

微机原理与嵌入式系统 实验报告

信息科学技术学院

姓名：胡睿 PB17061124

实 验 报 告

评分：

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO. _____

【实验题目】实验 1 基于 ASM 的 Project

【实验目的】

- 1、掌握 μ Vision 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

【实验内容】

1.2.1、Project 的建立、编译、连接

1.2.2、Project 的调试 (Debug)

运行前寄存器数值：

运行后寄存器数值：

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
xPSR	0x01000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	0
Sec	0.00000000
FPU	

Register	Value
Core	
R0	0x000000AB
R1	0x000000AB
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)	0x08000408
xPSR	0x21000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	108001587
Sec	9.00013225
FPU	

(4) 观察调试过程中通用寄存器的变化

实验报告

评分:

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO. _____

(5) 观察如下代码执行前后栈指针的变化

56	; 观察SP	
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

运行前:

Registers	
Register	Value
Core	
R0	0x00000034
R1	0x00000034
R2	0x00000000
R3	0x000000EF
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)	0x0800042A
xPSR	0xA1000000

Memory 1	
Address: 0x20000000	
0x2000037B:	00 00
0x20000396:	00 00
0x200003B1:	00 00
0x200003CC:	00 00
0x200003E7:	00 00
0x20000402:	00 00
0x2000041D:	00 00
0x20000438:	00 00
0x20000453:	00 00

运行后:

实 验 报 告

评分:

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO.

Registers	
Register	Value
Core	
R0	0x00000034
R1	0x00000001
R2	0x00000003
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)	0x0800043A
+ xPSR	0xA1000000

Memory 1	
Address: 0x20000000	
0x2000037B:	00 00
0x20000396:	00 00
0x200003B1:	00 00
0x200003CC:	00 00
0x200003E7:	00 03 00 00 00 01 00 00 00 00 00 00 00
0x20000402:	00 00
0x2000041D:	00 00
0x20000438:	00 00
0x20000453:	00 00

可以看出栈的指针是递减的，同时指向栈顶的元素。

(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:

Memory 1	
Address: 0x20000800	
0x20000800:	00 00
0x2000081B:	00 00
0x20000836:	00 00
0x20000851:	00 00
0x2000086C:	00 00
0x20000887:	00 00
0x200008A2:	00 00
0x200008BD:	00 00
0x200008D8:	00 00

执行后:

实 验 报 告

评分：

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO.

Memory 1																																		
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	00	00	00
0x2000081B:	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	00	00
0x20000836:	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	00	00
0x20000851:	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	00	00
0x2000086C:	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	00	00
0x20000887:	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	00	00
0x200008A2:	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	00	00
0x200008BD:	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	00	00

【1.3 思考题】

(7) 异常处理子程序 Reset_Handler 的入口地址是？

答：0x08000400

(8) 添加一行代码，使 xPSR 寄存器的 Z 标志位为 1。

答：MOV R1,#0x34

CMP R1,#0x34

(9) 示例代码中，为何使用的是 MSP 而不是 PSP。

答：因为是工作在 handler 模式下，只允许使用主堆栈指针 MSP。

(10) 请解释执行至 Reset_Handler 中第一行代码时，为何 MSP 为“0x20000400”？

答：因为在程序开头，定义了栈的大小为 0x00000400，然后开始执行时，会指向栈的顶部，即为 0x20000400，同时为递减堆栈。

(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：首先执行机器码 4803，LDR，然后将 pc 寄存器向下偏移 12 个字节的数据存入 R0 中。

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

实 验 报 告

评分：

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO. _____

答：0x08000448。

(13) 解释指示符（伪指令）“EXPORT”和“DCD”的作用？

答：export 伪指令用于程序中声明一个全局的标号，该标号可在其他的文件中引用。DCD 伪指令用于分配一片连续的字存储单元并用伪指令中指定的表达式初始化。

(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：递增 2 的指令：MOV R0, R1

递增 4 的指令：MOV R1, #0xAB。

实 验 报 告

评分：

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO. _____

【实验题目】实验 2 基于 C 的 Project

【实验目的】

- 1、掌握 μ Vision IDE 下创建 C 语言工程的基本步骤
- 2、了解 μ Vision 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

【实验内容】

2.2.1、建立基于 C 程序的 Project

2.2.2、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：首先设置了栈和堆的大小并进行设置，然后设置了向量表，然后调用 reset handler，引导程序进入 __main，最后进行用户堆栈的初始化，完成启动的过程。

(17) 分析示例代码中 C 调用汇编子程序的过程。

答：示例中，c 程序调用的是 strcpy 子程序，strcpy 首先将 R1 指向源字符串地址，取出字符内容存入 R2，然后 R0 指向目的字符串地址，R2 中内容存入 R0 指向内存单元，然后比较 R2 里面是不是全空，就可以实现复制字符串的功能。

(18) 分析示例代码中汇编调用 C 子程序的过程。

答：示例中，汇编调用的是 MY_C_FUNCTION 子程序，通过传入两个 int 数，

实 验 报 告

评分:

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO.

然后返回两个数的和。

2.2.3、代码分析技巧

(19) 在当前 Project 内所有文件中查找 “__get_xPSR” 关键字。

逐个文件查看关键字 “__get_xPSR” 出现的位置。

The screenshot shows an IDE with three tabs: `ustc_sample_c.c`, `ustc_sample_c.asm.s`, and `cmsis_armcc.h`. The `ustc_sample_c.c` tab is active, showing lines 43 to 52. Line 47 contains `__get_xPSR();`, which is highlighted with a red box. Below this, the `ustc_sample_c.asm.s` tab is active, showing lines 101 to 111. Line 105 contains `__STATIC_INLINE uint32_t __get_xPSR(void)`, which is also highlighted with a red box. At the bottom, a 'Find in Files' window is open, showing the search results for the keyword '`__get_xPSR`'. The search path is `F:\github_repository\Microcomputer-Principle-and-System_experiment\实验2 基于C的Project\ustc_sample_c.c(47)`. The search results show that the keyword was found in 2 files, with 2 lines matched and 2 files matched. The total files searched were 12.

(20) 分析示例代码中所调用的 CMSIS-core 函数的定义。如下图所示在函数位置点击鼠标右键（或热键 F12），跳转至 `__get_xPSR()` 的定义文件。浏览所打开的 “`cmsis_armcc.h`” 文件。

答：访问一些汇编指令和特殊寄存器。

The screenshot shows the definition of the `__get_xPSR` function in the `cmsis_armcc.h` file. The function is defined as a static inline function that returns the value of the xPSR register. The code is as follows:

```
105 __STATIC_INLINE uint32_t __get_xPSR(void)
106 {
107     register uint32_t __regXPSR      __ASM("xpsr");
108     return(__regXPSR);
109 }
```

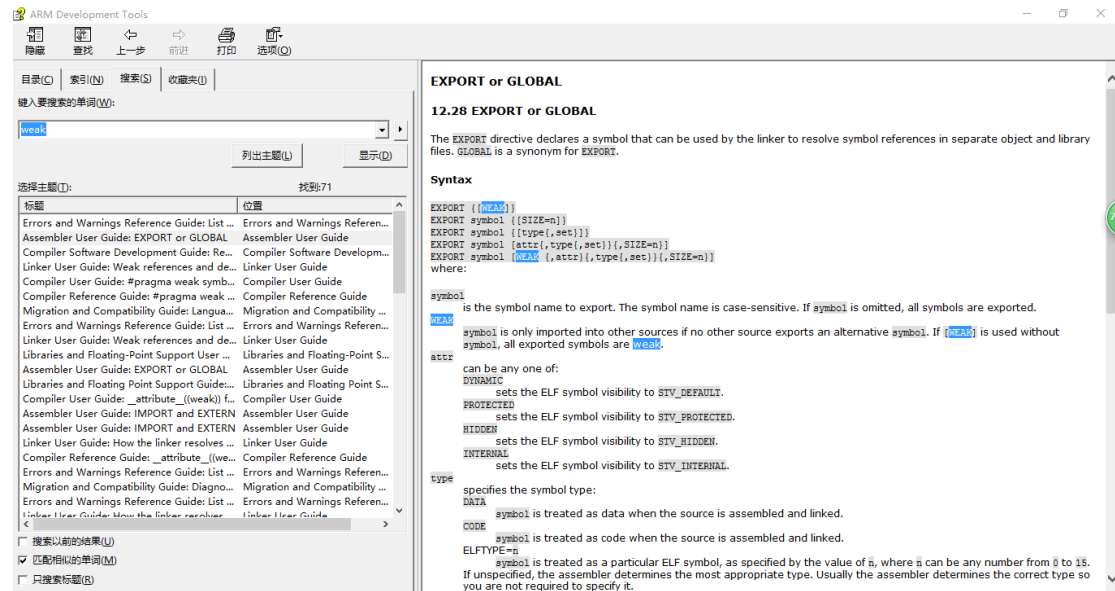

实 验 报 告

评分：

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO.

2.2.4 μ Vision 联机资源使用

(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。

AREA

12.6 AREA

The **AREA** directive instructs the assembler to assemble a new code or data section.

Syntax

```
AREA sectionname[,attr][,attr]...  
where:
```

sectionname
is the name to give to the section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker.
You can choose any name for your sections. However, names starting with a non-alphabetic character must be enclosed in bars or a missing section name error is generated. For example, `!1_DataArea`.
Certain names are conventional. For example, `!text` is used for code sections produced by the C compiler, or for code sections otherwise associated with the C library.

attr
are one or more comma-delimited section attributes. Valid attributes are:

ALIGN=expression
By default, ELF sections are aligned on a four-byte boundary. **expression** can have any integer value from 0 to 31. The section is aligned on a $2^{\text{expression}}$ -byte boundary. For example, if **expression** is 10, the section is aligned on a 1KB boundary. This is not the same as the way that the **ALIGN** directive is specified.

Note

Do not use **ALIGN=0** or **ALIGN=1** for ARM code sections.
Do not use **ALIGN=0** for Thumb code sections.

ASSOC=section
section specifies an associated ELF section. **sectionname** must be included in any link that includes **section**

CODE
Contains machine instructions. **READONLY** is the default.

CODEALIGN
Causes armasm to insert **NOP** instructions when the **ALIGN** directive is used after ARM or Thumb instructions within the section, unless the **ALIGN** directive specifies a different padding. **CODEALIGN** is the default for execute-only sections.

COMDEF
Is a common section definition. This ELF section can contain code or data. It must be identical to any other section of the same name in other source files.
Identical ELF sections with the same name are overlaid in the same section of memory by the linker. If any are different, the linker generates a warning and does not overlay the sections.

实 验 报 告

评分：

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO.

(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。

BLX

Performs a branch with link.

Syntax **BLX{cond} Rm**
BLX label

CPU **ARM9E only**

Description **BLX{cond} Rm**
Copy address of next instruction to R14 and jump to address in Rm.
BLX label
Copy address of next instruction to R14, change to ARM instruction set and jump to label. The jump distance must be within $\pm 4\text{MByte}$ of the current instruction. Note that this mnemonic is generated as two 16-bit Thumb instructions.

Condition Flags not modified.

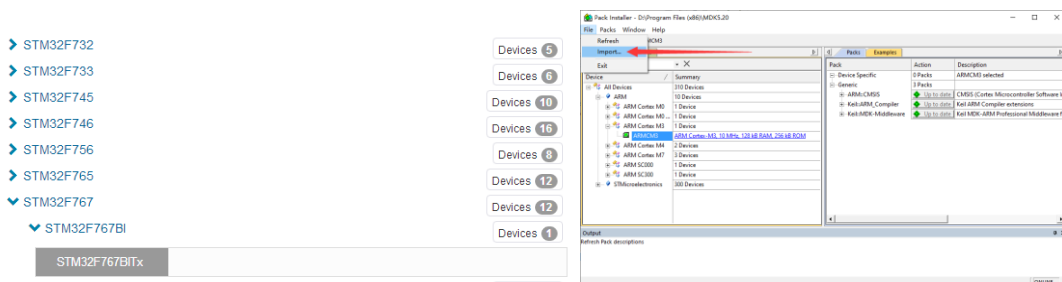
Example

```
BLX armfunc // call ARM function

LDR R6,=function
BLX R6 // call function
```

(24) 了解包管理器“Pack installer”的基本功能。

keil 中如果找不到自己要使用的芯片，可以使用 keil 的“Pack installer”找到目标芯片进行下载安装，首先我们要到 keil 的官网 <http://www.keil.com/dd2> 找到我们需要的芯片并下载对应的.pack 文件如下左图所示：



然后点击 keil 的“Pack installer”进入如上图右图界面，点击 file-import 找到下载好的.pack 文件即可安装。

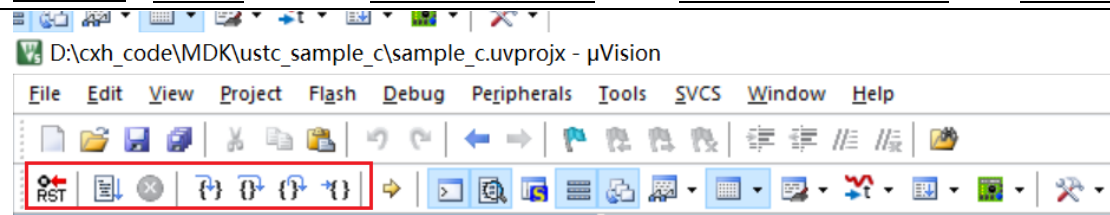
2.2.5、Project 的调试 (Debug)

(25) 通过试验分析下图所示 Debug 工具栏 (红色框内) 各个图标对应功能的区别。

实 验 报 告

评分：

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO.



答：第一个是重新设置 cpu，相当于清零大部分寄存器；

第二个是向下运行；

第三个是停止代码的执行；

第四个是进入函数的单步执行；

第五个是不进入函数的单步执行；

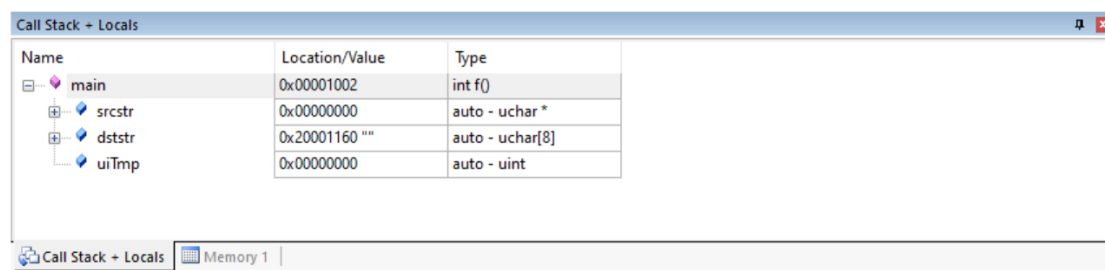
第六个是跳过当前函数执行；

第七个是执行到光标位置。

(26) 通过 Watch 窗口观察一下代码执行前后变量值。

```
26 char * srcstr = "0123456" ;
27 char dststr [ ] = "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);
```

观察变量“srcstr”和“dststr”

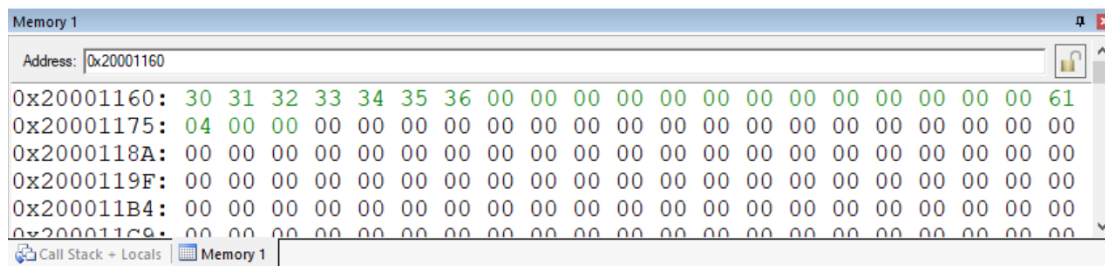


同时观察 Memory 窗口对应地址的值

实 验 报 告

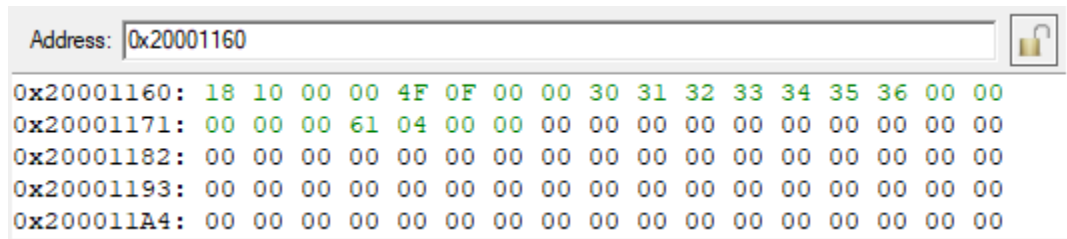
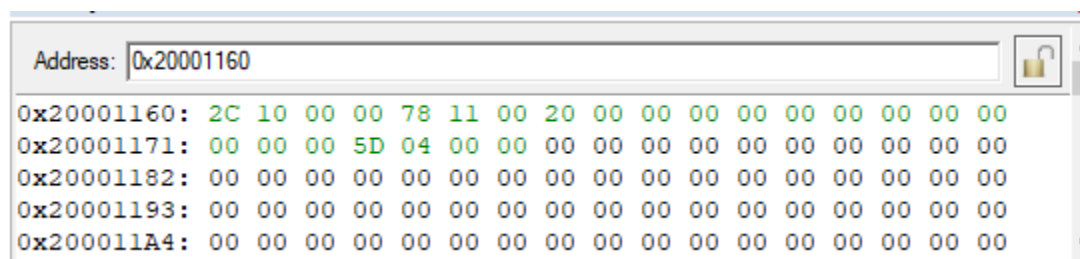
评分:

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO.



main	0x00000F46	int f()
dststr	<not in scope>	auto - uchar[8]
srcstr	<not in scope>	auto - uchar *
uiTmp	<not in scope>	auto - uint

Name	Location/Value	Type
main	0x00000F46	int f()
dststr	0x20001168 "0123456"	auto - uchar[8]
srcstr	<not in scope>	auto - uchar *
uiTmp	<not in scope>	auto - uint

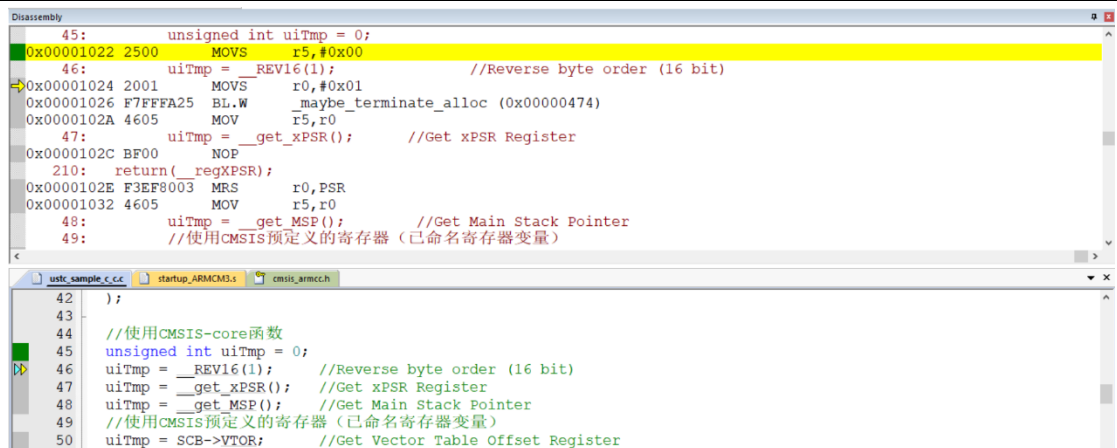


(27) 分析下图所示反汇编窗口中机器指令与源代码窗口中 C 代码的对应关系。

实 验 报 告

评分:

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO.



```
Disassembly
45: unsigned int uiTmp = 0;
0x00001022 2500 MOVS r5,#0x00
46: uiTmp = __REV16(1); //Reverse byte order (16 bit)
0x00001024 2001 MOVS r0,#0x01
0x00001026 F7FFFA25 BL.W _maybe_terminate_alloc (0x00000474)
0x0000102A 4605 MOV r5,r0
47: uiTmp = __get_xPSR(); //Get xPSR Register
0x0000102C BF00 NOP
210: return( __regXPSR);
0x0000102E F3EF8003 MRS r0,PSR
0x00001032 4605 MOV r5,r0
48: uiTmp = __get_MSP(); //Get Main Stack Pointer
49: //使用CMSIS预定义的寄存器 (已命名寄存器变量)

ustc_sample.c.c
startup_ARMCM3.s
cmsis_armcc.h
42: );
43:
44: //使用CMSIS-core函数
45: unsigned int uiTmp = 0;
46: uiTmp = __REV16(1); //Reverse byte order (16 bit)
47: uiTmp = __get_xPSR(); //Get xPSR Register
48: uiTmp = __get_MSP(); //Get Main Stack Pointer
49: //使用CMSIS预定义的寄存器 (已命名寄存器变量)
50: uiTmp = SCB->VTOR; //Get Vector Table Offset Register
```

答：第 45 行对应第一条机器指令。

第 46 行对应后面三条机器指令。

第 47 行对应后面四条机器指令。

【2.3】思考题（内容需要上机调试才能完成）

(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：表示如果有多个地方出现了 \$Handler_Name，那么就不改变原来的量的定义。

```
NMI_Handler PROC

EXPORT NMI_Handler [WEAK]

B .

ENDP
```

(29) 依据调试结果，示例 c 程序中如下行中字符串"Hello USTCcr\n"被保

实 验 报 告

评分：

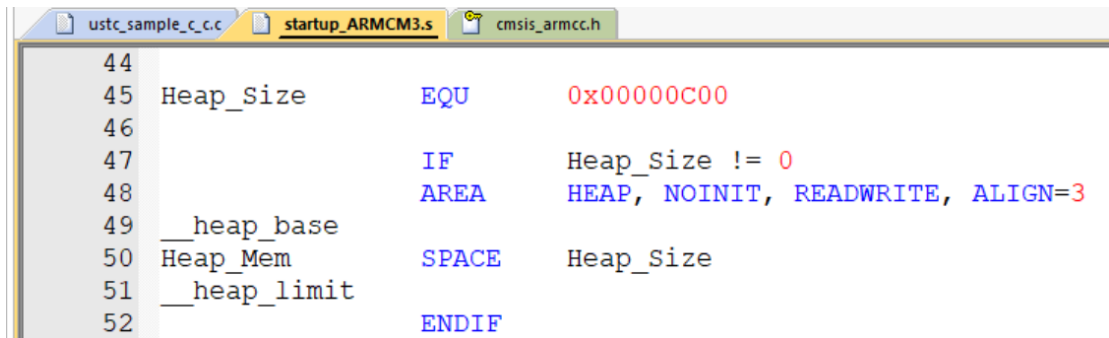
信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO. _____

存在存储器的什么位置（写出存储器地址）？

```
21 | printf("Hello USTCer\n");
```

答：0x00000F80

(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”？



```
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：初始化的堆栈头地址为 0x20000178，堆空间为 0x00000C00，栈空间为 0x00000400，初始化的栈顶为 0x20001178。栈是向下增长的，所以 dststr 存放的地址是 0x20001168。

实 验 报 告

评分：

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO. _____

【实验题目】实验 3 基于 STM32 库的 GPIO 与定时器

【实验目的】

- (1) 掌握 μ Vision 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 μ Vision IDE 中外设仿真模块（GPIO）的使用
 - a) 学会利用外设仿真模块（GPIO）观察 I/O 引脚输出
 - b) 学会利用外设仿真模块（GPIO）模拟 I/O 引脚的输入
- (5) 掌握 μ Vision IDE 逻辑分析模块（Logic Analyzer）的使用

【实验内容】

3.2.1 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

3.2.2 建立基于 STM32 库的 Project

3.2.3 配置 Project 的头文件目录、预编译参数、Simulator

3.2.4 Debug 时使用外设仿真功能验证 GPIO 输入和输出

- (40) 观察示例代码中一下代码行执行前后 GPIOD 的变化：

执行之前：

实 验 报 告

评分:

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO.

General Purpose I/O D (GPIO)

Pin	CNF
PD.0	Floating Input
PD.1	Floating Input
PD.2	Floating Input
PD.3	Floating Input
PD.4	Floating Input
PD.5	Floating Input
PD.6	Floating Input
PD.7	Floating Input

Selected Port Pin Configuration
MODE: 0: Input CNF: 1: Floating Input

Configuration & Mode Settings
GPIO_CRH: 0x44444444 GPIO_CRL: 0x44444444

GPIO
GPIO_IDR: 0x00000000 15 Bits 8 7 Bits 0
GPIO_ODR: 0x00000000 LCKK
GPIO_LCKR: 0x00000000
Pins: 0x00000000

Settings: Clock Disabled

执行之后:

General Purpose I/O D (GPIO)

Pin	CNF
PD.0	GP output push-pull
PD.1	Floating Input
PD.2	GP output push-pull
PD.3	Floating Input
PD.4	Floating Input
PD.5	Floating Input
PD.6	Floating Input
PD.7	Floating Input

Selected Port Pin Configuration
MODE: 3: Output (50 MHz) CNF: 0: GP output push-pull

Configuration & Mode Settings
GPIO_CRH: 0x44444444 GPIO_CRL: 0x44444343

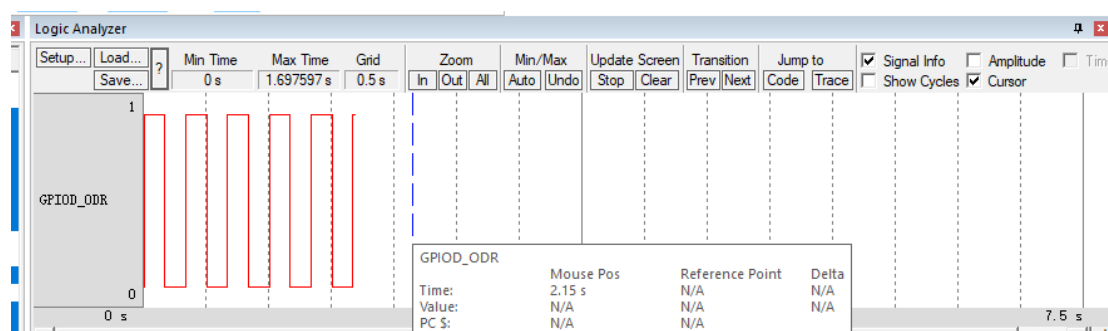
GPIO
GPIO_IDR: 0x00000005 15 Bits 8 7 Bits 0
GPIO_ODR: 0x00000005 LCKK
GPIO_LCKR: 0x00000000
Pins: 0x00000005

Settings: Clock Enabled

评分：

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO.

(41) 通过 μ Vision IDE 自带的逻辑分析模块 (Logic Analyzer) 观察 GPIOD 的变化。

[illegible]

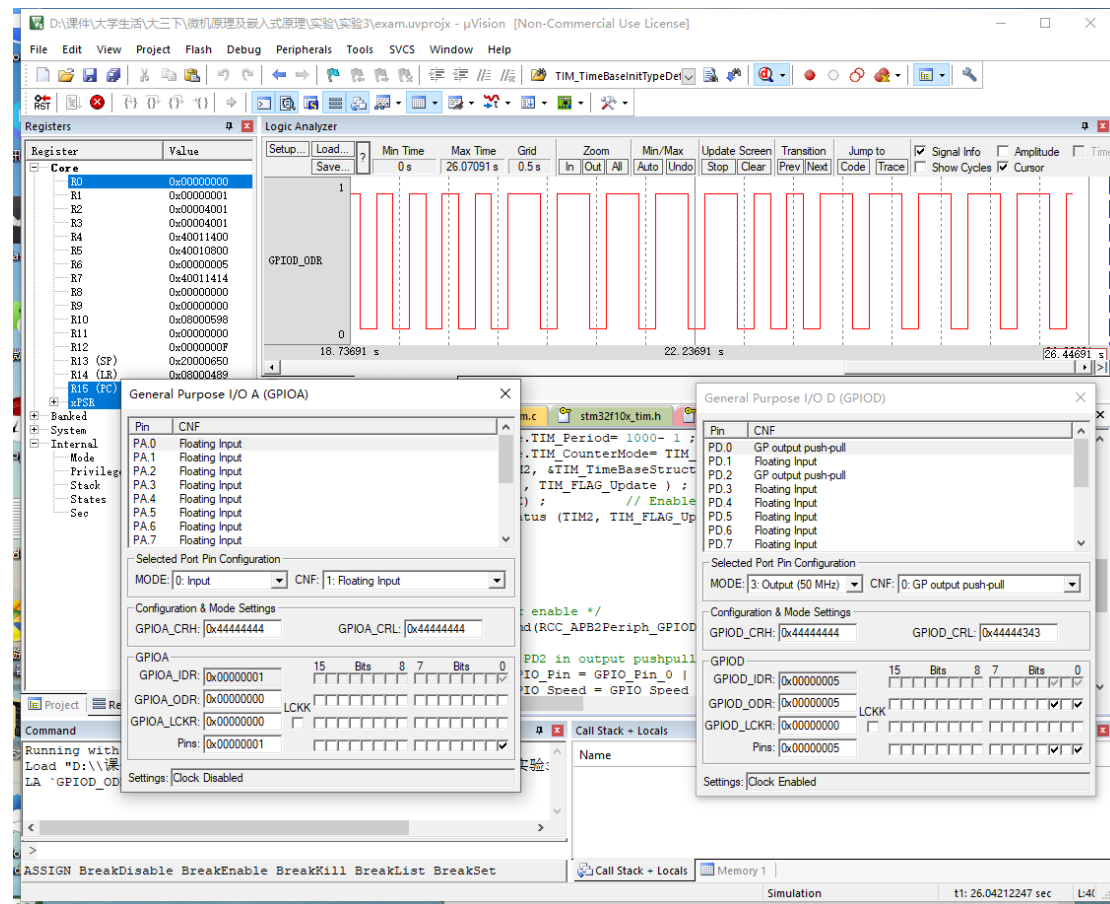
(42) 通过再 GPIOA 窗口单击 PA0 引脚位置, 模拟 PA0 输入 “1”, 观察 Logic

实验报告

评分:

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO.

Analyzer 的变化，并与示例代码进行验证。



3.2.7 定时器的配置

(43) 阅读代码的注释信息，了解该结构体各成员的含义。

```
typedef struct
{
    uint16_t TIM_Prescaler;          /*!< Specifies the prescaler value used to divide the TIM clock.
                                     This parameter can be a number between 0x0000 and 0xFFFF.

    uint16_t TIM_CounterMode;        /*!< Specifies the counter mode.
                                     This parameter can be a value of @ref TIM_Counter_Mode.

    uint16_t TIM_Period;             /*!< Specifies the period value to be loaded into the active
                                     Auto-Reload Register at the next update event.
                                     This parameter must be a number between 0x0000 and 0xFFFF.

    uint16_t TIM_ClockDivision;      /*!< Specifies the clock division.
                                     This parameter can be a value of @ref TIM_Clock_Division.

    uint8_t TIM_RepetitionCounter;   /*!< Specifies the repetition counter value. Each time the RC
                                     reaches zero, an update event is generated and counting
                                     from the RCR value (N).
                                     This means in PWM mode that (N+1) corresponds to:
                                     - the number of PWM periods in edge-aligned mode
                                     - the number of half PWM period in center-aligned mode
                                     This parameter must be a number between 0x00 and 0xFF.
                                     @note This parameter is valid only for TIM1 and TIM8. */
    TIM_TimeBaseInitTypeDef;
};
```

实 验 报 告

评分:

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO. _____

```
typedef struct
{
    uint16_t TIM_Prescaler; // 指定用于分频 TIM 时钟的预分频器值。该参数可以是 0x0000 和 0xFFFF 之间的数字

    uint16_t TIM_CounterMode; // 指定计数器模式。该参数可以是 @ref TIM_计数器_模式的值

    uint16_t TIM_Period; // 指定要加载到活动中的周期值。下一次更新事件时自动重新加载寄存器。该参数必须是 0x0000 和 0xFFFF 之间的数字。

    uint16_t TIM_ClockDivision; // 指定时钟划分。该参数可以是 @ref TIM_除法_CKD 的值

    uint8_t TIM_RepetitionCounter; // 指定重复计数器值。每次 RCR 向下计数。达到零时，会生成更新事件并重新开始计数。来自 RCR 值(N)。

    这意味着在脉宽调制模式下，(N-1)对应于:
    -边沿对齐模式下的脉宽调制周期数
    -中间对齐模式下半个脉宽调制周期的数量

    该参数必须是 0x00 和 0xFF 之间的数字。

    @注意该参数仅对 TIM1 和 TIM8 有效。
} TIM_TimeBaseInitTypeDef;
```

【3.3 思考题（部分内容需要上机调试才能完成）】

(44) 阅读讲义 8.5.3 小节, 及 STM32 芯片手册, 解释下图中“CRL、CRH、IDR、ODR、LCKR”几个寄存器的作用。

实验报告

评分：

信院系 17 级 姓名 胡 睿 日期 2020-7-23 NO.

General Purpose I/O D (GPIO)

Pin	CNF
PD.0	GP output push-pull
PD.1	Floating Input
PD.2	GP output push-pull
PD.3	Floating Input
PD.4	Floating Input
PD.5	Floating Input
PD.6	Floating Input
PD.7	Floating Input

Selected Port Pin Configuration

MODE: 3: Output (50 MHz) CNF: 0: GP output push-pull

Configuration & Mode Settings

GPIO_CRH: 0x44444444 GPIO_CRL: 0x44444343

GPIO

GPIO_IDR: 0x00000005

GPIO_ODR: 0x00000005

GPIO_LCKR: 0x00000000

Pins: 0x00000005

	15	Bits	8	7	Bits	0
LCKK	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Settings: Clock Enabled

答: CRL 寄存器: 端口配置寄存器, 配置 GPIO 工作模式, 用于控制 GPIOX

(X 表示 A—G) 的低 8 位 (Pin7-Pin0);

CRH 寄存器：端口配置寄存器，配置 GPIO 工作模式，用于控制 GPIOX (X 表示 A—G) 的高 8 位 (Pin15-Pin8)；

IDR 寄存器：端口输入数据寄存器，只用了低 16 位。该寄存器为只读寄存器，并且只能以 16 位的形式读出。读出的值为对应 IO 口的状态。

ODR 寄存器：端口输出数据寄存器，也只用了低 16 位。该寄存器虽然为可读写，但是从该寄存器读出来的数据都是 0。只有写是有效的。其作用就是控制端口的输出。

LCKR 寄存器：端口配置锁定寄存器，端口锁定后下次系统复位之前将不能再更改端口位的配置。

(45) 解释代码 “GPIO->BSRR = 0x00000085;” 的作用。

答：设置 I/O 端口 7，端口 2 和端口 0 为高，保持其他 I/O 端口不变。

(46) 解释代码 “GPIOD->BRR = 0x00000080;” 的作用。

答：设置 I/O 端口 7 为低，保持其他 I/O 端口不变。

实 验 报 告

评分：

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO. _____

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

答：用来读取 GPIO 的 I/O 端口 0 状态的速率。

实 验 报 告

评分：

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO. _____

【实验题目】实验 4 基于 STM32 库的中断

【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、NVIC 相关库函数
- 5、掌握 μ Vision IDE 中外设仿真模块（NVIC）的使用

【实验内容】

4.2.1 建立基于 STM32 库的 Project

4.2.2 配置 Project 的头文件目录、预编译参数、Simulator

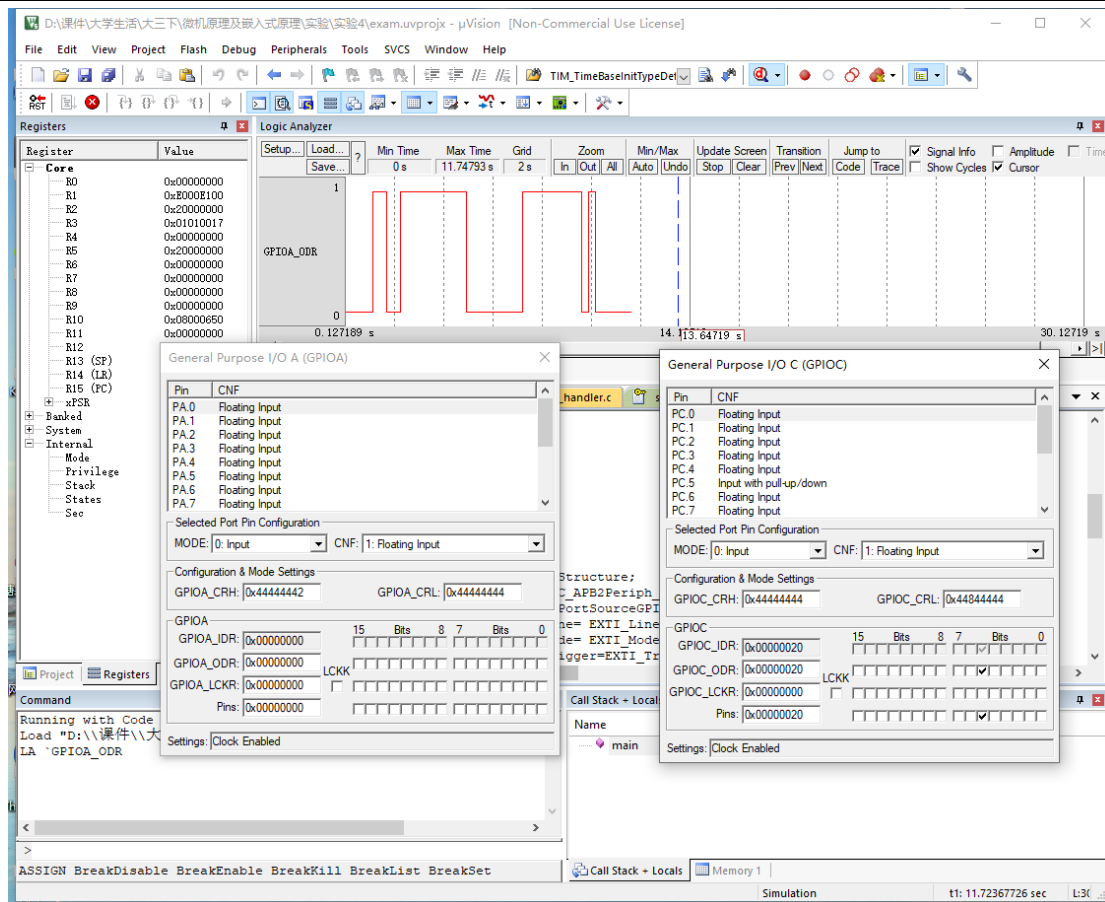
4.2.3 使用 Logic Analyzer 和外设仿真功能验证 EXIT 及 GPIO 输出

(59) F5 运行示例代码，在 GPIOC 窗口可以输入 PC5（模拟中断信号）。随着 PC5 信号的变化，可以在 Logic Analyzer 窗口和 GPIOA 窗口同时观察到 PA8 的变化。

实验报告

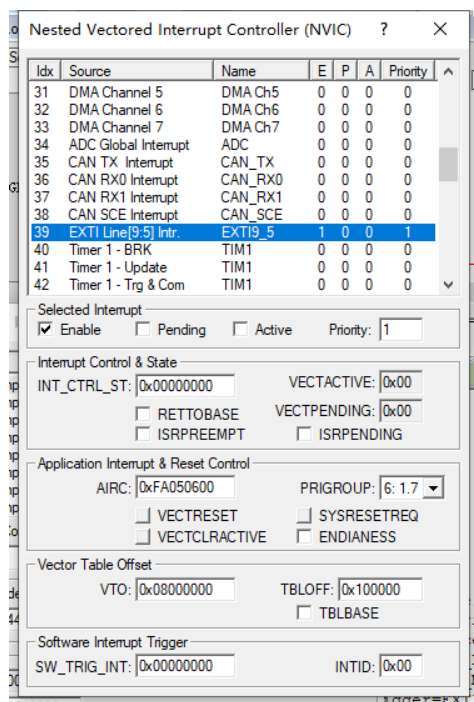
评分:

信院系 17 级 姓名 胡睿 日期 2020-7-23 NO.



4.2.4 观察 NVIC 寄存器组

(60) 观察与外部中断#5 有关的寄存器信息。



实 验 报 告

评分：

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO.

4.3 思考题

(61) 为什么通过 GPIOC 的窗口 (Peripherals General purpose I/O GPIOC)

模拟 PC5 (中断信号) 输入的时候, 改变两次 PC5 后 PA8 才会发生变化?

答: 因为在实际操作中, 每当产生中断信息后, 芯片会自动复位, 但是按照软件进行模拟的时候, 并没有自动产生复位, 所以需要改变两次 PC5, 相当于复位一次后, 才能发生变化。

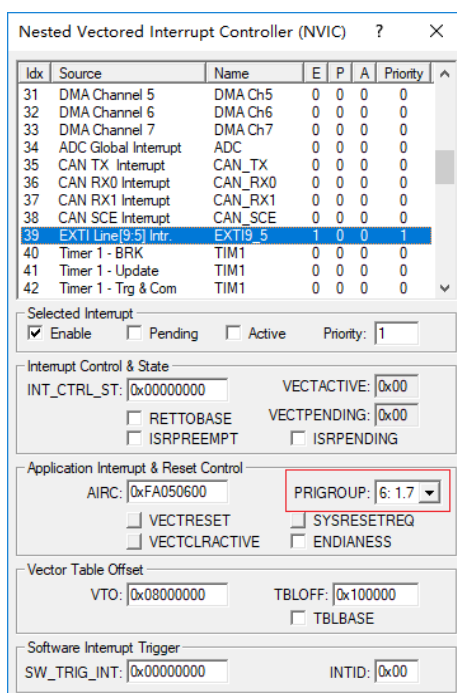
(62) 请通过调试获得 EXTI9_5_IRQHandler() 的入口地址 (应该是 0x800026A),

这个地址保存在异常向量表什么位置?

答: 入口地址为 0x8000026A

通过查询芯片手册可以知道, 会保存在 0x0000009C 的位置, 即向量表开始地址往后偏移 9C 的位置。

(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?



实 验 报 告

评分：

信院 系 17 级 姓名 胡 睿 日期 2020-7-23 NO. _____

答：这是复位控制寄存器的 PRIGROUP 域，通过设置该域来完成优先级配置。

此时表示抢占优先级域是 Bit[7]，子优先级域 Bit[6:0]。

PRIGROUP 是 32 位寄存器 AIRC 中的第[10:8]位。