# 嵌入式系统实验报告



|  |  |
| --- | --- |
| 实验名称： | Cortex-M 处理器异常处理与系统调用 |
| 姓 名： | 王小龙 |
| 学 号： | 2020211502 |
| 学 院(系)： | 计算机学院 |
| 专 业： | 网络工程 |
| 指导教师： | 戴志涛、刘健培 |

2022年 11 月 18 日

# 实验目的

1. 了解开发环境的使用。
2. 学会通过查阅文档和数据手册获取信息。
3. 掌握 cortex-M4 体系结构中核内寄存器和异常处理的使用方式。
4. 理解处理器异常上下文切换的实现方式。
5. 了解系统调用的实现方式。
6. 掌握基本的软件编写与调试方式。

# 实验环境

1. FS-STM32F407开发平台
2. ST-Link 仿真器
3. RealView MDK5.23集成开发软件
4. 串口调试工具
5. PC机Window7/8/10 (64bit)

# 实验要求

1. 本实验通过编写系统调用理解 Cortex-M 处理器异常处理的特点与流程。

2. 在给定的参考代码基础上，参考并分析已实现的系统调用框架，实现一个完

整的系统调用函数。

3. 二选一：可以选择实现指定的系统调用，也可以选择实现自拟的系统调用：

 指定：基于 SVC 指令实现一个提升特权的系统调用。即首先将程序设置到用户模式，然后调用此系统调用，通过异常进入特权模式，在特权模式中将程序权限提升到特权模块，异常返回后，用户程序将以特权模式运行。

 自拟：系统调用的功能自拟，例如，可以读取系统 Systick 时间、调用内核 API、控制任务状态、操作外设、控制用户 login 等等，简单的也可以打印输出、做简单的数学运算等等。

4. 编写一个测试程序验证所编系统调用成功执行。

# 实验原理

**1、SVC 异常与异常处理流程**

**2、Cortex-M4 处理异常的流程：**

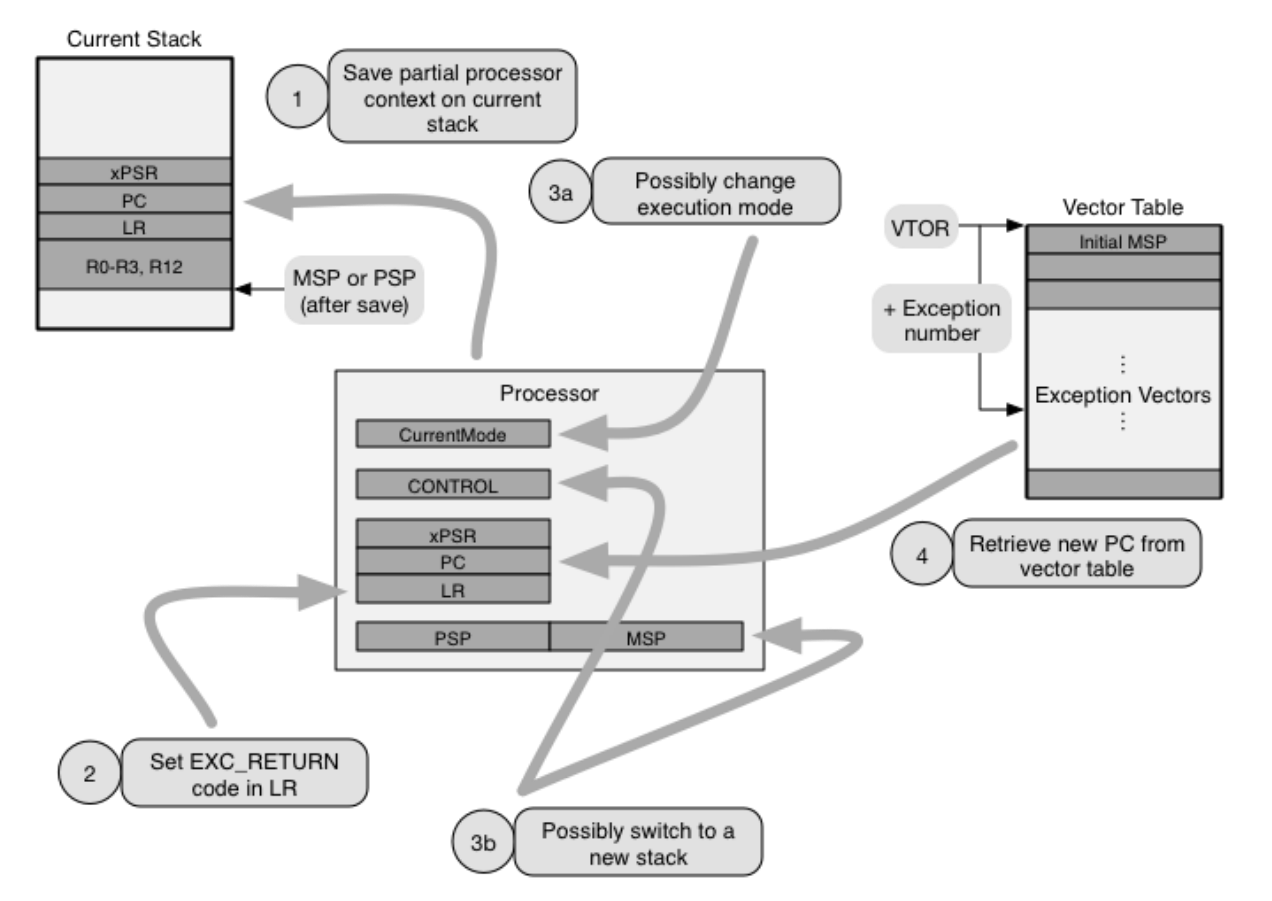
异常发生后，需要软件和硬件协作处理。

进入异常时，硬件会切换模式，并保存必要的寄存器。然后异常处理程序（软件）再根据需要保存需要的额外寄存器。

保护好现场后，就可以进行额外的处理。此时可以进一步准备好调用 c 函数需要的堆栈，然后调用 c 函数进行进一步处理。

退出异常时，流程基本与进入异常时相反，首先软件需要恢复部分寄存器，然后通过机器指令让硬件恢复之前硬件保存的寄存器，并跳转到异常发生时的地址继续执行（如果需要）。

处理器进入异常的处理流程大致如下：



**3、系统调用的原理**

系统调用（system call），指运行在用户空间的程序向操作系统内核请求需要更高权限运行的服务。系统调用提供用户程序与操作系统之间的接口。系统调用和普通库函数调用非常相似，只是系统调用由操作系统内核提供，运行于内核核心态，而普通的库函数调用由函数库或用户自己提供，运行于用户态。Linux 在 x86 上的系统调用通过 int 80h 实现，用系统调用号来区分入口函数。操作系统实现系统调用的基本过程是：

1. 应用程序调用库函数（API）；
2. API 将系统调用号存入 EAX，然后通过中断调用使系统进入内核态；
3. 内核中的中断处理函数根据系统调用号，调用对应的内核函数（系统调
4. 用）；
5. 系统调用完成相应功能，将返回值存入 EAX，返回到中断处理函数；
6. 中断处理函数返回到 API 中；
7. API 将 EAX 返回给应用程序。

应用程序调用系统调用的过程是：

1. 把系统调用的编号存入 EAX；
2. 把函数参数存入其它通用寄存器；
3. 触发 0x80 号中断（int 0x80）。

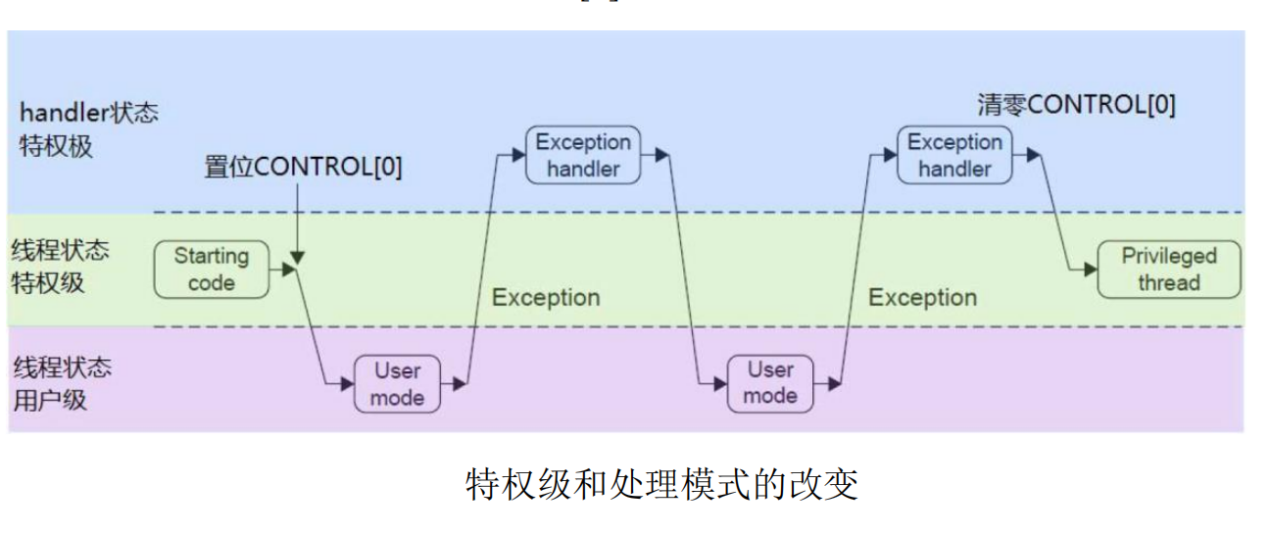
在嵌入式系统环境，嵌入式操作系统与用户应用程序往往链接在一个可执行文件中，往往也不区分用户程序与操作系统内核的权限。有些嵌入式操作系统也提供系统调用作为一个可配置的选项，大部分则默认将用户应用程序作为特权任务运行。

**4、Cortex-M 切换处理器特权级别**

当程序运行在用户态时，权限是受限的。在线程模式＋用户级下，对系统控制空间（SCS）的访问将被阻止。除此之外，还禁止使用 MSR 访问特殊功能寄存器（除了 APSR 有例外）。

在特权级下的代码可以通过置位 CONTROL[0]来进入用户级。而不管是任何原因产生了任何异常，处理器都将以特权级来运行其服务例程，异常返回后将回到产生异常之前的特权级。用户级下的代码不能再试图修改 CONTROL[0]来回到特权级。它必须通过一个异常 handler，由那个异常 handler 来修改CONTROL[0]，才能在返回到线程模式后拿到特权级。

CONTROL［0］只有在特权级下才能访问。用户级的程序如想进入特权级，通常都是使用一条“系统服务呼叫指令（SVC）”来触发“SVC 异常”，该异常的服务例程可以选择修改 CONTROL[0]。

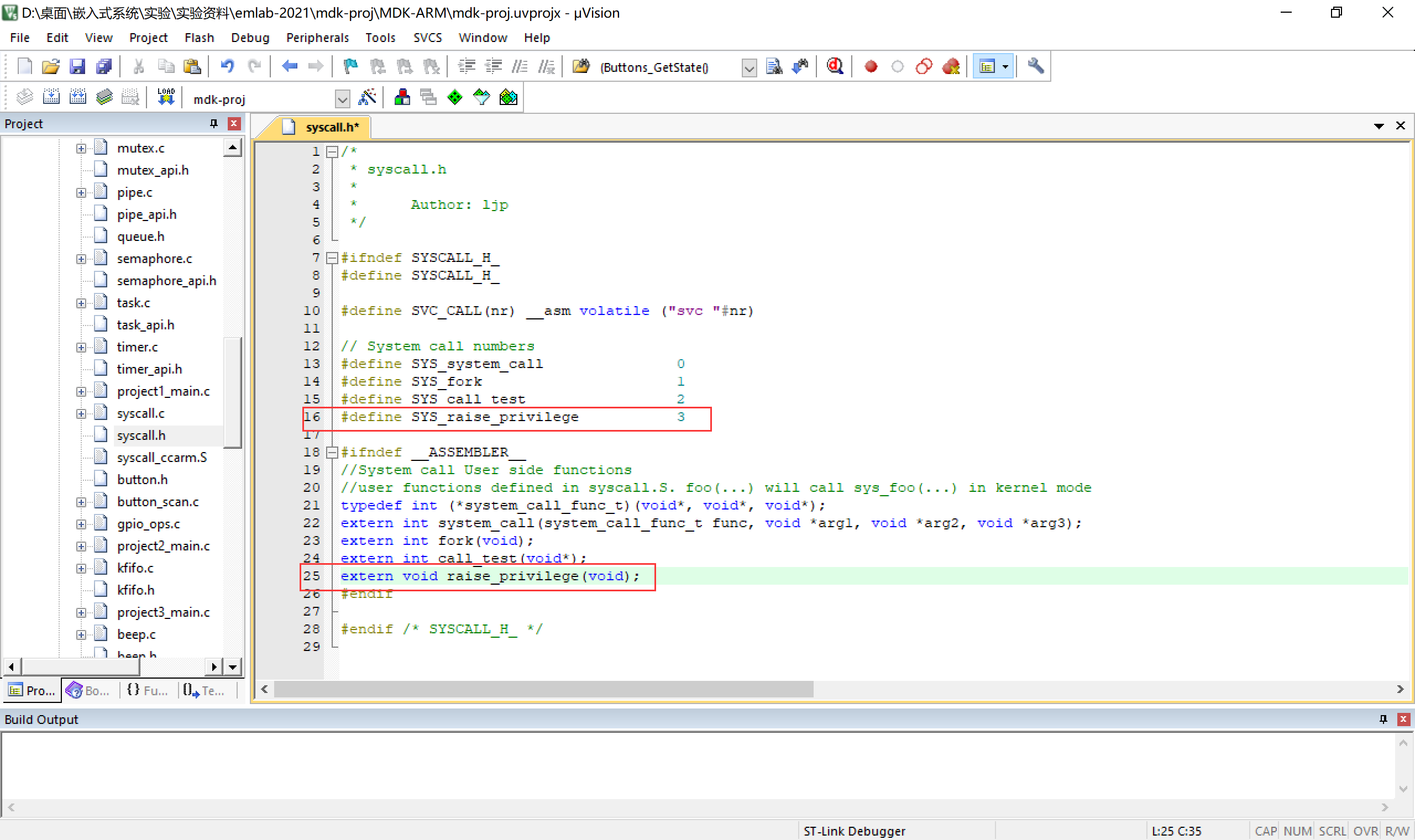


# 实验步骤

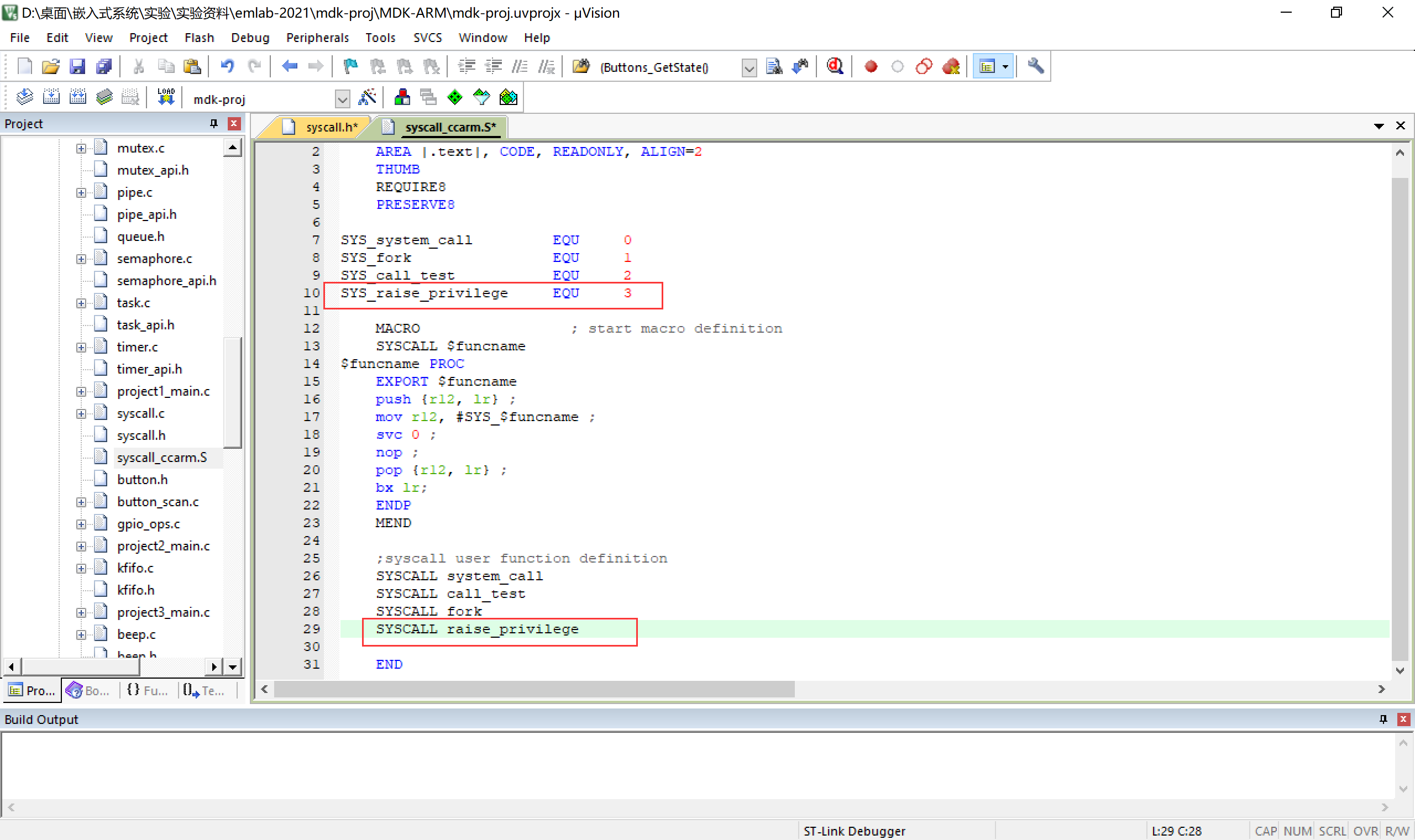
1、编写系统调用函数。

申明用户侧系统调用函数和系统调用号。

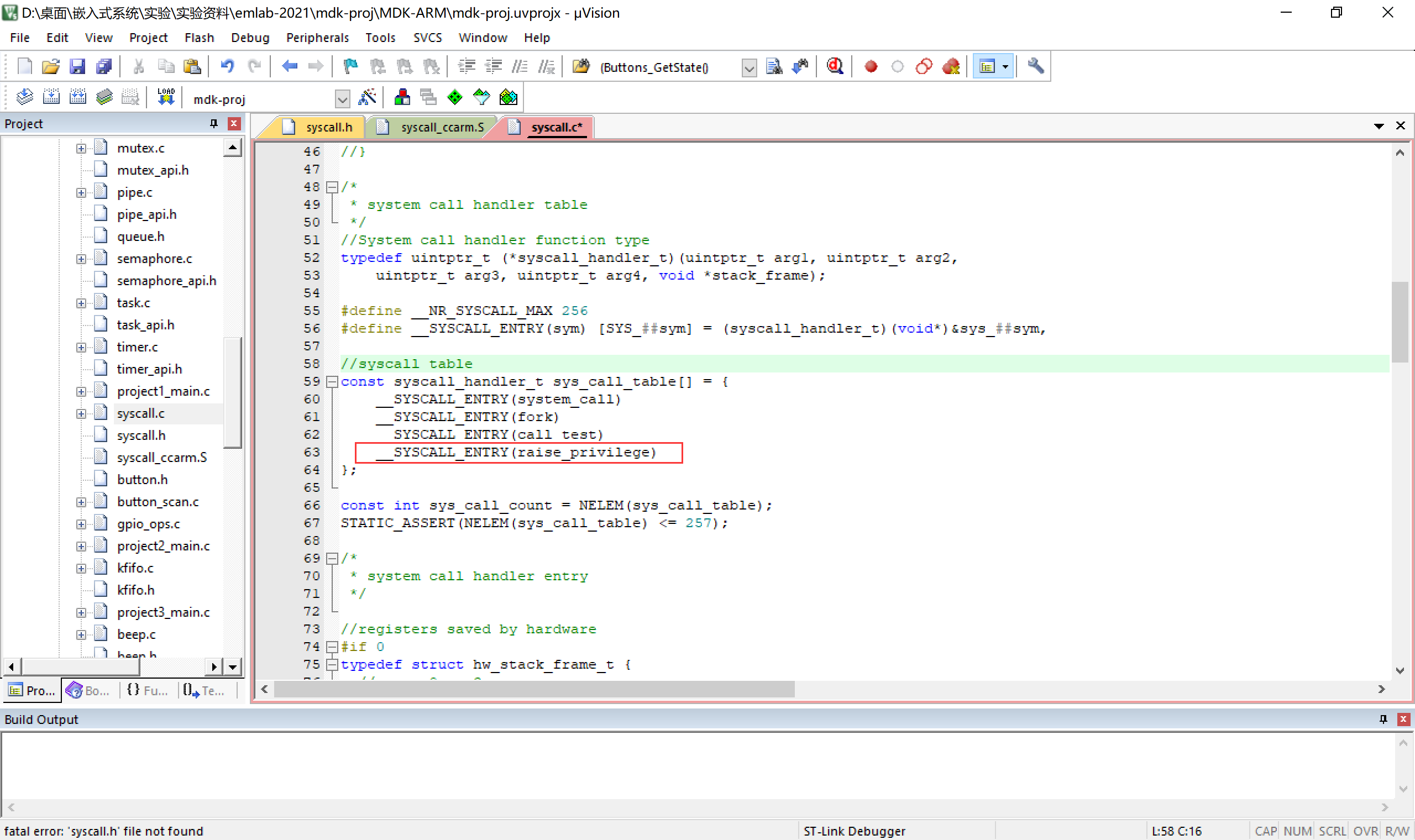
在syscall.h中申明用户侧函数 raise\_privilege，并定义其使用的系统调用号SYS\_raise\_privilege。



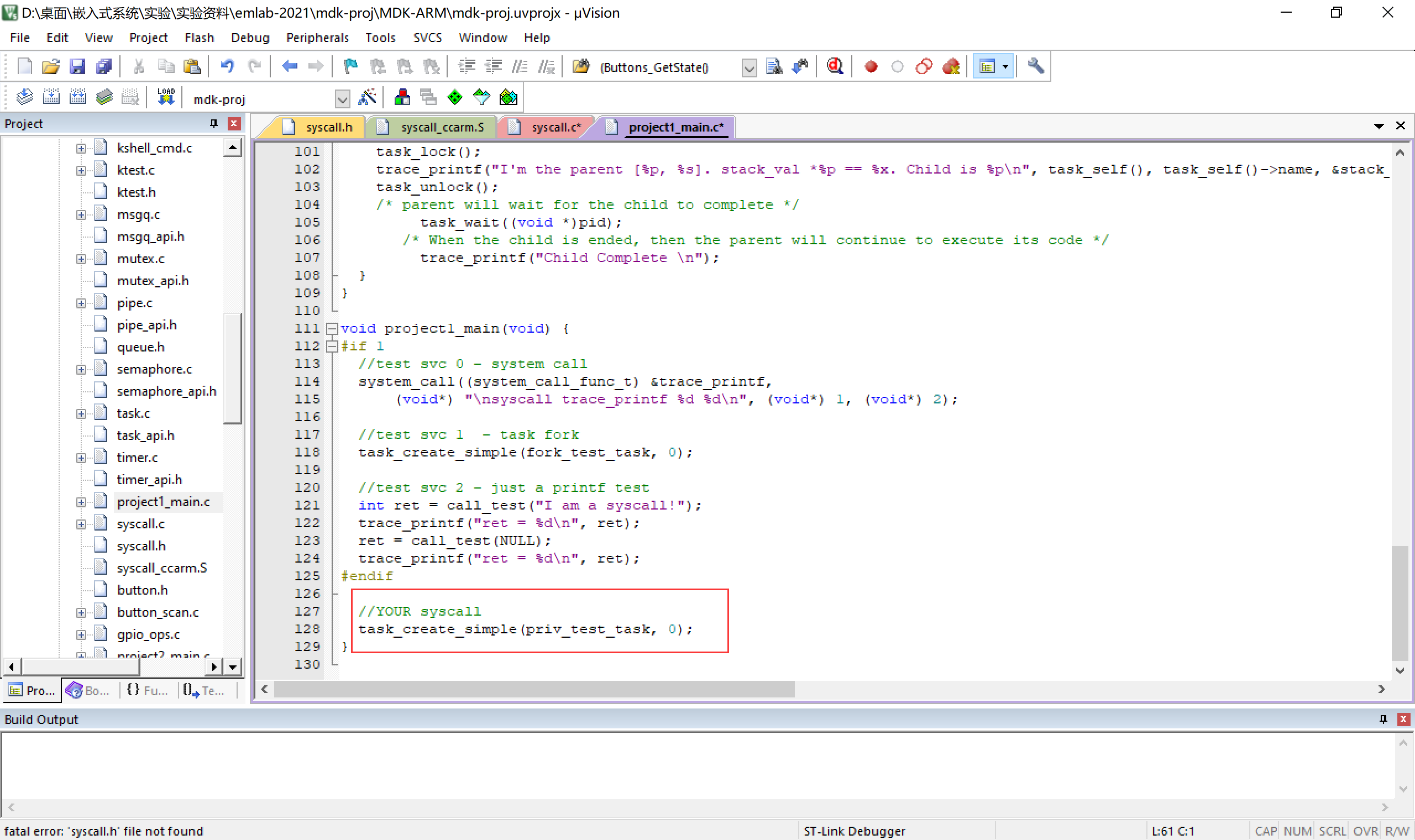
接着在 syscall\_ccarm.S中使用 EQU 伪指令定义系统调用号（与 syscall.h 中定义的值相同），并使用宏 SYSCALL 实现 raise\_privilege 汇编函数体。



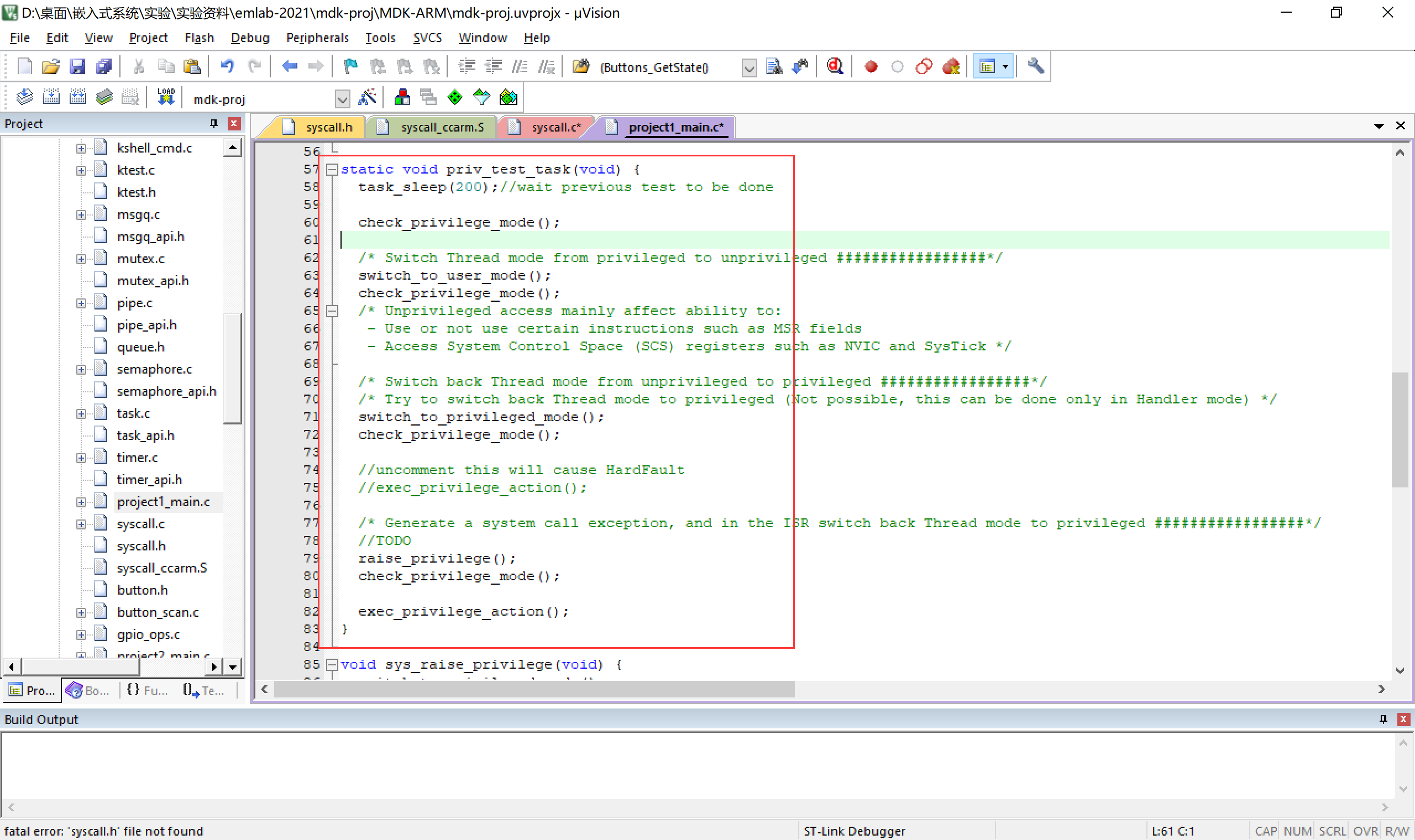
在syscall.c中的 sys\_call\_table[]表中增加表项 ：\_\_SYSCALL\_ENTRY(raise\_privilege)



2、在project1\_main函数中找到编写的系统调用的入口：



3、在 project1\_main.c 中编写测试程序如下：



上述static void priv\_test\_task(void)代码中，首先创建了一个运行在特权等级的任务，然后等待任务完成，接着通过check\_privilege\_mode()查看当前的状态，然后通过switch\_to\_user\_mode();任务将自身切换到用户态（是通过设置 CONTROL［0］=1实现的，任务在特权态是可以访问 CONTROL［0］的）。

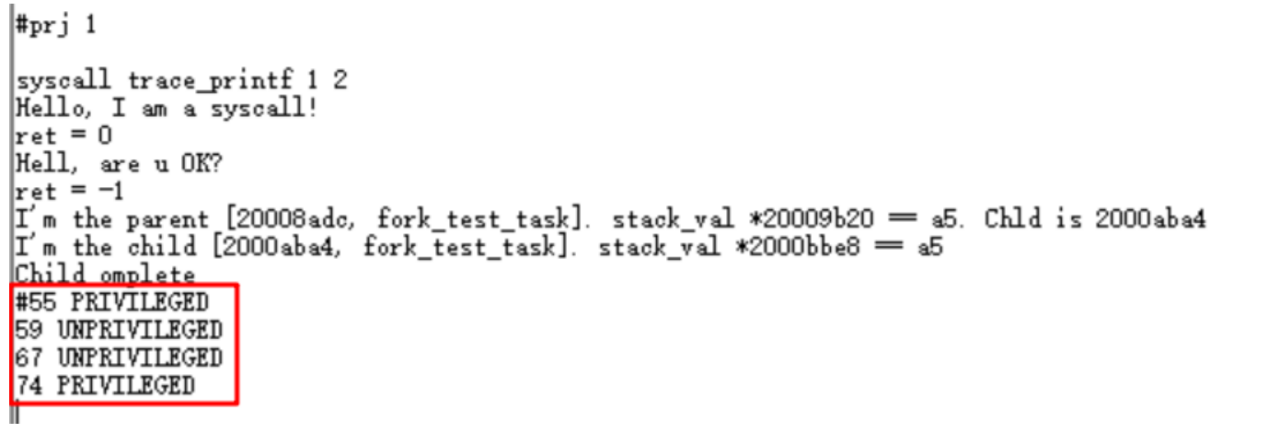
接下来调用的switch\_to\_privileged\_mode();没有起到作用，还是处于用户态，原因是此时任务处于用户态，无法调用特权操作。如果直接调用切换特权的函数，是没有任何效果的。如果调用其余特权指令，例如，访问systick或者NVIC寄存器等系统空间，会导致 HardFault 异常。后面用check\_privilege\_mode()查看当前的状态。

之后调用系统调用 raise\_privilege 函数切换特权。这是我们编写的系统调用，由这个函数调用陷入特权态，并在特权态设置 CONTROL［0］。此时任务就处于特权级别了，可以调用特权操作。

1. 调用测试程序，检查测试的输出：

在 sscom 中输入 prj 1 命令，将调用 project1\_main 函数。

程序输出如下：



符合预期。

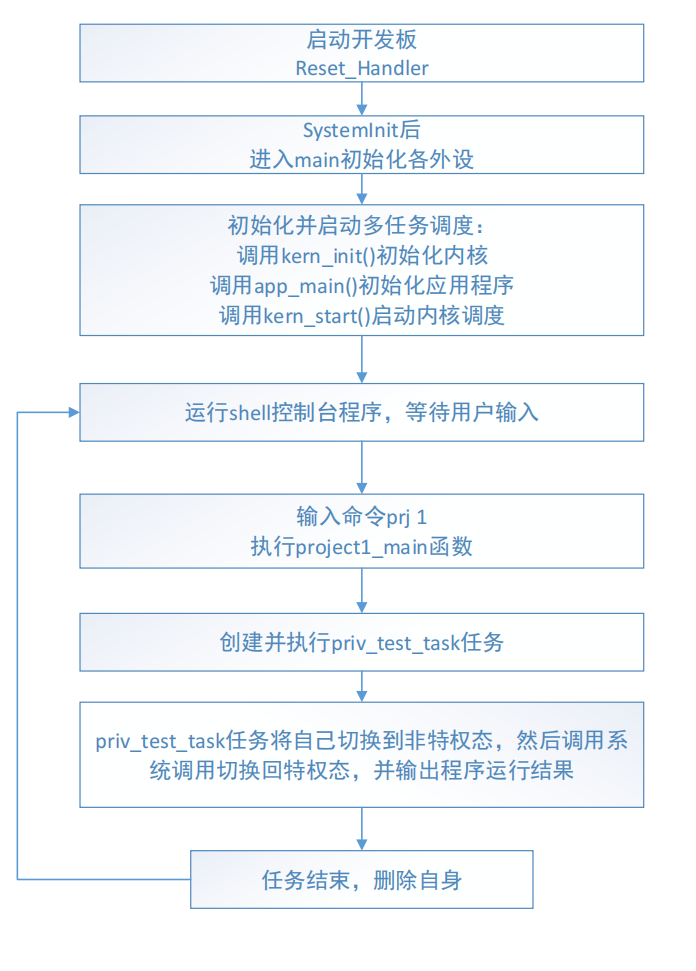
# 实验方案与实现

## 软件结构

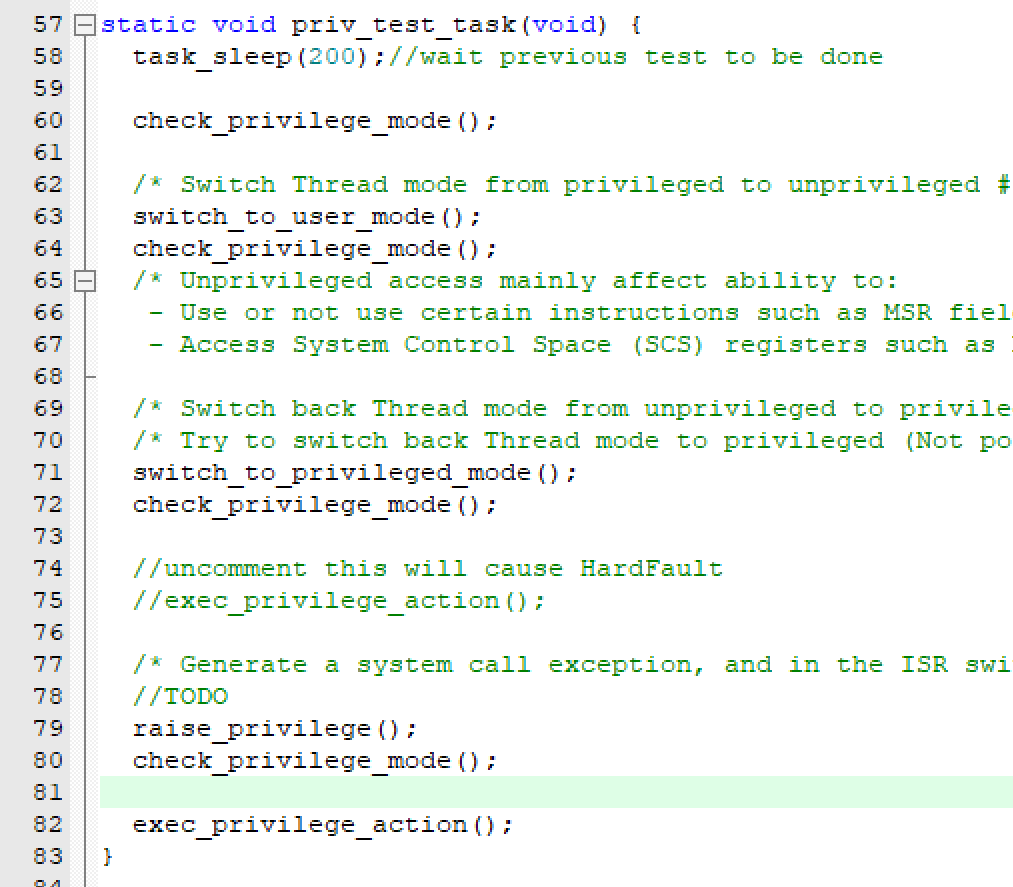
在MDK软件中编译成功后，进入调试模式，然后启动开发板Reset\_Handler，之后进行SystemInit，以及进入main初始化各外设，然后再初始化并启动多任务调度。再然后，运行shell控制台程序，输入命令prj 1，执行project1\_main函数，创建并执行priv\_test\_task任务，

priv\_test\_task任务将自己切换到非特权态，然后调用系统调用切换回特权态，并输出程序运行结果。

执行流程如下：



## 源代码



# 实验结果与分析

1.请描述系统调用的过程（从用户模式下开始系统调用到系统调用完成）。在此过程中，处理器的状态与进入异常处理时的堆栈是如何变化的？

答：通常情况下，调用系统调用和调用一个普通的自定义函数在应用层代码上并没有什么区别，但调用后发生的事情有很大不同。调用自定义函数是通过 bl指令直接跳转到该子函数的地址，继续运行。

而调用系统调用，实质上是去调用一个异常指令 svc，触发处理器的异常处理过程，切换处理器到内核态，在此期间将原函数对应的系统调用号和参数通过堆栈（or 寄存器，不同实现方式）传入到内核态，内核态中查表调用真正的实现函数，并将结果通过异常返回。所以用户态程序无法控制调用的子函数如何具体实现，只能通过传递信息的方式请求(call)系统(system)去完成该函数的功能。系统调用相当于一扇门，隔开了用户态/非特权级和内核态/特权级，使得敏感的特权指令在内核程序的控制之下，用户态的应用程序无法直接执行特权指令（否则 fault 异常伺候），需要通过这个“门”“喊话（call）”，引起内核态的“注意”，才能委托内核完成功能。这是处理器支持操作系统特权的一种系统机制。

异常发生后，需要软件和硬件协作处理。

进入异常时，硬件会切换模式，并保存必要的寄存器。然后异常处理程序（软件）再根据需要保存需要的额外寄存器。

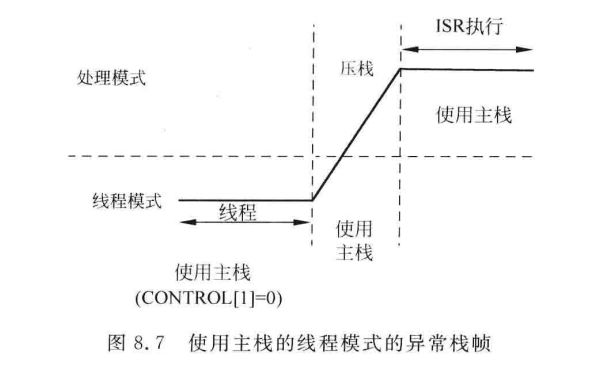
保护好现场后，就可以进行额外的处理。此时可以进一步准备好调用 c 函数需要的堆栈，然后调用 c 函数进行进一步处理。

退出异常时，流程基本与进入异常时相反，首先软件需要恢复部分寄存器，然后通过机器指令让硬件恢复之前硬件保存的寄存器，并跳转到异常发生时的地址继续执行（如果需要）。

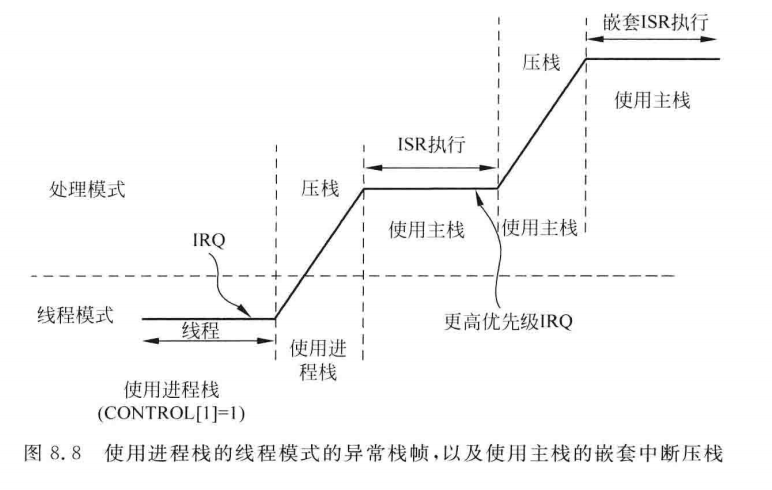
Cortex-M3和Cortex-M4处理器具有多个总线接口。在压栈操作的同时，处理器还可以开始取向量和取指。这样，由于压栈操作可以和Flash存储器访问同时进行，哈佛总线架构可以降低中断等待时间。若向量表位于SRAM或异常也位于SRAM，中断等待会稍微增加。

压栈操作中用的栈可以为主栈（使用主栈指针，MSP）或进程栈（使用进程栈，PSP）。

若处理器运行在线程模式且使用MSP（CONTROL寄存器的第0位为0，默认配置），则压栈操作在执行时使用主栈MSP。如下图：

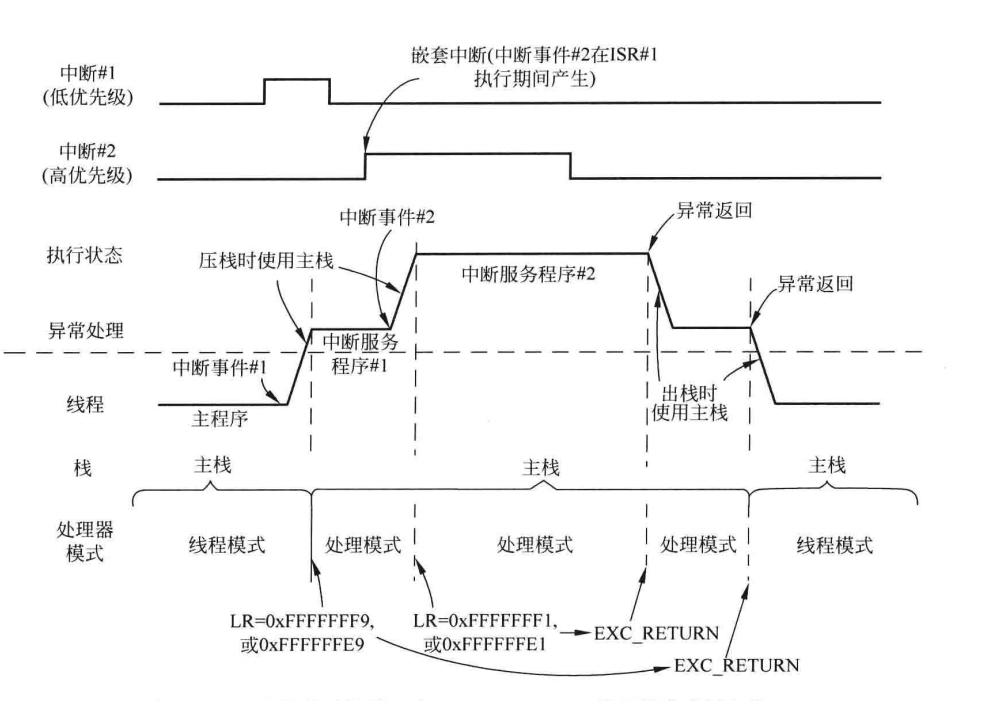


处理器运行在线程模式且使用进程栈（CONTROL寄存器的第1位为1），则压栈操作执行时使用进程栈PSP。在进入处理模式后，处理器必须使用MSP，所有嵌套中断的压栈操作执行时都使用主栈MSP。如下图：

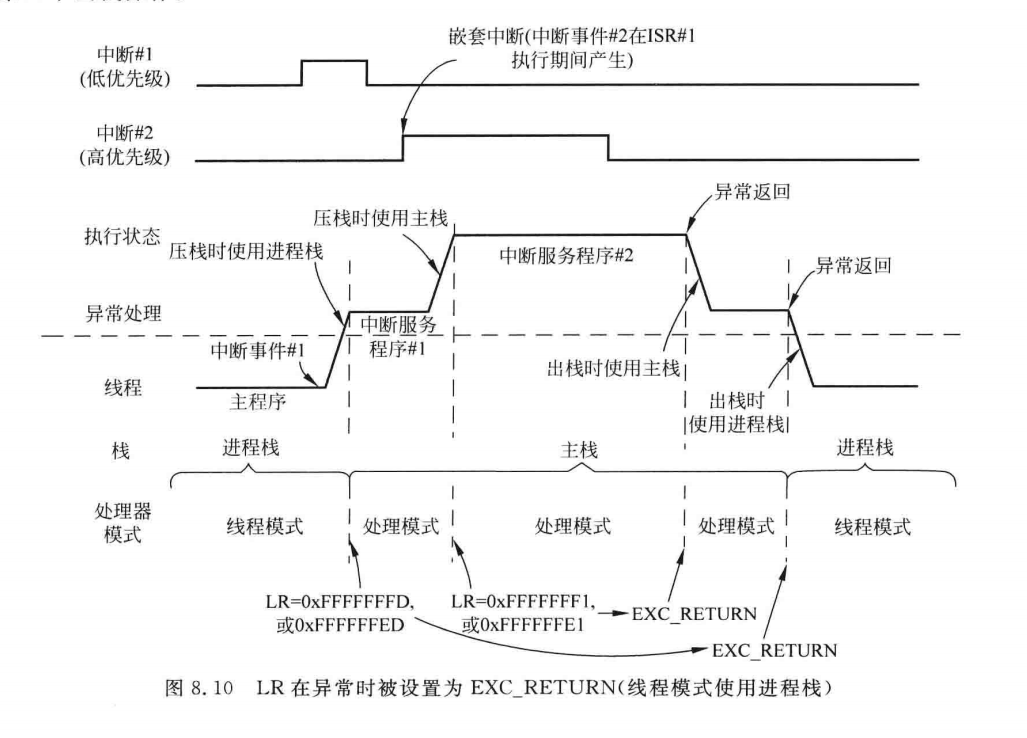


2.系统调用的返回值是何返回给调用者的？

在异常处理结束时，异常入口处生成的EXC\_RETURN数值的第2位用于确定提取栈帧时所用的栈指针。若第2位为0，则处理器会知道之前压栈时使用的是主栈，如下图；



若第2位为1，处理器就会知道压栈时使用的是进程栈。如下图：



知道了系统调用时所使用的栈指针，就可以找到调用前的栈，之后就可以读取栈中存储的值。

3. 实验中实现系统调用的方式最多能传递几个参数？为什么？

用于ARM架构的C编译器遵循ARM的一个名为AAPCS的规范。根据这份标准，C函数可以修改RO~R3、R12、R14（LR）以及PSR。若C函数需要使用R4~R11，就应该将这些寄存器保存到栈空间中，并且在函数结束前将它们恢复。

RO~R3、R12、LR以及PSR被称作“调用者保存寄存器”，若在函数调用后还需要使用这些寄存器的数值，在进行调用前，调用子程序的程序代码需要将这些寄存器的内容保存到内存中（如栈）。函数调用后不需要使用的寄存器数值则不用保存。

R4~R11为“被调用者保存寄存器”，被调用的子程序或函数需要确保这些寄存器在函数结束时不会发生变化（与进入函数时的数值一样）。这些寄存器的数值可能会在函数执行过程中变化，不过需要在函数退出前将它们恢复为初始值。

若Cortex-M处理器具有浮点单元,则浮点单元中的寄存器也有类似的需求：

* S0～S15为“调用者保存寄存器”。
* S16～S31为“被调用者保存寄存器”。

一般来说，函数调用将R0~R3作为输入参数，R0则用作返回结果。若返回值为64位，则R1也会用于返回结果。

1. 实验中使用的教学 RTOS 进行任务切换时，需要切换处理器状态（即核内寄存器），切换函数也是使用异常（pendsv 异常）实现的，是否理解通过 pendsv异常进行处理器状态切换的原理与过程？

PendSV(可挂起的系统调用)异常对OS操作也非常重要,其异常编号为14且具有可编程的优先级。可以写入中断控制和状态寄存器(ICSR)设置挂起位以触发PendSV异常。与SVC异常不同,它是不精确的。因此,它的挂起状态可在更高优先级异常处理内设置,且会在高优先级处理完成后执行。

利用该特性,若将PendSV设置为最低的异常优先级﹐可以让PendSV异常处理在所有其他中断处理任务完成后执行。这对于上下文切换非常有用,也是各种OS设计中的关键。

若中断请求(IRQ)在SysTick异常前产生,则 SysTick 异常可能会抢占IRQ处理。在这种情况下,OS不应执行上下文切换。否则,IRQ处理就会被延迟，为了解决这个问题, PendSV异常将上下文切换请求延迟到所有其他IRQ处理都已经完成后,此时需要将PendSV设置为最低优先级。若OS需要执行上下文切换,它会设置PendSV的挂起状态,并在 PendSV异常内执行上下文切换。

# 实验总结

由于是第一次接触这个软件，起初有一些迷茫，但后来认真看了实验指导书，并请教了周围同学后，逐渐找到了思路，最后成功得到了实验预期结果。

经过这次实验，了解了系统调用和异常处理有关的知识，同时锻炼了查阅资料的能力。