

微机原理与嵌入式系统自主实验实验报告

姓名：麦华煜

学号：PB17061254

实验一 基于 ASM 的 Project

一、实验目的

- 1、掌握µVision IDE 基本使用、了解一个项目编译、连接、调试的工作过程。
- 2、汇编代码编写的一般语法，掌握编写子程序的方法。
- 3、掌握常规代码调试技巧。
- 4、理解编程者模型。

二、实验内容

1、Project 的建立、编译、连接

(1) ustc_sample_asm 的建立

(2) 编译、连接 Project

2、Project 的 debug

(3) 调试

(4) 观察调试过程中通用寄存器的变化

开始调试:

Register	Value
Core	
R0	0x00000000
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000400
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000400
xFSR	0x01000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	0
Sec	0.00000000
FPU	

结束调试:

Register	Value
Core	
R0	0x20000800
R1	0xABABABAB
R2	0xABABABAB
R3	0x00000003
R4	0x00000001
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000400
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000446
xFSR	0xA1000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	69
Sec	0.00000575
FPU	

(5) 观察如下代码执行前后栈指针的变化:

```
57      MOV R1, #0x1
58      MOV R3, #0x3
59      PUSH {R1}           ;将R1压入堆栈
60      PUSH {R3}           ;将R3压入堆栈
61      POP {R2}            ;从堆栈弹出至R2
62      POP {R4}            ;从堆栈弹出至R4
```

执行前:

R13 (SP)	0x20000400
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x0800042A

执行第 59 行代码后:

R13 (SP)	0x200003FC
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000434

执行第 60 行代码后:

R13 (SP)	0x200003F8
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000436

执行第 61 行代码后:

R13 (SP)	0x200003FC
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x08000438

执行第 62 行代码后:

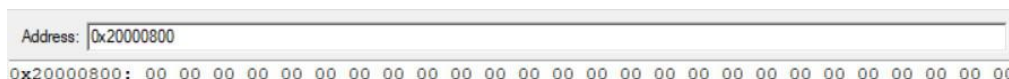
R13 (SP)	0x20000400
R14 (LR)	0xFFFFFFFF
R15 (PC)	0x0800043A

可见，由于压栈操作，因此栈指针发生了变化，而之后的出栈栈指针变化也对应了这点。

(6) 在如下代码执行前后观察存储器 0x20000800 地址的内容：

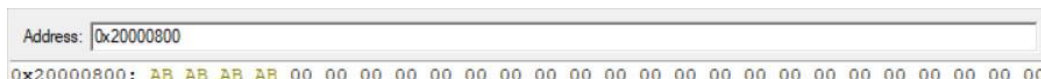
```
64      LDR R0, =0x20000800      ;把0x12345678装载到R0
65      MOV R1, #0xABABABAB      ;立即数需要满足特定的规则
66      STR R1, [R0]
67      LDR R2, [R0]
```

执行前：



Address: 0x20000800
0x20000800: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

执行后：



Address: 0x20000800
0x20000800: AB AB AB AB 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

三、思考题

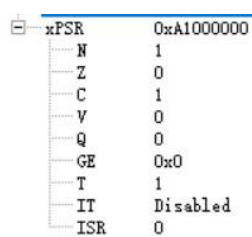
(7) 异常处理子程序 Reset_Handler 的入口地址是？

答：入口地址为 0x0800 0400。

(8) 添加一行代码，使 xPSR 寄存器的 Z 标志位为 1。

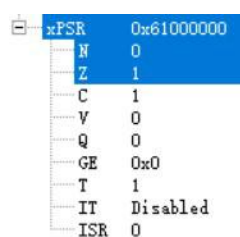
答：添加如下指令：MOVS R5, #0x0

执行前：



xPSR	0xA1000000
N	1
Z	0
C	1
V	0
Q	0
GE	0x0
T	1
IT	Disabled
ISR	0

执行后：



xPSR	0x61000000
N	0
Z	1
C	1
V	0
Q	0
GE	0x0
T	1
IT	Disabled
ISR	0

(9) 示例代码中，为何使用的是 MSP 而不是 PSP？

答：因为默认栈指针就是 MSP，CONTROL 寄存器也没有被重新设置，值一直都是 0x00，因此不会使用 PSP。

(10) 请解释至 Reset_Handler 中的第一行代码时，为何 MSP 的值为“0x20000400”？

答：由于栈大小设置为了 0x00000400，而在将 MSP 取出时初始化为 0x20000000，因此在执行到第一行代码时，MSP 指向的是 0x20000400。

(11) 请依据代码调试中观察到的机器指令解释伪指令“LDR R0, =0x20000800”被翻译为机器指令的执行过程。

答：机器指令为“LDR r0, [pc, #12]”，考虑到立即数是存在指令中的，因此是将 pc 的值加上 12 以此找到立即数的地址，然后再依照地址取出立即数，将其赋给 r0 寄存器。

(12) 伪指令“LDR R0, =0x20000800”中的“0x20000800”被存放在哪个地址？

答：由上得，存放在指令地址之中，具体偏移量是 12。

(13) 解释指示符（伪指令）“EXPORT”和“DCD”的作用？

答：EXPORT 用于在程序中声明一个全局的标号，该标号可在其他文件中引用；而 DCD 用于分配一段连续的字存储单元，并且可以用指定的数据初始化。

(14) 观察代码执行过程中，PC 变化的规律。并写出一条使 PC 递增 2 的指令，再写出一条使 PC 递增 4 的指令。

答：在代码执行过程中，PC 的增值是 2 或者 4，在涉及到对立即数的操作时，由于立即数是存储在指令的地址中的，因此指令会长一些，PC 增值基本为 4，而只涉及到寄存器时，PC 增值基本是 2；

一条增值为 2 的代码：MOV R0, R1；

一条增值为 4 的代码：MOV R1, #0xAB。

实验二 基于 C 的 Project

一、实验目的

- 1、掌握µVision IDE 下创建 C 语言工程的基本步骤。
- 2、了解µVision IDE 自带 CMSIS 库和 device 的启动文件。
- 3、掌握联机帮助查询技巧。
- 4、掌握代码分析技巧。
- 5、理解 ARM 汇编程序中的伪指令（指示符，Directive）。
- 6、掌握 C 和汇编混合编程方法。

二、实验内容

1、建立基于 C 程序的 Project

(15) 选择 AEMCM3 建立新的 Project

2、代码功能验证

(16) 阅读启动文件“startup_ARMCM3.s”和“startup_ARMCM3.c”，分析启动的大致过程。

答：startup_ARMCM3.s，首先是建立堆栈、对堆栈进行初始化，然后开始设置中断向量表，通过 DCD 伪指令对中断向量表的具体内容做了初始化定义，最后是子程序 Reset_Handler 的设置，这里调用了两个外部函数_main 和 SystemInit；startup_ARMCM3.c 文件则定义了一些关于时钟的变量和函数。

(17) 分析示例代码中 C 调用汇编子程序的过程。

答：C 调用的是汇编中定义的 strcpy(char *a, char *b) 函数：

```
8          EXPORT strcpy
9  strcpy PROC;必须与EXPORT后面标号一致
10
11 LOOP    LDRB    R2, [R1],#1 ;R1指向源字符串地址，取出字符内容存入R2
12          ;更新R1=R1+1，第一次调用时R1指向源字符串首地址
13          STRB    R2, [R0],#1 ;R0指向目的字符串地址，R2中内容存入R0指向内存单元，
14          ;更新R0=R0+1，第一次调用时R0指向目的字符串首地址
15          CMP    R2, #0
16          BNE    LOOP ;先执行后判断，源字符串的终止符'\0'也复制到目的字符串
17
18          MOV    PC, LR
19          ENDP
```

C 通过 extern void strcpy (char * d , char * s) 声明来调用汇编子程序 strcpy，C 中的传递参数 a 和 b 分别存储在寄存器 R0 和 R1 中，并在汇编程序中通过 LDR 和 STRB 指令，达成了对字符串复制的目的。

(18) 分析示例代码中汇编调用 C 子程序的过程。

答：汇编调用的是 C 中定义的 MY_C_FUNCTION(int a, int b) 函数：

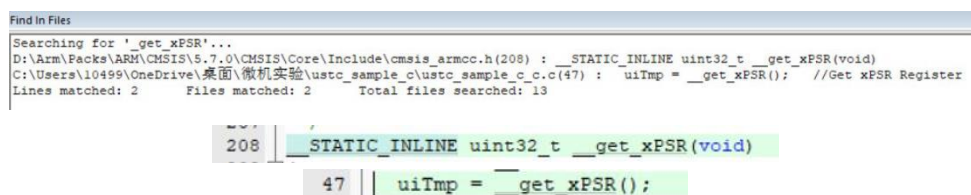
```
24          PRESERVE8 ;根据ATPCS,堆栈数据需要设置为8字节对齐
25          THUMB
26          EXPORT call_C
27          IMPORT MY_C_FUNCTION ;声明MY_C_FUNCTION为外部引用符号
28          ALIGN
29  call_C PROC;必须与EXPORT后面标号一致
30          MOV    R0, #1 ;参数1赋R0
31          MOV    R1, #2 ;参数2赋R1
32          BL     MY_C_FUNCTION
33          ENDP
```

汇编利用 call_C 调用了外部函数 MY_C_FUNCTION 函数，并通过指令以立即数的形式将

参数存储在 R0 和 R1 寄存器中，以此来完成参数的传递。

3、代码分析技巧

(19) 通过查找功能查找 “__get_xPSR” 关键字。



(20) 分析示例代码中所调用的 CMSIS-core 函数的定义。

答：调用的 CMSIS-core 函数分为三个：__REV16()、__get_xPSR()和__get_MSP()。

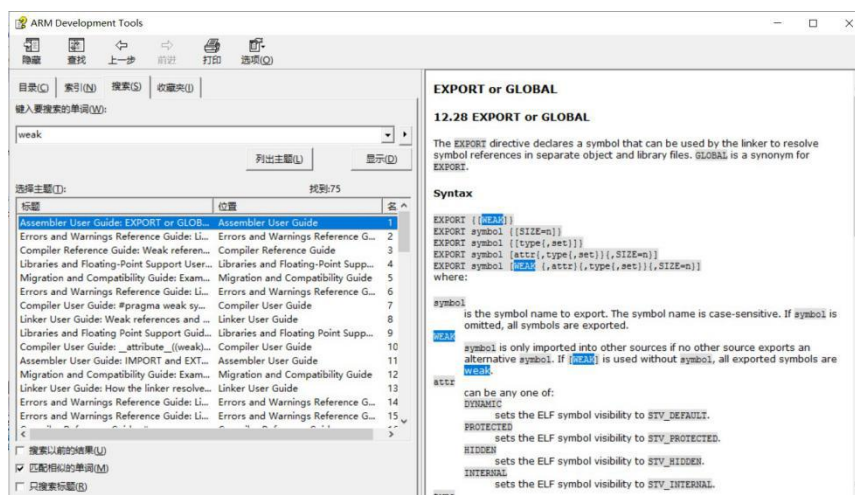
__REV16()函数是将一个 64bit 的数，先进行一次 32 位翻转，再进行一次 16 位翻转；

__get_xPSR()函数则是将 xpsr 寄存器的数值以 32 位的形式返回；

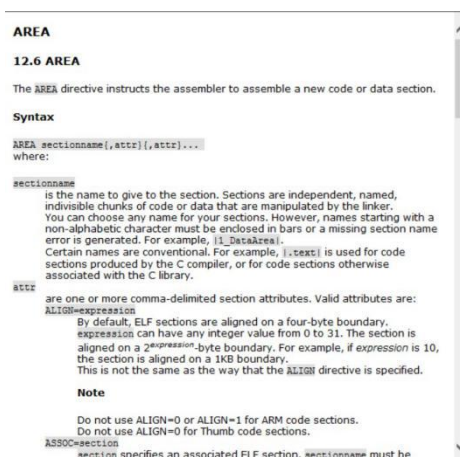
__get_MSP()函数是将 msp 寄存器的数值以 32 位的形式返回。

4、μVision 联机资源的使用

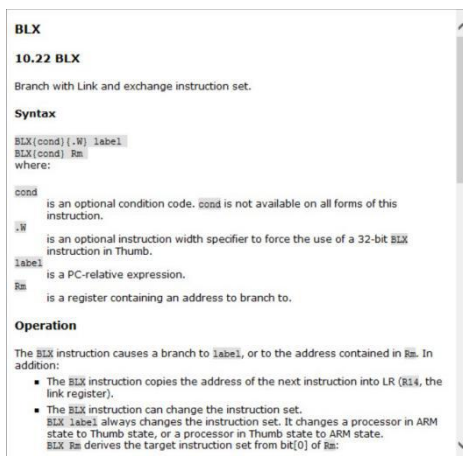
(21) 打开联机帮助文档，搜索指示符 “WEAK” 的作用。



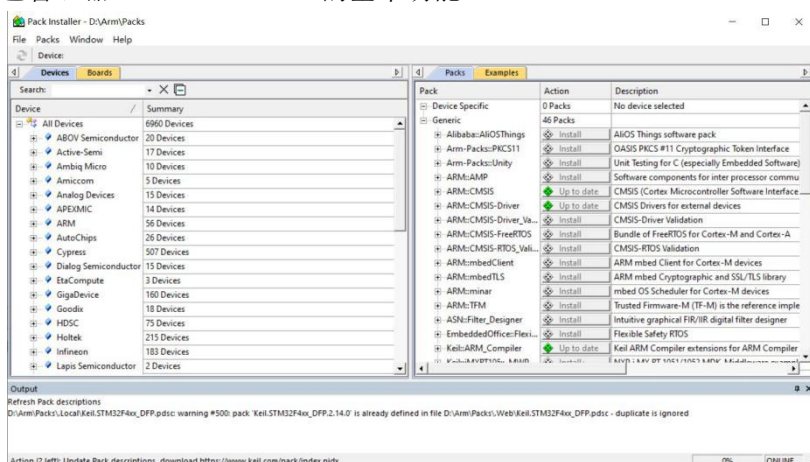
(22) 将光标停留在某个指示符（伪指令关键字）上，如 “AREA”，按键 “F1”，查阅自动弹出的帮助信息。



(23) 将光标停留在某条指令上，如“BLX”，按键“F1”，查阅自动弹出的帮助信息。



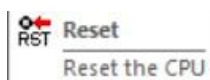
(24) 了解包管理器 Pack installer 的基本功能。



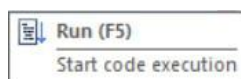
5、Project 的调试 (Debug)

(25) 分析 Debug 工具栏各个图标对应功能的区别。

Reset 复位：让程序复位到起点，调试设置恢复到初始状态。



Run 全速运行：可以让程序运行并查看运行状态，也可以在特定位置打断点，让程序运行到特定位置并查看运行状态。



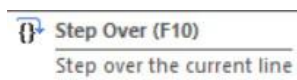
stop 停止运行：程序全速运行时（有效），点击该按钮可让程序停止运行。



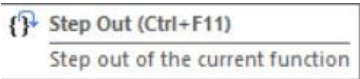
Step 单步调试：也就是每点一次按钮，程序运行一步，遇到函数会跳进函数执行。



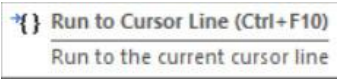
Step Over 逐行调试：也就是每点一次按钮，程序运行一行，遇到函数跳过函数执行。



Step Out 跳出调试：也就是每点一次按钮，程序跳出当前函数执行，直到跳出最外面的函数（main 函数）



Run to Cursor Line 运行到光标处：即将光标放在某一行，点击该按钮，程序执行到光标的位置就会停下来（前提是程序能执行到光标的位置）



（26）通过 watch 窗口观察以下代码执行前后，变量“srcstr”“dststr”的值。

```
26 char * srcstr = "0123456" ;
27 char dststr [ 8 ] = "abcdefg" ;
28 //根据ATPCS规则，子程序间通过寄存器R0~R3来传递参数
29 //dststr地址存放在R0，srcstr地址存放在R1
30 strcpy(dststr,srcstr);
```

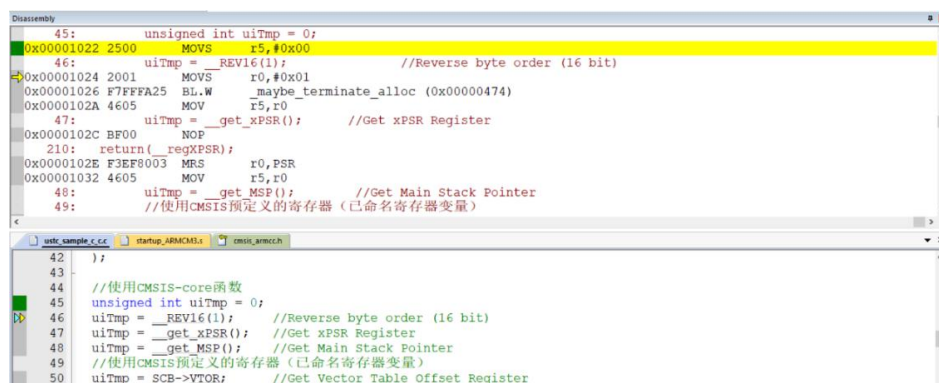
答：通过 watch 窗口，分别观察该段代码运行前、strcpy 函数调用前、strcpy 函数调用后三种情况下的变量的值：

Name	Location/Value	Type
main	0x00001002	int f()
srcstr	0x00000000	auto - uchar *
srcstr[0]	0x78 'x'	uchar
dststr	0x20001160 ""	auto - uchar[8]
dststr[0]	0x00	uchar
dststr[1]	0x00	uchar
dststr[2]	0x00	uchar
dststr[3]	0x00	uchar
dststr[4]	0x00	uchar
dststr[5]	0x00	uchar
dststr[6]	0x00	uchar
dststr[7]	0x00	uchar
uiTmp	0x00000000	auto - uint

Name	Location/Value	Type
main	0x00001002	int f()
srcstr	0x00001058 "0123456"	auto - uchar *
srcstr[0]	0x30 '0'	uchar
dststr	0x20001160 "abcdefg"	auto - uchar[8]
dststr[0]	0x61 'a'	uchar
dststr[1]	0x62 'b'	uchar
dststr[2]	0x63 'c'	uchar
dststr[3]	0x64 'd'	uchar
dststr[4]	0x65 'e'	uchar
dststr[5]	0x66 'f'	uchar
dststr[6]	0x67 'g'	uchar
dststr[7]	0x00	uchar
uiTmp	0x00000000	auto - uint

Name	Location/Value	Type
main	0x00001002	int f()
srcstr	0x00001058 "0123456"	auto - uchar *
srcstr[0]	0x30 '0'	uchar
dststr	0x20001160 "0123456"	auto - uchar[8]
dststr[0]	0x30 '0'	uchar
dststr[1]	0x31 '1'	uchar
dststr[2]	0x32 '2'	uchar
dststr[3]	0x33 '3'	uchar
dststr[4]	0x34 '4'	uchar
dststr[5]	0x35 '5'	uchar
dststr[6]	0x36 '6'	uchar
dststr[7]	0x00	uchar
uiTmp	0x00000000	auto - uint

(27) 分析下图所示反汇编窗口中机器指令与源代码窗口中 C 代码的对应关系。



答：“MOVNS r5, #0x00”对应变量的声明 unsigned int uiTmp=0, 此时 r5 寄存器存储 uiTmp 的值；

“MOVNS r0, #0x00”将 0 赋给 r0 寄存器，为接下来的函数调用做准备，“BL.W _maybe_terminate_alloc (0x00000474)”跳转地址，用于调用函数，最后的“MOV r5, r0”将函数结果赋给 r5，也就是 uiTmp 变量；

下面的三个机器指令就很好理解了，即，将 PSR 的值通过特殊指令 MRS 取出到 r0 寄存器中，再将 r0 中的值赋给 r5，也就是变量 uiTmp。

三、思考题

(28) 分析启动文件“startup_ARMCM3.s”中图下宏定义的含义？并写出\$Handler_Name 等于 NMI_Handler 时，该宏定义展开后的代码。

```
119 MACRO
120 Set_Default_Handler $Handler_Name
121 $Handler_Name PROC
122 EXPORT $Handler_Name [WEAK]
123 B .
124 ENDP
125 MEND
```

答：该宏定义的含义是，宏定义一个默认的异常/中断处理器，该程序是一个具有无限循环的弱符号；

\$Handler_Name 等于 NMI_Handler 时，宏定义展开后的代码：

```
NMI_Handler PROC
EXPORT NMI_Handler
B .
ENDP
```

查阅代码，此时 NMI_Handler 的值为-14，因此该段程序调用异常处理子函数，跳转到 -14 对应的中断处继续执行。

(29) 根据调试结果，示例代码中的“printf(“Hello USTCer\n”);”中字符串“Hello USTCer\n”被存储在存储器的什么位置？

答：运用 PC 相对寻址的方式存储在 (PC) + 0x20 地址中，即 0x00001020。

(30) 请结合“startup_ARMCM3.s”文件中如下代码分析示例中“srcstr”的地址为何是“0x20001160”？

```
ustc_sample_c.c  startup_ARMCM3.s  cmsis_armcc.h
44
45 Heap_Size      EQU      0x00000C00
46
47               IF      Heap_Size != 0
48               AREA     HEAP, NOINIT, READWRITE, ALIGN=3
49 __heap_base
50 Heap_Mem       SPACE    Heap_Size
51 __heap_limit
52               ENDIF
```

答：定义了 Stack_Size 为 0x0000 0400，Heap_Size 为 0x0000 0C00，启动方式为片内 SRAM 启动，起始地址从 0x00000000 映射到 0x20000000，所以此时的地址为 0x20001000。main 函数再加上之后的 printf 函数，一共占用 0x00000160，求和可得为 0x20001160，故“srcstr”的地址为 0x20001160。

实验三 基于 STM32 库的 GPIO 与定时器

一、实验目的

- 1、掌握µVision IDE 中基于 ST 公司 STM32 库建立 project 的流程。
- 2、了解 ST 公司提供的 TIM、GPIO 相关库函数。
- 3、了解 STM32F10X 系列芯片定时器相关的寄存器功能。
- 4、掌握µVision IDE 中外设仿真模块（GPIO）的使用，学会利用外设仿真模块（GPIO）观察 I/O 引脚输出，学会利用外设仿真模块（GPIO）模拟 I/O 引脚的输入。
- 5、掌握µVision IDE 逻辑分析模块（Logic Analyzer）的使用。

二、实验内容

1、下载 ST 公司 STM32 库及芯片手册

（31）下载 ST 公司关于 STM32F103 系列芯片的库文件。

（32）下载 ST 公司关于 STM32F103 系列芯片的文档。

2、建立基于 STM32 库的 Project

（33）建立 Project。

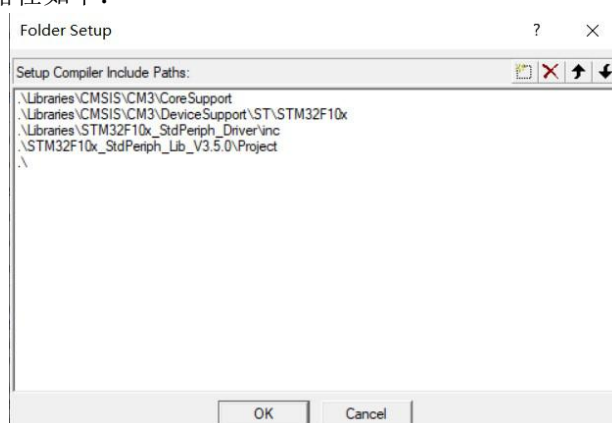
（34）拷贝 ST 公司的库文件。

（35）添加 ST 公司的库文件。

3、配置 Project 的头文件目录、预编译参数、Simulator

（36）配置“include file”路径执行 ST 库头文件目录

配置头文件路径如下：



（37）配置预编译选项 Preprocessor Symbols。

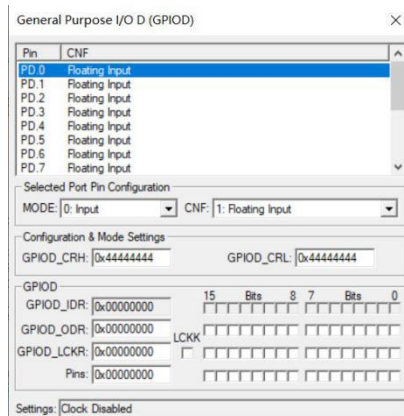
（38）设置采用 simulator 方式进行调试，配置为 STM32F103ZE 芯片的 simulator。

（39）Build Project

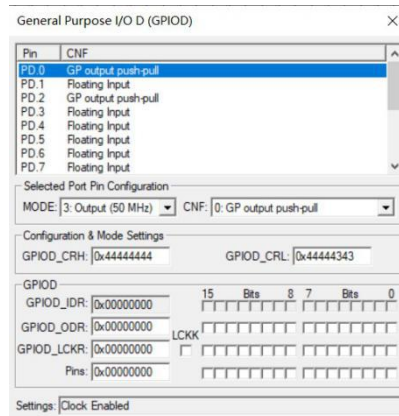
4、Debug 时使用外设仿真功能验证 GPIO 输入和输出

(40) 进入调试状态，打开 GPIO 窗口，观察执行前后 GPIOD 窗口的变化。

代码运行前：

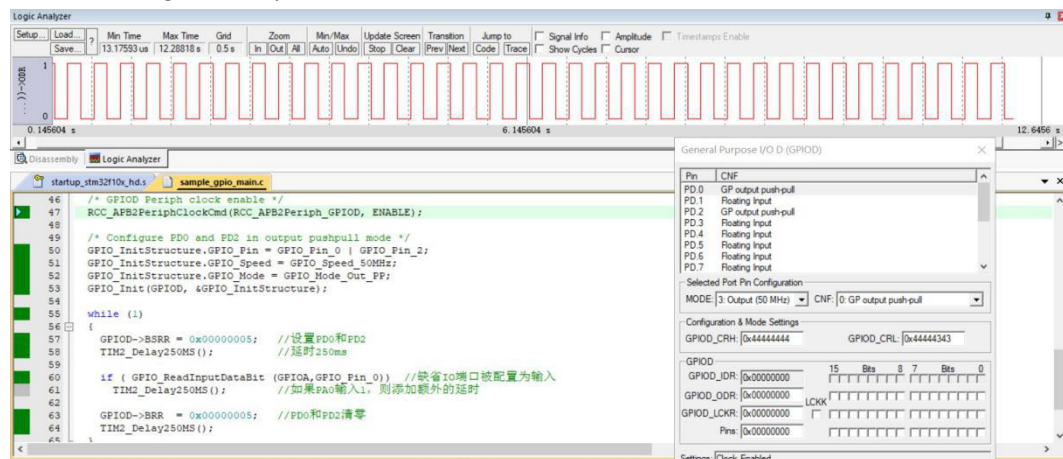


代码运行后：



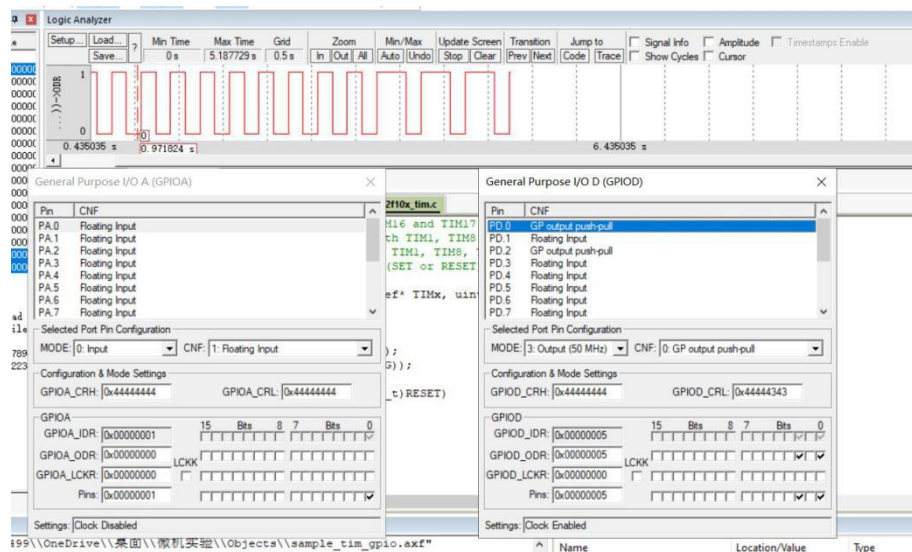
5、Debug 时使用 Logic Analyzer 观察 GPIO 输出

(41) 通过 Logic Analyzer 观察 GPIOD 的变化



6、使用外设仿真和 Logic Analyzer 验证 GPIO 输入和输出

(42) 在 GPIOA 窗口中单击 PA0 引脚，模拟 PA0 输入“1”，观察 Logic Analyzer 的变化，并与示例代码进行验证。



对应的代码:

```
55 while (1)
56 {
57     GPIOD->BSRR = 0x00000005; //设置PD0和PD2
58     TIM2_Delay250MS(); //延时250ms
59
60     if ( GPIO_ReadInputDataBit (GPIOA,GPIO_Pin_0)) //缺省IO端口被配置为输入
61         TIM2_Delay250MS(); //如果PA0输入1,则添加额外的延时
62
63     GPIOD->BRR = 0x00000005; //PD0和PD2清零
64     TIM2_Delay250MS();
65 }
66 }
```

由于在运行过程中将 GPIOA 的 PA0 端口输入改为 1，因此这里触发了条件“GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0)”，故而额外的时延函数 TIM_Delay250MS() 开始工作，因此在 Logic Analyzer 分析器上会看到一个更宽的高电平信号，而取消了这个输入以后，高电平信号又恢复到之前的宽度了。

7、定时器的配置

(43) 阅读 TIM_TimeBaseInitTypeDef 的定义，了解该结构体各成员的含义。

答：该结构体的具体定义如下：

```
typedef struct
{
    uint16_t TIM_Prescaler;
    uint16_t TIM_CounterMode;
    uint16_t TIM_Period;
    uint16_t TIM_ClockDivision;
    uint8_t TIM_RepetitionCounter;
} TIM_TimeBaseInitTypeDef;
```

阅读代码后，各项的具体含义：

TIM_Prescaler，用于划分 TIM 预分频寄存器 Prescaler 的值；

TIM_CounterMode，用于设置计数器模式；

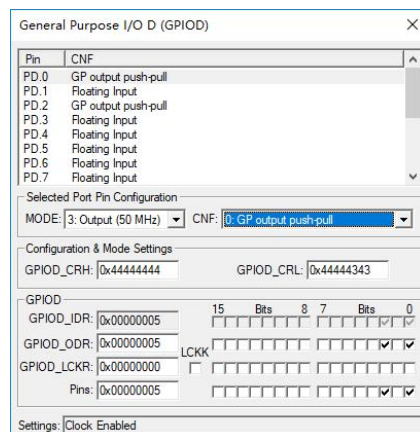
TIM_Period，在下一个更新事件装入 ARR 值；

TIM_ClockDivision，CK_INT 与死区发生器以及数字滤波器采样频率分频系数；

TIM_RepetitionCounter，重复定时器值。

三、思考题

(44) 阅读讲义 8.5.3 小节，及芯片手册，解释下图中“CRL、CRH、IDR、ODR、LCKR”几个寄存器的作用。



答：CRL 和 CRH 是两个配置寄存器，前者为低寄存器，后者为高寄存器，根据 I/O 端口号来决定具体选取哪个寄存器，每四位控制一个引脚，可对引脚的 I/O 模式进行配置；

IDR 是端口输入数据寄存器，仅低 16 位有效，用于读取端口上一次的输入数据，但在此之前要将端口设置为输入模式；

ODR 是端口输出数据寄存器，与 IDR 类似，也是低 16 位有效，写的时候，其值影响对应 I/O 口（即端口的引脚）的输出（0 或 1，即输出低/高电平），读的时候，ODR 反映了作为输出时，上一次写出的数据；

LCKR 是端口配置锁定寄存器，当执行正确的写序列设置了位 16(LCKK)时，该寄存器低 16 位[15:0]（LCKP[15:0]）用来锁定端口位的配置。当对相应端口位执行了 LOCK 序列后，在下次系统复位之前将不能再更改端口位的配置。每个锁定位锁定控制寄存器(CRL、CRH)中相应的 4 个位（即一个引脚的配置）。

（45）解释代码“GPIO->BSRR=0x00000085;”的作用。

答：该代码表示将引脚 0, 2, 6 设置为 1。

（46）解释代码“GPIO->BRR=0x00000080;”的作用。

答：该代码表示将引脚 6 复位为 0。

（47）解释库函数“GPIO_ReadInputDataBit（GPIO, GPIO_Pin_0）”的作用。

答：该函数的具体代码如下：

```
uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
    uint8_t bitstatus = 0x00;

    assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
    assert_param(IS_GET_GPIO_PIN(GPIO_Pin));

    if ((GPIOx->IDR & GPIO_Pin) != (uint32_t)Bit_RESET)
    {
        bitstatus = (uint8_t)Bit_SET;
    }
    else
    {
        bitstatus = (uint8_t)Bit_RESET;
    }
    return bitstatus;
}
```

assert_param 用于判断参数有没有错误（在这里就是那两个函数），随后将 GPIOx->IDR 和 GPIO_Pin 进行按位与操作，在此结果不全为零的情况下将时钟状态设置为置位，否则设置为复位状态。

（48）访问“<http://www.keil.com/support/docs/3726.htm>”，查询µVision IDE 是否支持 STM32F407ZG 芯片的 Simulation。

答：经查阅，µVision IDE 不支持 STM32F407ZG 芯片的 Simulation。

实验四 基于 STM32 库的中断

一、实验目的

- 1、掌握 EXTI 中断配置流程。
- 2、理解异常向量表。
- 3、理解异常优先级配置。
- 4、了解 ST 公司提供的 TIM 和 NVIC 相关库函数。
- 5、掌握µVision IDE 中外设仿真模块（NVIC）的使用。

二、实验内容

1、建立基于 STM32 库的 Project

(50) 建立 Project

(51) 拷贝 ST 公司的库文件

(52) 添加 ST 公司的库文件

2、配置 Project 的头文件目录、预编译参数、Simulator

(53) 配置 “include file” 路径执行 ST 库头文件目录

(54) 配置预编译选项 “Preprocessor Symbols” 为：USE_STDPERIPH_DRIVER

(55) 设置采用 simulator 方式进行调试，配置为 STM32F103ZE 芯片的 simulator

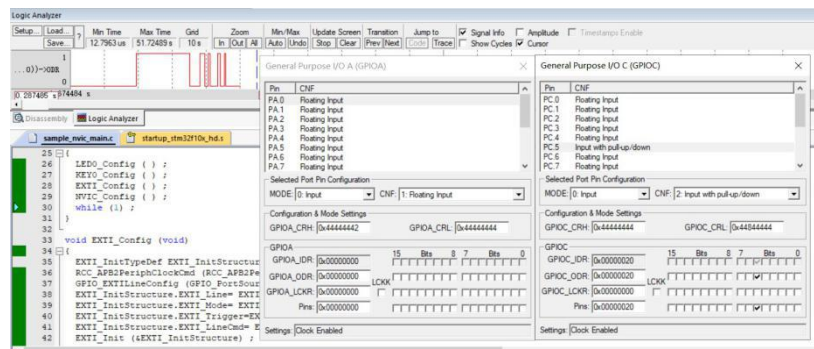
3、使用 Logic Analyzer 和外设仿真功能验证 EXIT 及 GPIO 输出

(56) 打开µVision IDE 自带的逻辑分析模块（Logic Analyzer），单击 “Setup” 设置 GPIOA->ODR 为需要观察的信号。并设置 GPIOA->ODR 信号的显示类型（Display Type）为 “Bit”

(57) 打开 GPIOA 的窗口

(58) 打开 GPIOC 的窗口

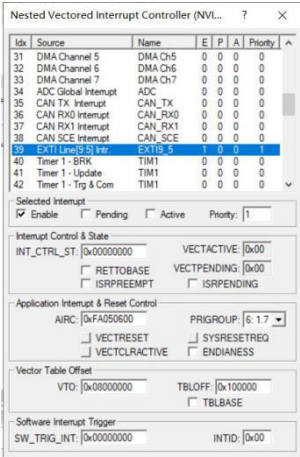
(59) 运行代码，并在 GPIOC 窗口输入 PC5 来模拟中断信号，可在 Logic Analyzer 和 GPIOA 窗口同时观察到 PA8 的变化。



经对 Logic Analyzer、GPIOA、GPIOC 窗口的观察分析, 可知, 在经历过一次模拟中断信号 (PC5) 的置位+复位过程之后, PA8 输出信号电平将会发生改变。

4、观察 NVIC 寄存器组

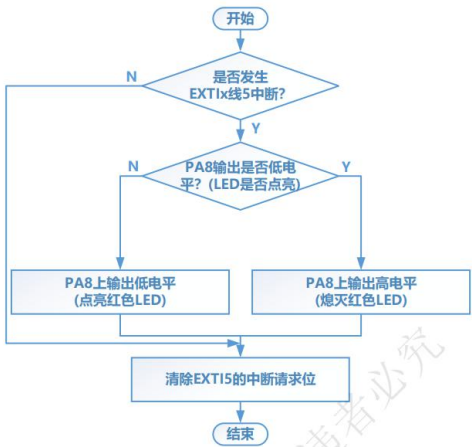
(60) 观察与外部中断#5 有关的寄存器信息。



三、思考题

(61) 为什么通过 GPIOC 的窗口模拟 PC5 (中断信号) 输入的时候, 改变两次 PC5 才会使 PA8 发生变化?

答: 讲义上给出的程序流程:



可知 PC5 改变触发中断并引起 PA8 改变后, 会清除其中断请求位, 因此下一次撤销中断请求位的操作不产生任何动作。所以 PC5 改变两次, PA8 改变一次。

(62) 请通过调试获得 EXTI9_5_IRQHandler() 的入口地址 (应该是 0x800026A), 这个地址保存在异常向量表什么位置?

答: 由下图可知入口地址为 0x800026A。

查阅代码可知, 该地址保存在第 40 个向量中, 位置应为 0x000000A0。

```

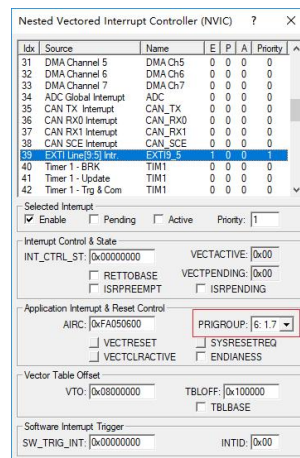
Disassembly
0x08000266 0002 DCW 0x0002
0x08000268 4770 BX lr
24:
0x0800026A B510 PUSH {r4,lr}
25: unsigned char temp=LEDO_IsOn ();
0x0800026C F00F96C BL.W LED0_IsOn (0x08000549)
0x08000270 4604 MOV r4,r0
26: if (EXTI_GetITStatus (EXTI_Line5) != RESET)
..

sample_nvic_main.c startup_stm32f10x_hds system_stm32f10x.c stm32f10x_rcc.c misc.c sample_nvic_exception_handler.c stm32f10x_gpio.c stm32f10x_exti.c

19 void LED0_On (void) ;
20 void LED0_Off (void) ;
21 unsigned char LED0_IsOn (void) ;
22
23 void EXTI9_5_IRQHandler (void)
24 {
25     unsigned char temp=LED0_IsOn ();
26     if (EXTI_GetITStatus (EXTI_Line5) != RESET)
27     {
28         if (temp)
29             LED0_Off ();
30         else
31             LED0_On ();
32         EXTI_ClearITPendingBit (EXTI_Line5) ;
33     }
34
35     void LED0_On (void)
36

```

(63)解释下图中PRIGROUP的含义(请查阅讲义5.5.1小节),PRIGROUP和AIRC(Application Interrupt & Reset Control 寄存器)是什么关系?



答: PRIGROUP 是 AIRC 寄存器的一个域, 包括 AIRC[10:8] 这三位, 用这三位来控制优先级分组, 一共有八个优先级分组。