**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

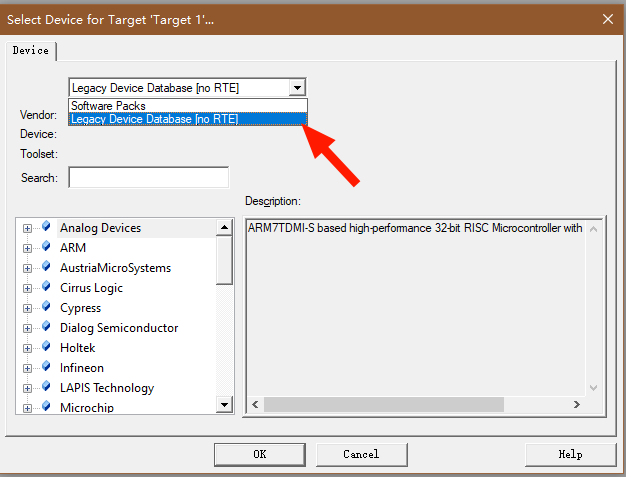
指导教师： 杨涛

2020 年 12月 1日

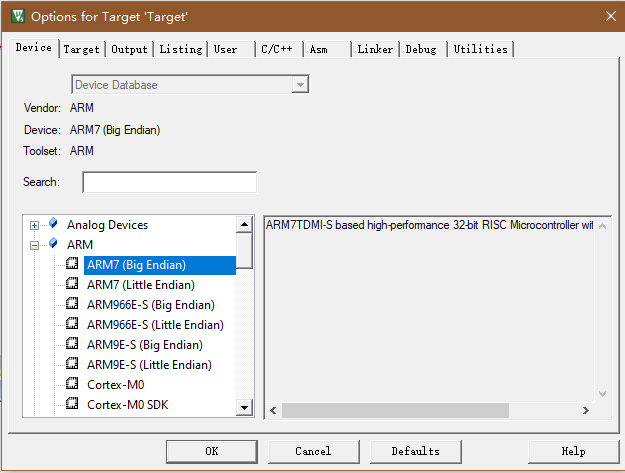
**一．验证实验**

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/>

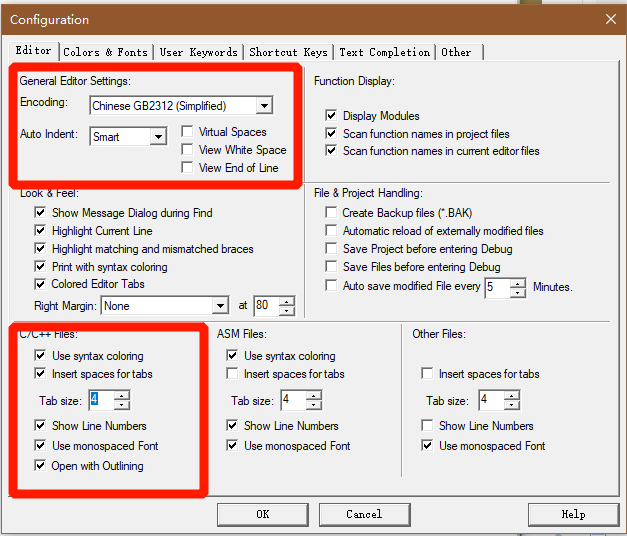
芯片包: <https://www.keil.com/dd2/pack/>



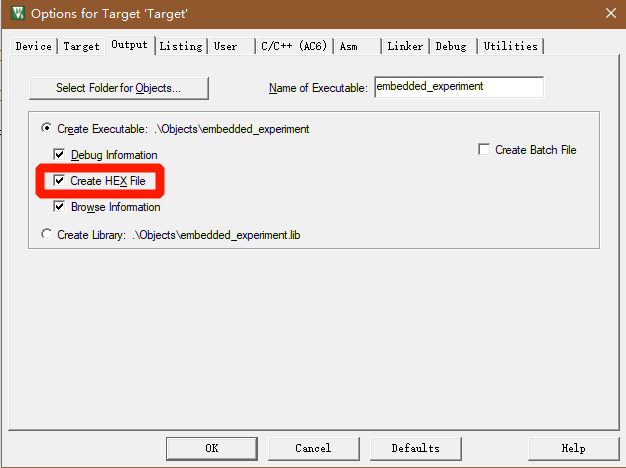
下载完成后新建工程时要注意选择Legacy Device Database,不要选择Software Packs,不然找不到旧的支持包，这里选择ARM7内核,不添加启动文件,不选择器件，只做纯软件仿真，否则需要添加启动代码。



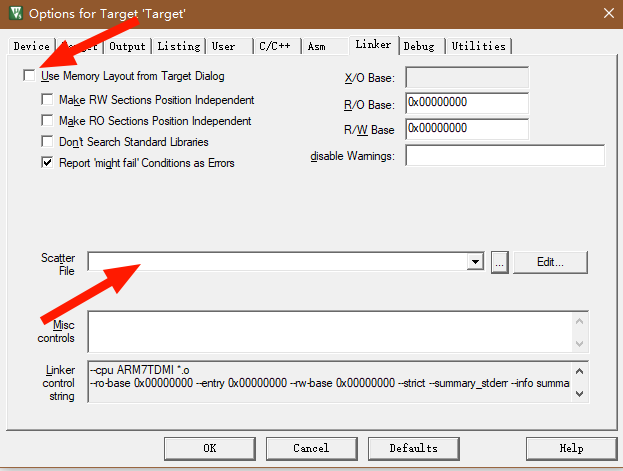
然后对工程设置做一些修改



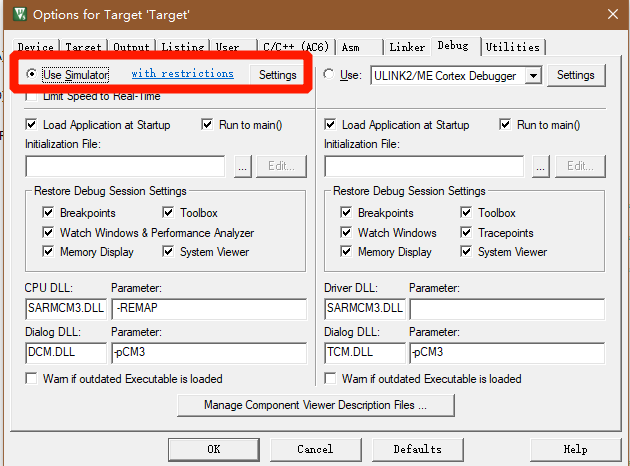
创建16进制文件



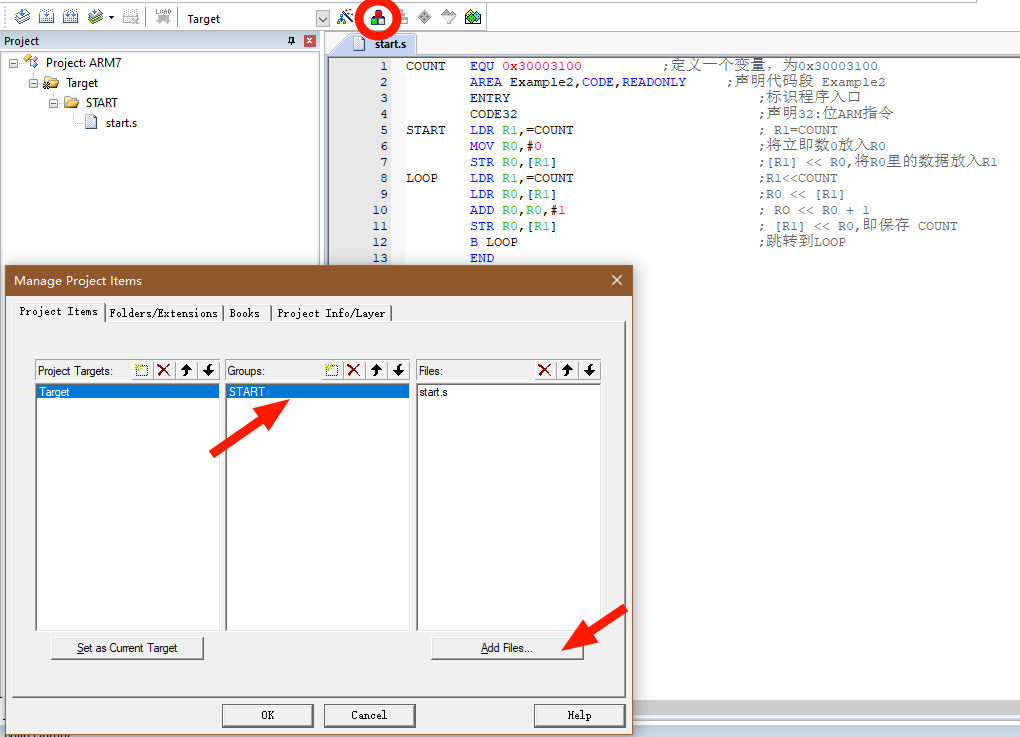
关闭连接文件，不使用分散加载文件,只进行纯软件仿真



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**

使用MOV指令访问ARM通用寄存器

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 ; RO << R0 + 1

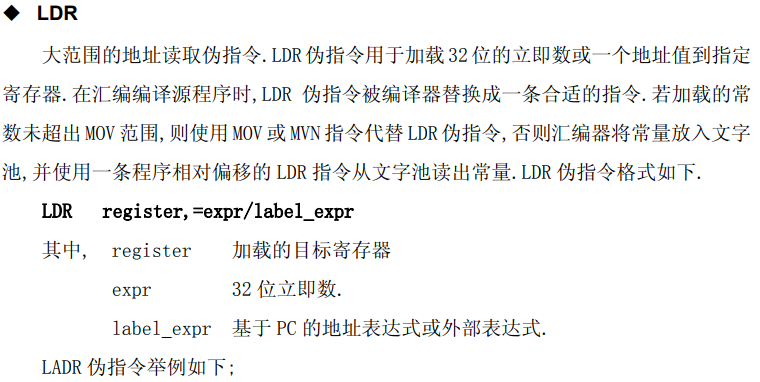
STR R0,[R1] ; [R1] << R0,即保存 COUNT

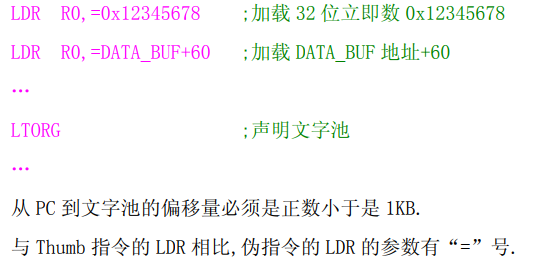
B LOOP ;跳转到LOOP

END

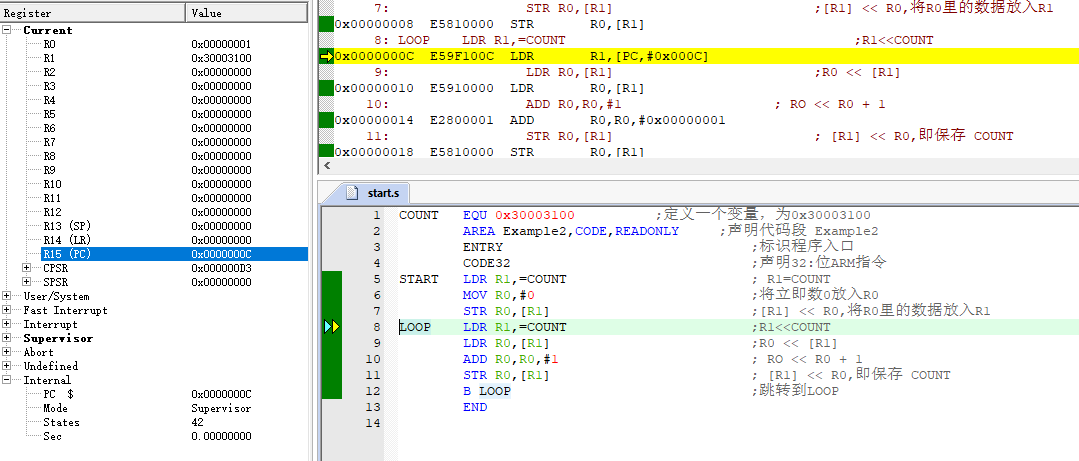
AREA Example2,CODE,READONLY ;声明了一个代码段，名字为Example2，类型为CODE,本段只读

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

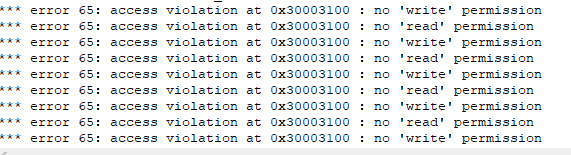


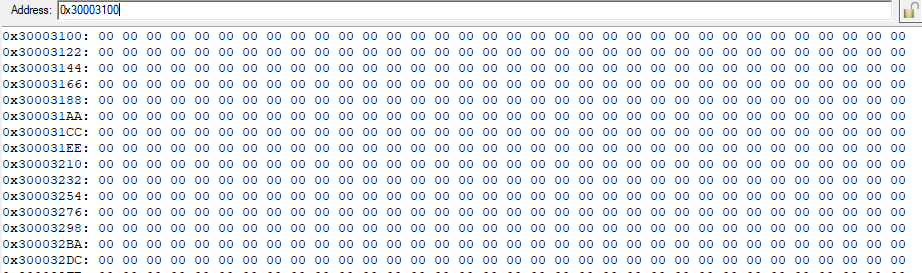


完成代码输入之后进行编译，编译无误进行仿真



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

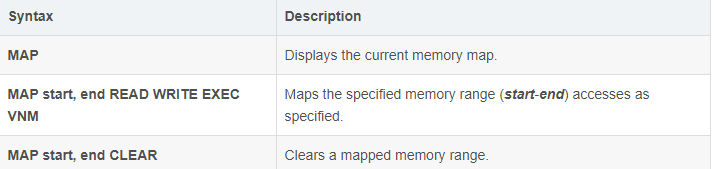




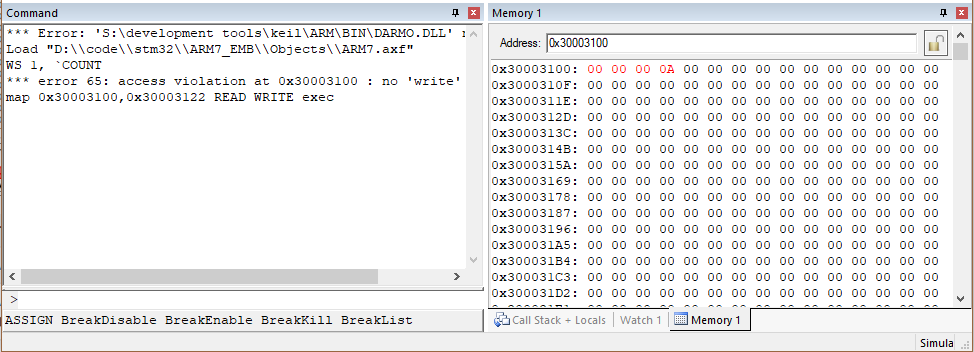
解决办法：在command窗口输入

map 0x30003100, 0x30003122 read write exec

START-END之间需要包含需要操作的地址，回车，复位，运行代码，即可解决问题



指定寄存器每隔一个循环+1，代码验证实现



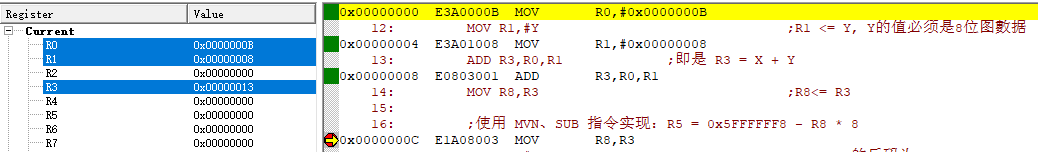
**1.2掌握ARM算术，逻辑运算指令**

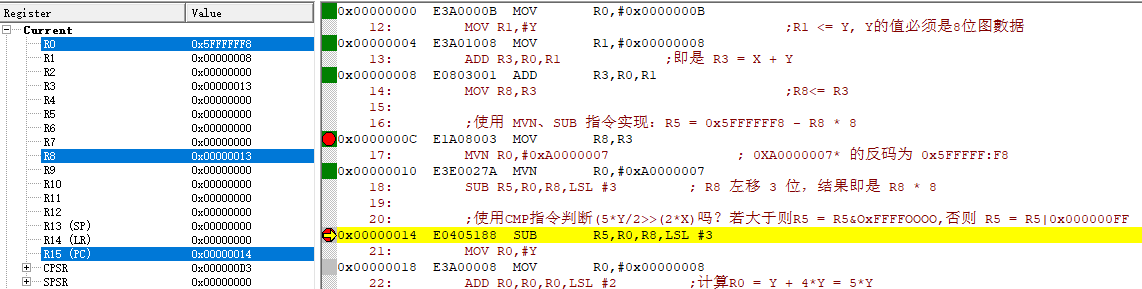
使用ADD、SUB、AND、ORR、CMP、TST等指令完成数据加减

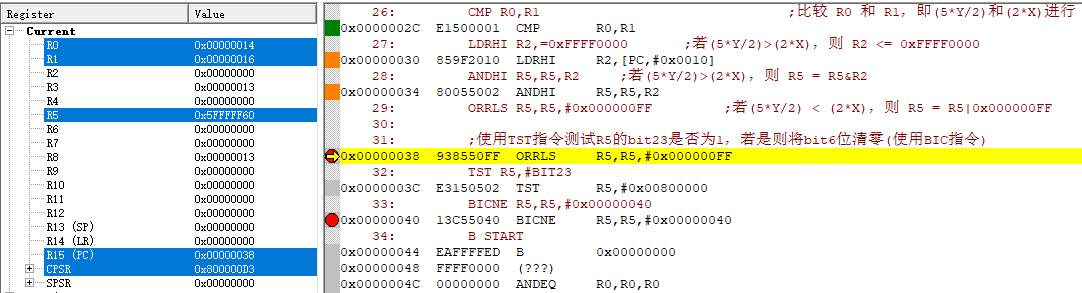
验证代码：

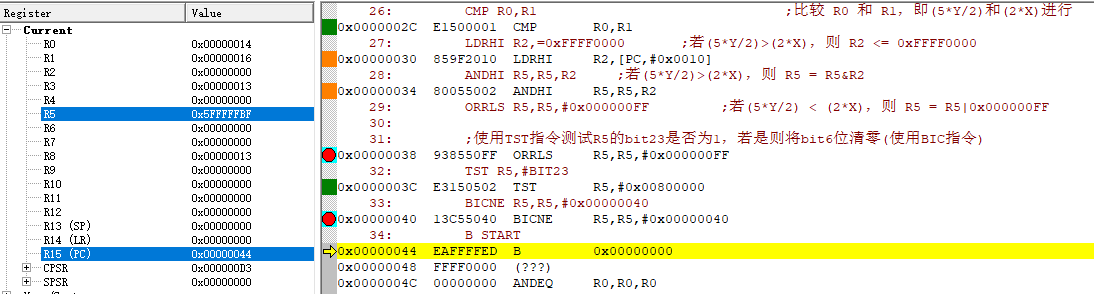


打上断点并进行仿真验证，详细指令使用规则可参见指令集





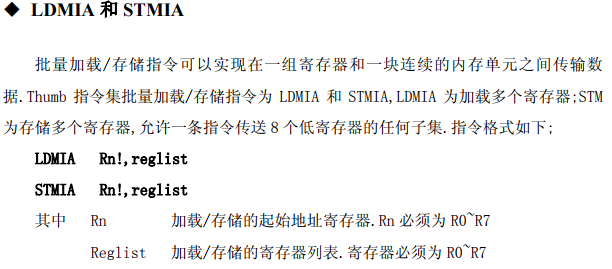


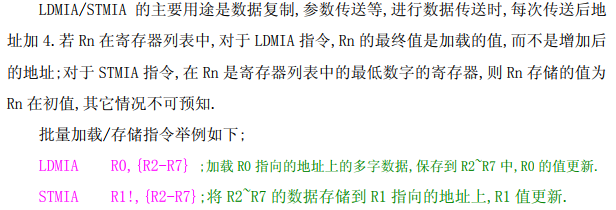


**1.3掌握ARM寻址方式**

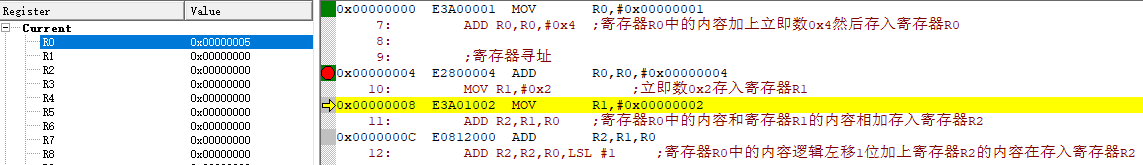
编写汇编程序，分别实现:立即数寻址,寄存器寻址,寄存器间接寻址,基址变址寻址,多寄存器寻址

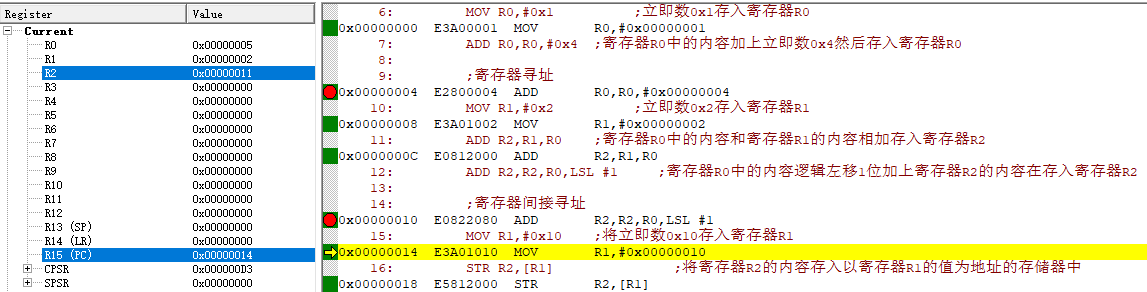
验证代码:





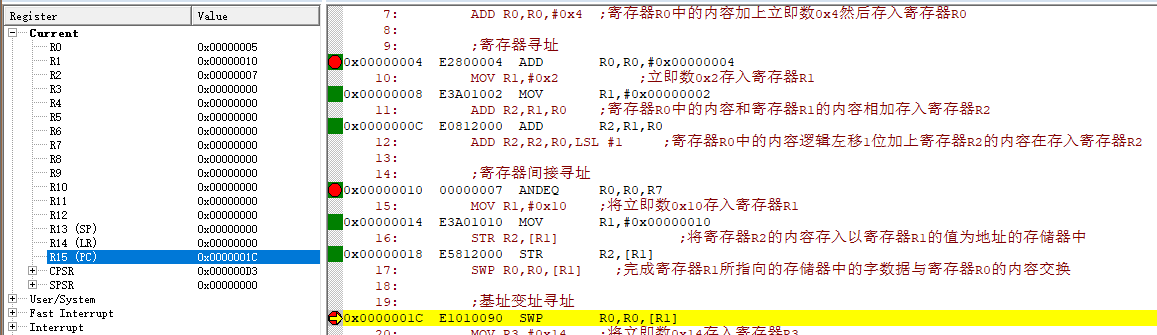
打上断点并进行仿真验证，需要在终端输入map 0x00, 0x100 read write exec

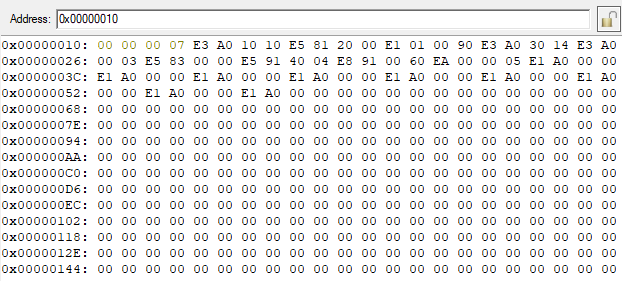


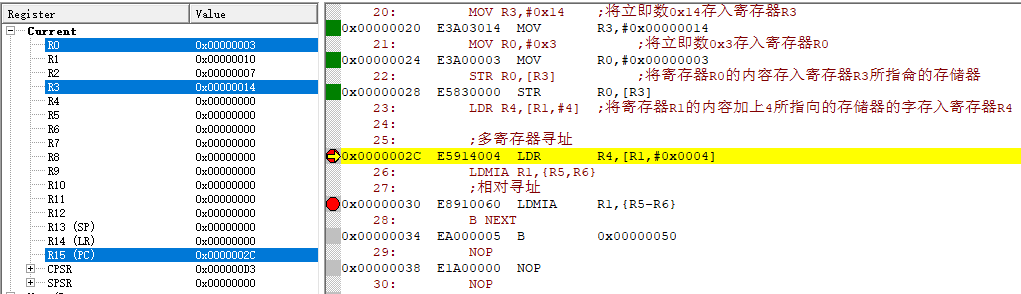


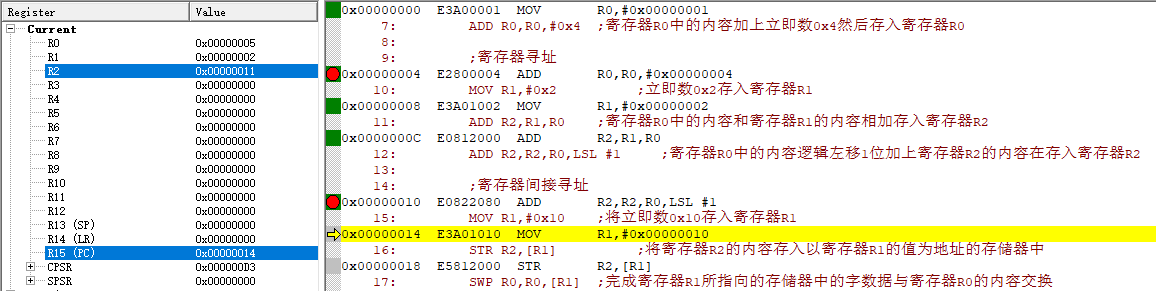


这个时候有提示：Non-aligned Access: ARM Instruction at 00000018H, Memory Access at 00000002H 非对齐访问,但实际上我认为这个地址是对齐的。可能只是内存空间没有实际映射，尝试修改内存地址为0x0000FF00。内存操作的地址必须与4字节边界对齐。这意味着地址必须是4的倍数，或者，地址的后两位必须为零(二进制)。但最后是因为，map 写入时注意写成32位。









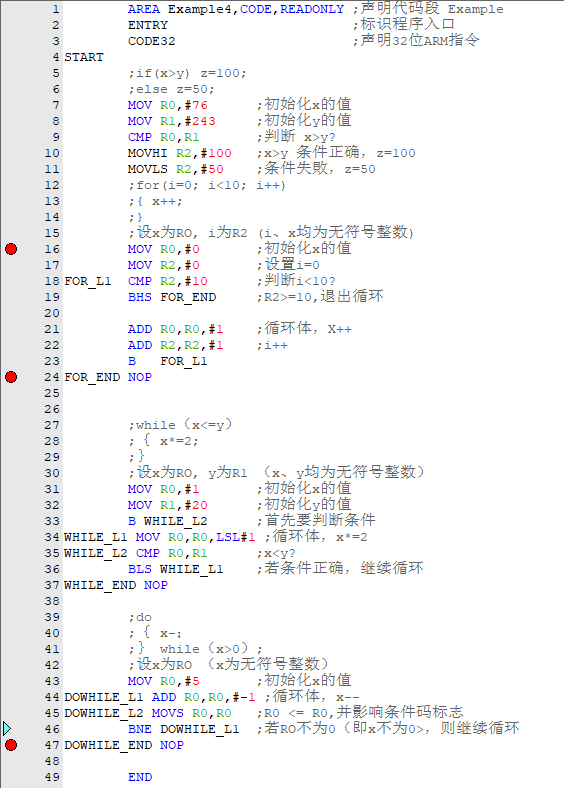
**1.4掌握ARM各种逻辑控制语句结构**  使用ARM汇编指令实现if条件执行

使用ARM汇编指令实现for循环结构

使用ARM汇编指令实现while循环结构

使用ARM汇编指令实现do...while循环结构

验证代码:



**CMP**：比较指令使用寄存器Rn的值减值operand2的值,根据操作的结果理新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

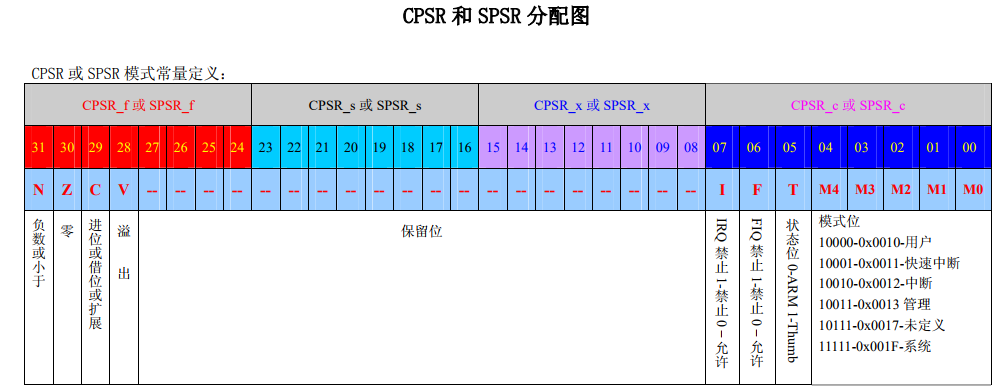


CMP指令与SUBS指令的区别在于CMP指令不保存运算结果.在进行两个数据大小判断时,常用CMP指令及相应的条件码来操作.

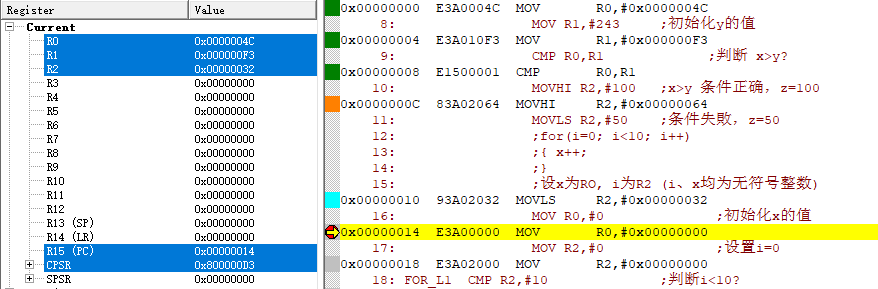
**BHS：**配合条件码的转跳指令，需要和状态寄存器配合

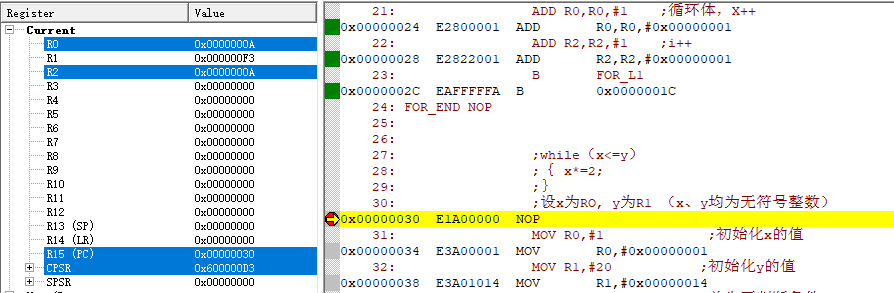
**BNE：**配合条件码的转跳指令，需要和状态寄存器配合

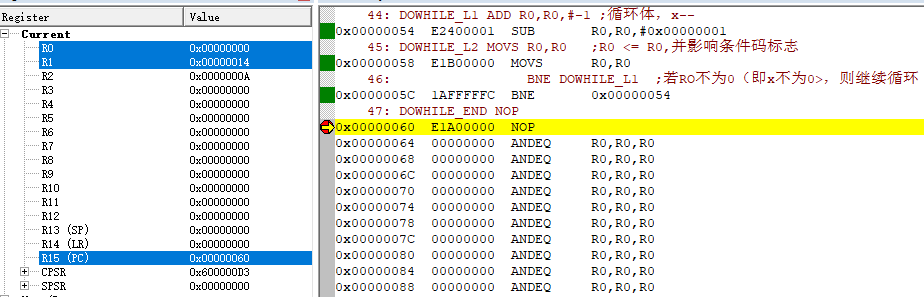




打上断点，调试仿真







**1.5ARM汇编编程-模式切换**

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

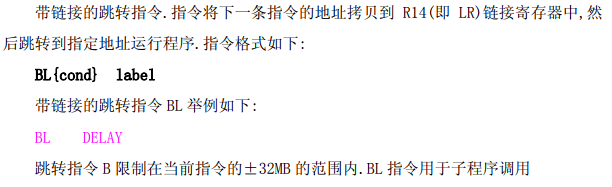
.\Objects\ARM7.axf: Error: L6221E: Execution region ER\_RO with Execution range [0x00000000,0x000000a8) overlaps with Execution region ER\_ZI with Execution range [0x00000000,0x00000240).



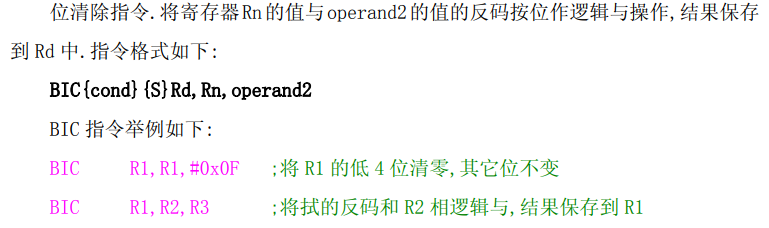
原因暂未知，应该是有未声明的变量



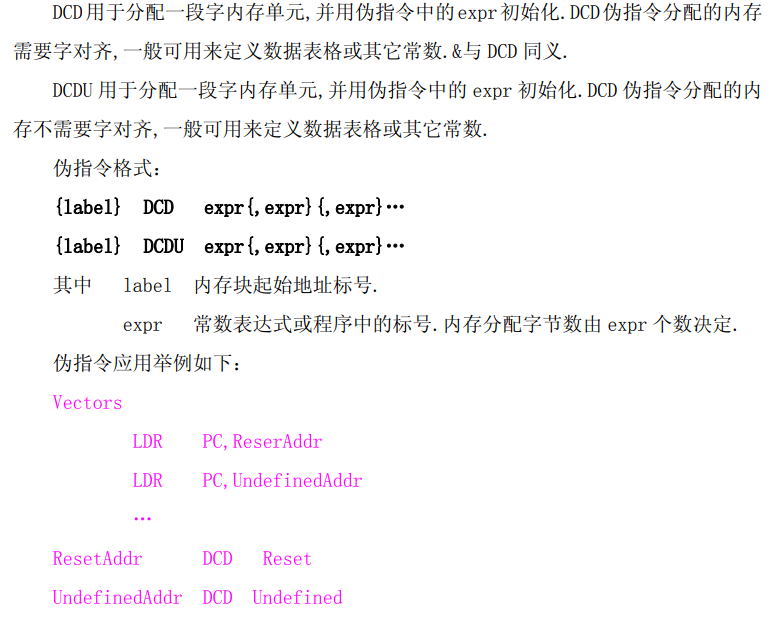
**BL：转跳指令**



**BIC：位带清除指令**

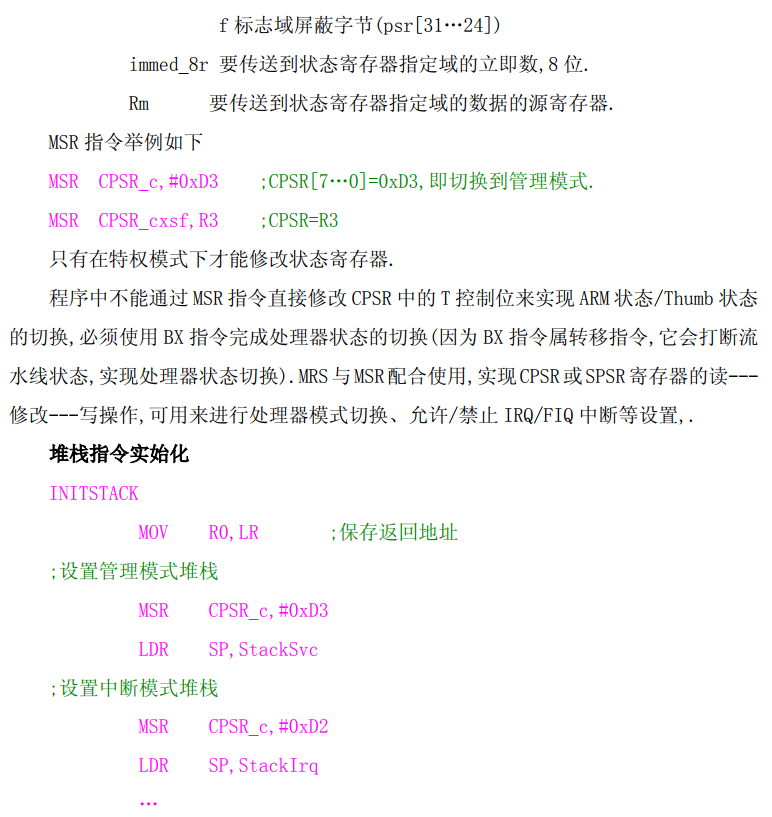


**DCD：**



**MSR：写状态寄存器**

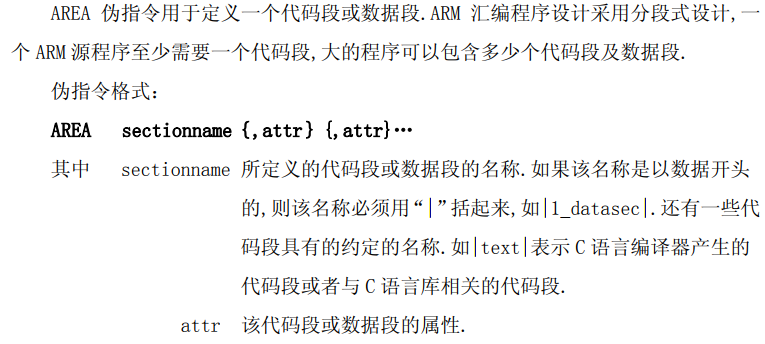


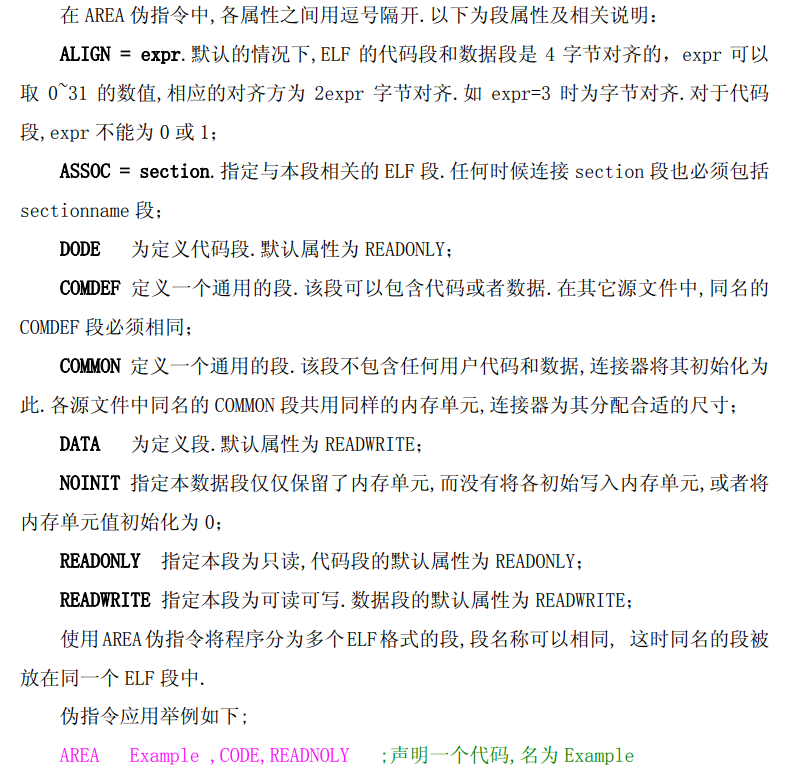


**LDR：加载**

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**





**SPACE：分配地址空间**



**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；

**1.6ARM汇编编程-模式切换**

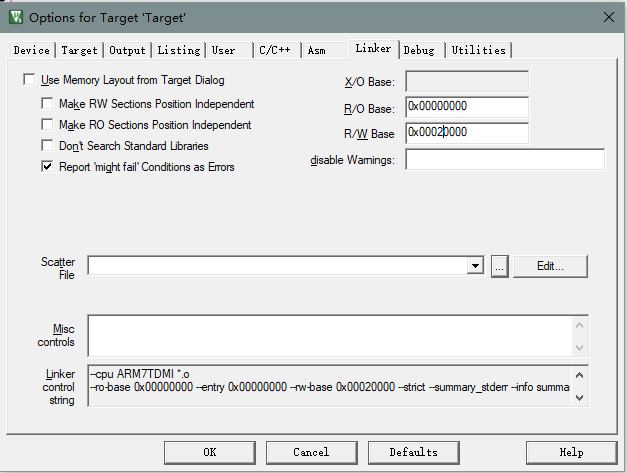
掌握ARM汇编指令与C语言的混合编程, 编写一个汇编程序文件Startup.S和一个C程序文件Test.c。汇编程序的功能是初始化堆栈指针和初始化C程序的运行环境，然后调跳转到C程序运行，这就是一个简单的启动程序。C程序使用加法运算来计算1+2+3+4+…N

在工程中添加并创建start.s main.c 并添加代码如下:

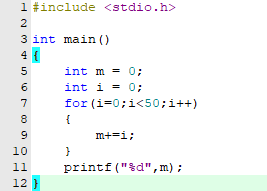
start.s



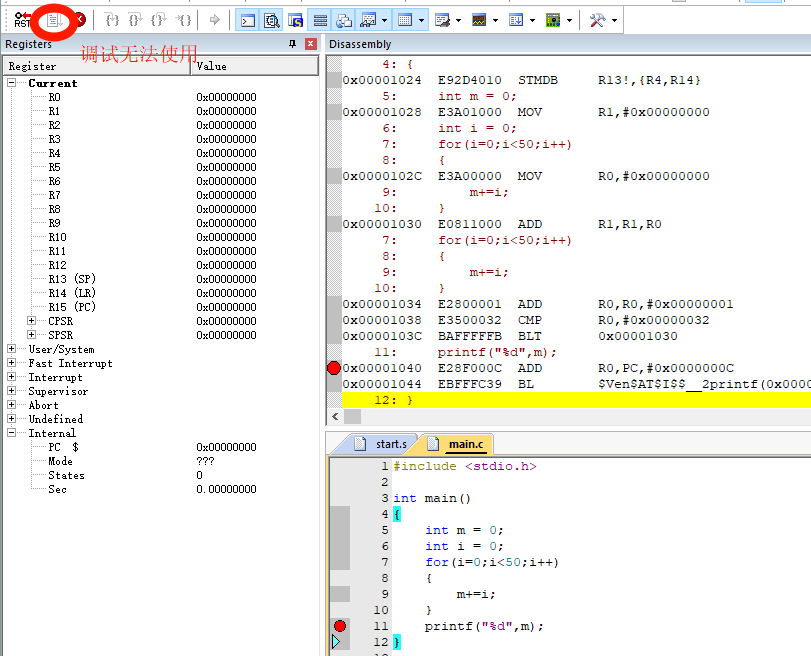
为RO标定起始地址



main.c



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，Zl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Zl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:

|lmage$$RO$$Base |: RO段起始地址

|lmage$$RO$$Limit|: RO段结束地址加1

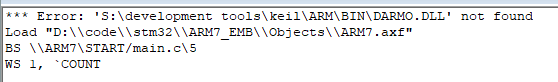
|lmage$$RW$$Base |: RW段起始地址

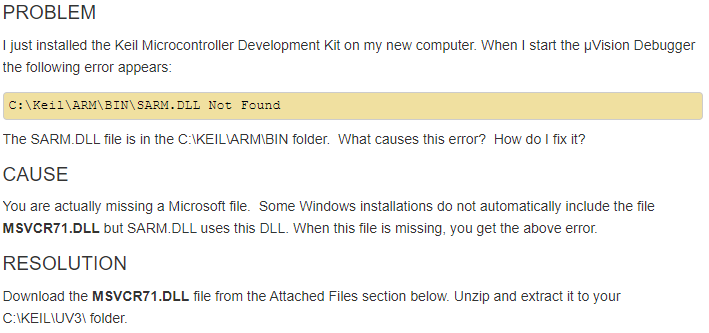
|lmage$$RW$$Limit|: Zl段结束地址加1

|lmage$$ZI$$Base |: Zl段起始地址

|lmage$$ZI$$Limit|: Zl段结束地址加1

启用调试会出现错误如下：在[keil官网](https://www.keil.com/support/docs/3192.htm)找到解决办法如下





[下载](https://www.keil.com/support/attachment/3298.htm)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

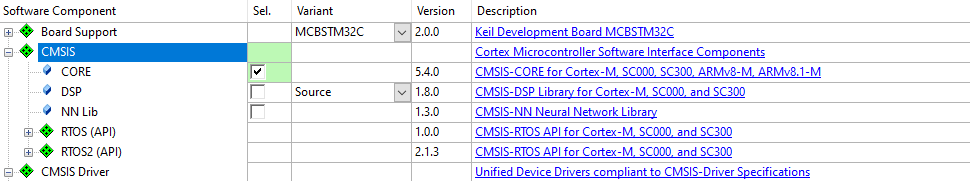
卡在某个点无法运行

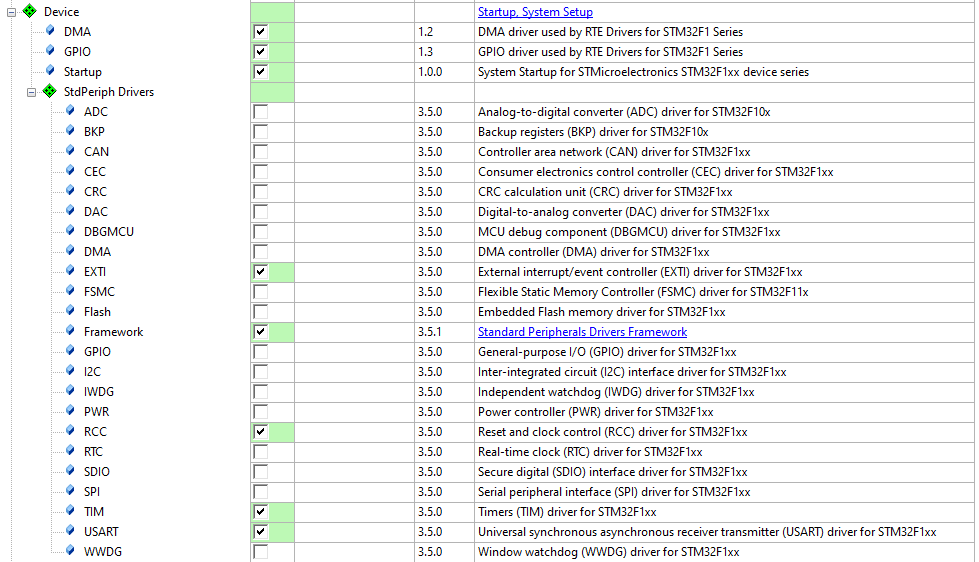


**二．仿真实验**

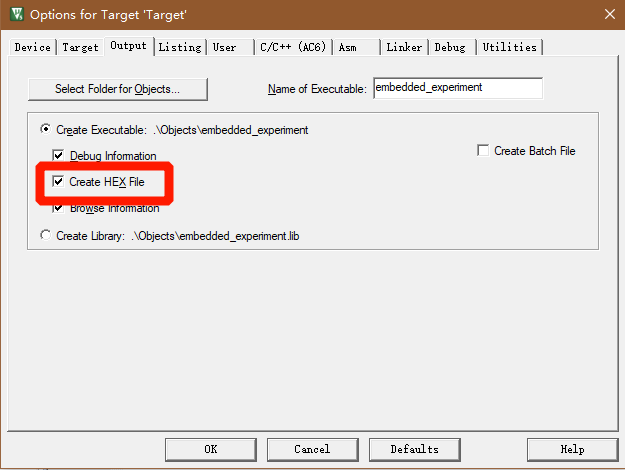
1. **前置配置**

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

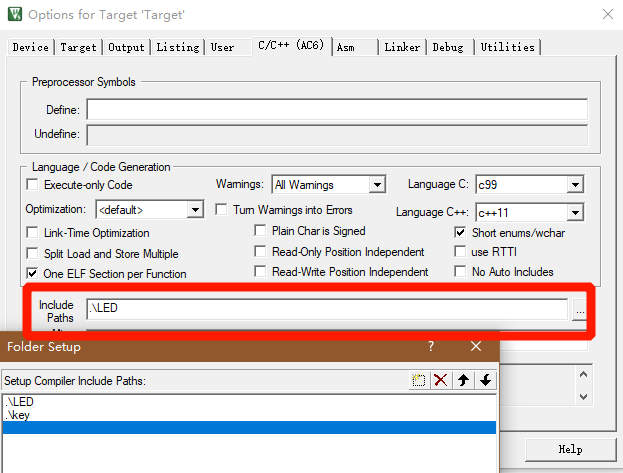




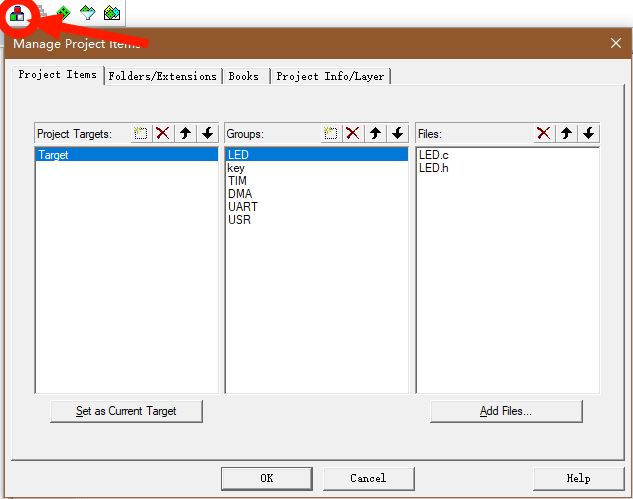
设置输出HEX文件



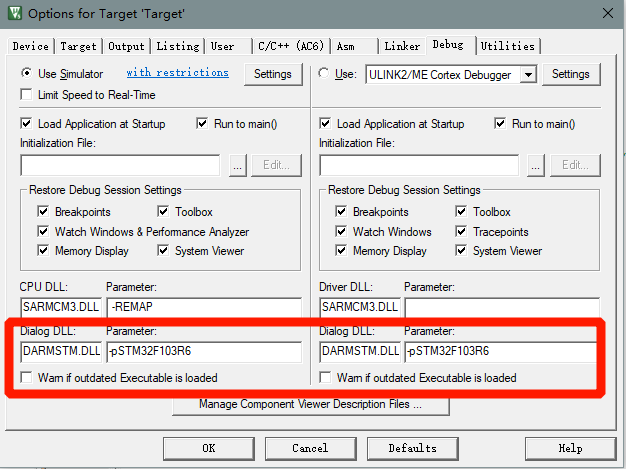
在include\_path设置中添加.h头文件的路径



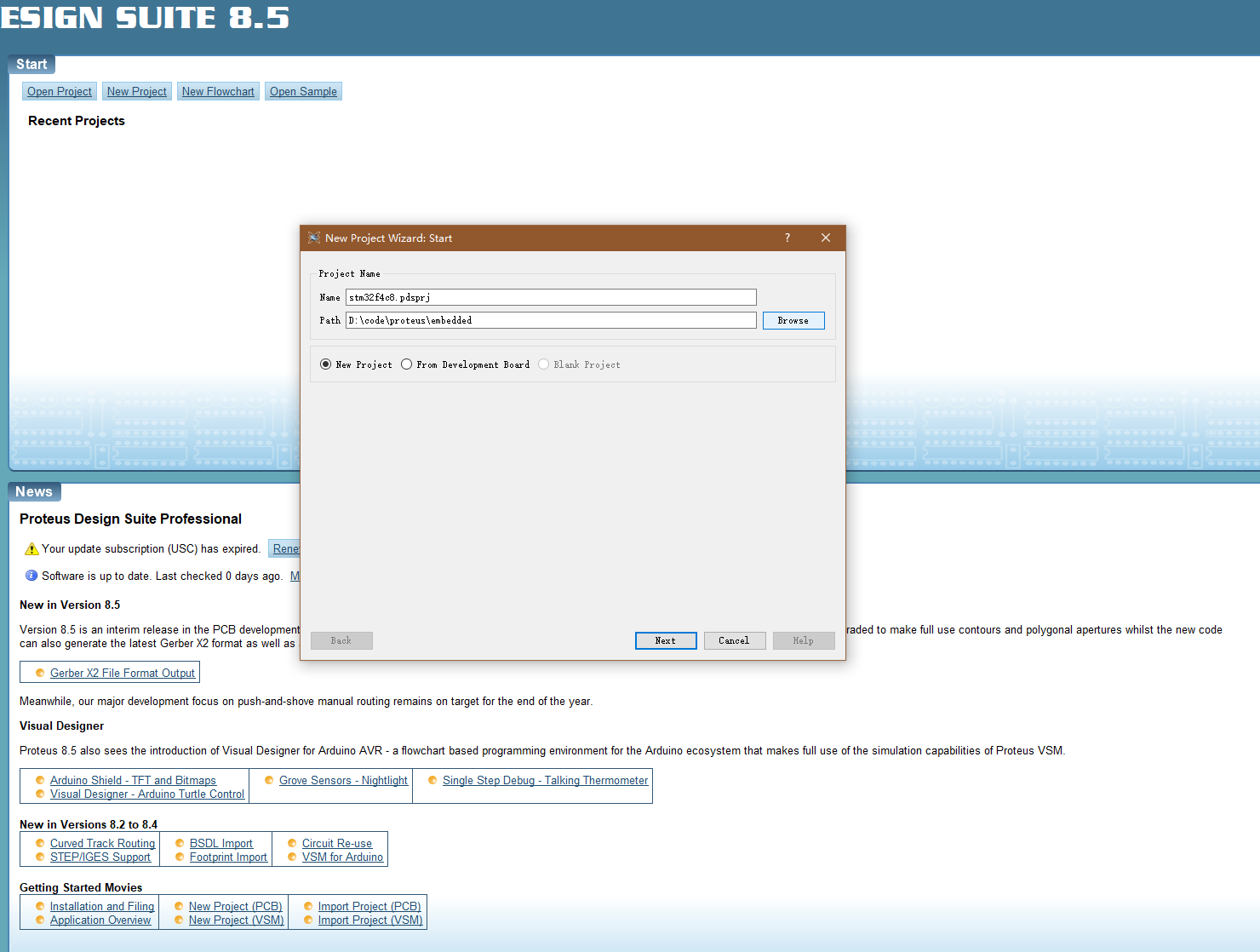
将源文件导入到工程中去



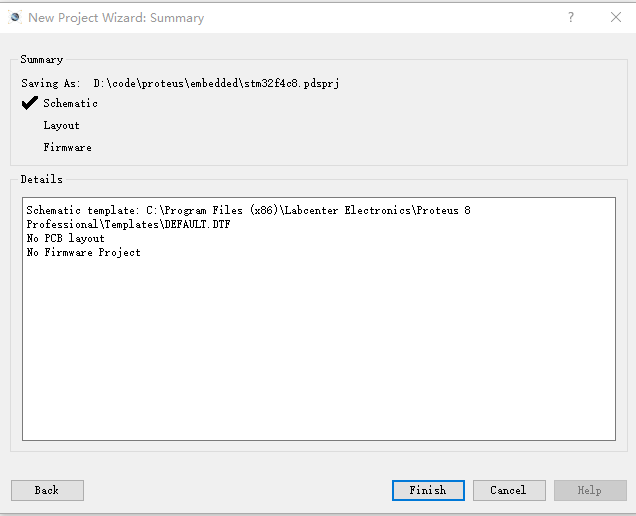
设置仿真



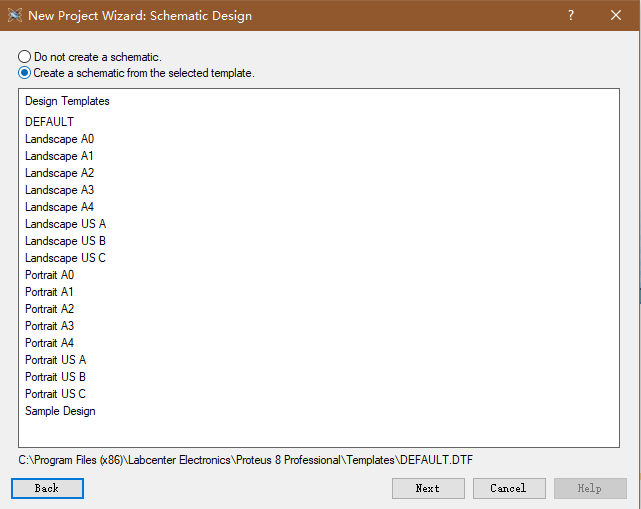
安装Protues8.9安装时要注意不要选择继承前版本设置(可能会导致出现库安装不完全的现象) 并建立 protues 工程,使用Cortex3内核仿真



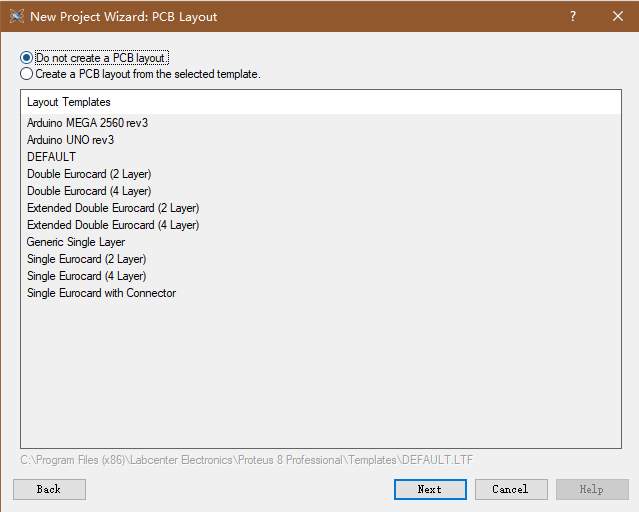
创建原理图



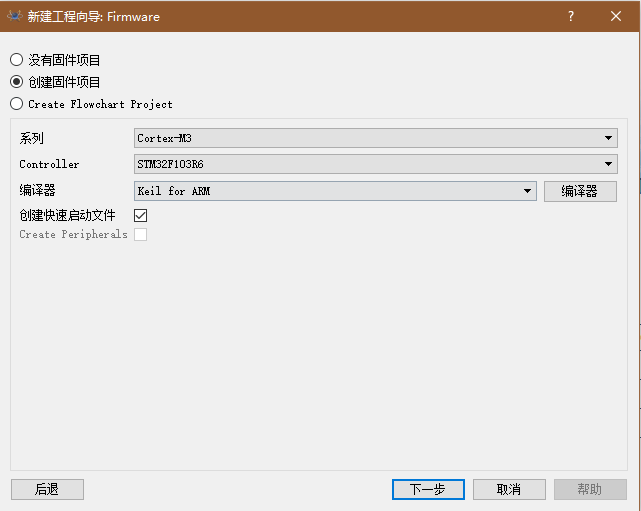
创建模板



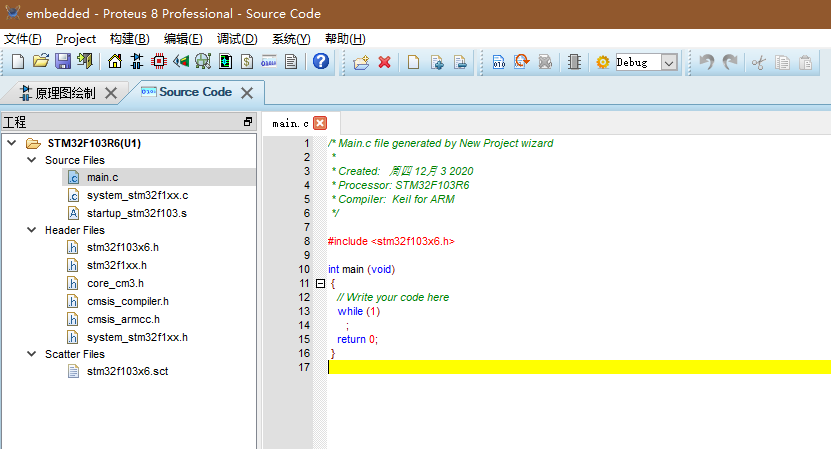
不创建PCB板



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



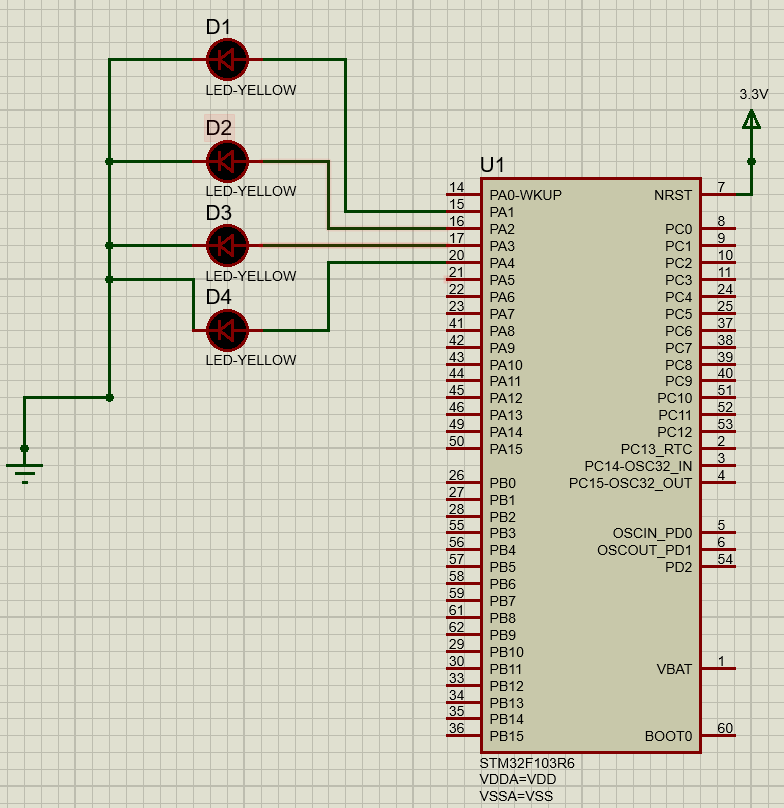
Arm7 与 Cortex3的比较



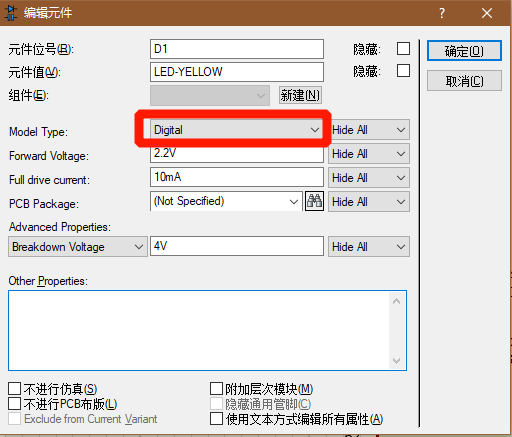
**2.实验验证**

**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）

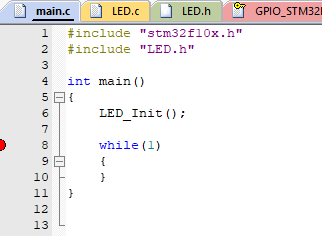
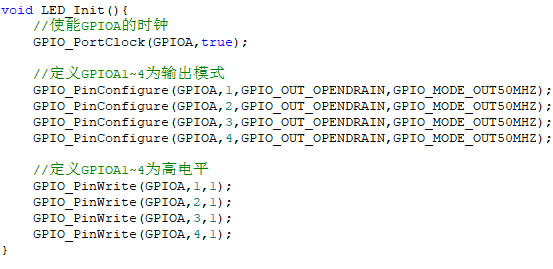


同时也需要在设置里配置供电网络，电源属性

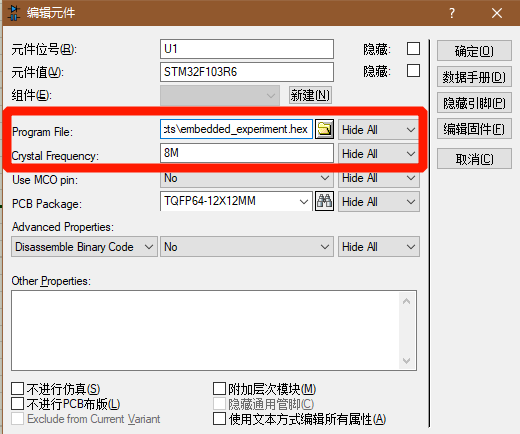
新建供电网络，并将其添加到3.3V逻辑电平上

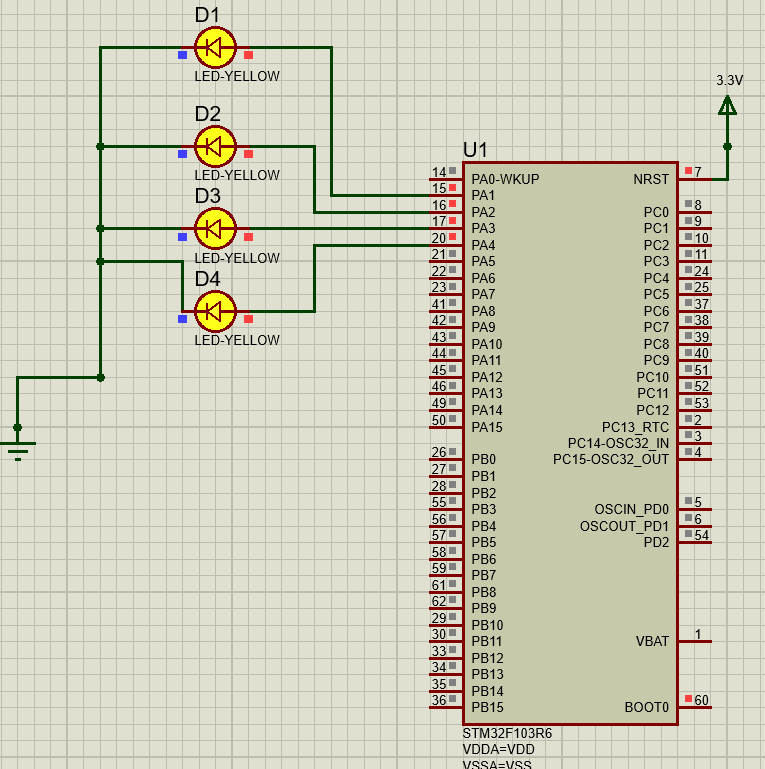


编写LED代码采用STM32F103R6 并使用RTE\_DeVice.h进行总控，编写LED\_Init()函数如下，并将其在main函数调用



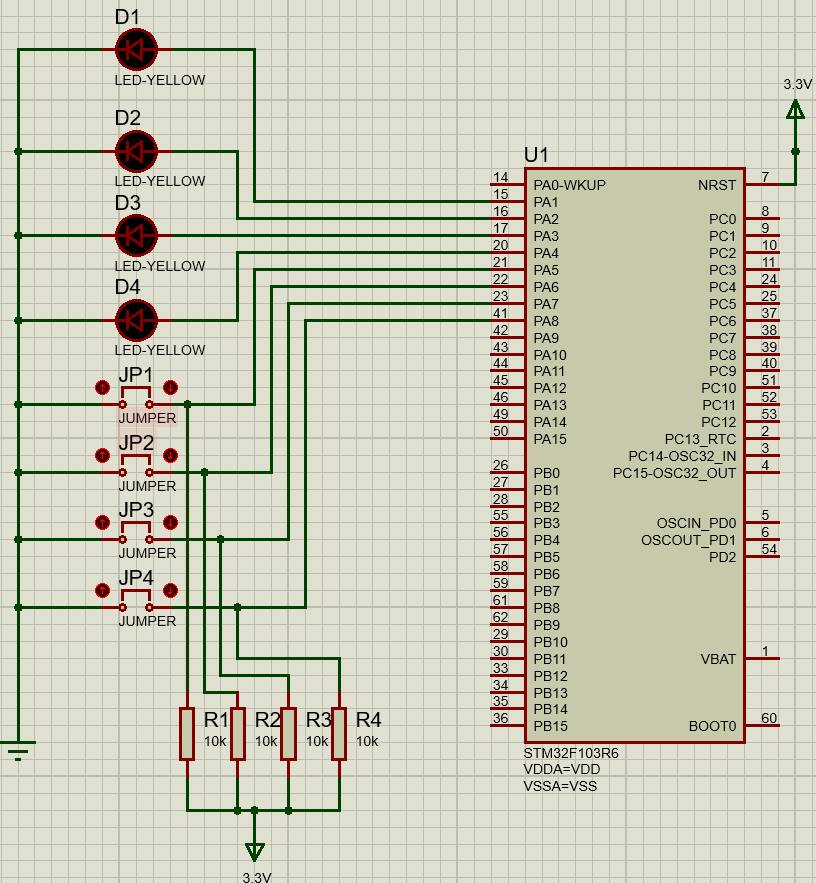
将编译好的hex文件载入proteus进行仿真可以看到LED灯被点亮





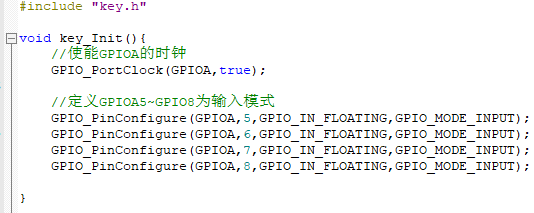
**2.2实现按键检测阵列**

**搭建电路图如下**

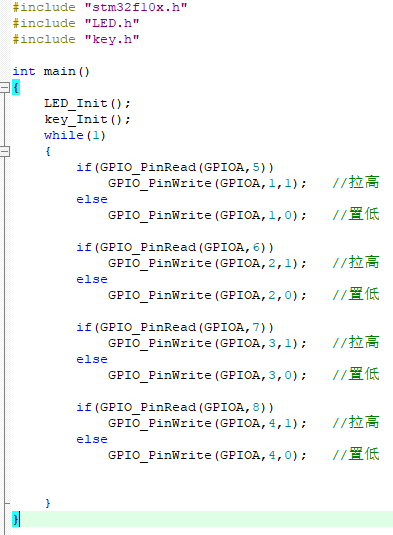


在2.1实验的基础上，绘制电路，并设定当JP1-JP4按下时D1-JP4熄灭

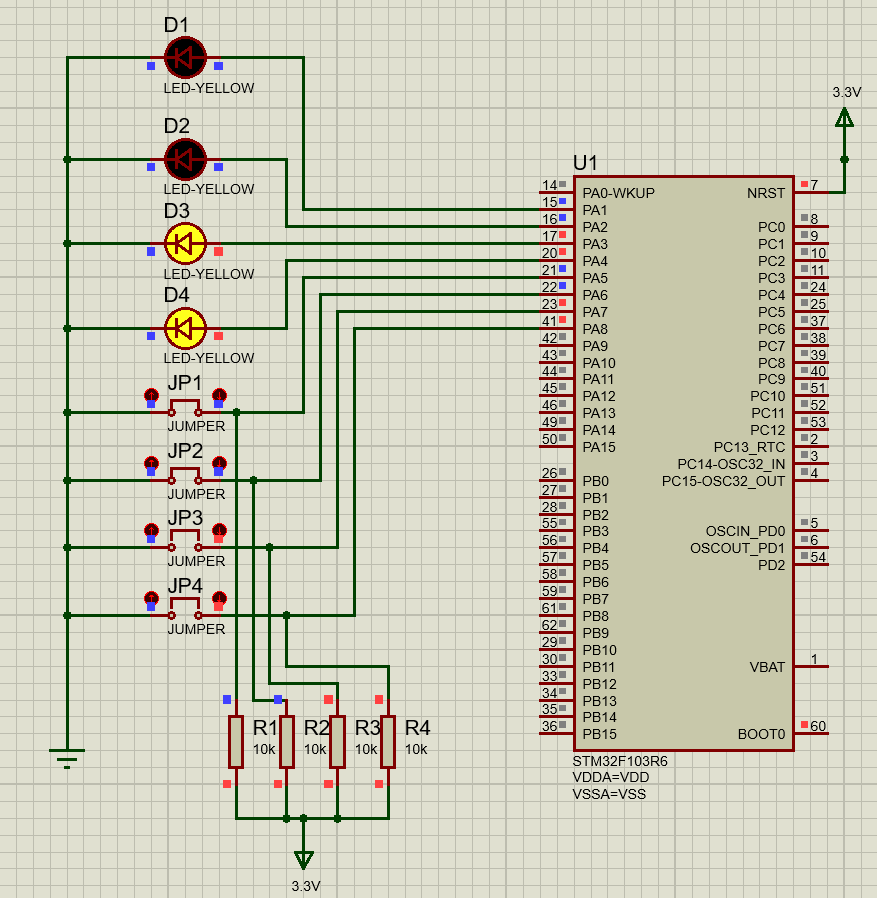
编写key.c代码，初始化并设定GPIO的输入属性



在man.c里实时检测key的状态并改变LED的状态



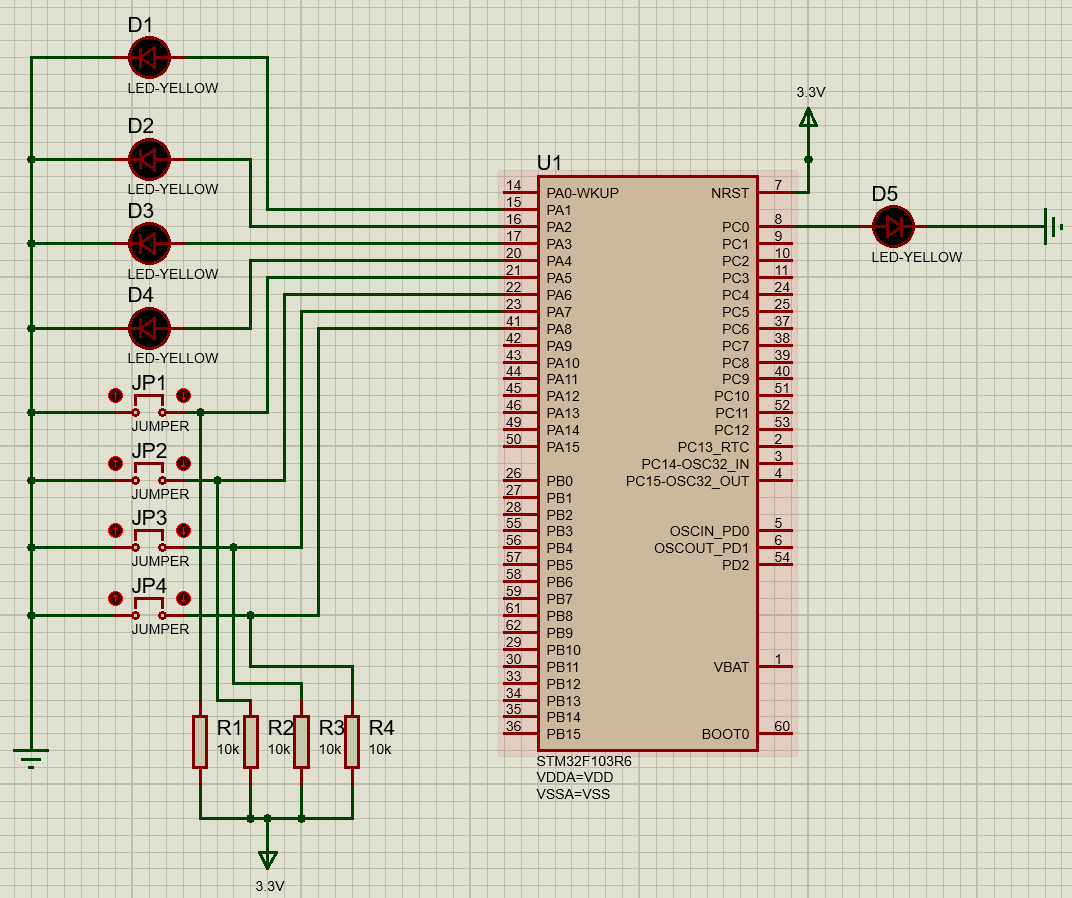
将编译好的hex文件载入proteus进行仿真可以看到当JP1-JP4按下时，LED1-LED4熄灭



**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。



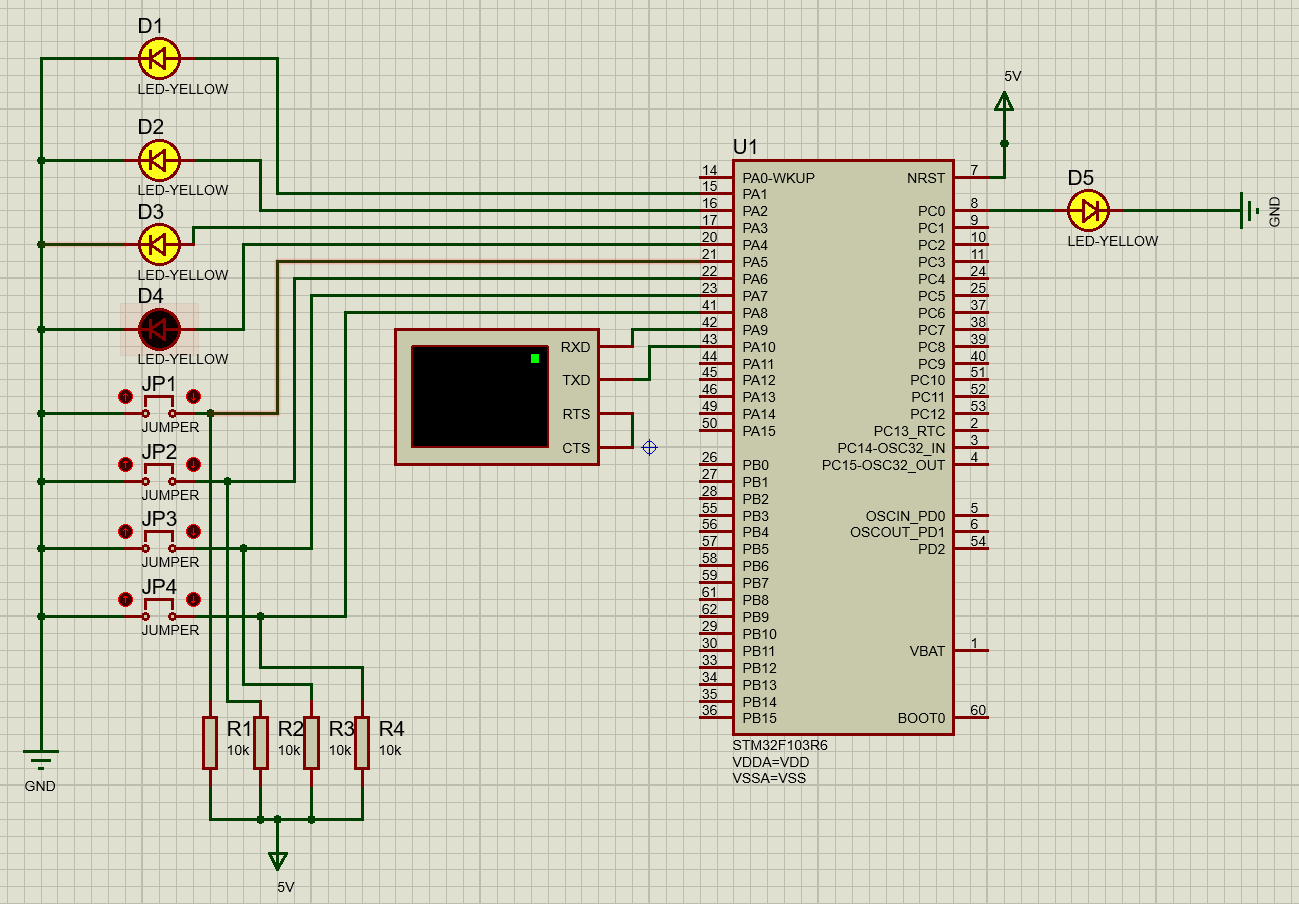
编写定时器初始化函数，并使能中断及设置中断优先级，编写中断服务函数。在中断服务函数中反转PC0的值实现LED闪烁的功能。

**具体实际效果可查看视频**

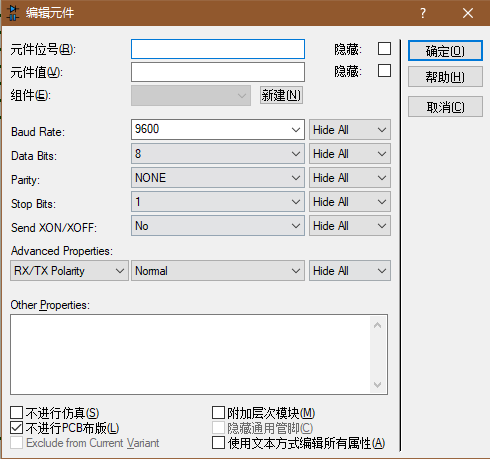


**2.4 UART串口通信**

在2.3的基础上，搭建电路图如下



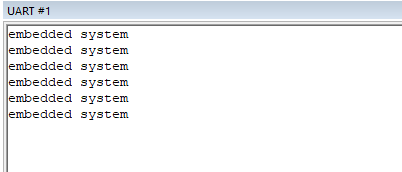
配置接收器属性如下

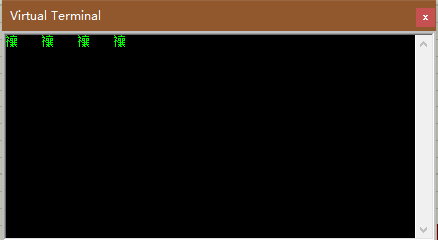


使用RTE\_DeVice.h控件进行初始化USART，并编写服务函数



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

（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\_convariance(float mean1,float mean2,int a[],int b[],int len)

{

float convariance;

int i;

for(i = 0;i<len;i++)

{

convariance = (a[i]-mean1)\*(b[i]-mean2);

convariance +=convariance;

}

convariance = convariance/len;

return convariance;

}

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,convariance34,coefficent34;

BubbleSort(a1,8);

mean3 = get\_mean(a3,12);

mean4 = get\_mean(a4,12);

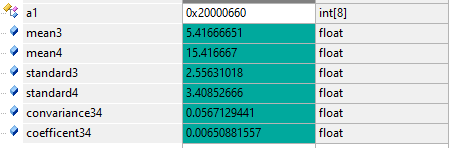
standard3 = get\_standard(a3,12,mean3);

standard4 = get\_standard(a4,12,mean4);

convariance34 = get\_convariance(mean3,mean4,a3,a4,12);

coefficent34 = convariance34/(standard3\*standard4);

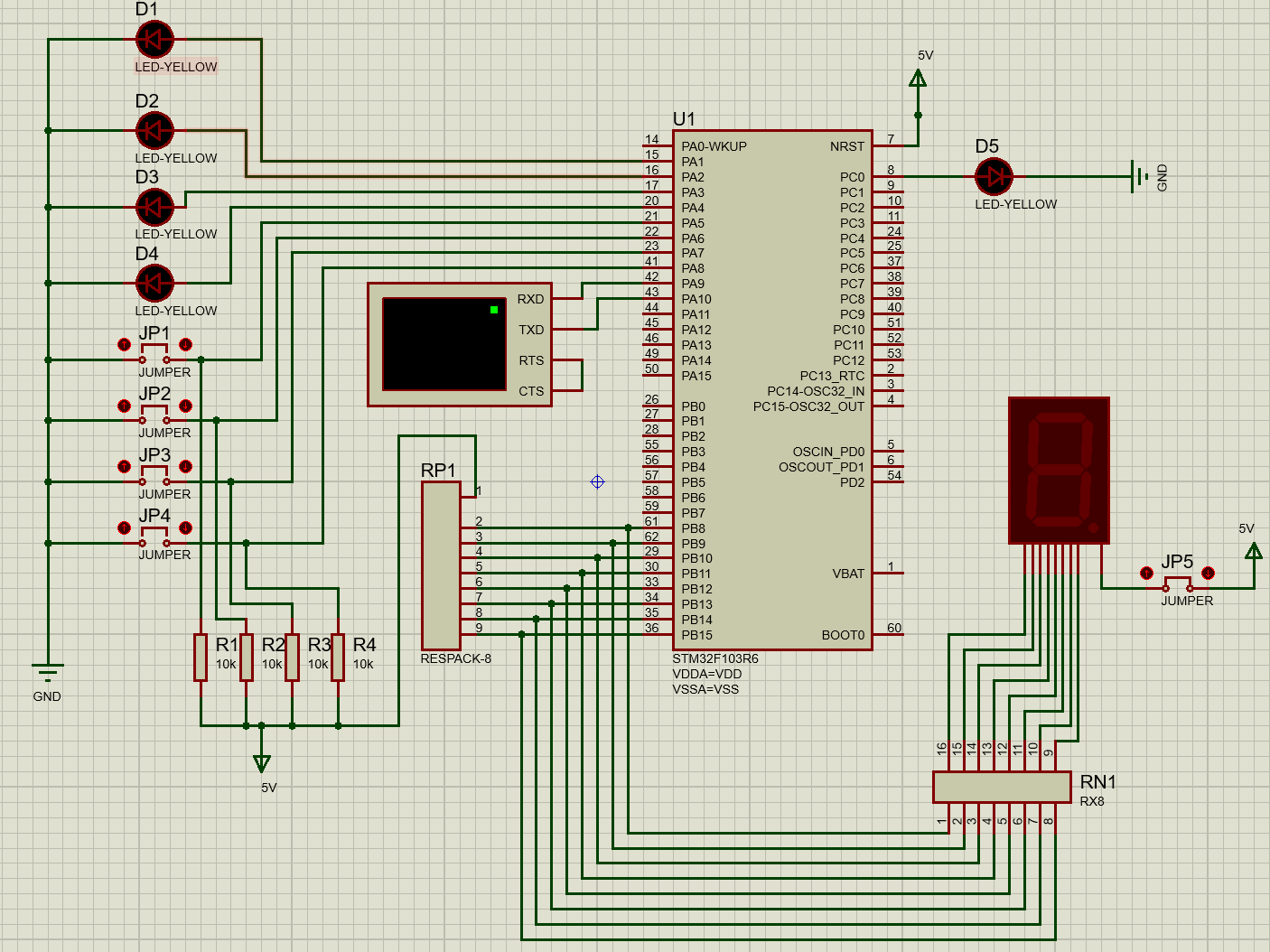
仿真结果如下

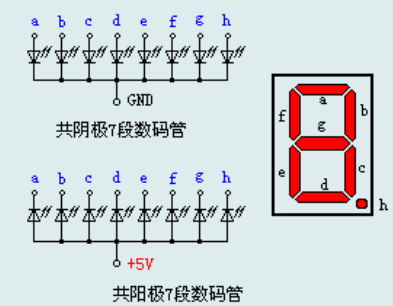


将字符串转存的汇编语句如下

**实验5:**

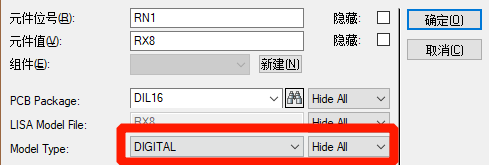
在前置电路的基础上搭建电路如下



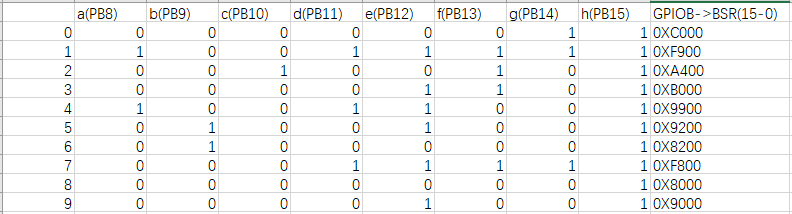


数码管可以分为共阳极与共阴极两种，共阳极就是把所有LED的阳极连接到共同接点com，使用时com接正5伏电源，而每个LED的阴极分别为a、b、c、d、e、f、g 及dp(小数点)﹔共阴极则是把所有LED的阴极连接到共同接点COM，使用时COM要将其接地。而每个LED的阳极分别为a、b、c、d、e、f、g 及dp (小数点)，8个LED的分布方式如图下图所示。图中的8个LED分别与上面那个图中的A~DP各段相对应，通过控制各个LED的亮灭来显示数字。

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



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





**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**

