



福州大学

嵌入式系统实践

姓 名：_____ 孟岩

学 号：_____ 111700426

学 院：_____ 物理与信息工程学院

专 业：_____ 微电子科学与工程

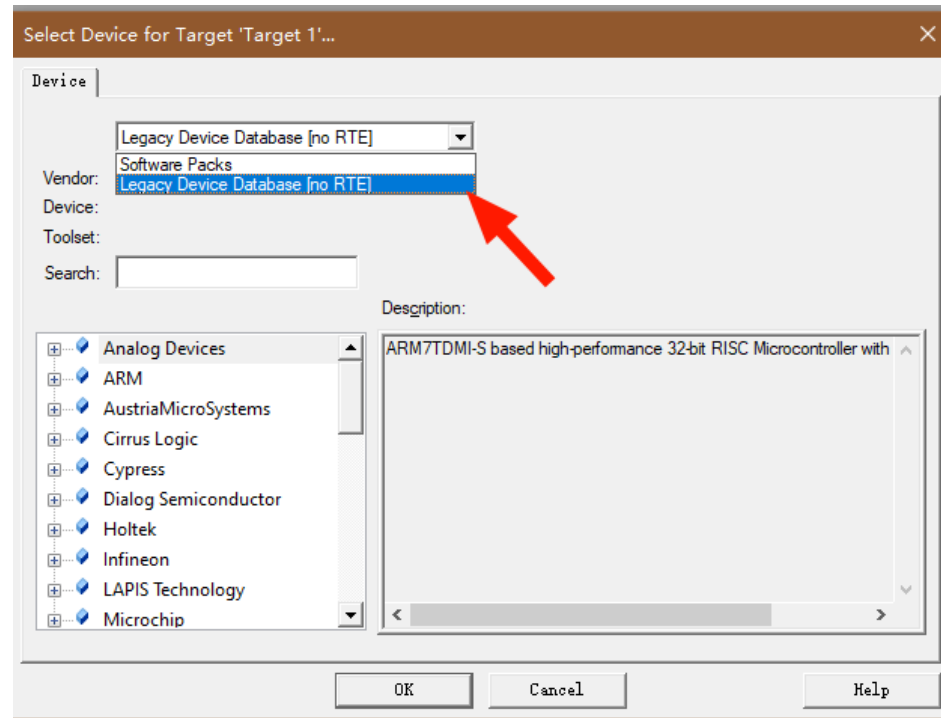
年 级：_____ 2017

指导教师：_____ 杨涛

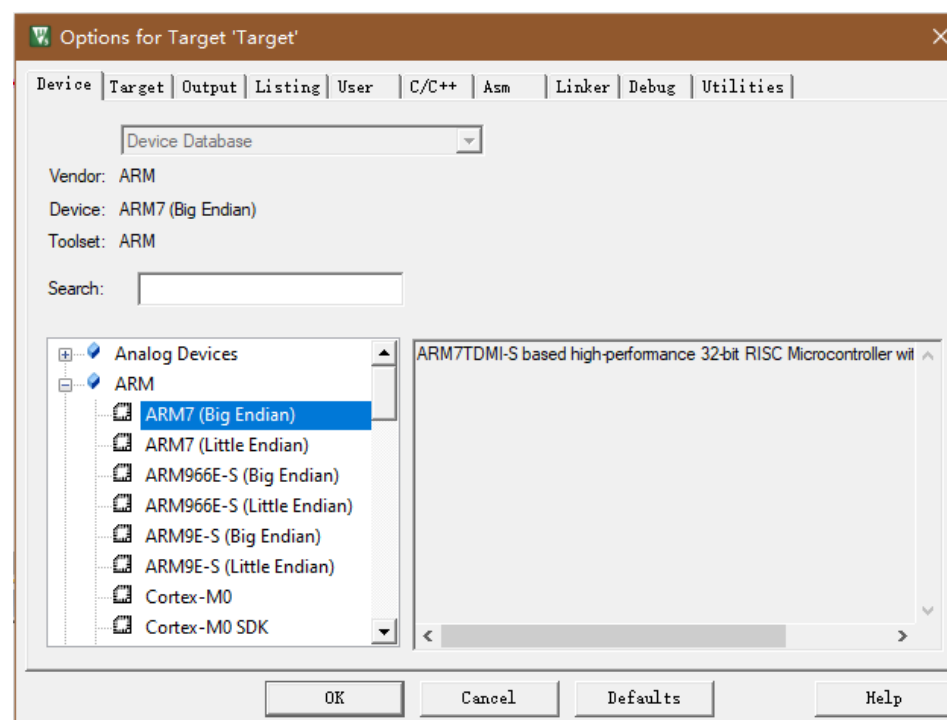
2020 年 12 月 1 日

一. 验证实验

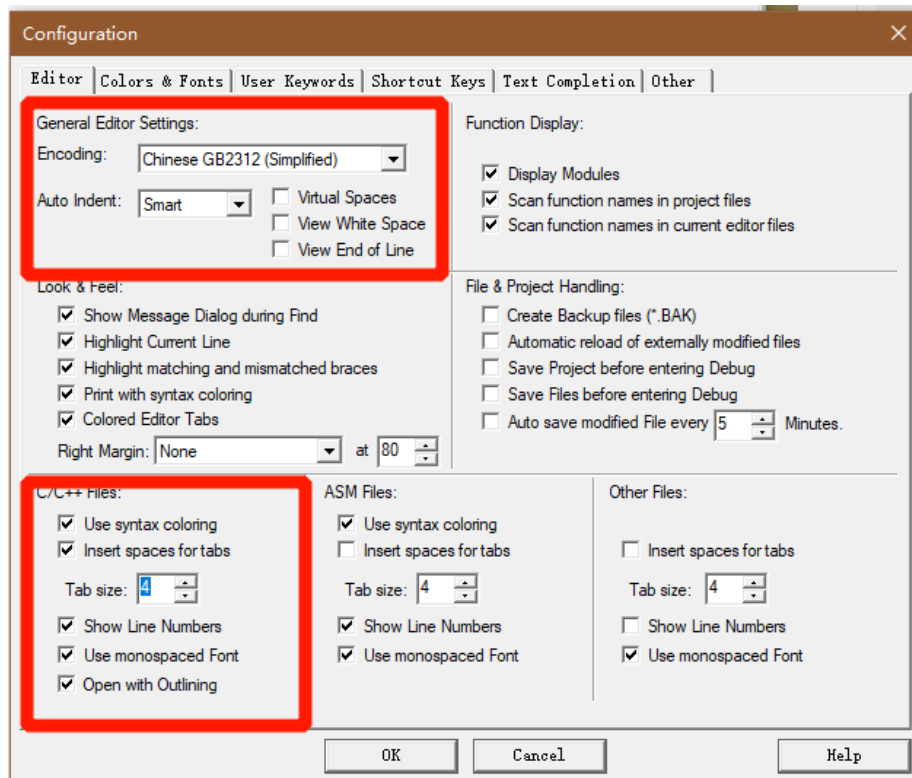
完成对书本上汇编代码的验证,keilV5.25 拓展包将是最后支持 Arm7 的版本, 所以安装 keilV5.25 拓展包 : <http://www2.keil.com/mdk5/legacy/>
芯片包: <https://www.keil.com/dd2/pack/>




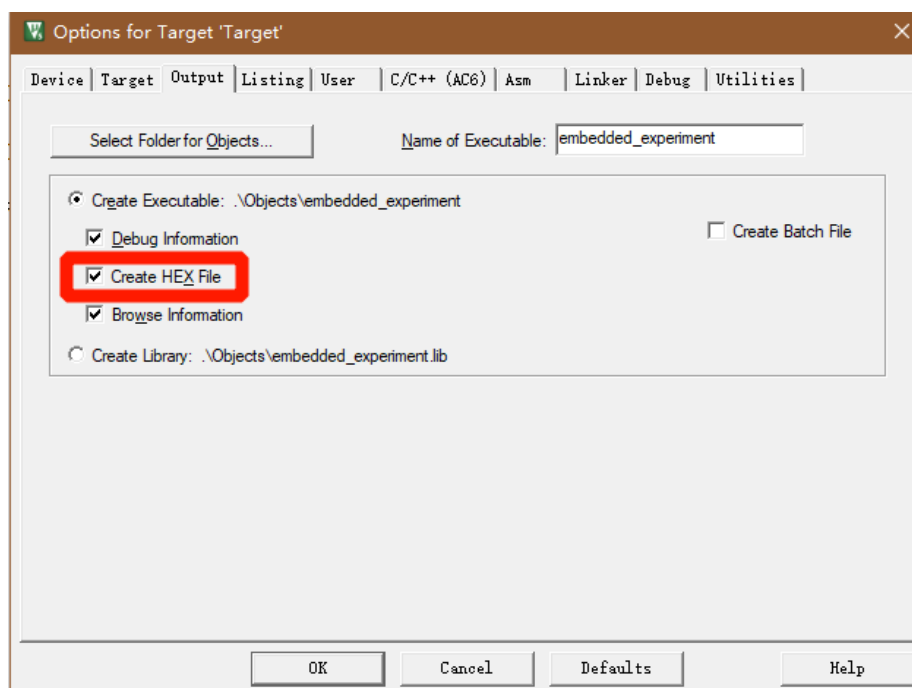
下载完成后新建工程时要注意选择 Legacy Device Database,不要选择 Software Packs,不然找不到旧的支持包, 这里选择 ARM7 内核,不添加启动文件,不选择器件, 只做纯软件仿真, 否则需要添加启动代码。



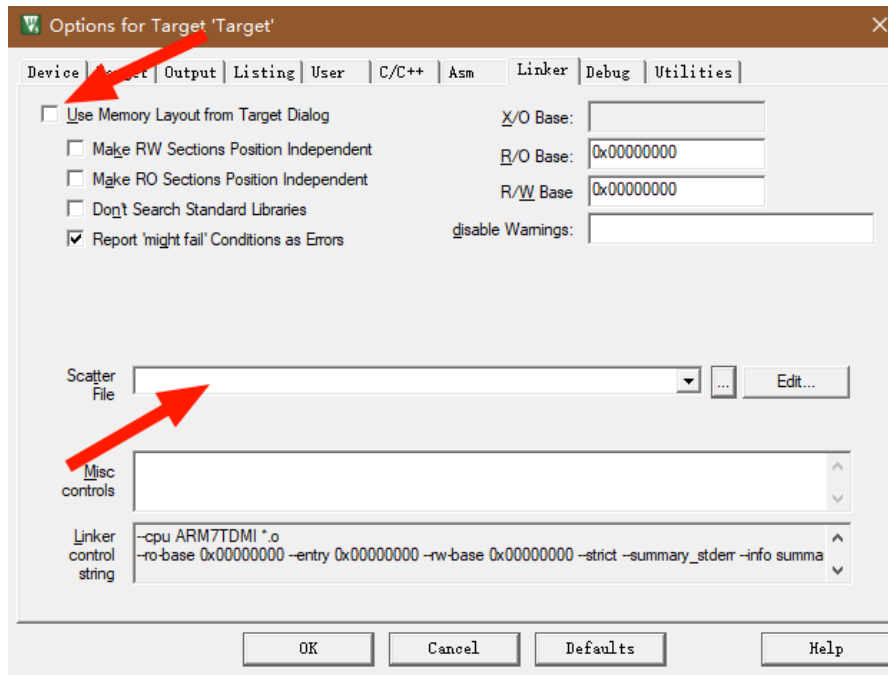
然后对工程设置做一些修改 



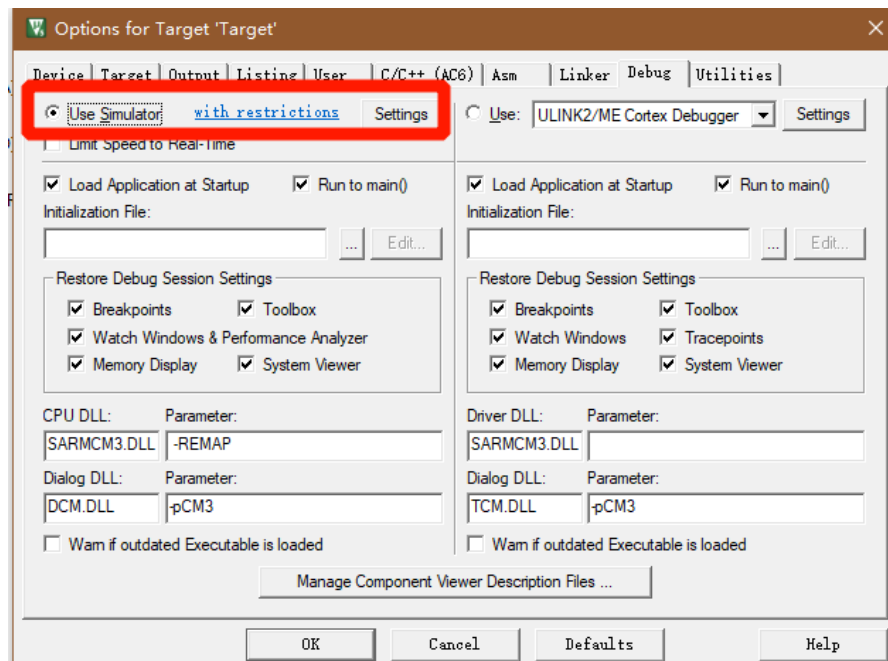
 创建 16 进制文件



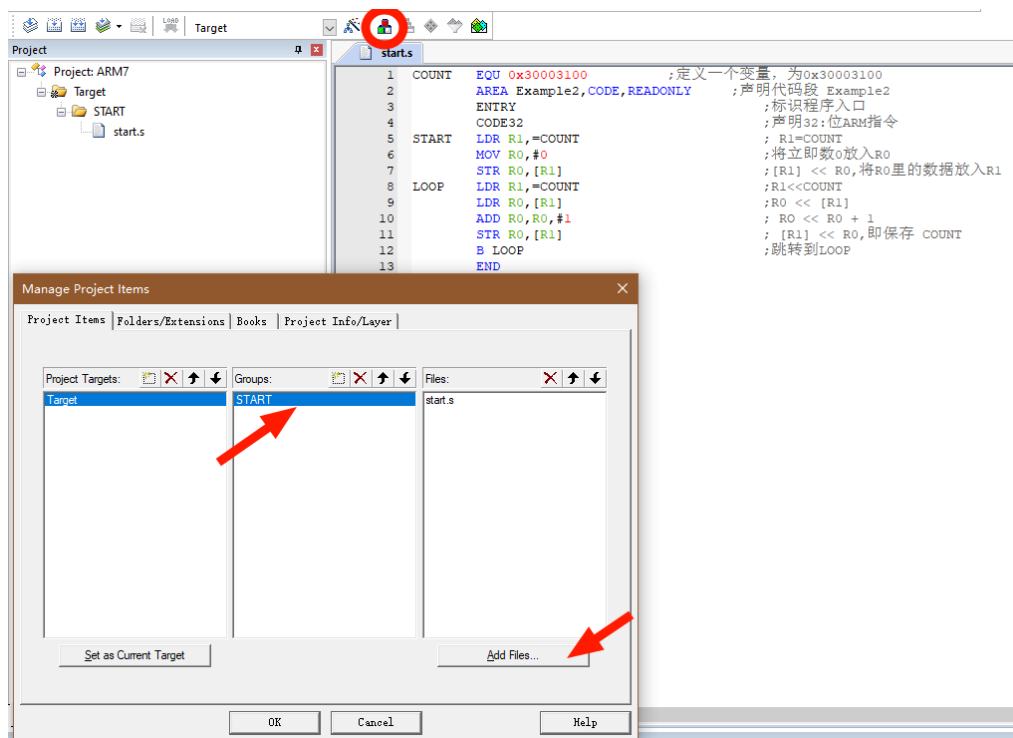
关闭连接文件，不使用分散加载文件,只进行纯软件仿真



使用软件仿真



添加工程文件



1.1 掌握 ARM 通用寄存器，存储器的访问方法

- ① 使用 MOV 指令访问 ARM 通用寄存器
- ② 使用 LDR/STR 指令完成存储器的访问

验证代码：

```

COUNT EQU 0x30003100      ; 定义一个变量，为 0x30003100
      AREA Example2, CODE, READONLY    ; 声明代码段 Example2
      ENTRY                          ; 标识程序入口
      CODE32                        ; 声明 32: 位 ARM 指令
START  LDR R1, =COUNT           ; R1=COUNT
      MOV R0, #0                  ; 将立即数 0 放入 R0
      STR R0, [R1]                ; [R1] << R0, 将 R0 里的数据放入 R1
LOOP   LDR R1, =COUNT           ; R1<<COUNT
      LDR R0, [R1]                ; R0 << [R1]
      ADD R0, R0, #1              ; R0 << R0 + 1
      STR R0, [R1]                ; [R1] << R0, 即保存 COUNT
      B LOOP                      ; 跳转到 LOOP
END
  
```

◆ AREA Example2, CODE, READONLY ; 声明了一个代码段，名字为 Example2，类型为 CODE, 本段只读

CODE16 和 CODE32, CODE16 表示汇编编译器后边的指令为 16 位的 Thumb 指令，CODE32 表示汇编编译器后边的指令为 32 位 arm 指令

◆ LDR

大范围的地址读取伪指令. LDR 伪指令用于加载 32 位的立即数或一个地址值到指定寄存器. 在汇编编译源程序时, LDR 伪指令被编译器替换成一条合适的指令. 若加载的常数未超出 MOV 范围, 则使用 MOV 或 MVN 指令代替 LDR 伪指令, 否则汇编器将常量放入文字池, 并使用一条程序相对偏移的 LDR 指令从文字池读出常量. LDR 伪指令格式如下.

LDR register, =expr/label_expr

其中, register 加载的目标寄存器

 expr 32 位立即数.

 label_expr 基于 PC 的地址表达式或外部表达式.

LADR 伪指令举例如下;

LDR R0, =0x12345678 ;加载 32 位立即数 0x12345678

LDR R0, =DATA_BUF+60 ;加载 DATA_BUF 地址+60

...

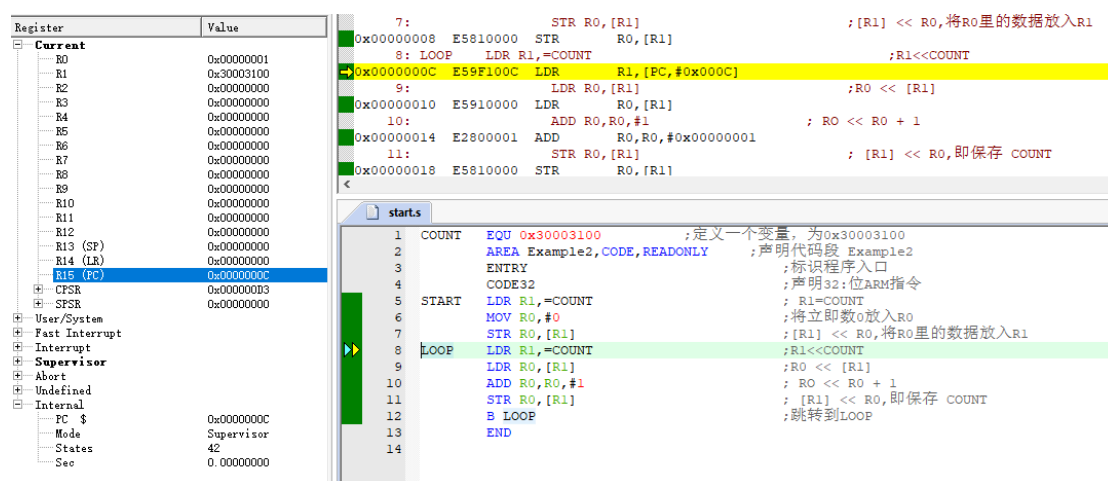
LTORG ;声明文字池

...

从 PC 到文字池的偏移量必须是正数小于是 1KB.

与 Thumb 指令的 LDR 相比, 伪指令的 LDR 的参数有 “=” 号.






完成代码输入之后进行编译   , 编译无误进行仿真 



Keil MDK-ARM 汇编界面截图。左侧显示寄存器列表，R15 (PC) 值为 0x0000000C。右侧显示汇编代码，包括 LDR、STR、ADD、SUB 等指令。代码中使用了 LDR 伪指令来加载立即数和地址。

Register	Value
R0	0x00000001
R1	0x30003100
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x0000000C
CPSR	0x00000003
SFSR	0x00000000

```
7: STR R0, [R1] ; [R1] << R0, 将R0里的数据放入R1
8: LOOP LDR R1, =COUNT ; R1<<COUNT
9: LDR R0, [R1] ; R0 << [R1]
10: ADD R0, R0, #1 ; R0 << R0 + 1
11: STR R0, [R1] ; [R1] << R0, 即保存 COUNT
12: B LOOP
13: END
```

使用调试工具      进行调试并查看寄存器变化, 发现没有修改权限

1.2 掌握 ARM 算术，逻辑运算指令

① 使用 ADD、SUB、AND、ORR、CMP、TST 等指令完成数据加减

验证代码：

```
1 X EQU 11 ;定义x的值为11
2 Y EQU 8 ;定义y的值为8
3 BIT23 EQU (1<<23) ;定义 BIT23 的值为 0x00800000
4
5
6 AREA Example2, CODE, READONLY ;声明代码段 Example2
7 ENTRY ;标识程序入口
8 CODE32 ;声明32位ARM指令
9 START ;使用 MOV、ADD 指令实现 R8 = R3 = X + Y
10
11 MOV R0, #X ;R0 <= X, X的值必须是8位图数据
12 MOV R1, #Y ;R1 <= Y, Y的值必须是8位图数据
13 ADD R3, R0, R1 ;即是 R3 = X + Y
14 MOV R8, R3 ;R8<= R3
15
16 ;使用 MVN、SUB 指令实现: R5 = 0x5FFFFFF8 - R8 * 8
17 MVN R0, #0xA0000007 ; 0xA0000007* 的反码为 0x5FFFFFF8
18 SUB R5, R0, R8, LSL #3 ; R8 左移 3 位, 结果即是 R8 * 8
19
20 ;使用CMP指令判断 (5*Y/2)>> (2*X) 吗? 若大于则R5 = R5&0xFFFF0000, 否则 R5 = R5|0x000000FF
21 MOV R0, #Y
22 ADD R0, R0, R0, LSL #2 ;计算R0 = Y + 4*Y = 5*Y
23 MOV R0, R0, LSR #1 ;计算R0 = 5*Y/2
24 MOV R1, #X
25 MOV R1, R1, LSL #1 ;计算 R1 = 2*X
26 CMP R0, R1 ;比较 R0 和 R1, 即 (5*Y/2) 和 (2*X) 进行
27 LDRHI R2, #0xFFFF0000 ;若 (5*Y/2)> (2*X), 则 R2 <= 0xFFFF0000
28 ANDHI R5, R5, R2 ;若 (5*Y/2)> (2*X), 则 R5 = R5&R2
29 ORRLS R5, R5, #0x000000FF ;若 (5*Y/2) < (2*X), 则 R5 = R5|0x000000FF
30
31 ;使用TST指令测试R5的bit23是否为1, 若是则将bit6位清零(使用BIC指令)
32 TST R5, #BIT23
33 BICNE R5, R5, #0x00000040
34 B START
35
36 END
```

打上断点并进行仿真验证，详细指令使用规则可参见指令集

Register	Value	0x00000000 E3A0000B MOV R0, #0x0000000B
Current		12: MOV R1, #Y ;R1 <= Y, Y的值必须是8位图数据
R0	0x00000008	0x00000004 E3A01008 MOV R1, #0x00000008 ;R1 <= Y, Y的值必须是8位图数据
R1	0x00000008	13: ADD R3, R0, R1 ;即是 R3 = X + Y
R2	0x00000000	0x00000008 E0803001 ADD R3, R0, R1 ;即是 R3 = X + Y
R3	0x00000013	14: MOV R8, R3 ;R8<= R3
R4	0x00000000	15: ;使用 MVN、SUB 指令实现: R5 = 0x5FFFFFF8 - R8 * 8
R5	0x00000000	0x0000000C E1A08003 MOV R8, R3
R6	0x00000000	16: ;使用 MVN、SUB 指令实现: R5 = 0x5FFFFFF8 - R8 * 8
R7	0x00000000	17: MVN R0, #0xA0000007 ; 0xA0000007* 的反码为 0x5FFFFFF8
R8	0x00000013	18: SUB R5, R0, R8, LSL #3 ; R8 左移 3 位, 结果即是 R8 * 8
R9	0x00000000	19: ;使用CMP指令判断 (5*Y/2)>> (2*X) 吗? 若大于则R5 = R5&0xFFFF0000, 否则 R5 = R5 0x000000FF
R10	0x00000000	0x00000010 E3E027A MVN R0, #0xA0000007 ; 0xA0000007* 的反码为 0x5FFFFFF8
R11	0x00000000	20: ;使用CMP指令判断 (5*Y/2)>> (2*X) 吗? 若大于则R5 = R5&0xFFFF0000, 否则 R5 = R5 0x000000FF
R12	0x00000000	0x00000014 E0405188 SUB R5, R0, R8, LSL #3
R13 (SP)	0x00000000	21: MOV R0, #Y
R14 (LR)	0x00000000	0x00000018 E3A00008 MOV R0, #0x00000008 ;R1 <= Y, Y的值必须是8位图数据
R15 (PC)	0x00000014	22: ADD R0, R0, R0, LSL #2 ;计算R0 = Y + 4*Y = 5*Y
CSR	0x00000003	
SPSR	0x00000000	

Register	Value				
Current		26:	CMP R0,R1		;比较 R0 和 R1, 即 (5*Y/2) 和 (2*X) 进行
R0	0x00000014	0x0000002C	E1500001	CMP	R0,R1
R1	0x00000016	27:	LDRHI R2,=0xFFFF0000		;若 (5*Y/2) > (2*X), 则 R2 <= 0xFFFF0000
R2	0x00000000	0x00000030	859F2010	LDRHI	R2,[PC,#0x0010]
R3	0x00000013	28:	ANDHI R5,R5,R2		;若 (5*Y/2) > (2*X), 则 R5 = R5&R2
R4	0x00000000	0x00000034	80055002	ANDHI	R5,R5,R2
R5	0x5FFFFFF60	29:	ORRLS R5,R5,#0x000000FF		;若 (5*Y/2) < (2*X), 则 R5 = R5 0x000000FF
R6	0x00000000	30:			
R7	0x00000000	31:			;使用TST指令测试R5的bit23是否为1, 若是则将bit6位清零(使用BIC指令)
R8	0x00000013	0x00000038	938550FF	ORRLS	R5,R5,#0x000000FF
R9	0x00000000	32:	TST R5,#BIT23		
R10	0x00000000	0x0000003C	E3150502	TST	R5,#0x00800000
R11	0x00000000	33:	BICNE R5,R5,#0x00000040		
R12	0x00000000	0x00000040	13C55040	BICNE	R5,R5,#0x00000040
R13 (SP)	0x00000000	34:	B START		
R14 (LR)	0x00000000	0x00000044	EAffFFFED	B	0x00000000
R15 (PC)	0x00000038	0x00000048	FFFF0000	(???)	
CPSR	0x80000003	0x0000004C	00000000	ANDEQ	R0,R0,R0
SFSR	0x00000000				

Register	Value				
Current		26:	CMP R0,R1		;比较 R0 和 R1, 即 (5*Y/2) 和 (2*X) 进行
R0	0x00000014	0x0000002C	E1500001	CMP	R0,R1
R1	0x00000016	27:	LDRHI R2,=0xFFFF0000		;若 (5*Y/2) > (2*X), 则 R2 <= 0xFFFF0000
R2	0x00000000	0x00000030	859F2010	LDRHI	R2,[PC,#0x0010]
R3	0x00000013	28:	ANDHI R5,R5,R2		;若 (5*Y/2) > (2*X), 则 R5 = R5&R2
R4	0x00000000	0x00000034	80055002	ANDHI	R5,R5,R2
R5	0x5FFFFFF60	29:	ORRLS R5,R5,#0x000000FF		;若 (5*Y/2) < (2*X), 则 R5 = R5 0x000000FF
R6	0x00000000	30:			
R7	0x00000000	31:			;使用TST指令测试R5的bit23是否为1, 若是则将bit6位清零(使用BIC指令)
R8	0x00000013	0x00000038	938550FF	ORRLS	R5,R5,#0x000000FF
R9	0x00000000	32:	TST R5,#BIT23		
R10	0x00000000	0x0000003C	E3150502	TST	R5,#0x00800000
R11	0x00000000	33:	BICNE R5,R5,#0x00000040		
R12	0x00000000	0x00000040	13C55040	BICNE	R5,R5,#0x00000040
R13 (SP)	0x00000000	34:	B START		
R14 (LR)	0x00000000	0x00000044	EAffFFFED	B	0x00000000
R15 (PC)	0x00000044	0x00000048	FFFF0000	(???)	
CPSR	0x00000003	0x0000004C	00000000	ANDEQ	R0,R0,R0
SFSR	0x00000000				

1.3 掌握 ARM 寻址方式

① 编写汇编程序，分别实现:立即数寻址,寄存器寻址,寄存器间接寻址,基址变址寻址,多寄存器寻址

验证代码:

◆ LDMIA 和 STMIA

批量加载/存储指令可以实现在一组寄存器和一块连续的内存单元之间传输数据. Thumb 指令集批量加载/存储指令为 LDMIA 和 STMIA, LDMIA 为加载多个寄存器; STM 为存储多个寄存器, 允许一条指令传送 8 个低寄存器的任何子集. 指令格式如下;

LDMIA Rn!,reglist

STMIA Rn!,reglist

其中 Rn 加载/存储的起始地址寄存器. Rn 必须为 R0~R7

Reglist 加载/存储的寄存器列表. 寄存器必须为 R0~R7

LDMIA/STMIA 的主要用途是数据复制, 参数传送等, 进行数据传送时, 每次传送后地址加 4. 若 Rn 在寄存器列表中, 对于 LDMIA 指令, Rn 的最终值是加载的值, 而不是增加后的地址; 对于 STMIA 指令, 在 Rn 是寄存器列表中的最低数字的寄存器, 则 Rn 存储的值为 Rn 在初值, 其它情况不可预知.

批量加载/存储指令举例如下;

LDMIA R0, {R2~R7} ;加载 R0 指向的地址上的多字数据, 保存到 R2~R7 中, R0 的值更新.

STMIA R1!, {R2~R7} ;将 R2~R7 的数据存储到 R1 指向的地址上, R1 值更新.

```

1  AREA TEST, CODE, READONLY
2  ENTRY
3  CODE32
4  ;立即数寻址
5
6  MOV R0, #0x1 ;立即数0x1存入寄存器R0
7  ADD R0, R0, #0x4 ;寄存器R0中的内容加上立即数0x4然后存入寄存器R0
8
9  ;寄存器寻址
10 MOV R1, #0x2 ;立即数0x2存入寄存器R1
11 ADD R2, R1, R0 ;寄存器R0中的内容和寄存器R1的内容相加存入寄存器R2
12 ADD R2, R2, R0, LSL #1 ;寄存器R0中的内容逻辑左移1位加上寄存器R2的内容存入寄存器R2
13
14 ;寄存器间接寻址
15 MOV R1, #0x10 ;将立即数0x10存入寄存器R1
16 STR R2, [R1] ;将寄存器R2的内容存入以寄存器R1的值为地址的存储器中
17 SWP R0, R0, [R1] ;完成寄存器R1所指向的存储器中的字数据与寄存器R0的内容交换
18
19 ;基址变址寻址
20 MOV R3, #0x14 ;将立即数0x14存入寄存器R3
21 MOV R0, #0x3 ;将立即数0x3存入寄存器R0
22 STR R0, [R3] ;将寄存器R0的内容存入寄存器R3所指派的存储器
23 LDR R4, [R1, #4] ;将寄存器R1的内容加上4所指派的存储器的字存入寄存器R4
24
25 ;多寄存器寻址
26 LDMIA R1, {R5, R6}
27 ;相对寻址
28 B NEXT
29 NOP
30 NOP
31 NOP
32 NOP
33 NOP
34 NOP
35 NEXT
36 NOP
37 NOP
38 NOP
39 END

```

打上断点并进行仿真验证，需要在终端输入 **map 0x00, 0x100 read write exec**

Register	Value
Current	0x00000005
R0	0x00000005
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)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000014
CPSR	0x00000003
SFSR	0x00000000

Address	Instruction	Comment
0x00000000	MOV R0, #0x1	;立即数0x1存入寄存器R0
0x00000004	ADD R0, R0, #0x4	;寄存器R0中的内容加上立即数0x4然后存入寄存器R0
0x00000008	MOV R1, #0x2	;立即数0x2存入寄存器R1
0x0000000C	ADD R2, R1, R0	;寄存器R0中的内容和寄存器R1的内容相加存入寄存器R2
0x00000010	ADD R2, R2, R0, LSL #1	;寄存器R0中的内容逻辑左移1位加上寄存器R2的内容存入寄存器R2
0x00000014	MOV R1, #0x10	;将立即数0x10存入寄存器R1
0x00000018	STR R2, [R1]	;将寄存器R2的内容存入以寄存器R1的值为地址的存储器中
0x0000001C	SWP R0, R0, [R1]	;完成寄存器R1所指向的存储器中的字数据与寄存器R0的内容交换
0x00000020	MOV R3, #0x14	;将立即数0x14存入寄存器R3
0x00000024	MOV R0, #0x3	;将立即数0x3存入寄存器R0
0x00000028	STR R0, [R3]	;将寄存器R0的内容存入寄存器R3所指派的存储器
0x0000002C	LDR R4, [R1, #4]	;将寄存器R1的内容加上4所指派的存储器的字存入寄存器R4
0x00000030	LDMIA R1, {R5, R6}	
0x00000034	B NEXT	
0x00000038	NOP	
0x0000003C	NOP	
0x00000040	NOP	
0x00000044	NOP	
0x00000048	NOP	
0x0000004C	NOP	
0x00000050	NEXT	
0x00000054	NOP	
0x00000058	NOP	
0x0000005C	NOP	
0x00000060	NOP	
0x00000064	END	

Register	Value
Current	0x00000004
R0	0x00000004
R1	0x00000002
R2	0x00000001
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x0000001C
CPSR	0x00000003
SFSR	0x00000000

Address	Instruction	Comment
0x00000000	MOV R0, #0x1	;立即数0x1存入寄存器R0
0x00000004	ADD R0, R0, #0x4	;寄存器R0中的内容加上立即数0x4然后存入寄存器R0
0x00000008	MOV R1, #0x2	;立即数0x2存入寄存器R1
0x0000000C	ADD R2, R1, R0	;寄存器R0中的内容和寄存器R1的内容相加存入寄存器R2
0x00000010	ADD R2, R2, R0, LSL #1	;寄存器R0中的内容逻辑左移1位加上寄存器R2的内容存入寄存器R2
0x00000014	MOV R1, #0x10	;将立即数0x10存入寄存器R1
0x00000018	STR R2, [R1]	;将寄存器R2的内容存入以寄存器R1的值为地址的存储器中
0x0000001C	SWP R0, R0, [R1]	;完成寄存器R1所指向的存储器中的字数据与寄存器R0的内容交换
0x00000020	MOV R3, #0x14	;将立即数0x14存入寄存器R3
0x00000024	MOV R0, #0x3	;将立即数0x3存入寄存器R0
0x00000028	STR R0, [R3]	;将寄存器R0的内容存入寄存器R3所指派的存储器
0x0000002C	LDR R4, [R1, #4]	;将寄存器R1的内容加上4所指派的存储器的字存入寄存器R4
0x00000030	LDMIA R1, {R5, R6}	
0x00000034	B NEXT	
0x00000038	NOP	
0x0000003C	NOP	
0x00000040	NOP	
0x00000044	NOP	
0x00000048	NOP	
0x0000004C	NOP	
0x00000050	NEXT	
0x00000054	NOP	
0x00000058	NOP	
0x0000005C	NOP	
0x00000060	NOP	
0x00000064	END	

这个时候有提示：Non-aligned Access: ARM Instruction at 00000018H, Memory

Access at 00000002H 非对齐访问,但实际上我认为这个地址是对齐的。可能只是内存空间没有实际映射,尝试修改内存地址为 0x0000FF00。内存操作的地址必须与 4 字节边界对齐。这意味着地址必须是 4 的倍数,或者,地址的后两位必须为零(二进制)。但最后是因为, map 写入时注意写成 32 位。

Register	Value	
Current		
R0	0x00000005	7: ADD R0,R0,#0x4 ;寄存器R0中的内容加上立即数0x4然后存入寄存器R0
R1	0x00000010	8: ;寄存器寻址
R2	0x00000007	9: E2800004 ADD R0,R0,#0x00000004 ;立即数0x2存入寄存器R1
R3	0x00000000	10: MOV R1,#0x2 ;立即数0x2存入寄存器R1
R4	0x00000000	11: E3A01002 MOV R1,#0x00000002 ;寄存器R0中的内容和寄存器R1的内容相加存入寄存器R2
R5	0x00000000	12: ADD R2,R1,R0 ;寄存器R0中的内容逻辑左移1位加上寄存器R2的内容存入寄存器R2
R6	0x00000000	13: E0812000 ADD R2,R1,R0 ;寄存器R0中的内容逻辑左移1位加上寄存器R2的内容存入寄存器R2
R7	0x00000000	14: ;寄存器间接寻址
R8	0x00000000	15: E0000010 00000007 AND EQ R0,R0,R7 ;将立即数0x10存入寄存器R1
R9	0x00000000	16: MOV R1,#0x10 ;将立即数0x10存入寄存器R1
R10	0x00000000	17: E3A01010 MOV R1,#0x00000010 ;将寄存器R2的内容存入以寄存器R1的值为地址的存储器中
R11	0x00000000	18: STR R2,[R1] ;完成寄存器R1所指向的存储器中的字数据与寄存器R0的内容交换
R12	0x00000000	19: E5812000 STR R2,[R1]
R13 (SP)	0x00000000	20: SWP R0,R0,[R1]
R14 (LR)	0x00000000	21: ;基址变址寻址
R15 (PC)	0x00000010	22: E1010090 SWP R0,R0,[R1]
CPSR	0x00000003	23: MOV R3,#0x14 ;将立即数0x14存入寄存器R3
SPSR	0x00000000	24: E3A03014 MOV R3,#0x00000014 ;将立即数0x3存入寄存器R0
User/System		25: MOV R0,#0x00000003 ;将寄存器R0的内容存入寄存器R3所指定的存储器
Fast Interrupt		26: E5830000 STR R0,[R3] ;将寄存器R1的内容加上4所指向的存储器的字存入寄存器R4
Interrupt		27: LDR R4,[R1,#4] ;多寄存器寻址
		28: E5914004 LDR R4,[R1,#0x0004]
		29: LDmia R1,{R5,R6} ;相对寻址
		30: E8910060 LDmia R1,{R5-R6}
		31: B NEXT ;B NEXT
		32: EA000005 B 0x00000005
		33: NOP
		34: E1A00000 NOP
		35: NOP

Address: 0x00000010	
0x00000010: 00 00 00 07 E3 A0 10 10 E5 81 20 00 E1 01 00 90 E3 A0 30 14 E3 A0	
0x00000026: 00 03 E5 83 00 00 E5 91 40 04 E8 91 00 60 EA 00 00 05 E1 A0 00 00	
0x0000003C: E1 A0 00 00 E1 A0 00 00 E1 A0 00 00 E1 A0 00 00 E1 A0 00 00 E1 A0	
0x00000052: 00 00 E1 A0 00 00 E1 A0 00 00 00 00 00 00 00 00 00 00 00 00 00	
0x00000068: 00	
0x0000007E: 00	
0x00000094: 00	
0x000000AA: 00	
0x000000C0: 00	
0x000000D6: 00	
0x000000EC: 00	
0x00000102: 00	
0x00000118: 00	
0x0000012E: 00	
0x00000144: 00	

Register	Value	
Current		
R0	0x00000003	20: MOV R3,#0x14 ;将立即数0x14存入寄存器R3
R1	0x00000010	21: E3A03014 MOV R3,#0x00000014 ;将立即数0x3存入寄存器R0
R2	0x00000007	22: MOV R0,#0x00000003 ;将寄存器R0的内容存入寄存器R3所指定的存储器
R3	0x00000014	23: E5830000 STR R0,[R3] ;将寄存器R1的内容加上4所指向的存储器的字存入寄存器R4
R4	0x00000000	24: LDR R4,[R1,#4] ;多寄存器寻址
R5	0x00000000	25: ;多寄存器寻址
R6	0x00000000	26: E5914004 LDR R4,[R1,#0x0004]
R7	0x00000000	27: LDmia R1,{R5,R6} ;相对寻址
R8	0x00000000	28: E8910060 LDmia R1,{R5-R6}
R9	0x00000000	29: B NEXT ;B NEXT
R10	0x00000000	30: EA000005 B 0x00000005
R11	0x00000000	31: NOP
R12	0x00000000	32: E1A00000 NOP
R13 (SP)	0x00000000	33: NOP
R14 (LR)	0x00000000	34: E1A00000 NOP
R15 (PC)	0x0000002C	35: NOP
CPSR	0x00000003	
SPSR	0x00000000	

Register	Value	
Current		
R0	0x00000005	7: E3A00001 MOV R0,#0x00000001 ;寄存器R0中的内容加上立即数0x4然后存入寄存器R0
R1	0x00000002	8: ADD R0,R0,#0x4 ;寄存器寻址
R2	0x00000011	9: E2800004 ADD R0,R0,#0x00000004 ;立即数0x2存入寄存器R1
R3	0x00000000	10: MOV R1,#0x2 ;立即数0x2存入寄存器R1
R4	0x00000000	11: E3A01002 MOV R1,#0x00000002 ;寄存器R0中的内容和寄存器R1的内容相加存入寄存器R2
R5	0x00000000	12: ADD R2,R1,R0 ;寄存器R0中的内容逻辑左移1位加上寄存器R2的内容存入寄存器R2
R6	0x00000000	13: E0812000 ADD R2,R1,R0 ;寄存器R0中的内容逻辑左移1位加上寄存器R2的内容存入寄存器R2
R7	0x00000000	14: ;寄存器间接寻址
R8	0x00000000	15: E022080 ADD R2,R2,R0,LSL #1 ;将立即数0x10存入寄存器R1
R9	0x00000000	16: MOV R1,#0x10 ;将立即数0x10存入寄存器R1
R10	0x00000000	17: E3A01010 MOV R1,#0x00000010 ;将寄存器R2的内容存入以寄存器R1的值为地址的存储器中
R11	0x00000000	18: STR R2,[R1] ;完成寄存器R1所指向的存储器中的字数据与寄存器R0的内容交换
R12	0x00000000	19: E5812000 STR R2,[R1]
R13 (SP)	0x00000000	20: SWP R0,R0,[R1]
R14 (LR)	0x00000000	
R15 (PC)	0x00000014	
CPSR	0x00000003	
SPSR	0x00000000	

1.4 掌握 ARM 各种逻辑控制语句结构

- ① 使用 ARM 汇编指令实现 if 条件执行
- ② 使用 ARM 汇编指令实现 for 循环结构
- ③ 使用 ARM 汇编指令实现 while 循环结构
- ④ 使用 ARM 汇编指令实现 do...while 循环结构

验证代码：

```
1      AREA Example4, CODE, READONLY ;声明代码段 Example
2      ENTRY                          ;标识程序入口
3      CODE32                         ;声明32位ARM指令
4  START
5          ;if(x>y) z=100;
6          ;else z=50;
7      MOV R0, #76                    ;初始化x的值
8      MOV R1, #243                   ;初始化y的值
9      CMP R0, R1                     ;判断 x>y?
10     MOVHI R2, #100                 ;x>y 条件正确, z=100
11     MOVLS R2, #50                  ;条件失败, z=50
12     ;for(i=0; i<10; i++)
13     ;{ x++;
14     ;}
15     ;设x为R0, i为R2 (i、x均为无符号整数)
16     MOV R0, #0                    ;初始化x的值
17     MOV R2, #0                    ;设置i=0
18  FOR_L1 CMP R2, #10                ;判断i<10?
19     BHS FOR_END                    ;R2>=10, 退出循环
20
21     ADD R0, R0, #1                ;循环体, X++
22     ADD R2, R2, #1                ;i++
23     B FOR_L1
24  FOR_END NOP
25
26
27     ;while (x<=y)
28     ; { x*=2;
29     ; }
30     ;设x为R0, y为R1 (x、y均为无符号整数)
31     MOV R0, #1                    ;初始化x的值
32     MOV R1, #20                   ;初始化y的值
33     B WHILE_L2                    ;首先要判断条件
34  WHILE_L1 MOV R0, R0, LSL#1        ;循环体, x*=2
35  WHILE_L2 CMP R0, R1              ;x<y?
36     BLS WHILE_L1                  ;若条件正确, 继续循环
37  WHILE_END NOP
38
39     ;do
40     ; { x-;
41     ; } while (x>0);
42     ;设x为R0 (x为无符号整数)
43     MOV R0, #5                    ;初始化x的值
44  DOWHILE_L1 ADD R0, R0, #-1        ;循环体, x--
45  DOWHILE_L2 MOVS R0, R0            ;R0 <= R0, 并影响条件码标志
46     BNE DOWHILE_L1                ;若R0不为0 (即x不为0), 则继续循环
47  DOWHILE_END NOP
48
49     END
```

CMP: 比较指令使用寄存器 Rn 的值减值 operand2 的值, 根据操作的结果更新 CPSR 中的相应条件标志位, 以便后面的指令根据相应的条件标志来判断是否执行. 指令格式

CMP {cond} Rn, operand2

CMP 指令举例如下:

CMP R1, #10 ;R1 与 10 比较, 设置相关标志位

CMP R1, R2 ;R1 与 R2 比较, 设置相关标志位

CMP 指令与 SUBS 指令的区别在于 CMP 指令不保存运算结果. 在进行两个数据大小判断时, 常用 CMP 指令及相应的条件码来操作.

BHS: 配合条件码的转跳指令, 需要和状态寄存器配合
BNE: 配合条件码的转跳指令, 需要和状态寄存器配合

指令条件码列表

条件码助记符	标志	含义
EQ	Z=1	相等
NE	Z=0	不相等
CS/HS	C=1	无符号数大于或等于
CC/LO	C=0	无符号数小于
MI	N=1	负数
PL	N=0	正数或零
VS	V=1	溢出
VC	V=0	没有溢出
HI	C=1, Z=0	无符号数大于
LS	C=0, Z=1	无符号数小于或等于
GE	N=V	带符号数大于或等于
LT	N!=V	带符号数小于
GT	Z=0, N=V	带符号数大于
LE	Z=1, N!=V	带符号数小于或等于
AL	任何	无条件执行 (指令默认条件)

CPSR 和 SPSR 分配图

CPSR 或 SPSR 模式常量定义:

CPSR_f 或 SPSR_f								CPSR_s 或 SPSR_s								CPSR_x 或 SPSR_x								CPSR_c 或 SPSR_c							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
N	Z	C	V	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	I	F	T	M4	M3	M2	M1	M0
负数或小于	零	进位或借位或扩展	溢出	保留位																		IRQ 禁止 = 禁止, 0 = 允许	FIQ 禁止 = 禁止, 0 = 允许	状态位 0-ARM, 1-Thumb	模式位 10000-0x0010-用户 10001-0x0011-快速中断 10010-0x0012-中断 10011-0x0013 管理 10111-0x0017-未定义 11111-0x001F-系统						

打上断点, 调试仿真

Register	Value
Current	
R0	0x0000004C
R1	0x000000F3
R2	0x00000032
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000014
CPSR	0x800000D3
SPSR	0x00000000

0x00000000	E3A0004C	MOV	R0,#0x0000004C	
8:			MOV R1,#243	;初始化y的值
0x00000004	E3A010F3	MOV	R1,#0x000000F3	
9:			CMP R0,R1	;判断 x>y?
0x00000008	E1500001	CMP	R0,R1	
10:			MOVHI R2,#100	;x>y 条件正确, z=100
0x0000000C	83A02064	MOVHI	R2,#0x00000064	
11:			MOVL R2,#50	;条件失败, z=50
12:			;for(i=0; i<10; i++)	
13:			{ x++;	
14:			};	
15:			;设x为R0, i为R2 (i、x均为无符号整数)	
0x00000010	93A02032	MOVL	R2,#0x00000032	
16:			MOV R0,#0	;初始化x的值
0x00000014	E3A00000	MOV	R0,#0x00000000	
17:			MOV R2,#0	;设置i=0
0x00000018	E3A02000	MOV	R2,#0x00000000	
18: FOR_L1	CMP R2,#10			;判断i<10?
21:			ADD R0,R0,#1	;循环体, X++
0x00000024	E2800001	ADD	R0,R0,#0x00000001	
22:			ADD R2,R2,#1	;i++
0x00000028	E2822001	ADD	R2,R2,#0x00000001	
23:			B FOR_L1	
0x0000002C	EAF00000	B	0x0000001C	
24: FOR_END	NOP			
25:				
26:				
27:				;while (x<=y)
28:			{ x*=2;	
29:			};	
30:			;设x为R0, y为R1 (x、y均为无符号整数)	
0x00000030	E1A00000	NOP		
31:			MOV R0,#1	;初始化x的值
0x00000034	E3A00001	MOV	R0,#0x00000001	
32:			MOV R1,#20	;初始化y的值
0x00000038	E3A01014	MOV	R1,#0x00000014	
44: DOWHILE_L1	ADD R0,R0,#-1			;循环体, x--
0x00000054	E2400001	SUB	R0,R0,#0x00000001	
45: DOWHILE_L2	MOVS R0,R0			;R0 <= R0,并影响条件码标志
0x00000058	E1B00000	MOVS	R0,R0	
46:			BNE DOWHILE_L1	;若R0不为0 (即x不为0), 则继续循环
0x0000005C	1AFF0000	BNE	0x00000054	
47: DOWHILE_END	NOP			
0x00000060	E1A00000	NOP		
0x00000064	00000000	ANDEQ	R0,R0,R0	
0x00000068	00000000	ANDEQ	R0,R0,R0	
0x0000006C	00000000	ANDEQ	R0,R0,R0	
0x00000070	00000000	ANDEQ	R0,R0,R0	
0x00000074	00000000	ANDEQ	R0,R0,R0	
0x00000078	00000000	ANDEQ	R0,R0,R0	
0x0000007C	00000000	ANDEQ	R0,R0,R0	
0x00000080	00000000	ANDEQ	R0,R0,R0	
0x00000084	00000000	ANDEQ	R0,R0,R0	
0x00000088	00000000	ANDEQ	R0,R0,R0	

Register	Value
Current	
R0	0x00000000
R1	0x00000014
R2	0x0000000A
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000060
CPSR	0x600000D3
SPSR	0x00000000

44: DOWHILE_L1	ADD R0,R0,#-1			;循环体, x--
0x00000054	E2400001	SUB	R0,R0,#0x00000001	
45: DOWHILE_L2	MOVS R0,R0			;R0 <= R0,并影响条件码标志
0x00000058	E1B00000	MOVS	R0,R0	
46:			BNE DOWHILE_L1	;若R0不为0 (即x不为0), 则继续循环
0x0000005C	1AFF0000	BNE	0x00000054	
47: DOWHILE_END	NOP			
0x00000060	E1A00000	NOP		
0x00000064	00000000	ANDEQ	R0,R0,R0	
0x00000068	00000000	ANDEQ	R0,R0,R0	
0x0000006C	00000000	ANDEQ	R0,R0,R0	
0x00000070	00000000	ANDEQ	R0,R0,R0	
0x00000074	00000000	ANDEQ	R0,R0,R0	
0x00000078	00000000	ANDEQ	R0,R0,R0	
0x0000007C	00000000	ANDEQ	R0,R0,R0	
0x00000080	00000000	ANDEQ	R0,R0,R0	
0x00000084	00000000	ANDEQ	R0,R0,R0	
0x00000088	00000000	ANDEQ	R0,R0,R0	

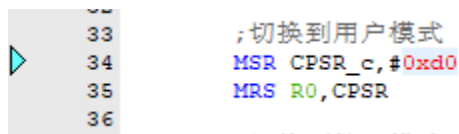
1.5 ARM 汇编编程-模式切换

① 掌握 ARM 7 种工作模式及切换方法使用 MRS/MSR 指令切换工作模式, 并初始化各种模式下堆栈针; 观察 ARM 处理器在各种模式下寄存器的区别。

验证代码:

编译出错

. \Objects\ARM7.axf: Error: L6221E: Execution region ER_R0 with Execution range [0x00000000,0x000000a8) overlaps with Execution region ER_ZI with Execution range [0x00000000,0x00000240).



```
33 ;切换到用户模式
34 MSR CPSR_c, #0xd0
35 MRS R0, CPSR
36
```

原因暂未知, 应该是有未声明的变量

```

1 ;定义堆栈的大小
2 USR_STACK_LEGTH EQU 64
3 SVC_STACK_LEGTH EQU 0
4 FIQ_STACK_LEGTH EQU 16
5 IRQ_STACK_LEGTH EQU 64
6 ABT_STACK_LEGTH EQU 0
7 UND_STACK_LEGTH EQU 0
8
9     AREA Example5, CODE, READONLY ;声明代码段 Example5
10    ENTRY                          ;标识程序入口
11    CODE32                          ;声明32位ARM指令
12 START
13    MOV R0, #0
14    MOV R1, #1
15    MOV R2, #2
16    MOV R3, #3
17    MOV R4, #4
18    MOV R5, #5
19    MOV R6, #6
20    MOV R7, #7
21    MOV R8, #8
22    MOV R9, #9
23    MOV R10, #10
24    MOV R11, #11
25    MOV R12, #12
26    BL InitStack;初始化各模式下的堆栈指针
27
28    ;打开IRQ中断(将CPSR寄存器的I位清零)
29    MRS R0, CPSR                ;RO <= CPSR
30    BIC R0, R0, #0x80
31    MSR CPSR_cxsf, R0          ;CPSR <= R0
32
33    ;切换到用户模式
34    MSR CPSR_c, #0xd0
35    MRS R0, CPSR
36
37    ;切换到管理模式
38    MSR CPSR_c, #0xdf
39    MRS R0, CPSR
40
41 HALT    B HALT
42 ;名称: InitStack
43 ;功能: 堆栈初始化, 即初始化各模式下的堆栈指针。
44 ;入口参数: 无
45 ;出口参数: 无
46 ;说明: 在特权模式下调用此子程序, 比如复位后的管理模式
47 InitStack
48    MOV R0, LR                ;RO<=LR, 因为各种模式下R0是相同的
49    ;设置管理模式堆栈
50    MSR CPSR_c, #0xd3
51    LDR SP, StackSvc
52    ;设置中断模式堆栈
53    MSR CPSR_c, #0xd2
54    LDR SP, StackIrq
55    ;设置快速中断模式堆栈
56    MSR CPSR_c, #0xd1
57    LDR SP, StackFiq
58    ;设置中止模式堆栈
59    MSR CPSR_c, #0xd7
60    LDR SP, StackAbt
61    ;设置未定义模式堆栈
62    MSR CPSR_c, #0xdb
63    LDR SP, StackUnd
64    ;设置系统模式堆栈
65    MSR CPSR_c, #0xdf
66    LDR SP, StackUsr
67    MOV PC, R0
68 StackUsr DCD UsrStackSize + (USR_STACK_LEGTH - 1) * 4
69 StackSvc DCD SvcStackSize + (SVC_STACK_LEGTH - 1) * 4
70 StackIrq DCD IrqStackSize + (IRQ_STACK_LEGTH - 1) * 4
71 StackFiq DCD FiqStackSize + (FIQ_STACK_LEGTH - 1) * 4
72 StackAbt DCD AbtStackSize + (ABT_STACK_LEGTH - 1) * 4
73 StackUnd DCD UndtStackSize + (UND_STACK_LEGTH - 1) * 4
74 ;分配堆栈空间
75    AREA MyStacks, DATA, NOINIT, ALIGN=2
76 UsrStackSize SPACE USR_STACK_LEGTH * 4 ;用户(系统)模式堆栈空间
77 SvcStackSize SPACE SVC_STACK_LEGTH * 4 ;管理模式堆栈空间
78 IrqStackSize SPACE IRQ_STACK_LEGTH * 4 ;中断模式堆栈空间
79 FiqStackSize SPACE FIQ_STACK_LEGTH * 4 ;快速中断模式堆栈空间
80 AbtStackSize SPACE ABT_STACK_LEGTH * 4 ;中止模式堆栈空间
81 UndtStackSize SPACE UND_STACK_LEGTH * 4 ;未定义模式堆栈
82
83    END

```


BL: 转跳指令

带链接的跳转指令. 指令将下一条指令的地址拷贝到 R14(即 LR)链接寄存器中, 然后跳转到指定地址运行程序. 指令格式如下:

BL{cond} label

带链接的跳转指令 BL 举例如下:

BL DELAY

跳转指令 B 限制在当前指令的±32MB 的范围内. BL 指令用于子程序调用

BIC: 位带清除指令

位清除指令. 将寄存器 Rn 的值与 operand2 的值的反码按位作逻辑与操作, 结果保存到 Rd 中. 指令格式如下:

BIC{cond} {S} Rd, Rn, operand2

BIC 指令举例如下:

BIC R1, R1, #0x0F ;将 R1 的低 4 位清零, 其它位不变

BIC R1, R2, R3 ;将 R1 的反码和 R2 相逻辑与, 结果保存到 R1

DCD:

DCD 用于分配一段字内存单元, 并用伪指令中的 expr 初始化. DCD 伪指令分配的内存需要字对齐, 一般可用来定义数据表格或其它常数. & 与 DCD 同义.

DCDU 用于分配一段字内存单元, 并用伪指令中的 expr 初始化. DCD 伪指令分配的内存不需要字对齐, 一般可用来定义数据表格或其它常数.

伪指令格式:

{label} DCD expr {, expr} {, expr} ...

{label} DCDU expr {, expr} {, expr} ...

其中 label 内存块起始地址标号.

expr 常数表达式或程序中的标号. 内存分配字节数由 expr 个数决定.

伪指令应用举例如下:

Vectors

LDR PC, ReserAddr

LDR PC, UndefinedAddr

...

ResetAddr DCD Reset

UndefinedAddr DCD Undefined

MSR: 写状态寄存器

写状态寄存器指令. 在 ARM 处理器中, 只有 MSR 指令可以直接设置状态寄存器 CPSR 或 SPSR. 指令格式如下

MSR{cond} psr_fields, #immed_8r

MSR{cond} psr_fields, Rm

其中: psr CPSR 或 SPSR

fields 指定传送的区域. Fields 可以是以下的一种或多种 (字母必须为小写):

c 控制域屏蔽字节 (psr[7...0])

x 扩展域屏蔽字节 (psr[15...8])

s 状态域屏蔽字节 (psr[23...16])

f 标志域屏蔽字节 (psr[31...24])

immed_8r 要传送到状态寄存器指定域的立即数, 8 位.

Rm 要传送到状态寄存器指定域的数据的源寄存器.

MSR 指令举例如下

MSR CPSR_c, #0xD3 ;CPSR[7...0]=0xD3, 即切换到管理模式.

MSR CPSR_cxsf, R3 ;CPSR=R3

只有在特权模式下才能修改状态寄存器.

程序中不能通过 MSR 指令直接修改 CPSR 中的 T 控制位来实现 ARM 状态/Thumb 状态的切换, 必须使用 BX 指令完成处理器状态的切换 (因为 BX 指令属转移指令, 它会打断流水线状态, 实现处理器状态切换). MRS 与 MSR 配合使用, 实现 CPSR 或 SPSR 寄存器的读---修改---写操作, 可用来进行处理器模式切换、允许/禁止 IRQ/FIQ 中断等设置, .

堆栈指令实初始化

INITSTACK

MOV R0, LR ;保存返回地址

;设置管理模式堆栈

MSR CPSR_c, #0xD3

LDR SP, StackSvc

;设置中断模式堆栈

MSR CPSR_c, #0xD2

LDR SP, StackIrq

...

LDR: 加载

为大范围的地址读取伪指令. LDR 伪指令用于加载 32 位的立即数或一个地址值到指定寄存器. 若汇编器将常量放入文字池, 并使用一条程序相对偏移的 LDR 指令从文字池读出常量, 则从 PC 到文字池的偏移量必须小于 4KB.

AREA: 定义

AREA 伪指令用于定义一个代码段或数据段. ARM 汇编程序设计采用分段式设计, 一个 ARM 源程序至少需要一个代码段, 大的程序可以包含多少个代码段及数据段.

伪指令格式:

AREA **sectionname** {, **attr**} {, **attr**}...

其中 **sectionname** 所定义的代码段或数据段的名称. 如果该名称是以数据开头的, 则该名称必须用“|”括起来, 如|l_datasec|. 还有一些代码段具有的约定的名称. 如|text|表示 C 语言编译器产生的代码段或者与 C 语言库相关的代码段.

attr 该代码段或数据段的属性.

在 AREA 伪指令中, 各属性之间用逗号隔开. 以下为段属性及相关说明:

ALIGN = expr. 默认的情况下, ELF 的代码段和数据段是 4 字节对齐的, **expr** 可以取 0~31 的数值, 相应的对齐方为 2**expr** 字节对齐. 如 **expr**=3 时为字节对齐. 对于代码段, **expr** 不能为 0 或 1;

ASSOC = section. 指定与本段相关的 ELF 段. 任何时候连接 **section** 段也必须包括 **sectionname** 段;

DODE 为定义代码段. 默认属性为 READONLY;

COMDEF 定义一个通用的段. 该段可以包含代码或者数据. 在其它源文件中, 同名的 **COMDEF** 段必须相同;

COMMON 定义一个通用的段. 该段不包含任何用户代码和数据, 连接器将其初始化为 0. 各源文件中同名的 **COMMON** 段共用同样的内存单元, 连接器为其分配合适的尺寸;

DATA 为定义段. 默认属性为 READWRITE;

NOINIT 指定本数据段仅仅保留了内存单元, 而没有将各初始写入内存单元, 或者将内存单元值初始化为 0;

READONLY 指定本段为只读, 代码段的默认属性为 READONLY;

READWRITE 指定本段为可读可写. 数据段的默认属性为 READWRITE;

使用 AREA 伪指令将程序分为多个 ELF 格式的段, 段名称可以相同, 这时同名的段被放在同一个 ELF 段中.

伪指令应用举例如下;

AREA **Example** , CODE, READNOLY ;声明一个代码, 名为 **Example**

SPACE: 分配地址空间

SPACE 用于分配一块内存单元, 并用 0 初始化. %与 SPACE 同义.

伪指令格式:

{label} SPACE expr

其中 label 内存块起始地址标号.

expr 所要分配的内存字节数.

伪指令应用举例如下:

```
                AREA    DataRA, DATA, READWRITE    ;声明一数据段, 名为 DataRAM
DataBuf SPACE    1000                                ;分配 1000 字节空间
```

七种工作状态:

- 1、用户模式(Usr): 用于正常执行程序;
- 2、快速中断模式(FIQ): 用于高速数据传输;
- 3、外部中断模式(IRQ): 用于通常的中断处理;
- 4、管理模式(svc): 操作系统使用的保护模式;
- 5、数据访问终止模式(abt): 当数据或指令预取终止时进入该模式, 可用于虚拟存储以及存储保护;
- 6、系统模式(sys): 运行具有特权的操作系统任务;
- 7、未定义指令中止模式(und): 当未定义的指令执行时进入该模式, 可用于支持硬件;

Arm 的工作模式切换有两种方法:

被动切换: 在 arm 运行的时候产生一些异常或者中断来自动进行模式切换

主动切换: 通过软件改变, 即软件设置寄存器来经行 arm 的模式切换, arm 的工作模式都是可以通过相应寄存器的赋值来切换的。

当处理器运行在用户模式下, 某些被保护的系统资源是不能被访问的。

除用户模式外, 其余 6 种工作模式都属于特权模式; 特权模式中除了系统模式以外的其余 5 种模式称为异常模式; 大多数程序运行于用户模式; 进入特权模式是为了处理中断、异常、或者访问被保护的系统资源;

1.6 ARM 汇编编程-模式切换

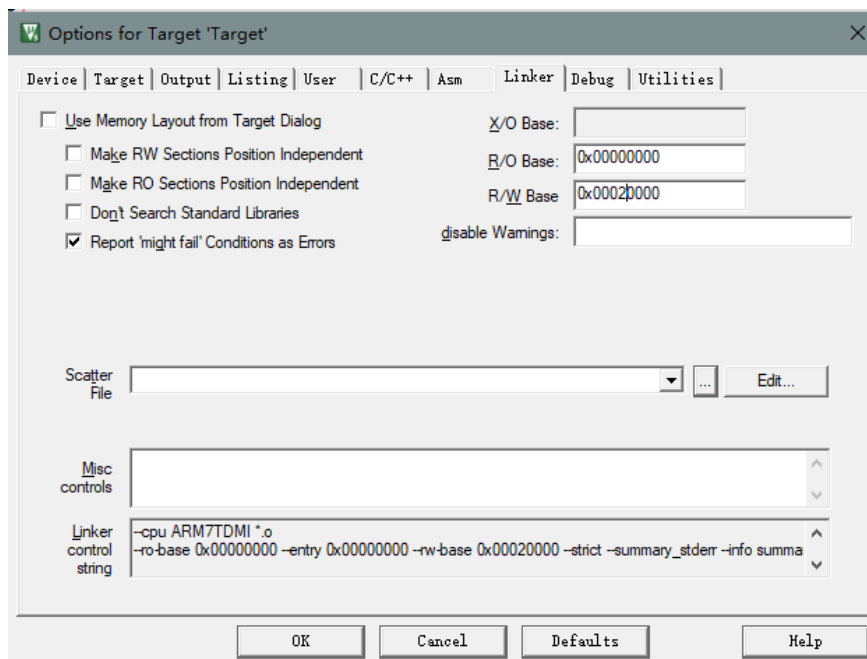
① 掌握 ARM 汇编指令与 C 语言的混合编程，编写一个汇编程序文件 Startup.S 和一个 C 程序文件 Test.c。汇编程序的功能是初始化堆栈指针和初始化 C 程序的运行环境，然后调跳转到 C 程序运行，这就是一个简单的启动程序。C 程序使用加法运算来计算 $1+2+3+4+\dots+N$

在工程中添加并创建 start.s main.c 并添加代码如下：

start.s

```
1      IMPORT |Image$$RO$$Limit| ;https://www.cnblogs.com/panfengyou/articles/11741259.html
2      IMPORT |Image$$RW$$Base| ;https://www.xuebuyuan.com/1716271.html
3      IMPORT |Image$$ZI$$Base|
4      IMPORT |Image$$ZI$$Limit|
5      IMPORT main ;声明c程序中的Main()函数
6      AREA Start, CODE, READONLY ;声明代码段 Start
7      PRESERVE8
8      ENTRY ;标识程序入口
9      CODE32 ;声明32位ARM指令
10 Reset LDR SP, =0x40003F00
11 ;初始化c程序的运行环境
12 LDR R0, =|Image$$RO$$Limit|
13 LDR R1, =|Image$$RW$$Base|
14 LDR R3, =|Image$$ZI$$Base|
15 CMP R0, R1
16 BEQ LOOP1
17 LOOP0 CMP R1, R3
18 LDRCC R2, [R0], #4
19 STRCC R2, [R1], #4
20 BCC LOOP0
21
22 LOOP1 LDR R1, =|Image$$ZI$$Limit|
23 MOV R2, #0
24 LOOP2 CMP R3, R1
25 STRCC R2, [R3], #4
26 BCC LOOP2
27
28 B main
29
30 END
```

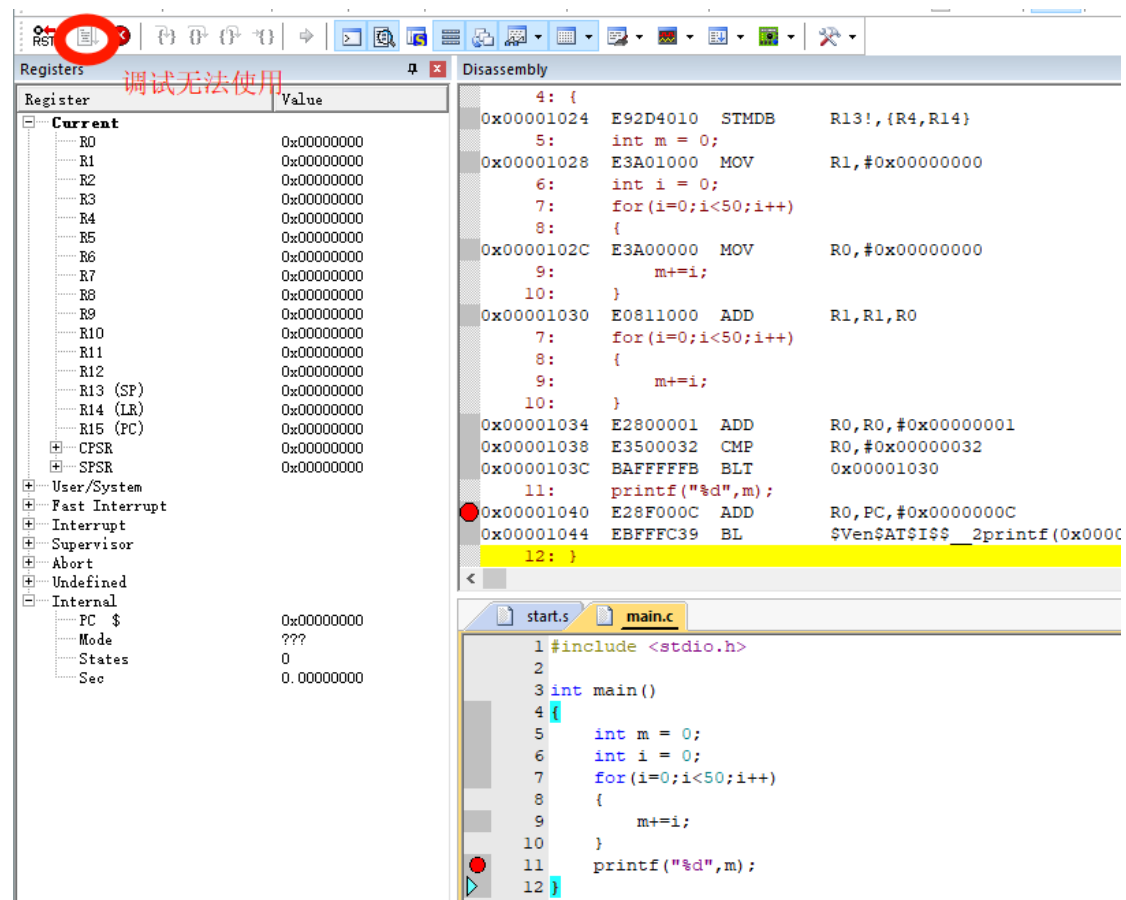
为 R0 标定起始地址



main.c

```
1 #include <stdio.h>
2
3 int main()
4 {
5     int m = 0;
6     int i = 0;
7     for(i=0;i<50;i++)
8     {
9         m+=i;
10    }
11    printf("%d",m);
12 }
```

为什么不能调试呢？



一个简易的映像文件包括以下几个部分：

- 一个只读（RO）区域；
- 一个读写（RW）区域；
- 一个被 0 初始化（ZI）的区域。

一个 ARM 由 RO，RW 和 ZI 三个段组成，其中 RO 为代码段，RW 是已初始化的全局变量，ZI 是未初始化的全局变量（与 TEXT，DATA 和 BSS 相对应）。

RO 段是只读的，在运行的时候不可以改变，所以，在运行的时候，RO 段可以驻留在 Flash 里。

RW 段是可以读写的，所以，在运行的时候必须被装载到 SDRAM 或者 SRAM 里，所以 Boot 要将 RW 段复制到 RAM 中，并将 ZI 段清零，以保证程序可以正确运行。

编译器使用下列符号来记录各段的起始和结束地址：

|Image\$\$RO\$\$Base |: RO 段起始地址
|Image\$\$RO\$\$Limit |: RO 段结束地址加 1
|Image\$\$RW\$\$Base |: RW 段起始地址
|Image\$\$RW\$\$Limit |: ZI 段结束地址加 1
|Image\$\$ZI\$\$Base |: ZI 段起始地址
|Image\$\$ZI\$\$Limit |: ZI 段结束地址加 1

启用调试会出现错误如下：在 [keil 官网](#)找到解决办法如下

```
*** Error: 'S:\development tools\keil\ARM\BIN\DARMO.DLL' not found
Load "D:\\code\\stm32\\ARM7_EMB\\Objects\\ARM7.axf"
BS \\ARM7\\START/main.c\5
WS 1, `COUNT
```

PROBLEM

I just installed the Keil Microcontroller Development Kit on my new computer. When I start the µVision Debugger the following error appears:

```
C:\Keil\ARM\BIN\SARM.DLL Not Found
```

The SARM.DLL file is in the C:\KEIL\ARM\BIN folder. What causes this error? How do I fix it?

CAUSE

You are actually missing a Microsoft file. Some Windows installations do not automatically include the file **MSVCR71.DLL** but SARM.DLL uses this DLL. When this file is missing, you get the above error.

RESOLUTION

Download the **MSVCR71.DLL** file from the Attached Files section below. Unzip and extract it to your C:\KEIL\UV3\ folder.

[下载](#)并解压到指定文件夹下，但不知道为什么仍然无法进入 startup.s 程序里卡在某个点无法运行

Disassembly

```

main:
0x00000000 EB000000 BL      __scatterload(0x00000008)
0x00000004 EB000024 BL      __rt_entry(0x0000009C)

__scatterload:
0x00000008 E28F002C ADD     R0,PC,#0x0000002C
0x0000000C E8900C00 LDMIA   R0,{R10-R11}
0x00000010 E08AA000 ADD     R10,R10,R0
0x00000014 E08BB000 ADD     R11,R11,R0
0x00000018 E24A7001 SUB     R7,R10,#0x00000001

__scatterload_main:
0x0000001C E15A000B CMP     R10,R11
0x00000020 1A000000 BNE     0x00000028
0x00000024 EB00001C BL      __rt_entry(0x0000009C)
0x00000028 E8BA000F LDMIA   R10!,{R0-R3}
0x0000002C E24FE018 SUB     R14,PC,#0x00000018
0x00000030 E3130001 TST     R3,#0x00000001
0x00000034 1047F003 SUBNE  PC,R7,R3
0x00000038 E12FFF13 BX      R3
0x0000003C 000001CC ANDEQ   R0,R0,R12,ASR #3
0x00000040 000001DC ???EQ

__scatterload_zeroinit:
0x00000044 E3B03000 MOVS    R3,#0x00000000
0x00000048 E3B04000 MOVS    R4,#0x00000000
0x0000004C E3B05000 MOVS    R5,#0x00000000

```

main.c
Startup.s

```

1      IMPORT |Image$$RO$$Limit| ;https://www.cnblogs.com/panfengyou/articles/11741259.html
2      IMPORT |Image$$RW$$Base| ;https://www.xuebuyuan.com/1716271.html
3      IMPORT |Image$$ZI$$Base|
4      IMPORT |Image$$ZI$$Limit|
5      IMPORT main ;声明c程序中的Main()函数
6      AREA Start,CODE,READONLY ;声明代码段 Start
7      PRESERVE8
8      ENTRY ;标识程序入口
9      CODE32 ;声明32位ARM指令
10     Reset LDR SP,=0x40003F00
11           ;初始化c程序的运行环境
12           LDR R0,=|Image$$RO$$Limit|
13           LDR R1,=|Image$$RW$$Base|
14           LDR R3,=|Image$$ZI$$Base|
15           CMP R0,R1
16           BEQ LOOP1
17     LOOP0 CMP R1,R3
18           LDRCC R2,[R0],#4
19           STRCC R2,[R1],#4
20           BCC LOOP0

```

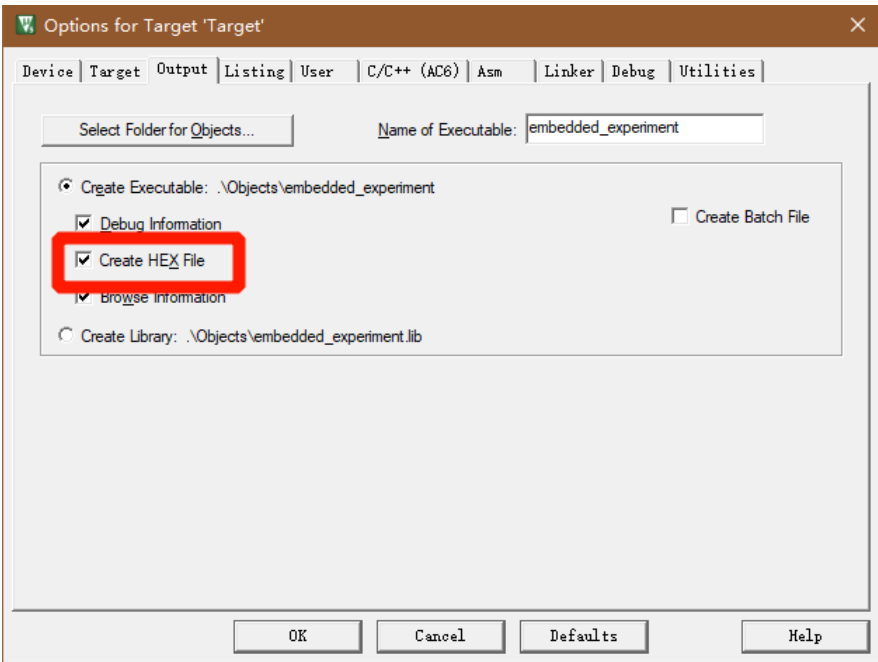

二. 仿真实验

1. 前置配置

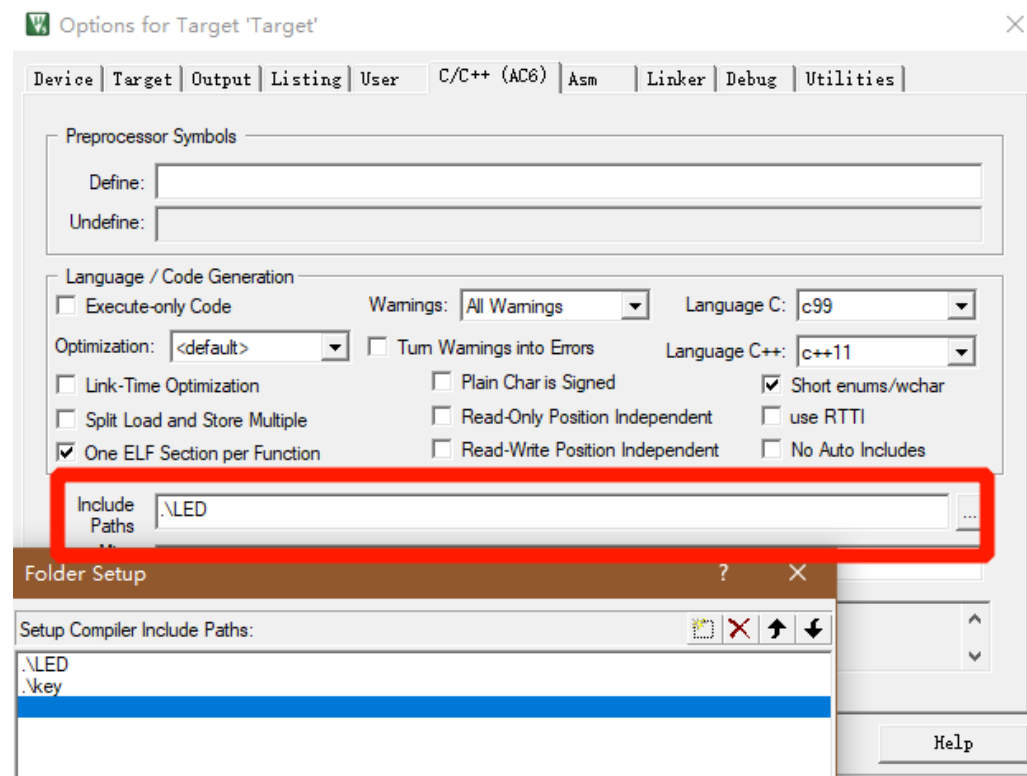
使用 KEIL 建立新项目并选择芯片 STM32R6，使用 RTE_DeVice.h 进行总控编写控件，在其中添加必要启动代码，系统文件等

CMSIS	<input checked="" type="checkbox"/>			Cortex Microcontroller Software Interface Components
CORE	<input checked="" type="checkbox"/>		5.4.0	CMSIS-CORE for Cortex-M, SC000, SC300, ARMv8-M, ARMv8.1-M
DSP	<input type="checkbox"/>	Source	1.8.0	CMSIS-DSP Library for Cortex-M, SC000, and SC300
NN Lib	<input type="checkbox"/>		1.3.0	CMSIS-NN Neural Network Library
RTOS (API)	<input type="checkbox"/>		1.0.0	CMSIS-RTOS API for Cortex-M, SC000, and SC300
RTOS2 (API)	<input type="checkbox"/>		2.1.3	CMSIS-RTOS API for Cortex-M, SC000, and SC300
CMSIS Driver	<input type="checkbox"/>			Unified Device Drivers compliant to CMSIS-Driver Specifications
Device	<input checked="" type="checkbox"/>			Startup, System Setup
DMA	<input checked="" type="checkbox"/>		1.2	DMA driver used by RTE Drivers for STM32F1 Series
GPIO	<input checked="" type="checkbox"/>		1.3	GPIO driver used by RTE Drivers for STM32F1 Series
Startup	<input checked="" type="checkbox"/>		1.0.0	System Startup for STMicroelectronics STM32F1xx device series
StdPeriph Drivers	<input type="checkbox"/>			
ADC	<input type="checkbox"/>		3.5.0	Analog-to-digital converter (ADC) driver for STM32F10x
BKP	<input type="checkbox"/>		3.5.0	Backup registers (BKP) driver for STM32F10x
CAN	<input type="checkbox"/>		3.5.0	Controller area network (CAN) driver for STM32F1xx
CEC	<input type="checkbox"/>		3.5.0	Consumer electronics control controller (CEC) driver for STM32F1xx
CRC	<input type="checkbox"/>		3.5.0	CRC calculation unit (CRC) driver for STM32F1xx
DAC	<input type="checkbox"/>		3.5.0	Digital-to-analog converter (DAC) driver for STM32F1xx
DBGMCU	<input type="checkbox"/>		3.5.0	MCU debug component (DBGMCU) driver for STM32F1xx
DMA	<input type="checkbox"/>		3.5.0	DMA controller (DMA) driver for STM32F1xx
EXTI	<input checked="" type="checkbox"/>		3.5.0	External interrupt/event controller (EXTI) driver for STM32F1xx
FSMC	<input type="checkbox"/>		3.5.0	Flexible Static Memory Controller (FSMC) driver for STM32F11x
Flash	<input type="checkbox"/>		3.5.0	Embedded Flash memory driver for STM32F1xx
Framework	<input checked="" type="checkbox"/>		3.5.1	Standard Peripherals Drivers Framework
GPIO	<input type="checkbox"/>		3.5.0	General-purpose I/O (GPIO) driver for STM32F1xx
I2C	<input type="checkbox"/>		3.5.0	Inter-integrated circuit (I2C) interface driver for STM32F1xx
IWDG	<input type="checkbox"/>		3.5.0	Independent watchdog (IWDG) driver for STM32F1xx
PWR	<input type="checkbox"/>		3.5.0	Power controller (PWR) driver for STM32F1xx
RCC	<input checked="" type="checkbox"/>		3.5.0	Reset and clock control (RCC) driver for STM32F1xx
RTC	<input type="checkbox"/>		3.5.0	Real-time clock (RTC) driver for STM32F1xx
SDIO	<input type="checkbox"/>		3.5.0	Secure digital (SDIO) interface driver for STM32F1xx
SPI	<input type="checkbox"/>		3.5.0	Serial peripheral interface (SPI) driver for STM32F1xx
TIM	<input checked="" type="checkbox"/>		3.5.0	Timers (TIM) driver for STM32F1xx
USART	<input checked="" type="checkbox"/>		3.5.0	Universal synchronous asynchronous receiver transmitter (USART) driver for STM32F1xx
WWDG	<input type="checkbox"/>		3.5.0	Window watchdog (WWDG) driver for STM32F1xx

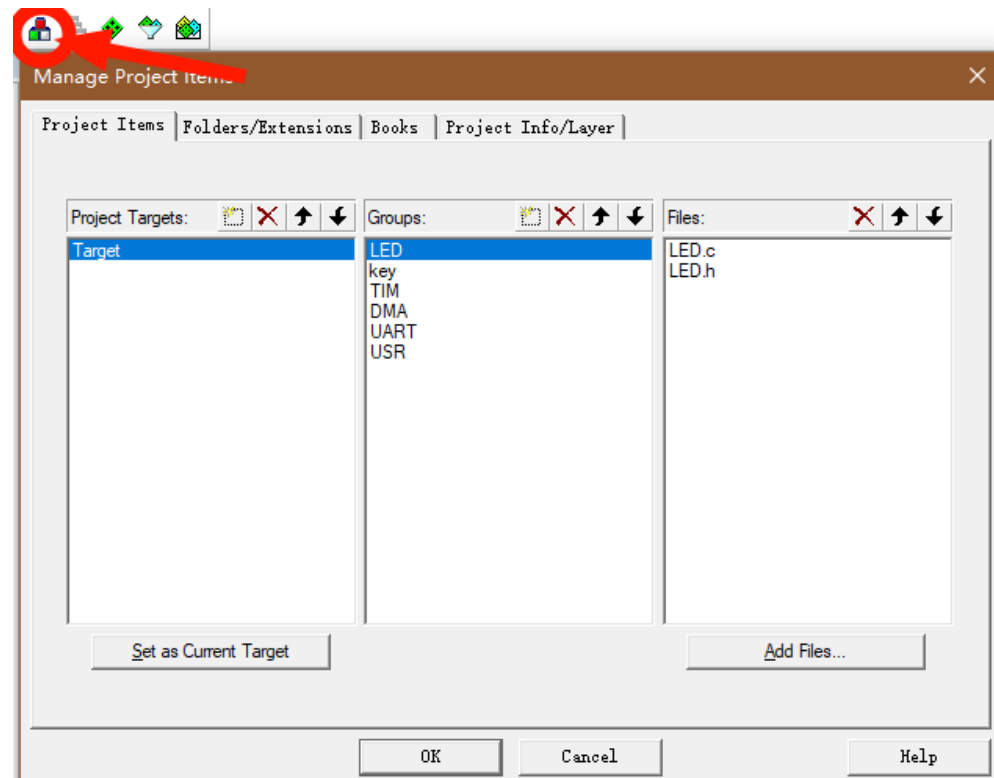
设置输出 HEX 文件



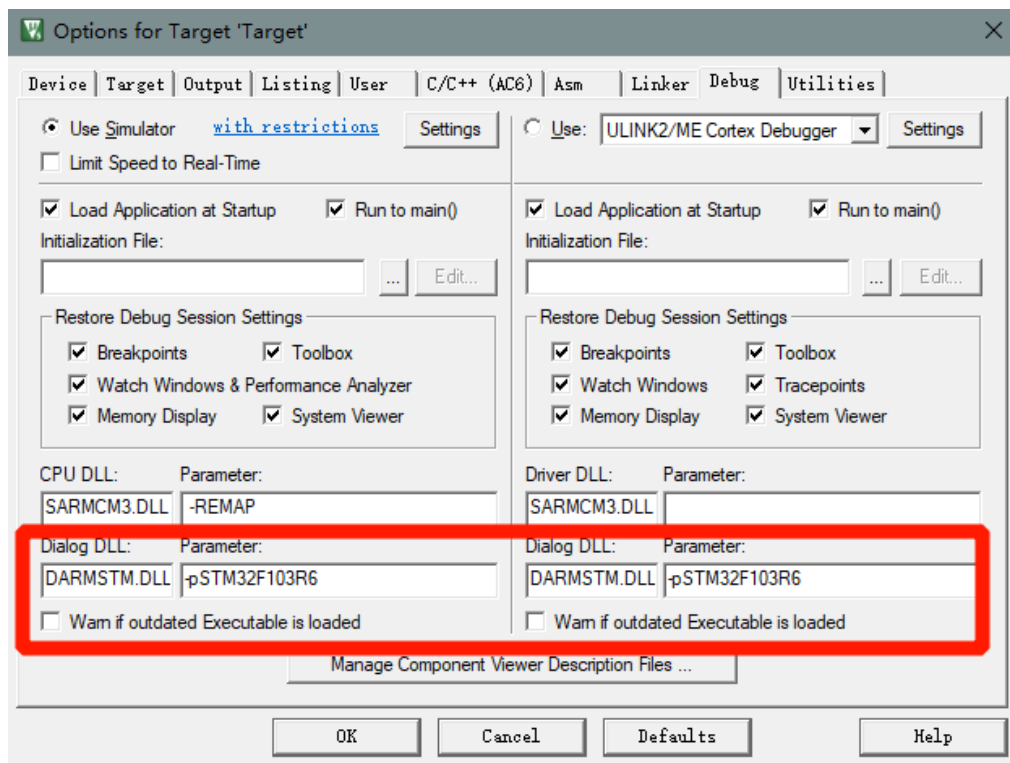
在 include_path 设置中添加.h 头文件的路径



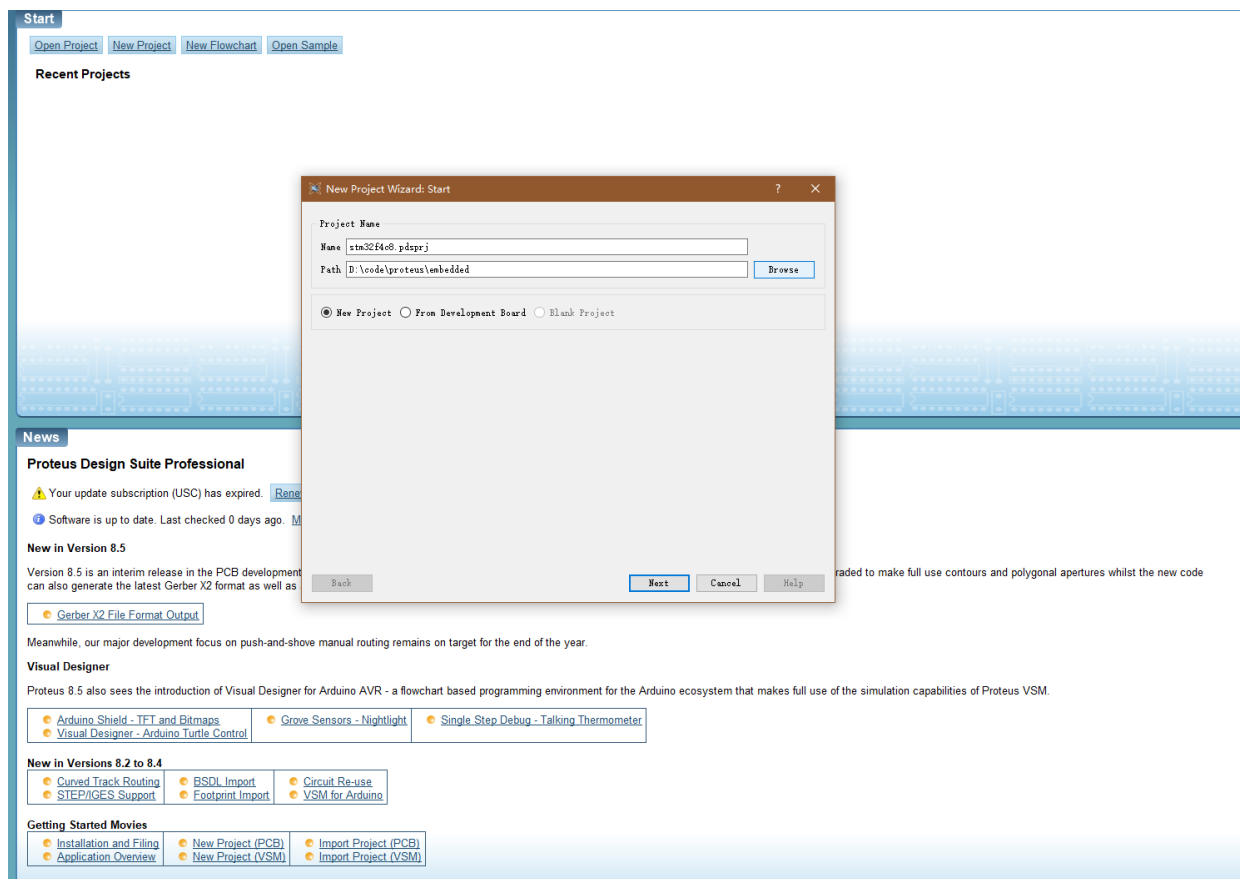
将源文件导入到工程中去



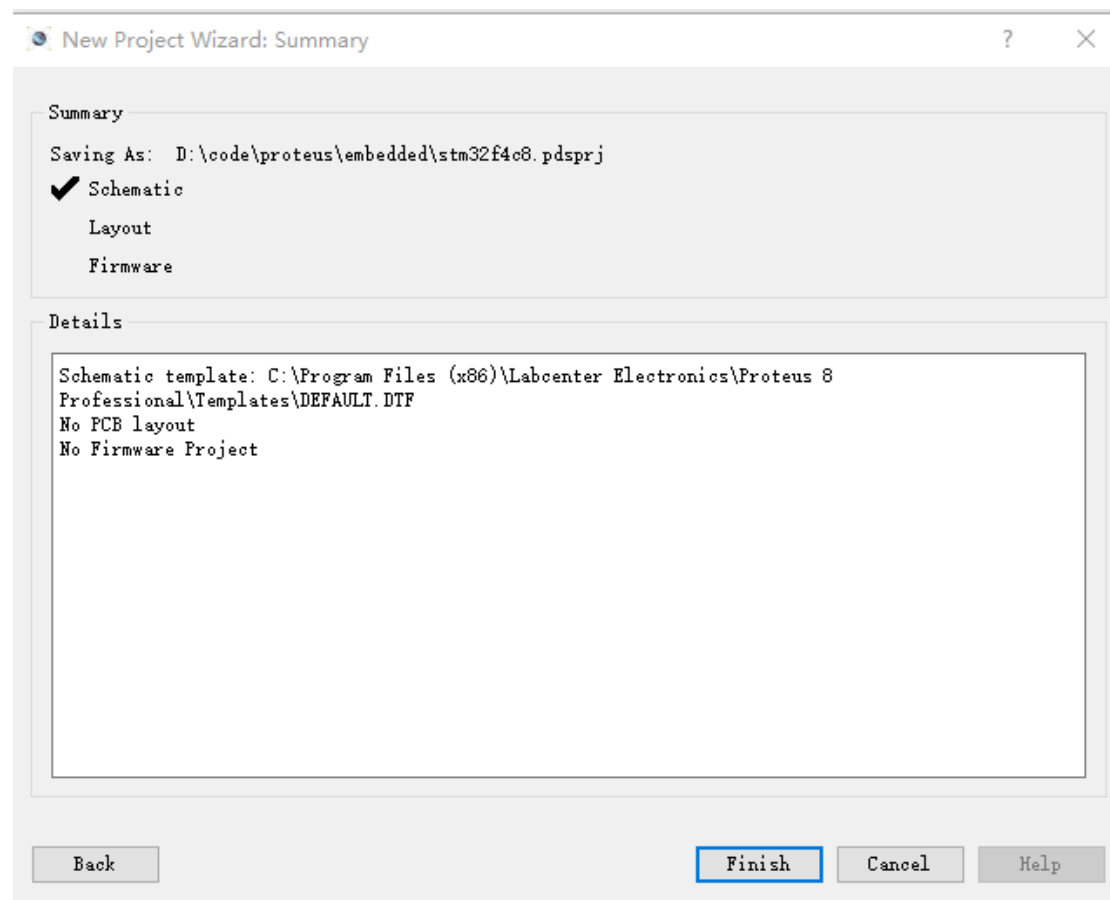
设置仿真



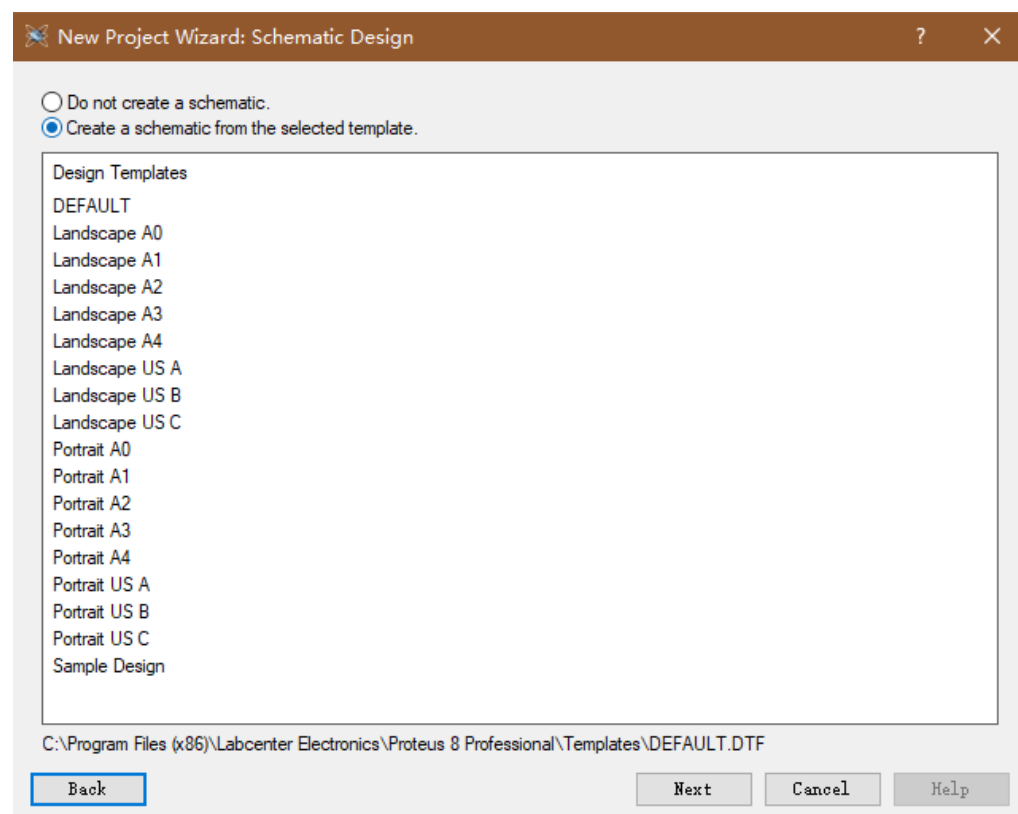
安装 Protues8.9 安装时要注意不要选择继承前版本设置(可能会导致出现库安装不完全的现象) 并建立 protues 工程,使用 Cortex3 内核仿真



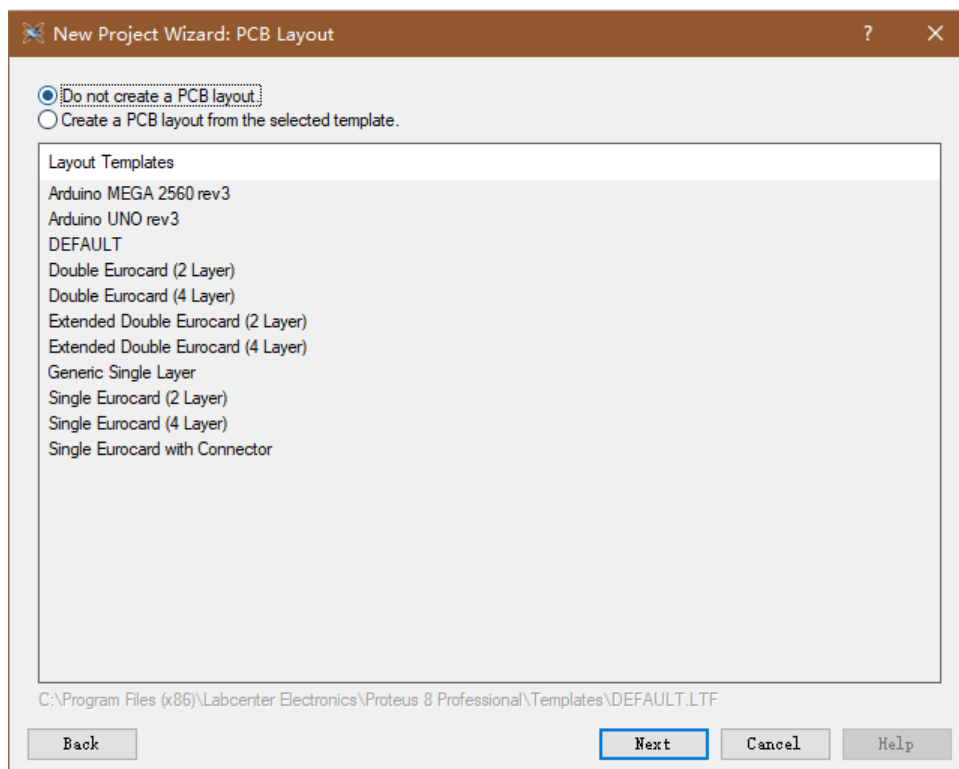
创建原理图



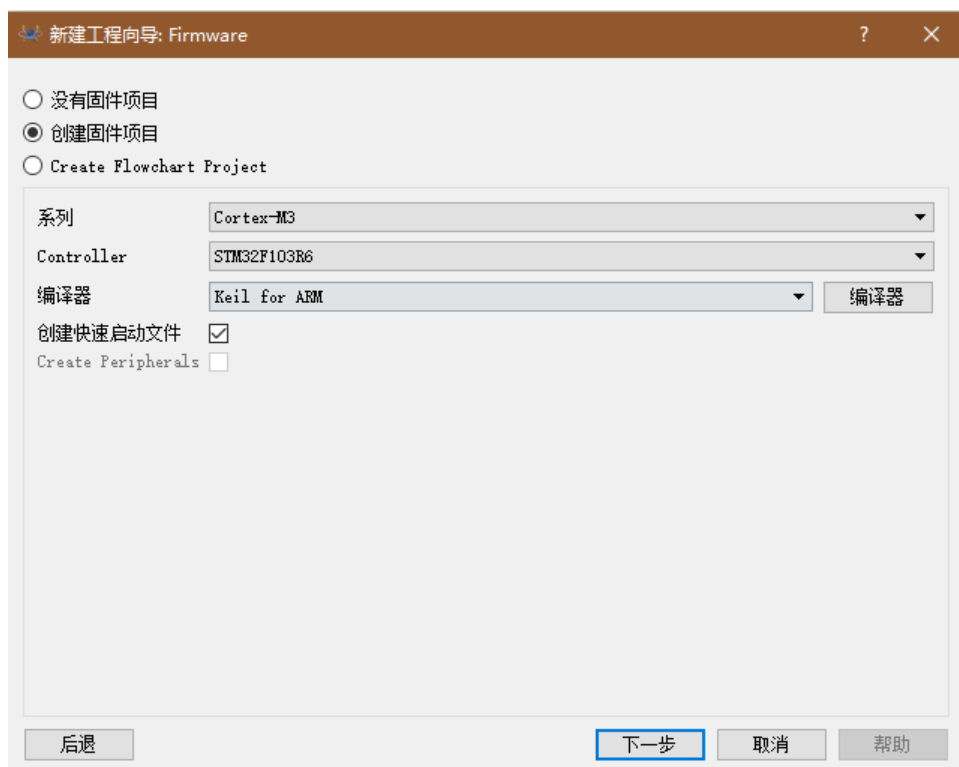
创建模板



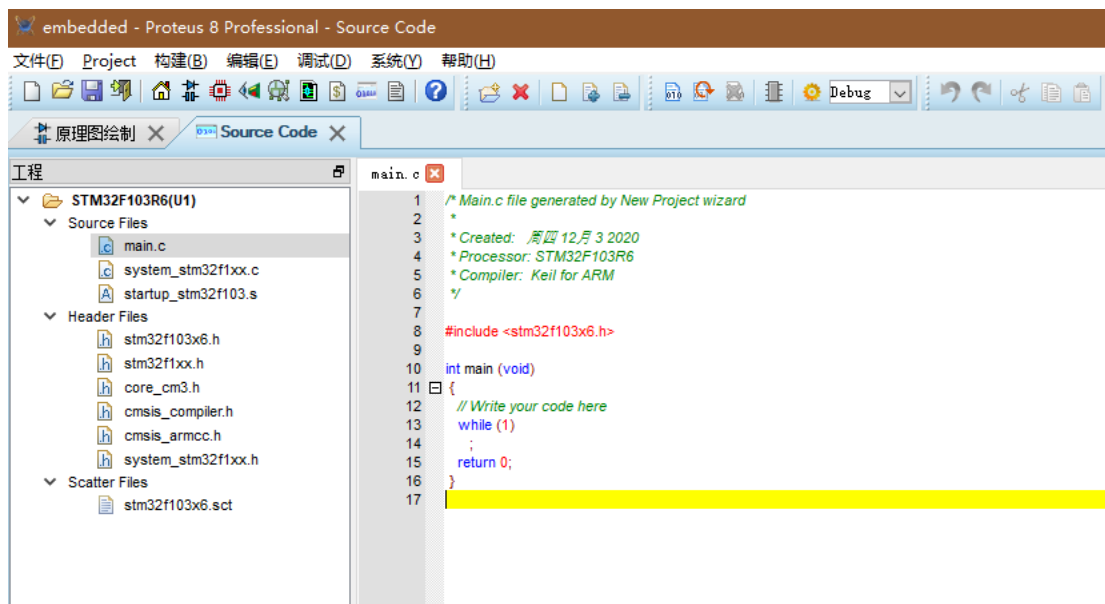
不创建 PCB 板



选择 STM32R6



如果创建时选择了创建快速启动文件，那么将会自动添加启动代码和初始工程，效果如下，可以在做简单实验时使用，若自身本来有在 keil 上编写代码，那么则不需要添加，一般比较鸡肋



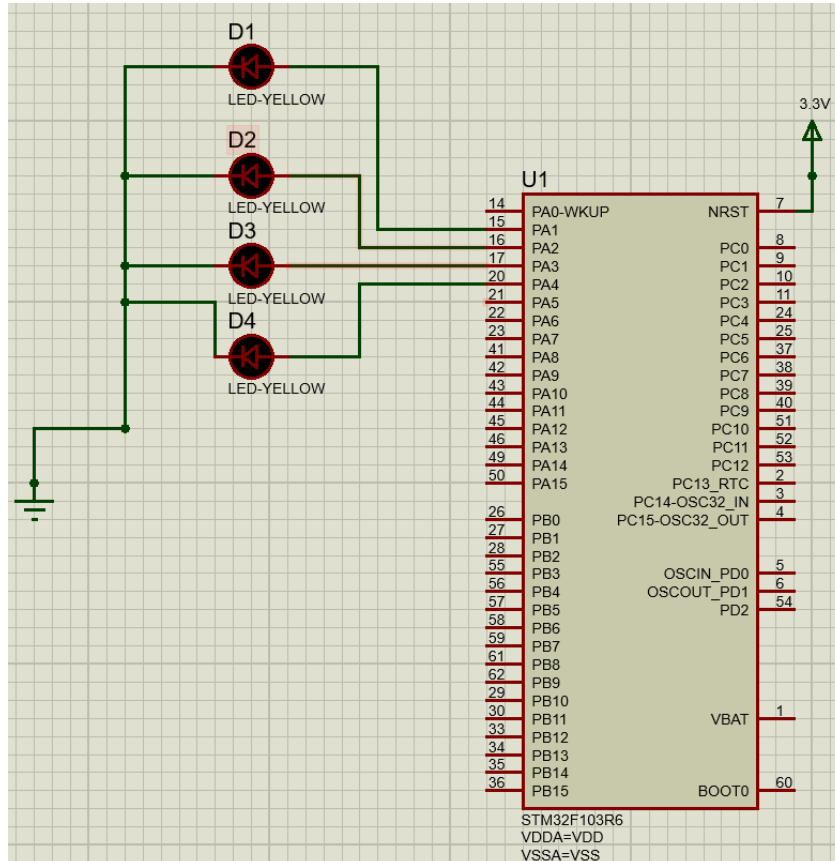
Arm7 与 Cortex3 的比较

Cortex-M3和ARM7的比较		
比较项目	ARM7	Cortex-M3
架构	ARMv4T（冯诺依曼） 指令和数据总线共用，会出现瓶颈	ARMv7-M（哈佛） 指令和数据总线分开，无瓶颈
指令集	32位ARM指令+16位Thumb指令 两套指令之间需要进行状态切换	Thumb/Thumb-2指令集 16位和32位 指令可直接混写，无需状态切换
流水线	3级流水线 若出现转移则需要刷新流水线，损失惨重	3级流水线+分支预测 出现转移时流水线无需刷新，几乎无损失
性能	0.95DMIPS/MHz（ARM模式）	1.25DMIPS/MHz
功耗	0.28mW/MHz	0.19mW/MHz
低功耗模式	无	内置睡眠模式
面积	0.62mm ² （仅内核）	0.86mm ² （内核+外设）
中断	普通中断IRQ和快速中断FIQ太少，大量外设不得不复用中断	不可屏蔽中断NMI+1-240个物理中断 每个外设都可以独占一个中断，效率高
中断延迟	24-42个时钟周期，缓慢	12个时钟周期，最快只需6个
中断压栈	软件手工压栈，代码长且效率低	硬件自动压栈，无需代码且效率高
存储器保护	无	8段存储器保护单元（MPU）
内核寄存器	寄存器分为多组、结构复杂、占核面积多	寄存器不分组（SP除外），结构简单
工作模式	7种工作模式，比较复杂	只有线程模式和处理模式两种，简单
乘除法指令	多周期乘法指令，无除法指令	单周期乘法指令，2-12周期除法指令
位操作	无 访问外设寄存器需分“读-改-写”三步走	先进的Bit-band位操作技术，可直接访问外设寄存器的某个值
系统节拍定时	无	内置系统节拍定时器，有利于操作系统移植

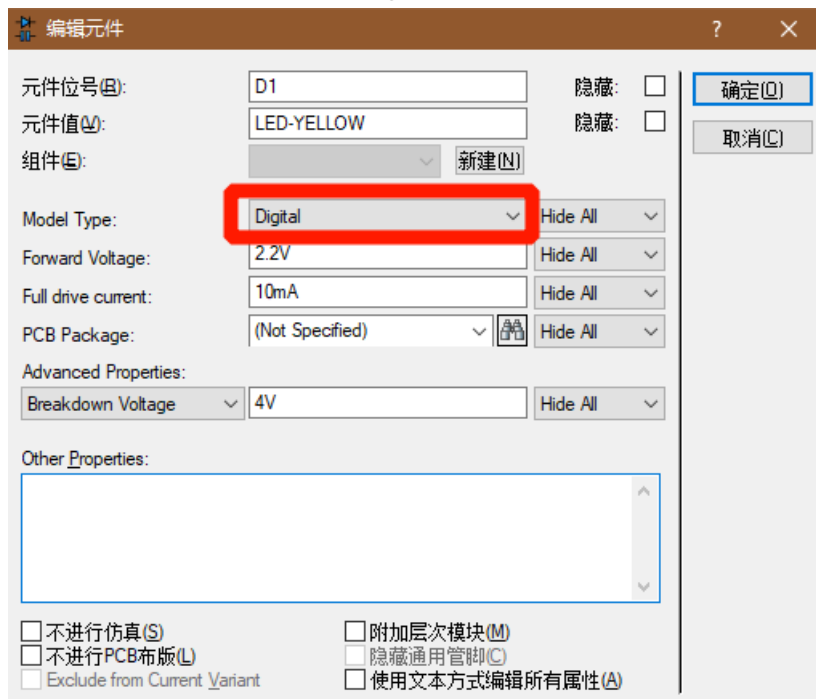
2.实验验证

2.1 实现 led 阵列显示

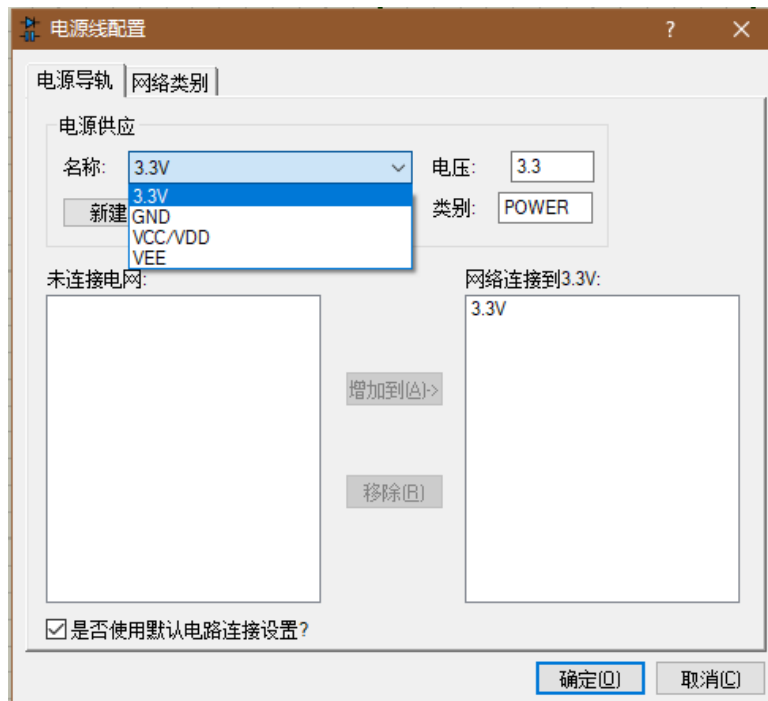
搭建电路图如下



这里需要注意将 LED 设置为 digital 模式，不然没办法进行高低电平点亮（坑）



同时也需要在设置里配置供电网络，电源属性
新建供电网络，并将其添加到 3.3V 逻辑电平上

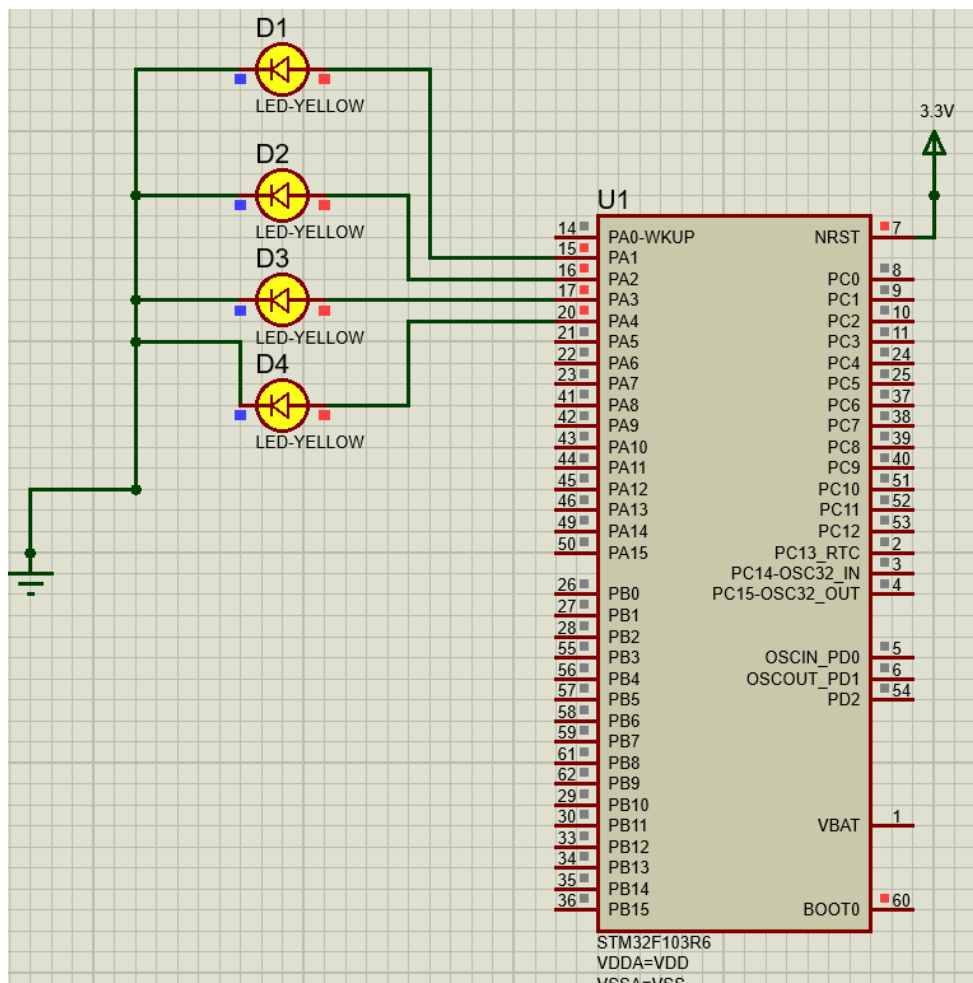
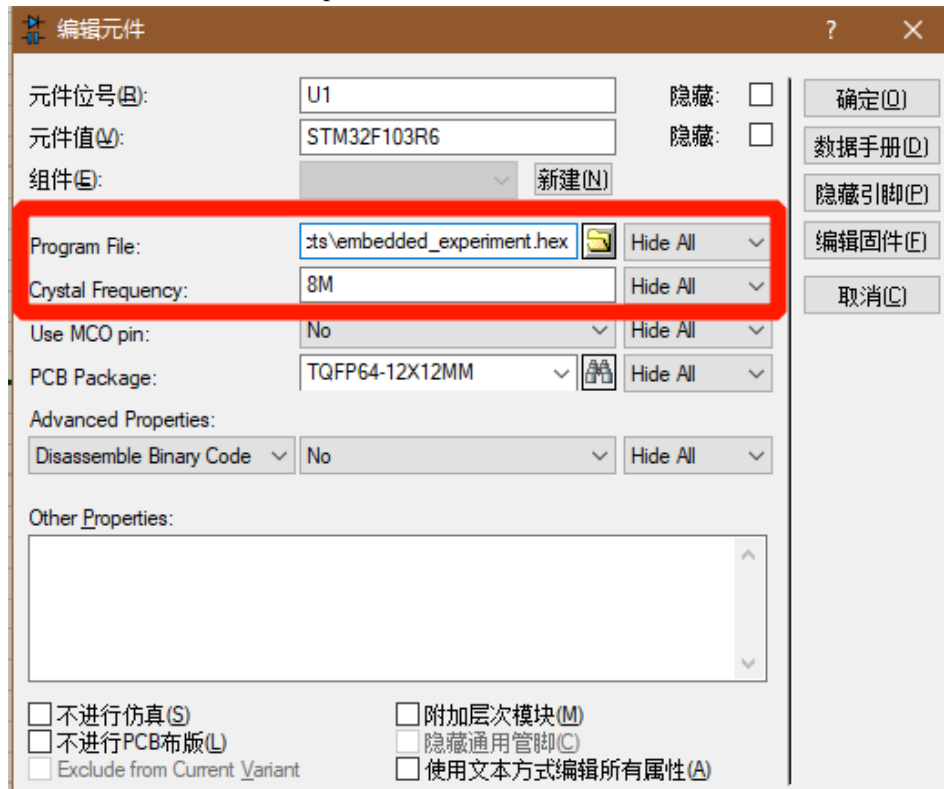


编写 LED 代码采用 STM32F103R6 并使用 RTE_DeVice.h 进行总控，编写 LED_Init()函数如下，并将其在 main 函数调用

```
void LED_Init() {  
    //使能GPIOA的时钟  
    GPIO_PortClock(GPIOA, true);  
  
    //定义GPIOA1~4为输出模式  
    GPIO_PinConfigure(GPIOA, 1, GPIO_OUT_OPENDRAIN, GPIO_MODE_OUT50MHZ);  
    GPIO_PinConfigure(GPIOA, 2, GPIO_OUT_OPENDRAIN, GPIO_MODE_OUT50MHZ);  
    GPIO_PinConfigure(GPIOA, 3, GPIO_OUT_OPENDRAIN, GPIO_MODE_OUT50MHZ);  
    GPIO_PinConfigure(GPIOA, 4, GPIO_OUT_OPENDRAIN, GPIO_MODE_OUT50MHZ);  
  
    //定义GPIOA1~4为高电平  
    GPIO_PinWrite(GPIOA, 1, 1);  
    GPIO_PinWrite(GPIOA, 2, 1);  
    GPIO_PinWrite(GPIOA, 3, 1);  
    GPIO_PinWrite(GPIOA, 4, 1);  
}
```

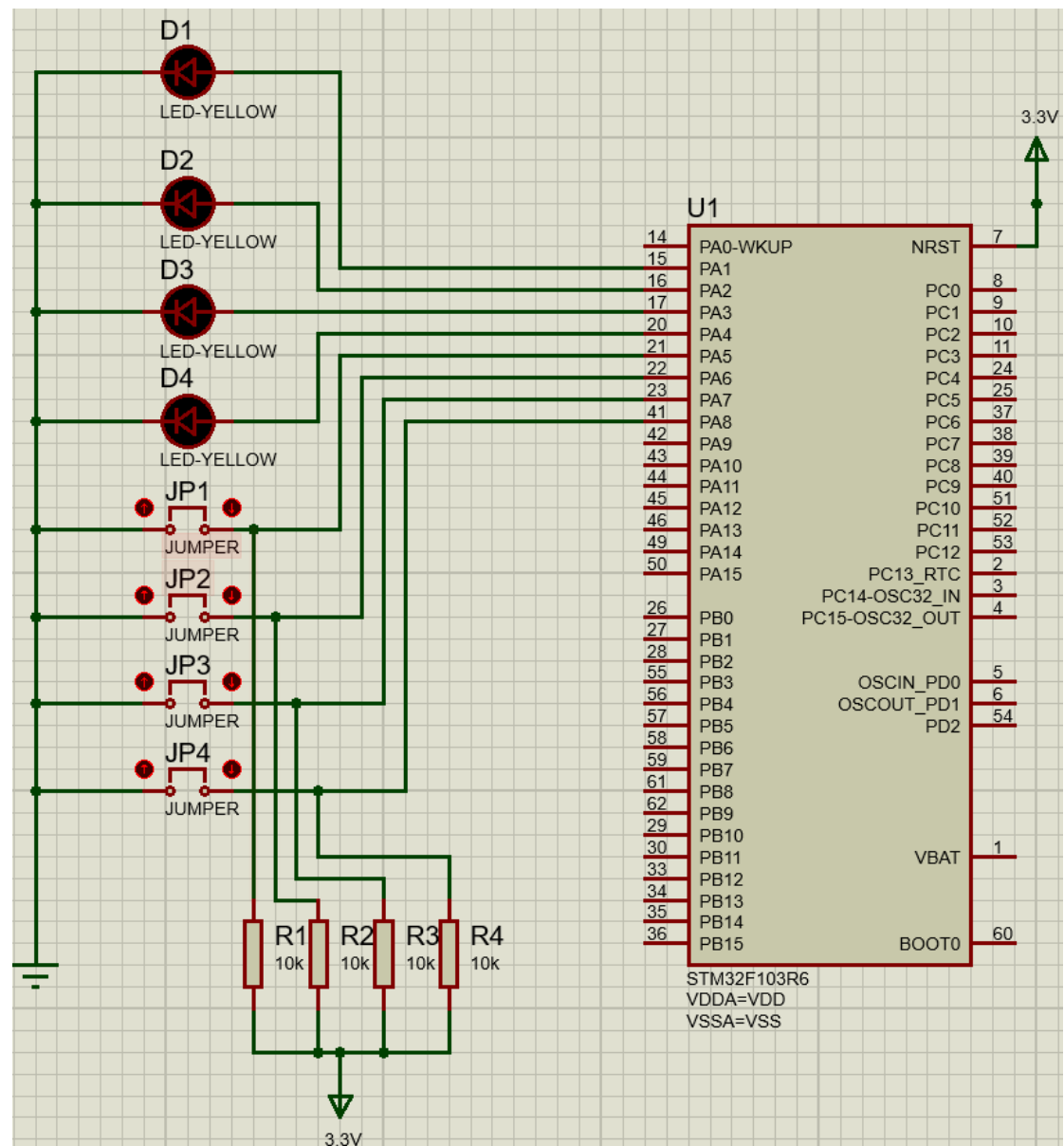
```
main.c | LED.c | LED.h | GPIO_STM32I  
1  #include "stm32f10x.h"  
2  #include "LED.h"  
3  
4  int main()  
5  {  
6      LED_Init();  
7  
8      while(1)  
9      {  
10  
11      }  
12  }  
13
```


将编译好的 hex 文件载入 proteus 进行仿真可以看到 LED 灯被点亮



2.2 实现按键检测阵列

搭建电路图如下



在 2.1 实验的基础上，绘制电路，并设定当 JP1-JP4 按下时 D1-JP4 熄灭
编写 key.c 代码，初始化并设定 GPIO 的输入属性

```
#include "key.h"

void key_Init() {
    //使能GPIOA的时钟
    GPIO_PortClock(GPIOA, true);

    //定义GPIOA5~GPIOA8为输入模式
    GPIO_PinConfigure(GPIOA, 5, GPIO_IN_FLOATING, GPIO_MODE_INPUT);
    GPIO_PinConfigure(GPIOA, 6, GPIO_IN_FLOATING, GPIO_MODE_INPUT);
    GPIO_PinConfigure(GPIOA, 7, GPIO_IN_FLOATING, GPIO_MODE_INPUT);
    GPIO_PinConfigure(GPIOA, 8, GPIO_IN_FLOATING, GPIO_MODE_INPUT);
}
```

在 man.c 里实时检测 key 的状态并改变 LED 的状态

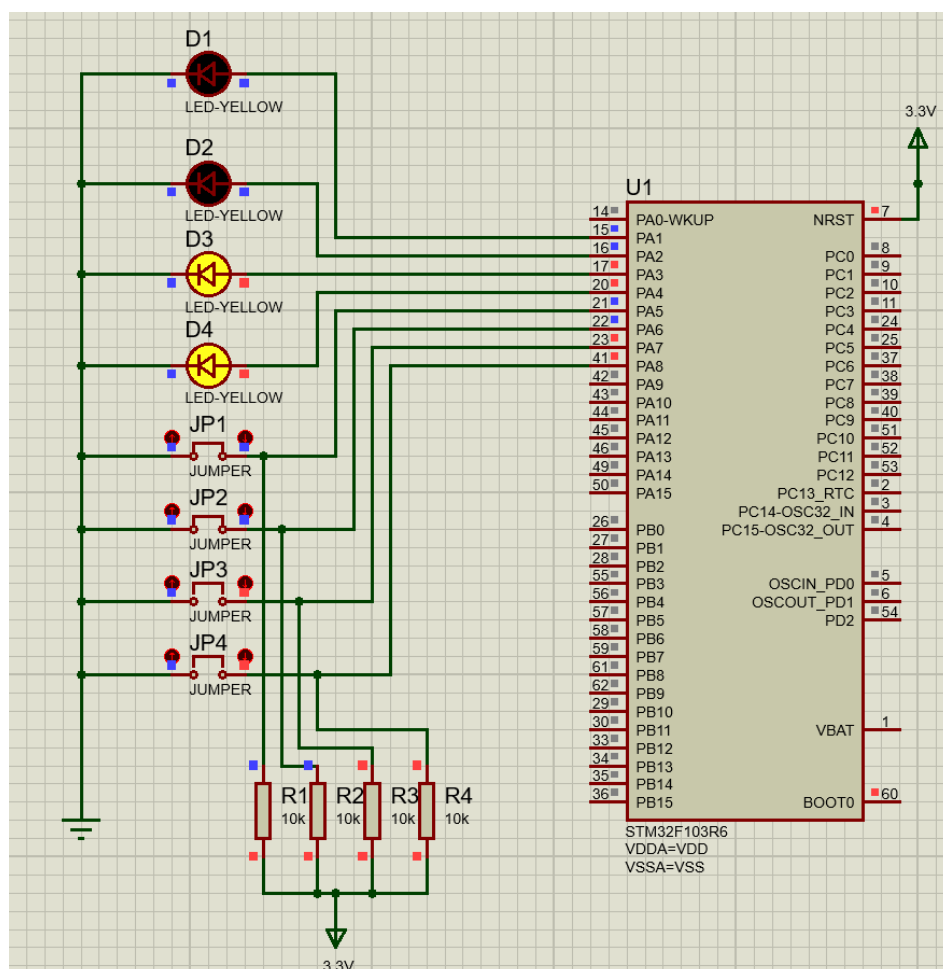
```
int main()
{
    LED_Init();
    key_Init();
    while(1)
    {
        if(GPIO_PinRead(GPIOA,5))
            GPIO_PinWrite(GPIOA,1,1); //拉高
        else
            GPIO_PinWrite(GPIOA,1,0); //置低

        if(GPIO_PinRead(GPIOA,6))
            GPIO_PinWrite(GPIOA,2,1); //拉高
        else
            GPIO_PinWrite(GPIOA,2,0); //置低

        if(GPIO_PinRead(GPIOA,7))
            GPIO_PinWrite(GPIOA,3,1); //拉高
        else
            GPIO_PinWrite(GPIOA,3,0); //置低

        if(GPIO_PinRead(GPIOA,8))
            GPIO_PinWrite(GPIOA,4,1); //拉高
        else
            GPIO_PinWrite(GPIOA,4,0); //置低
    }
}
```

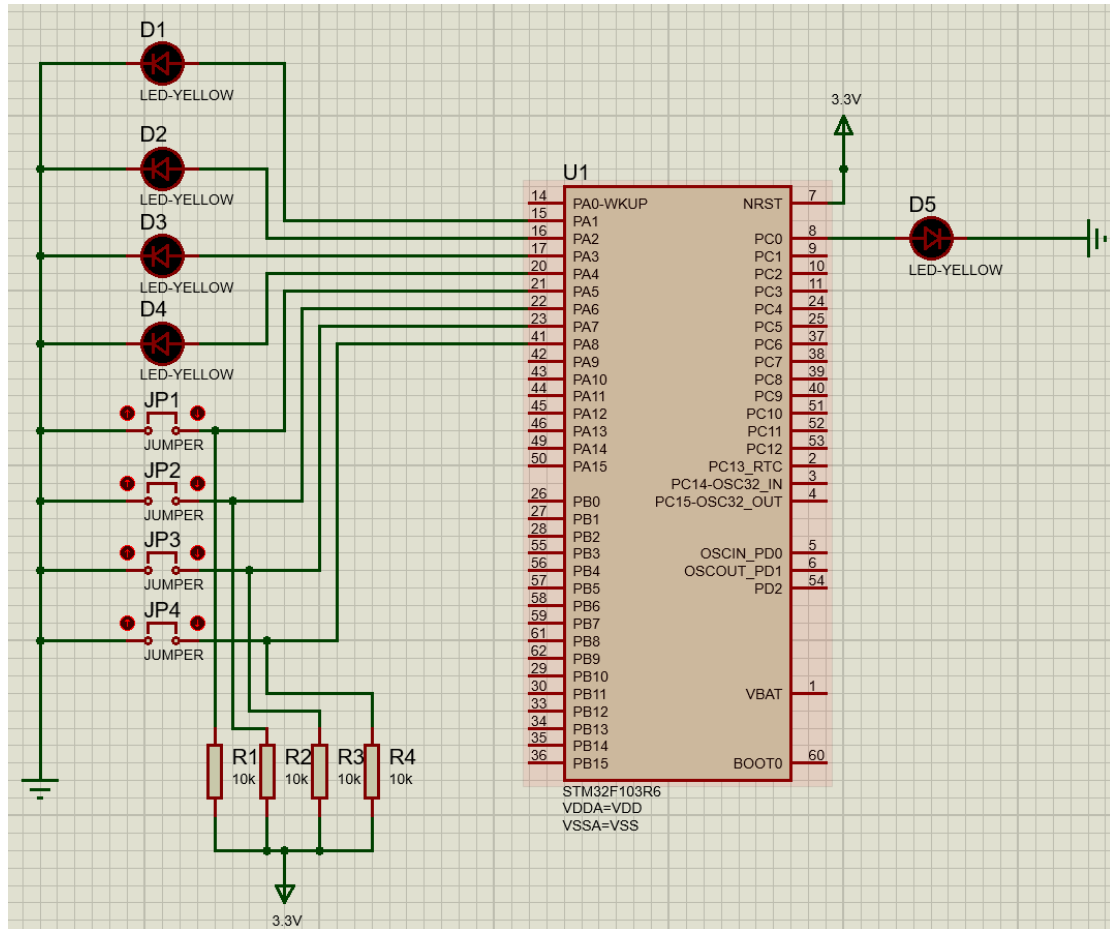
将编译好的 hex 文件载入 proteus 进行仿真可以看到当 JP1-JP4 按下时，LED1-LED4 熄灭



2.3 定时器中断使用

搭建电路如下

在 2.2 的基础上，在 PC0 端口加入一个 LED 灯，使用定时器控制其按照 0.5HZ 的频率进行闪烁，即亮 1S 暗 1S。



编写定时器初始化函数，并使能中断及设置中断优先级，编写中断服务函数。在中断服务函数中反转 PC0 的值实现 LED 闪烁的功能。

具体实际效果可查看[视频](#)

```

#include "TIM.h"

//通用定时器3中断初始化
//arr: 自动重装值。
//psc: 时钟预分频数
//这里使用的是定时器3!
void TIM3_Init(ul6 arr,ul6 psc)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE); //时钟使能

    //定时器TIM3初始化
    TIM_TimeBaseStructure.TIM_Period = arr; //设置在下一个更新事件装入活动的自动重装载寄存器周期的值
    TIM_TimeBaseStructure.TIM_Prescaler = psc; //设置用来作为TIMx时钟频率除数的预分频值
    TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1; //设置时钟分割:TDTS = Tck_tim
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //TIM向上计数模式
    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure); //根据指定的参数初始化TIMx的时间基数单位

    TIM_ITConfig(TIM3, TIM_IT_Update, ENABLE ); //使能指定的TIM3中断,允许更新中断

    //中断优先级NVIC设置
    NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn; //TIM3中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; //先占优先级0级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3; //从优先级3级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //IRQ通道被使能
    NVIC_Init(&NVIC_InitStructure); //初始化NVIC寄存器

    TIM_Cmd(TIM3, ENABLE); //使能TIMx
}

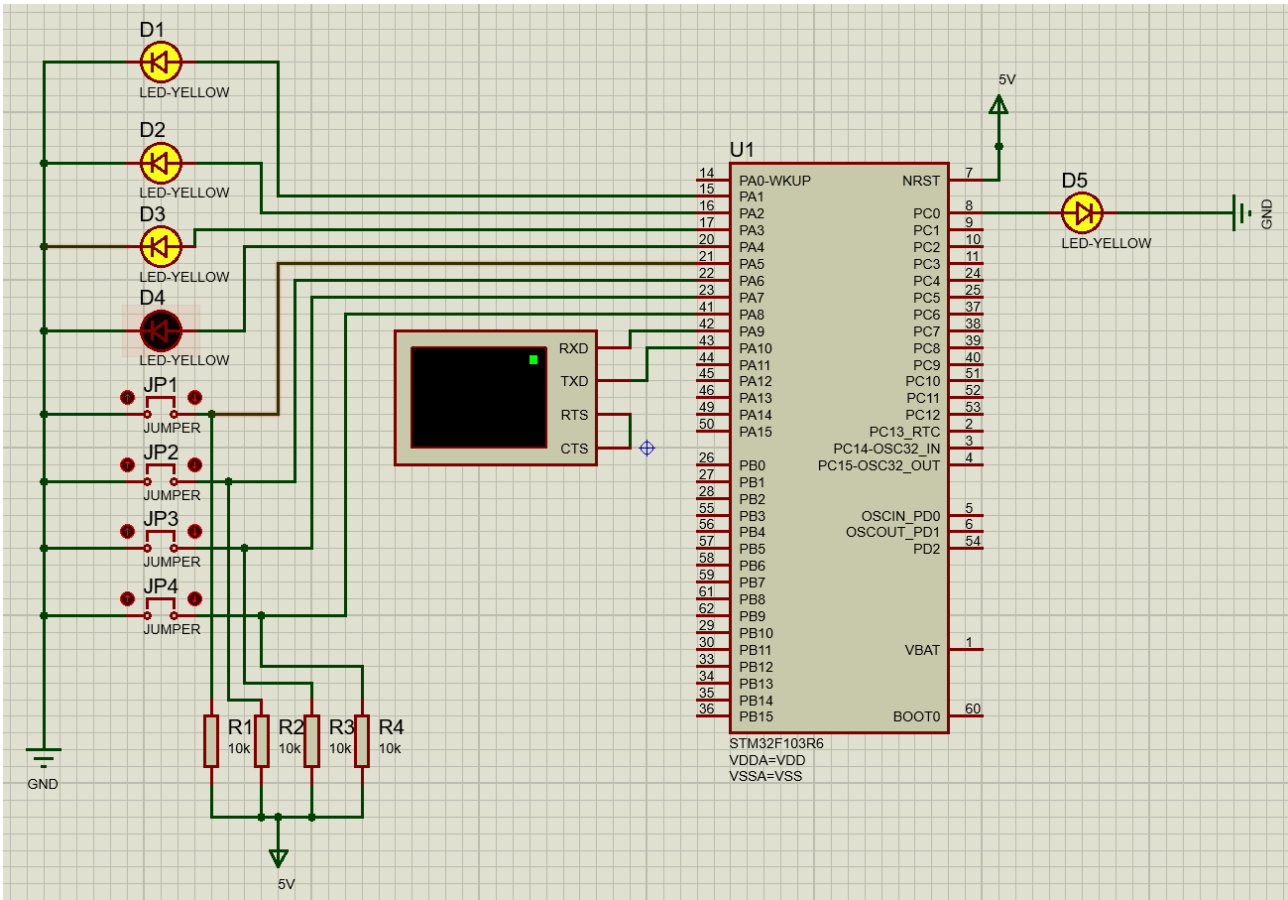
//定时器3中断服务程序
void TIM3_IRQHandler(void) //TIM3中断
{
    if (TIM_GetITStatus(TIM3, TIM_IT_Update) != RESET) //检查TIM3更新中断发生与否
    {
        TIM_ClearITPendingBit(TIM3, TIM_IT_Update); //清除TIMx更新中断标志

        if(GPIO_PinRead(GPIOC,0))
            GPIO_PinWrite(GPIOC,0,0);
        else
            GPIO_PinWrite(GPIOC,0,1);
    }
}

```

2.4 UART 串口通信

在 2.3 的基础上，搭建电路图如下



配置接收器属性如下

编辑元件

元件位号(B): 隐藏: ☐

元件值(V): 隐藏: ☐

组件(E): 新建(N)

Baud Rate: 9600 Hide All

Data Bits: 8 Hide All

Parity: NONE Hide All

Stop Bits: 1 Hide All

Send XON/XOFF: No Hide All

Advanced Properties:

RX/TX Polarity: Normal Hide All

确定(O) 帮助(H) 取消(C)

使用 RTE_DeVice.h 控件进行初始化 USART，并编写服务函数

```
#include "usart.h"
#include "stm32f10x.h"

#if 1
#pragma import( _use_no_semihosting)
//标准库需要的支持函数
struct __FILE
{
    int handle;

};

FILE __stdout;
//定义 sys_exit() 以避免使用半主机模式
sys_exit(int x)
{
    x = x;
}
//重定义 fputc 函数
int fputc(int ch, FILE *f)
{
    while((USART1->SR&0X40)==0); //循环发送,直到发送完毕
    USART1->DR = (u8) ch;
    return ch;
}
#endif

u8 USART_RX_BUF[USART_REC_LEN]; //接收缓冲,最大USART_REC_LEN个字节.末字节为换行符
u16 USART_RX_STA; //接收状态标记

void uart_init(u32 bound){
    //GPIO端口设置
    USART_InitTypeDef USART_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE); //使能USART1, GPIOA时钟
    GPIO_PortClock(GPIOA,true);

    //定义GPIOA5~GPIO8为输入模式
    GPIO_PinConfigure(GPIOA,9,GPIO_AF_PUSHPULL,GPIO_MODE_OUT50MHZ);
    GPIO_PinConfigure(GPIOA,10,GPIO_IN_FLOATING,GPIO_MODE_INPUT);

    //Usart1 NVIC 配置
    NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority=3 ;//抢占优先级3
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1; //子优先级3
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; //IRQ通道使能
    NVIC_Init(&NVIC_InitStructure); //根据指定的参数初始化VIC寄存器

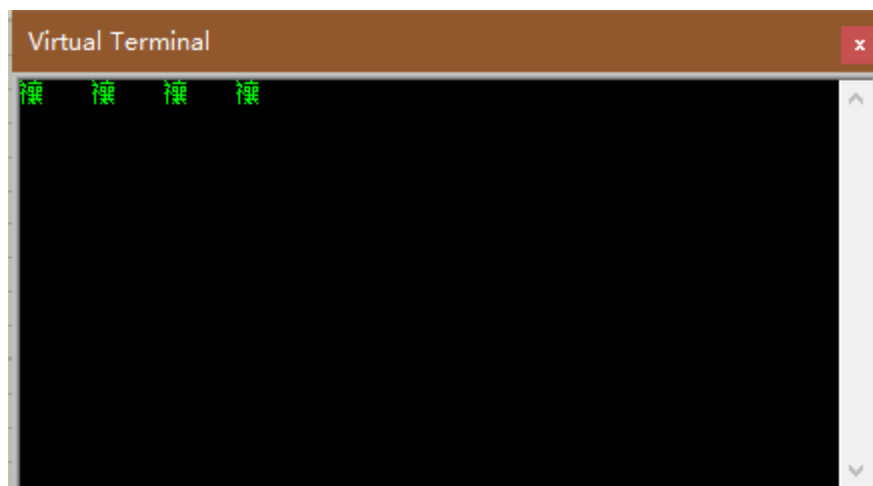
    //USART 初始化设置
    USART_InitStructure.USART_BaudRate = bound;//串口波特率
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;//字长为8位数据格式
    USART_InitStructure.USART_StopBits = USART_StopBits_1;//一个停止位
    USART_InitStructure.USART_Parity = USART_Parity_No;//无奇偶校验位
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;//无硬件数据流控制
    USART_InitStructure.USART_Mode = USART_Mode_Rx|USART_Mode_Tx; //收发模式

    USART_Init(USART1, &USART_InitStructure); //初始化串口1
    USART_ITConfig(USART1, USART_IT_RXNE, ENABLE); //开启串口接受中断
    USART_Cmd(USART1, ENABLE); //使能串口1
}
```

但由于仿真的波特率总是与实际设置的不一致，在 keil 里调试时正常的，但在 Proteus 里接收总是乱码

```
UART #1
embedded system
embedded system
embedded system
embedded system
embedded system
embedded system
```

(keil 发送)



(proteuse 接收)

三.综合实验

1. 定义数组[12, 15, 4, 9, 7, 10, 11, 2], 用 arm 指令实现数组的从大到小排序
2. 在内存中, 定义一个字符串, 拷贝到内存 0x10000000 起始地址处
3. 有数据[1, 10, 9, 5, 3, 4, 5, 2, 7, 9, 2, 6], 计算其均值与标准差。
4. 有数据[11, 20, 19, 20, 8, 16, 15, 12, 17, 19, 12, 16], 计算此数据与 3 题中数据的相关系数
5. 搭建 arm 基础电路, 实现按键中断触发下的 LED 数码管滚动显示数值的增减, 显示总体开关功能

其中 1-4 题编码如下:

```
#include "arm_math.h"
```

```
//大在前, 小在后,排序
```

```
void BubbleSort(int a[], int len)
```

```
{
    int i, j, temp;
    for (j = len - 1; j > 0; j--)
    {
        for (i = len - 1; i > len - 1 - j; i--)
            if (a[i] > a[i - 1])
            {
                temp = a[i];
                a[i] = a[i - 1];
                a[i - 1] = temp;
            }
    }
}
```

```
//计算平均值
```

```
float get_mean(int a[], int len)
```

```
{
    float mean;
    int i;
    for(i=0; i<len; i++)
    {
        mean += a[i];
    }
    mean = mean/len;
    return mean;
}
```

```
//计算标准差
```

```
float get_standard(int a[], int len, float mean)
```

```

{
    float standard;
    int i,temp;
    for(i=0;i<len;i++)
    {
        temp = a[i]-mean;
        temp = temp*temp;
        standard+=temp;
    }

    standard = standard/len;
    standard = sqrt(standard);

    return standard;
}

```

//计算协方差

```

float get_covariance(float mean1,float mean2,int a[],int b[],int len)
{
    float covariance;
    int i;
    for(i = 0;i<len;i++)
    {
        covariance = (a[i]-mean1)*(b[i]-mean2);
        covariance +=covariance;
    }
    covariance = covariance/len;

    return covariance;
}

```

```
int a1[8]={12,15,4,9,7,10,11,2};
```

```
int a3[12]={1,10,9,5,3,6,5,2,7,9,2,6};
```

```
int a4[12]={11,20,19,20,8,16,15,12,17,19,12,16};
```

```
char* b = {"Embedded system"};
```

```
float mean1,mean3,mean4;
```

```
float standard3,standard4,covariance34,coefficient34;
```

```
BubbleSort(a1,8);
```

```
mean3 = get_mean(a3,12);
```

```
mean4 = get_mean(a4,12);
```








```
standard3 = get_standard(a3,12,mean3);
```

```
standard4 = get_standard(a4,12,mean4);
```

```
covariance34 = get_covariance(mean3,mean4,a3,a4,12);
```

```
coefficient34 = covariance34/(standard3*standard4);
```

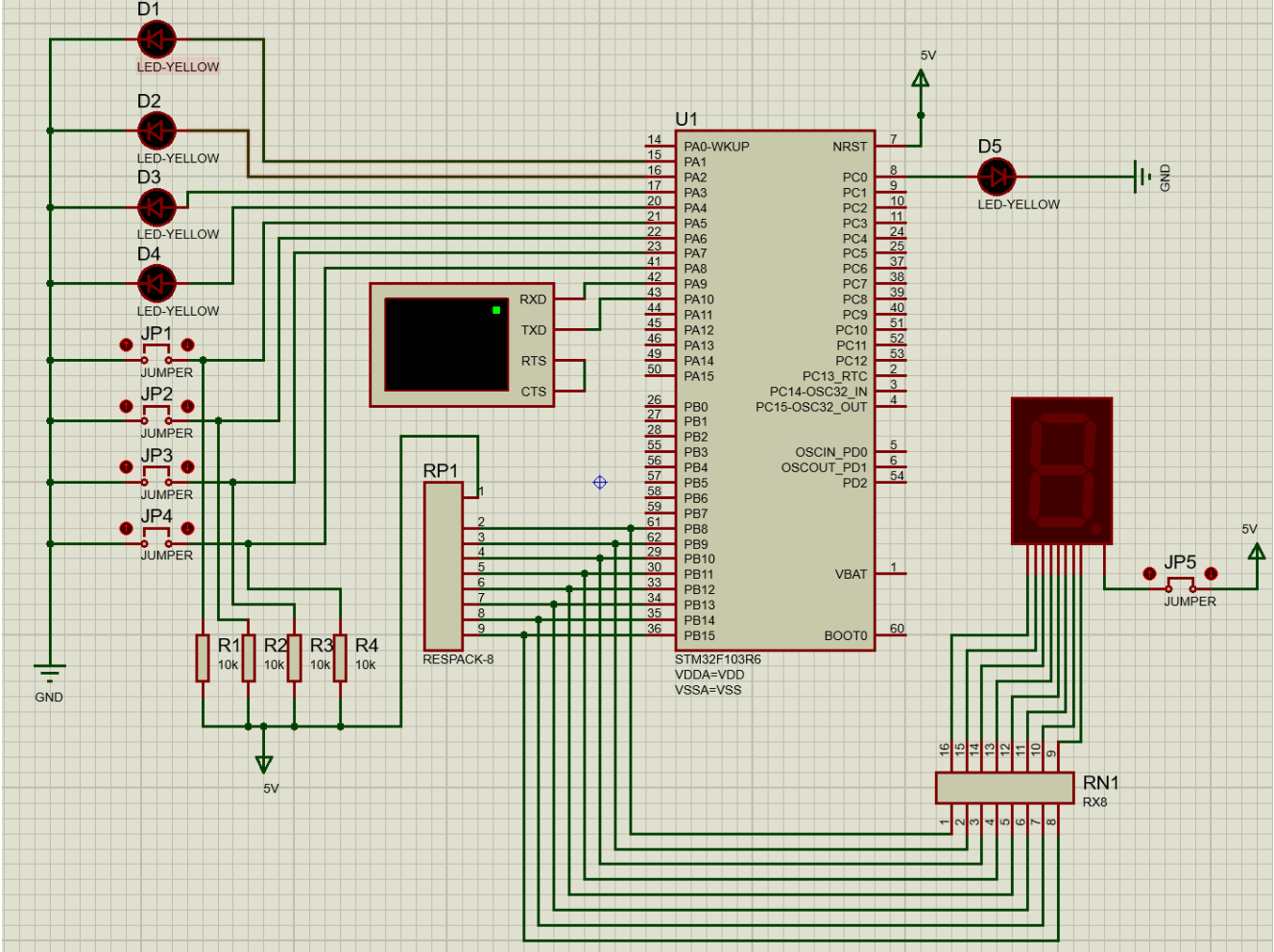
仿真结果如下

 a1	0x20000660	int[8]
...  mean3	5.4166651	float
...  mean4	15.416667	float
...  standard3	2.55631018	float
...  standard4	3.40852666	float
...  covariance34	0.0567129441	float
...  coefficent34	0.00650881557	float

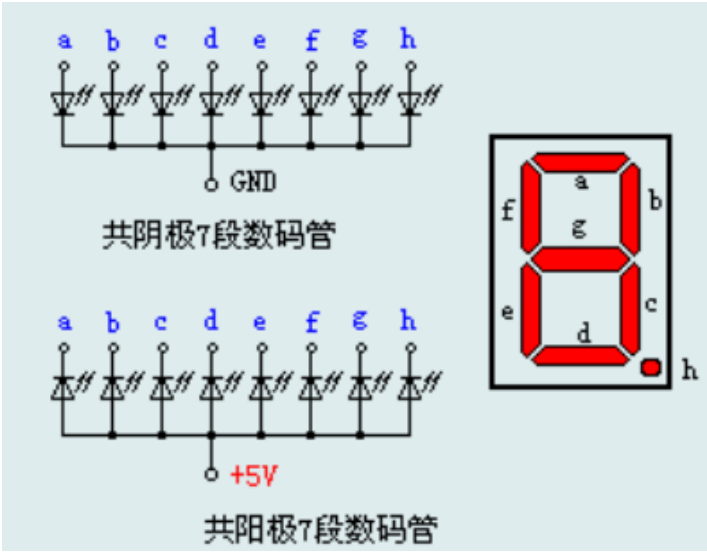
将字符串转存的汇编语句如下

实验 5:

在前置电路的基础上搭建电路如下



数码管可以分为共阳极与共阴极两种，共阳极就是把所有 LED 的阳极连接到共同接点 com，使用时 com 接正 5 伏电源，而每个 LED 的阴极分别为 a、b、c、d、e、f、g 及 dp(小数点)；共阴极则是把所有 LED 的阴极连接到共同接点 COM，使用时 COM 要将其接地。而每个 LED 的阳极分别为 a、b、c、d、e、f、g 及 dp(小数点)，8 个 LED 的分布方式如图下图所示。图中的 8 个 LED 分别与上面那个图中的 A~DP 各段相对应，通过控制各个 LED 的亮灭来显示数字。



切记要将排阻和上拉电阻选择 DIGITAL 数字模式，不然功能无法正常实现

元件位号(R): 隐藏: ☐

元件值(V): 隐藏: ☐

组件(E):

PCB Package:

LISA Model File:

Model Type:

由于这里选用的是共阳极数码管，且使用 PB8 - PB15 驱动，编码表如下

	a(PB8)	b(PB9)	c(PB10)	d(PB11)	e(PB12)	f(PB13)	g(PB14)	h(PB15)	GPIOB->BSR(15-0)
0	0	0	0	0	0	0	1	1	0XC000
1	1	0	0	1	1	1	1	1	0XF900
2	0	0	1	0	0	1	0	1	0XA400
3	0	0	0	0	1	1	0	1	0XB000
4	1	0	0	1	1	0	0	1	0X9900
5	0	1	0	0	1	0	0	1	0X9200
6	0	1	0	0	0	0	0	1	0X8200
7	0	0	0	1	1	1	1	1	0XF800
8	0	0	0	0	0	0	0	1	0X8000
9	0	0	0	0	1	0	0	1	0X9000

[illegible]

编写代码如下：

```
u16 LED_Tube[10] =
{0xC000,0xF900,0xA400,0xB000,0x9900,0x9200,0x8200,0xF800,0x8000,0x9000};

void Tube_Init(){
    //使能 GPIOB 的时钟
    GPIO_PortClock(GPIOB,true);

    //定义 GPIOB8~15 为输出模式
    GPIO_PinConfigure(GPIOB,8,GPIO_OUT_PUSH_PULL,GPIO_MODE_OUT50MHZ);
    GPIO_PinConfigure(GPIOB,9,GPIO_OUT_PUSH_PULL,GPIO_MODE_OUT50MHZ);
    GPIO_PinConfigure(GPIOB,10,GPIO_OUT_PUSH_PULL,GPIO_MODE_OUT50MHZ);
    GPIO_PinConfigure(GPIOB,11,GPIO_OUT_PUSH_PULL,GPIO_MODE_OUT50MHZ);
    GPIO_PinConfigure(GPIOB,12,GPIO_OUT_PUSH_PULL,GPIO_MODE_OUT50MHZ);
    GPIO_PinConfigure(GPIOB,13,GPIO_OUT_PUSH_PULL,GPIO_MODE_OUT50MHZ);
    GPIO_PinConfigure(GPIOB,14,GPIO_OUT_PUSH_PULL,GPIO_MODE_OUT50MHZ);
    GPIO_PinConfigure(GPIOB,15,GPIO_OUT_PUSH_PULL,GPIO_MODE_OUT50MHZ);
}

//定时器 3 中断服务程序
void TIM3_IRQHandler(void)    //TIM3 中断
{
    static int i =0;
    if (TIM_GetITStatus(TIM3, TIM_IT_Update) != RESET)    //检查 TIM3 更新中断发生与
    {
        TIM_ClearITPendingBit(TIM3, TIM_IT_Update);    //清除 TIMx 更新中断标志
        printf("embedded system \r\n");

        if(GPIO_PinRead(GPIOC,0))
            GPIO_PinWrite(GPIOC,0,0);
        else
            GPIO_PinWrite(GPIOC,0,1);

        if(i<10)
        {
            GPIOB->ODR &= 0x00FF;
            GPIOB->BSRR = LED_Tube[i++];
        }
        else
            i=0;
    }
}
```

最终数码管将按照一定的时间从 0-9 变化，具体效果见[视频](#)

