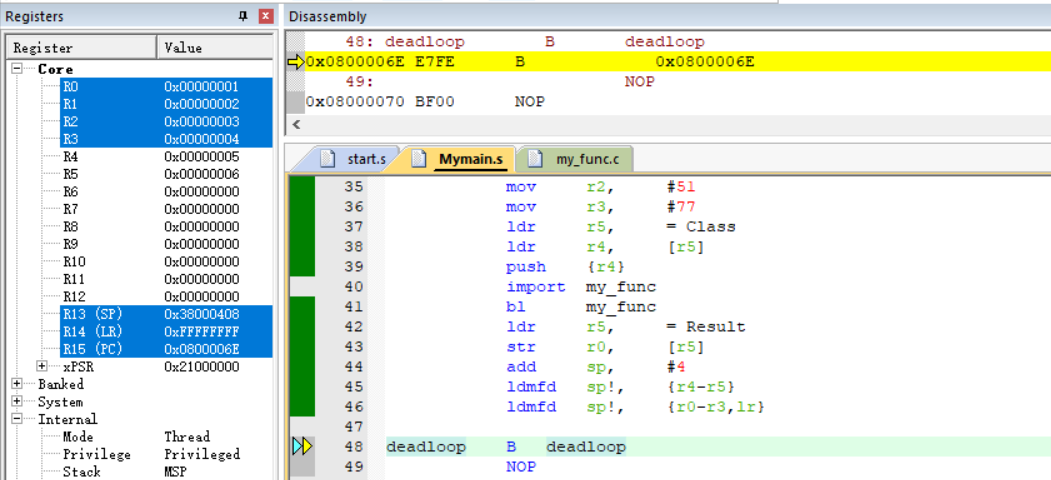


图 36：恢复现场

**（三）GPIO基础实验**

1. 通过配置 GPIO，点亮 LED 灯

初始化函数的函数原型声明为：



图 37：初始化函数

主函数调用：



图 38：电亮LED灯的主函数

灯的初始化函数的定义如下：

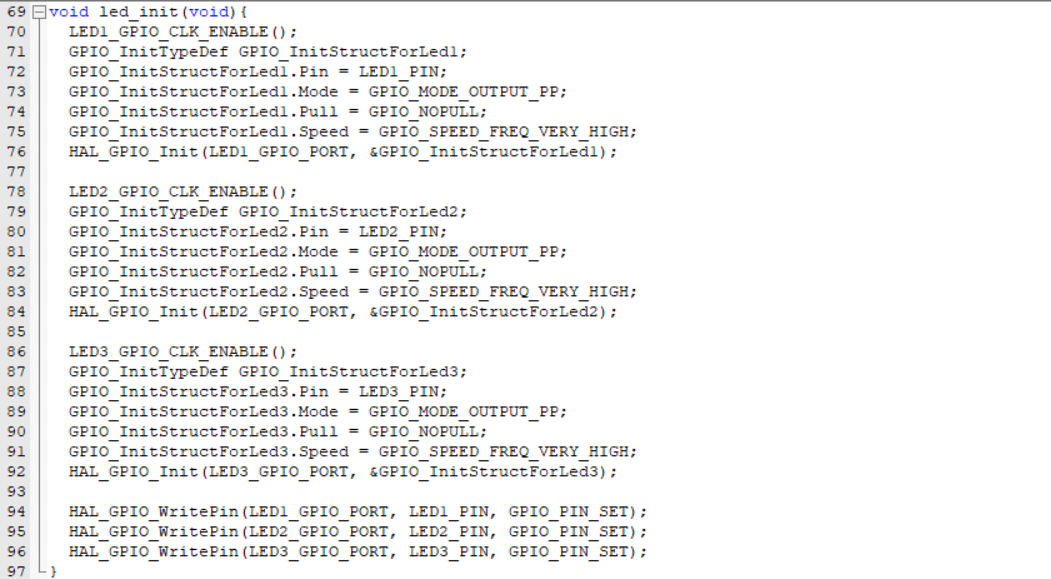


图 39：灯的初始化函数

通过在main函数中调用led\_init()可以使得Nucleo板的三个LED亮，实验现象如图40所示。

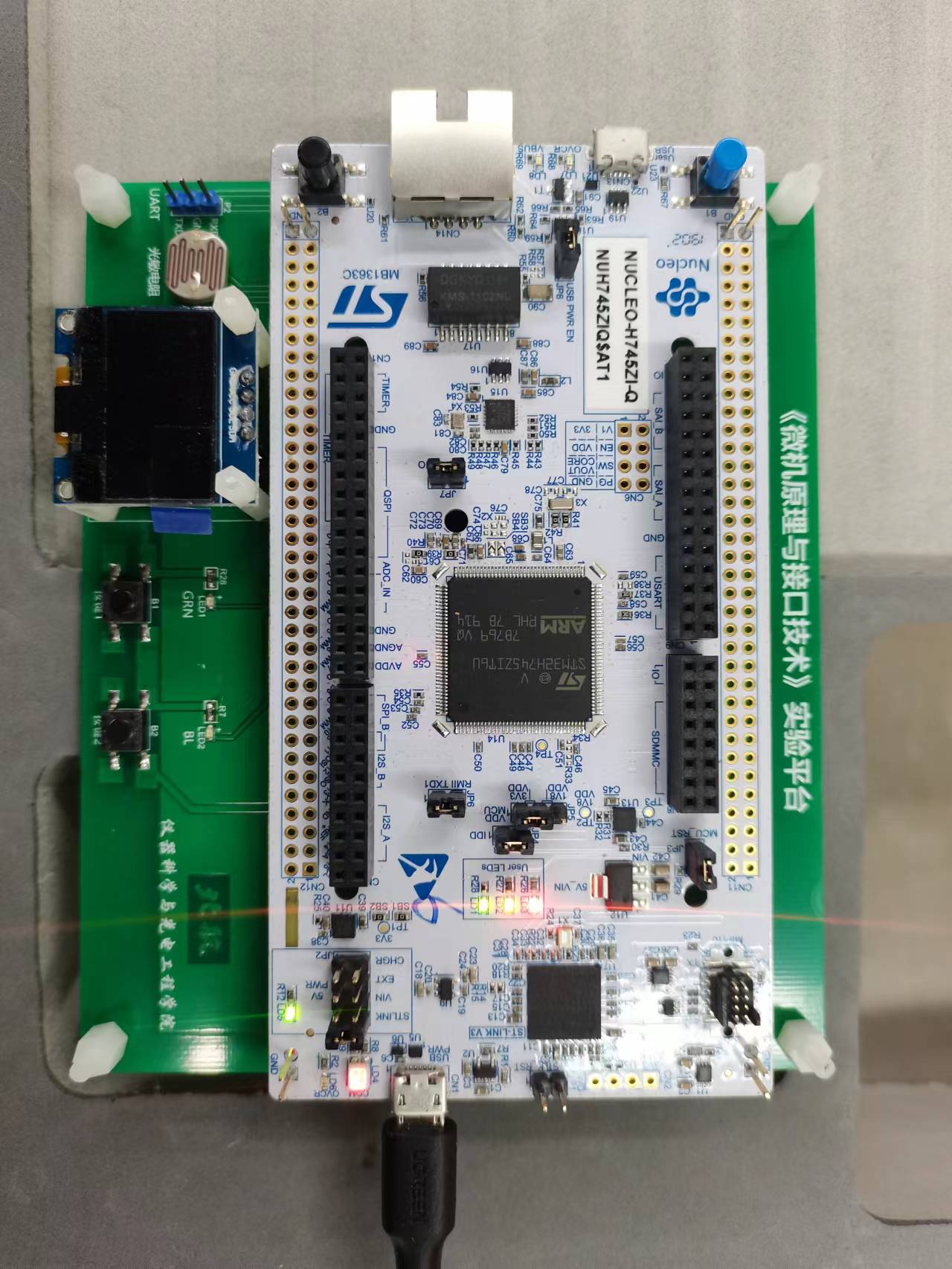


图 40：三个LED灯亮

1. 用硬延时 delay，实现 LED1 的循环亮灭（闪烁）

在main函数使用while循环语句并配合HAL\_Delay 函数和 HAL\_GPIO\_WritePin函数即可达到LED1闪烁的效果（视频在附件中），代码如图所示：

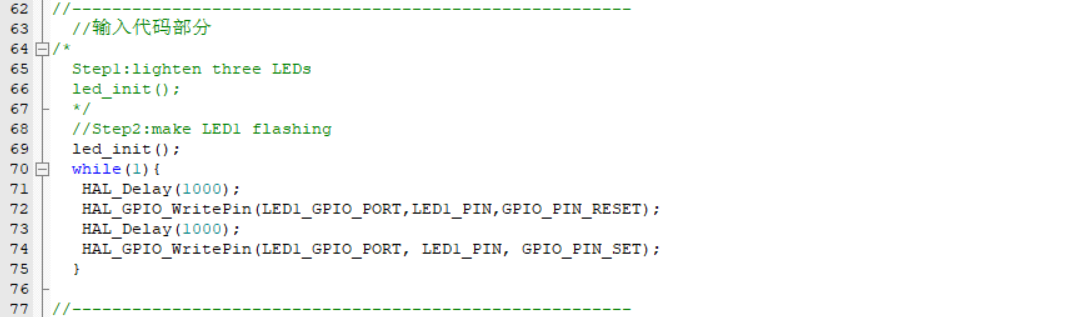


图 41：实现LED1的闪烁

1. 接收按键输入信号，当按键按下，点亮 LED1；当按键断开，LED1 熄灭

在stm32h7xx\_nucleo.h 文件中找到按键的宏定义：

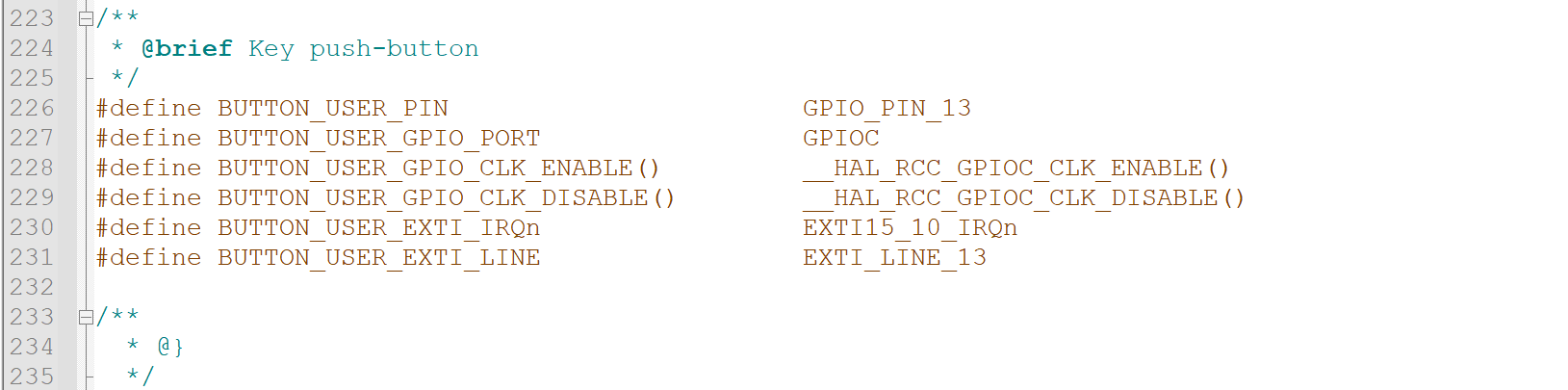


图 42：按键的宏定义

key\_init函数的函数声明：



图 43：key\_init函数的函数声明

key\_init函数的函数定义：

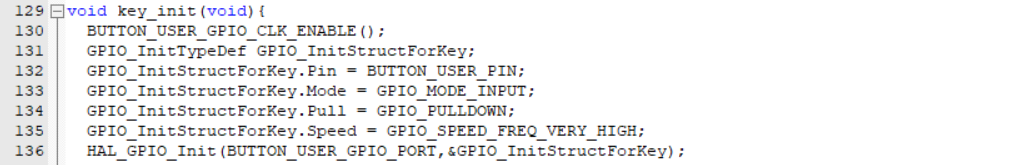


图 44：key\_init函数的函数定义

main()函数里调用key\_init 函数完成按键的初始化，随后完成按键控制 LED1 的亮灭操作：

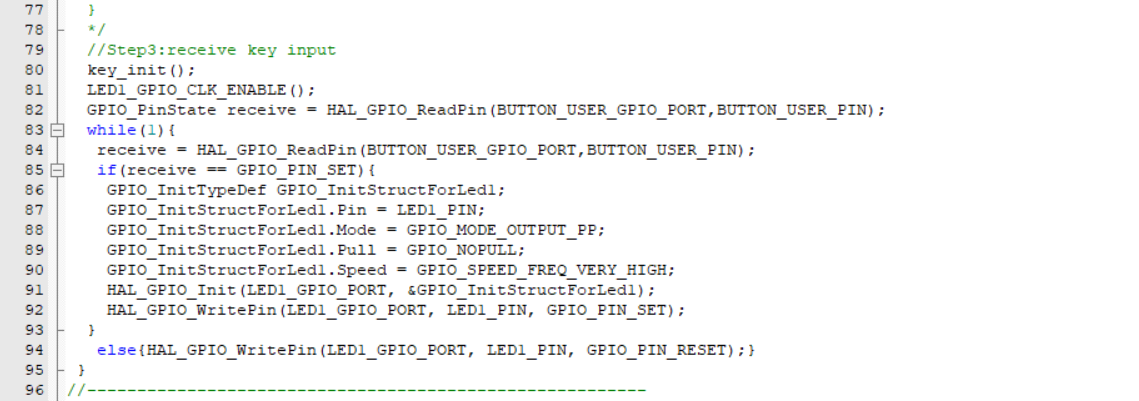


图 45：实现按键控制灯的亮灭

调试过程如下：

* 全速执行到断点处：

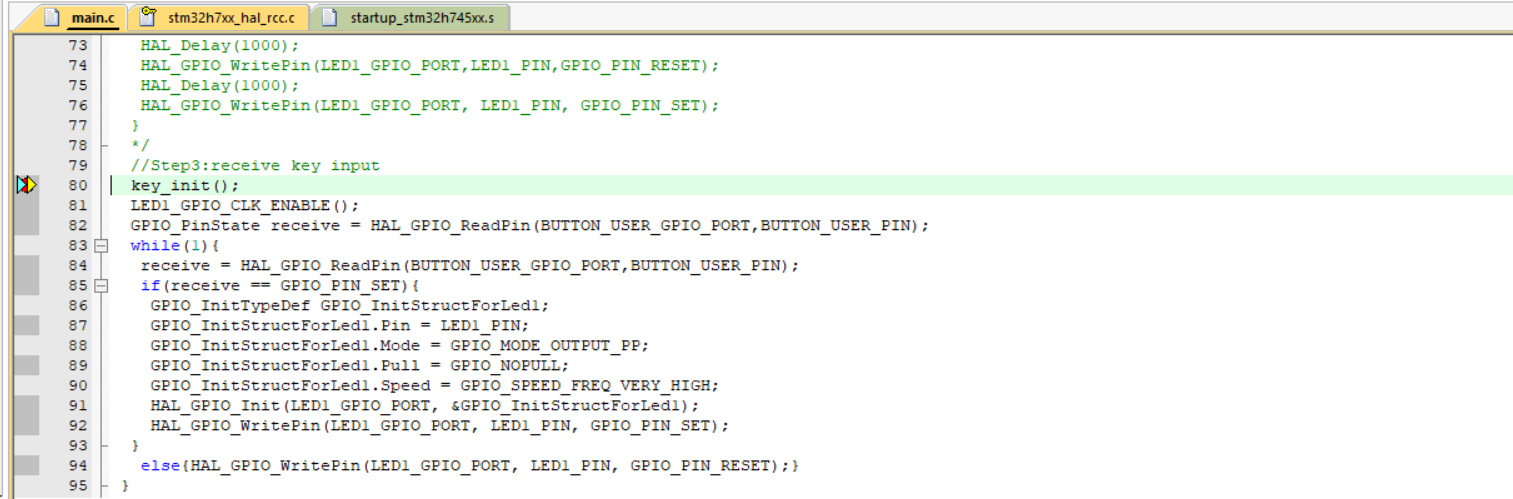
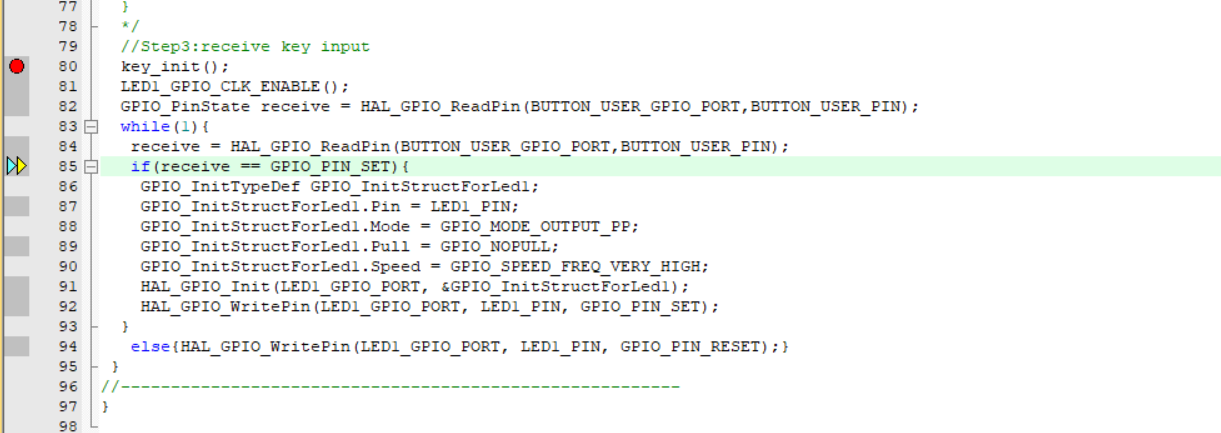


图 46：全速执行到断点

第81行代码将LED1对应端口的实在使能，为灯的亮灭作准备，第82行代码定义一个GPIO\_PinState 型变量用以接收key状态，83-93行的循环体中不停判断key是否处于按下的状态来控制LED1的亮灭。

按键不按下时，从图47可知receive中的值为0（GPIO\_PIN\_RESET），灯不亮，如图48所示。



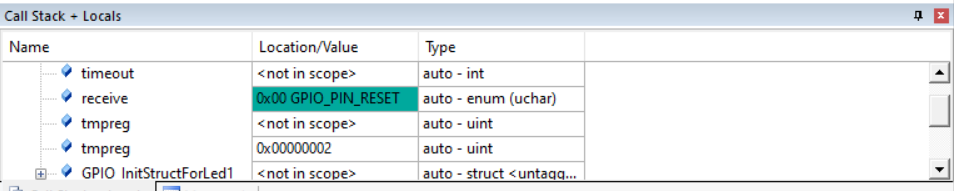


图 47：按键不按

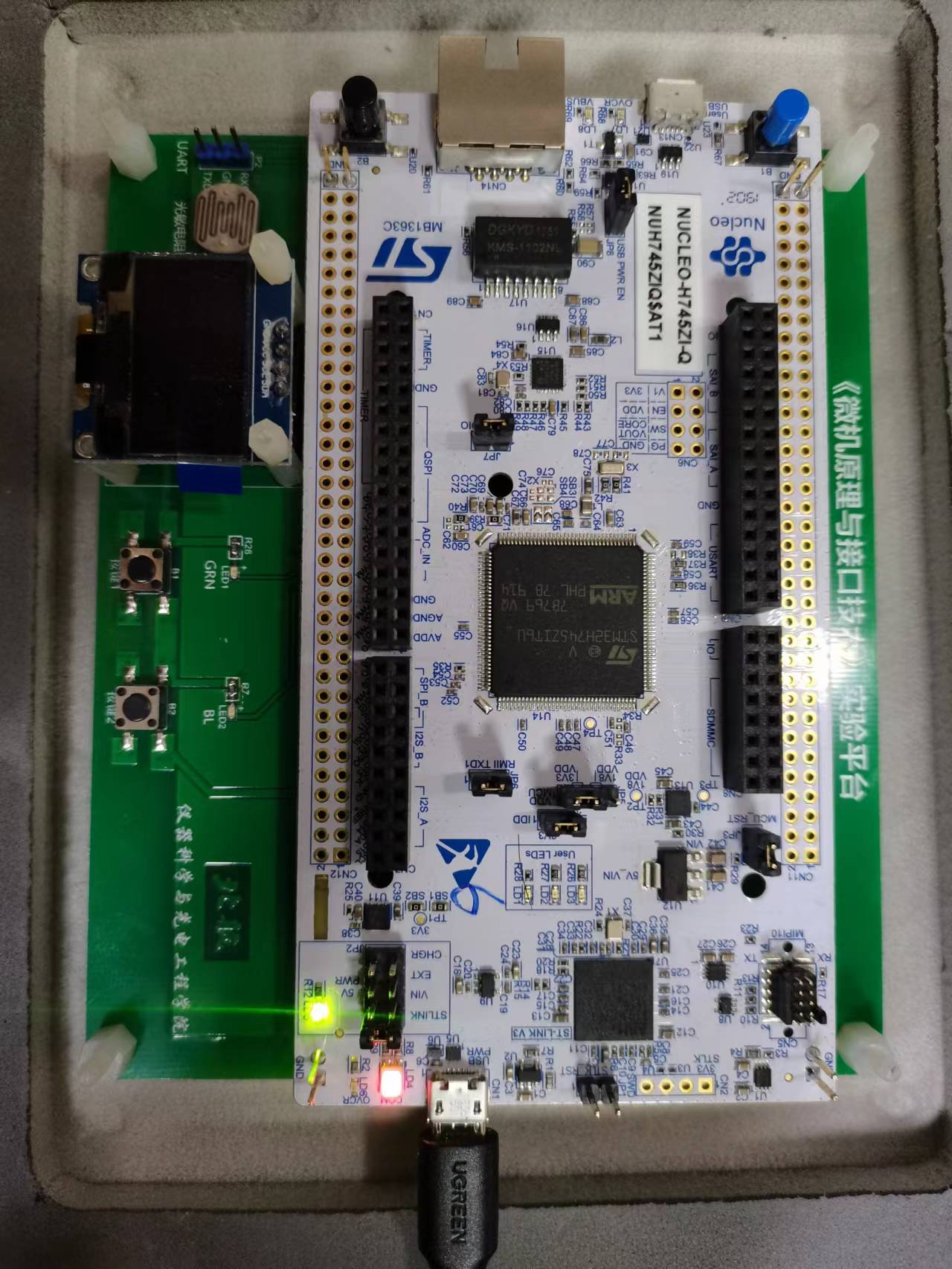
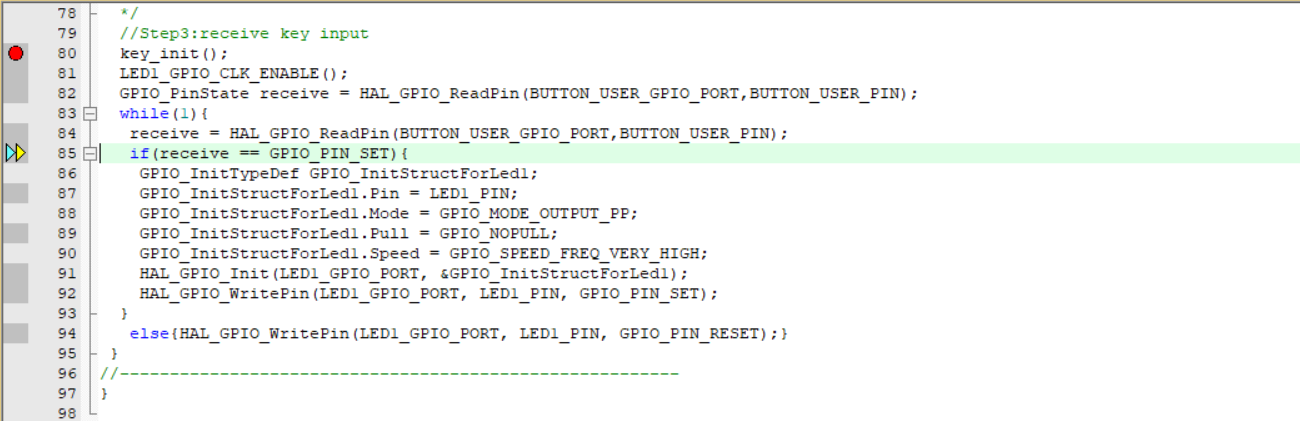


图 48：按键不按时的现象

按键按下时，从图49可知receive中的值为1（GPIO\_PIN\_SET），灯亮，如图50所示。



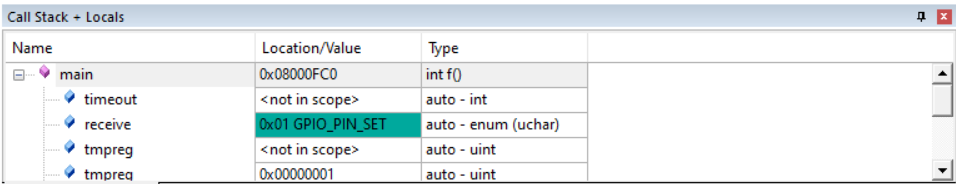


图 49：按下按键



图 50：按键按下时的现象

具体现象见附件中的视频。

1. 按一下按键 LED 灯点亮，再按一下 LED 灯熄灭。

用以实现该功能的代码如图51所示，具体现象见附件中的视频。



图 51：步骤4代码

1. 按键消抖实验：用硬延时 delay，检测按键是否按下。（本实验用消抖的思想完善第四步的功能）

代码如图52所示，具体现象见附件中的视频，按键质量挺好的，消抖前和消抖后均可正常完成按一下按键 LED 灯点亮，再按一下 LED 灯熄灭的功能。

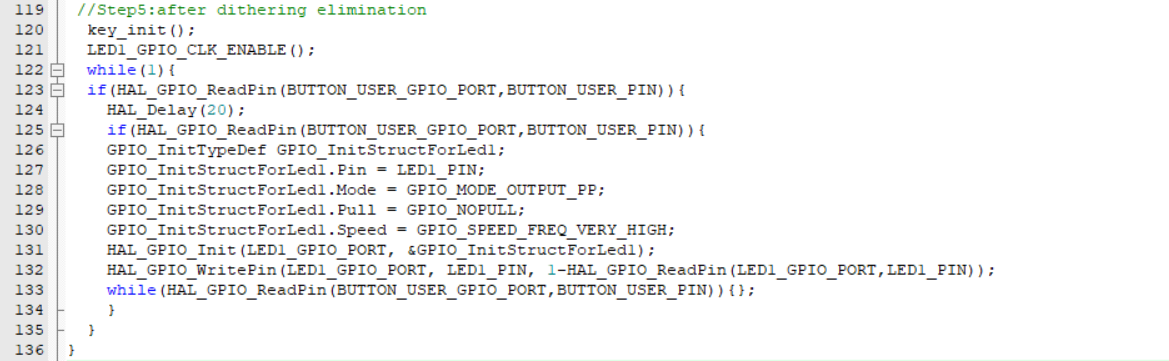


图 52：步骤5代码

**（四）GPIO提高实验1**

通过按键切换，实现 LED 灯的快闪，慢闪，双闪等功能。

初始时刻，Nucleo 板三个 LED，Disco 板二个 LED 灯初始化保持在点亮状态，首次按下按键时，LED1 开始快速闪烁，其余 LED 灯保持点亮。再次按下按键时，LED2 开始慢速闪烁，其余 LED 灯保持点亮。再次按下按键，LED 灯同时开始闪烁。再次按下按键恢复至初始点亮状态。

代码如图53所示，这里使用了HAL\_GetTick()函数充当计时功能，并据此实现闪烁。之所以不能使用硬延时来实现闪烁，是因为一旦进入硬延时delay函数，则无法检测按键是否被按下，而这不无法容许的。

实验现象见附件的视频。



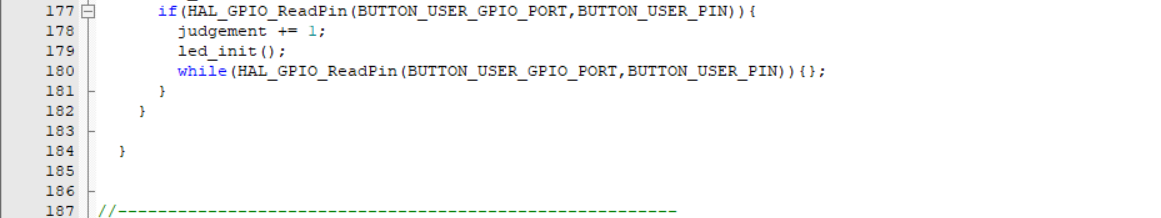


图 53：提高实验1代码

**（五）GPIO提高实验2**

实现长按住按键 5s 后，LED1 灯开始闪烁；短按按键时 LED1 灯实现亮灭切换，即短按按键 LED1 熄灭，再次短按按键 LED1 点亮，每次短按按键时长最长不超过1s。

实现计算按键按下的时间依旧使用了HAL\_GetTick()函数，具体代码如图54所示。

实验现象见附件的视频，长按过程中会保持按之前状态不变，结束后模式切换。

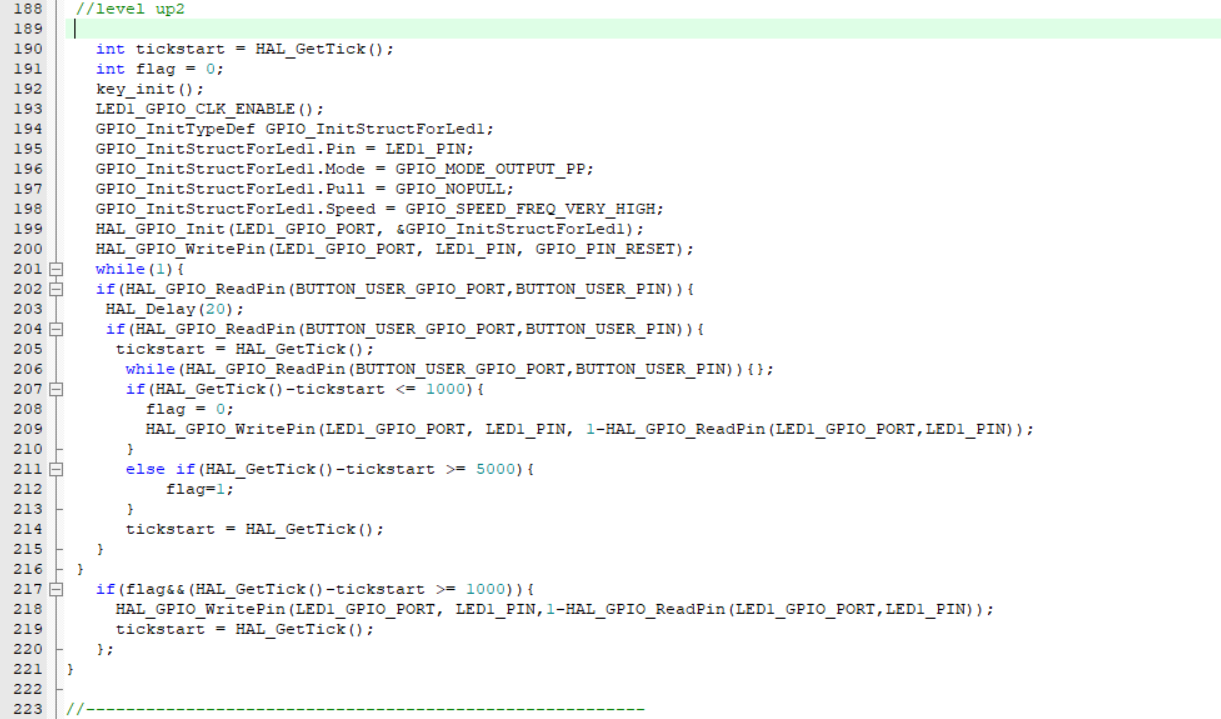


图 54：提高实验2代码