# 电子科技大学信息与软件工程学院

# 实验指导书

（实验）课程名称：《ARM 处理器体系结构及应用》

**电子科技大学教务处制表**

# 实验 1　指令系统及寻址方式

实验所属系列:《ARM 处理器体系结构及应用》课内实验　　实验对象：本科

相关课程及专业：嵌入式软件　　实验时数（学分）：4 学时

实验类别　课内上机

实验开发教师：　兰刚

## 【实验目的】

(1) 熟悉 Keil MDK 开发环境，掌握开发平台 MDK -ARM 的使用；

(2) 能正确建立 ARM 汇编程序工程文件，掌握汇编程序调试方法；

(3) 熟练掌握 ARM 的指令系统以及指令寻址方式；

(4) 熟练掌握 ARM 汇编语言的程序框架，能正确编写 ARM 汇编程序。

## 【实验内容】

(1) 学习 Keil MDK-ARM 开发平台的的使用，包括新建一个工程、在建立的工程中编写相关程序。

(2) 掌握对 ARM 汇编工程编译、调试的方法。

(3) 对 40 多条常用的指令，使用不同的寻址方式进行测试，并通过查看寄存器、存储器、程序状态寄存器的内容，检查是否与期望一致。

(4) 针对这 40 多条指令中，可以加 S 标记和条件的指令，使用 S 标记和不同的条件进行测试，并通过查看寄存器、存储器、程序状态寄存器的内容，检查是否与期望一致。

## 【实验环境】

Keil MDK-ARM uVision5 开发工具。

## 【实验设备】

PC 机一台。

## 【实验原理】

### 1. MDK 使用简单步骤及注意事项：

(1) 如果 Keil 软件没有安装就先安装 Keil 软件；

(2) 启动 MDK，屏幕上是“Keil uVision4”图标；

(3) 在菜单中找到“project”并点击,在打开的菜单中点击“new uVision project”;

(4) 在弹出的菜单中选择所建项目的路径及项目名称，然后点击“保存”按钮;

(5) 在弹出的界面中选择 CPU，请选择 SAMSUNG 的 S3C2440A，点击“OK”按钮;

(6) 在弹出的界面中选“否”，不拷贝启动文件;

(7) 在菜单中点击“File”，点击“New”;

(8) 在窗口中输入源程序;

(9) 在菜单中点击“File”，点击“Save as”，选择保存文件的路径及文件名，注意后缀为.s;

(10) 展开 project，在源文件组上右击，在弹出的菜单中点击“添加文件到组”，把刚才建立的文件加入到组中;

(11) 在菜单中点击“project”，再点击“Rbuild all target files”，生成可执行文件;

(12) 在菜单中点击“debug”，点击“start/stop debug…”;

(13) 根据菜单提示可以单步运行程序，然后查看寄存器和存储器中的数据;

(14) 注意三星 S3C2440A 的数据存储器的起始地址为：0X40000000;

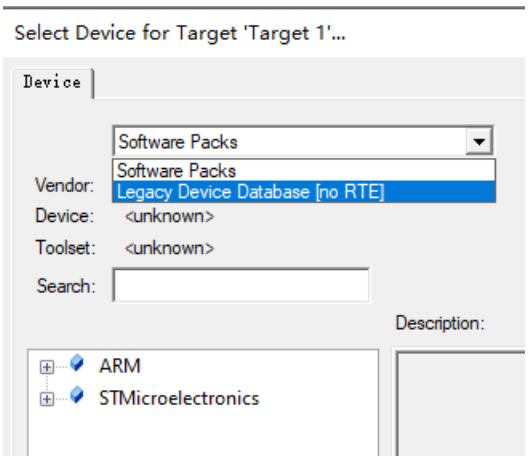
(15) 如果实验中涉及到有用 C 语言编程的，那就要将厂家的启动程序拷贝过来，将自己写的程序放在启动程序的最后，如果编译时提示有些变量没有定义，就把未定义的变量屏蔽掉。

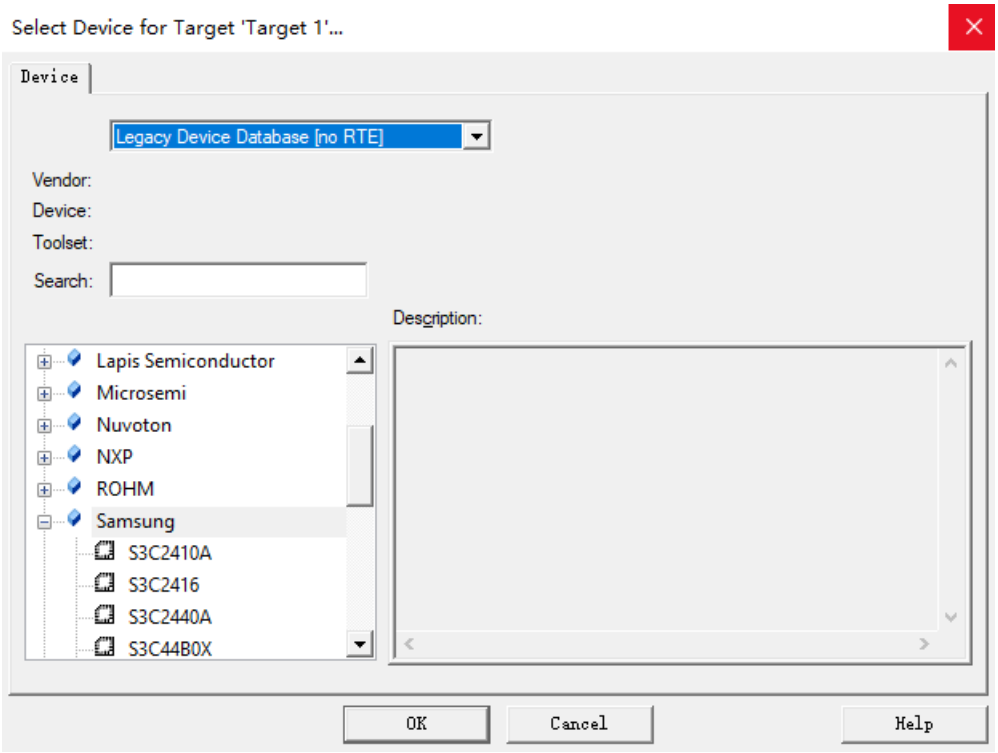
(16) 对于实验一，**不需要拷贝启动文件**。

## 2. 关于 Keil uVision5 及扩展支持包的安装问题

Keil uVision5（或 uVi4ion5）安装好以后可以不用破解，直接使用免费版，只是代码有 32K 的限制，但这对于实验足够了。

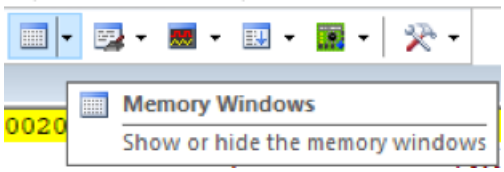
Keil uVision5 安装好以后，缺省是不包含 S3C2440 处理器芯片的，需要安装扩展支持包。扩展支持包见群文件“MDK79525(MDK5-ARM9 支持包)”。安装完后，在创建新工程的“器件选择”界面上，将会多出一个下拉菜单“Legacy Device Database[no RTE]”，如下图所示。选择该下拉菜单，就可以找到 s3c2440A 这个芯片。



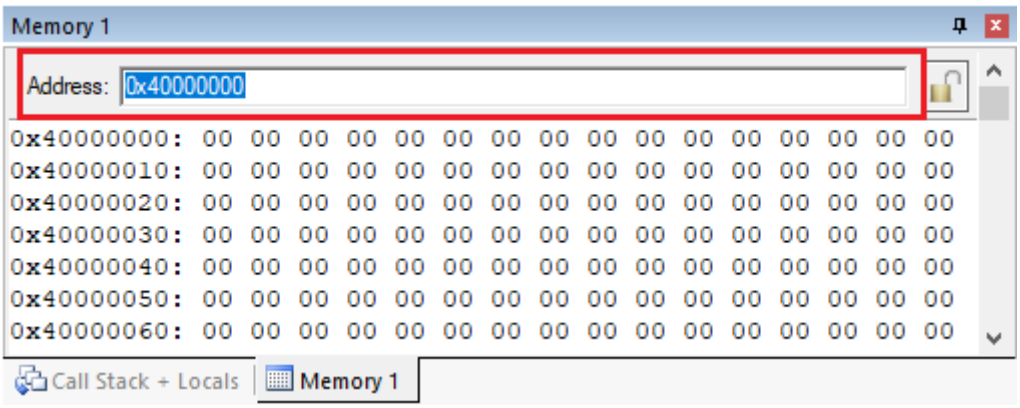




**3. 查看内存单元和 CPSR 标志位方法简述。**

(1) 点击“Memory Windows”按键，打开“Memory Windows” 观察窗，如果已将打开则可以忽略该步骤；注：该窗口及相关按钮，必须在调试状态下才可见。

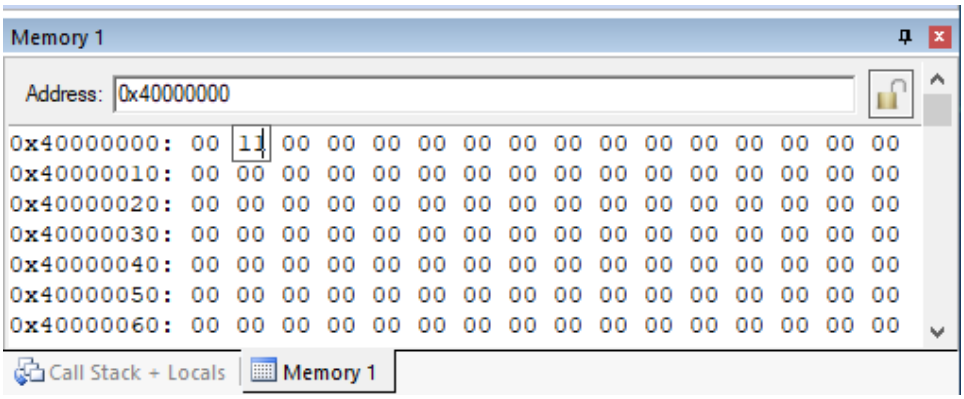



(2) “Memory Windows”观察窗打开后，界面如下（一般在整个界面的右下角）：

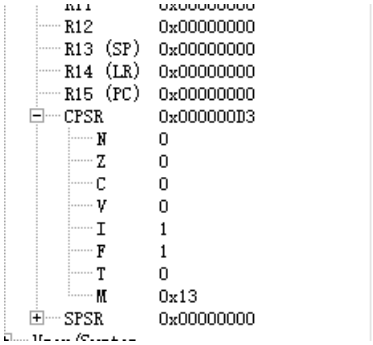



可以在“address”地址栏输入要观察的内存单元的起始地址，以便对指定地址单元进行观察。

也可以双击某单元内容，在该观察窗内直接手动修改某地址单元的内容。如下图所示。

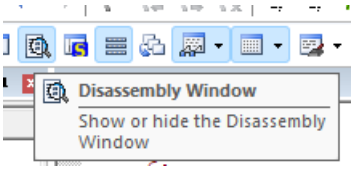


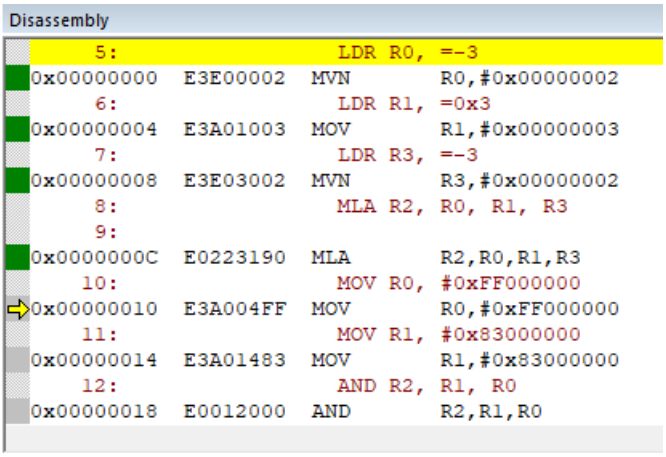

查看 CPSR 标志位时，可以点击 CPSR 寄存器旁的“+”号，把 CPSR 寄存器中的内容展开，按位查看。如下图所示：

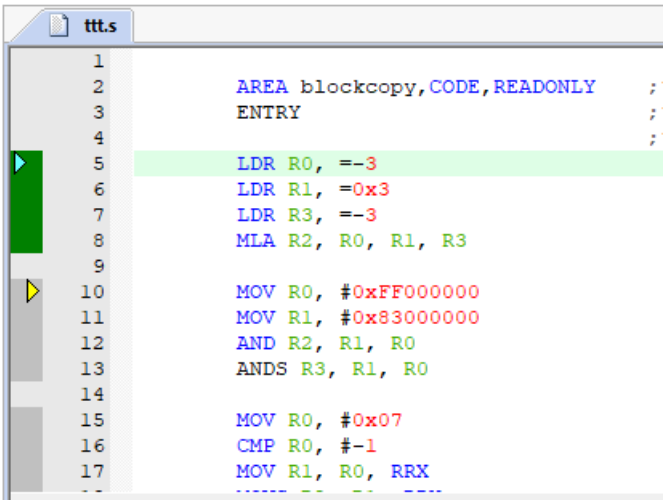



## 4. 查看反汇编后的代码

打开 Disassembly 窗口，就可以查看实际机器码及其对应的反汇编后的指令。

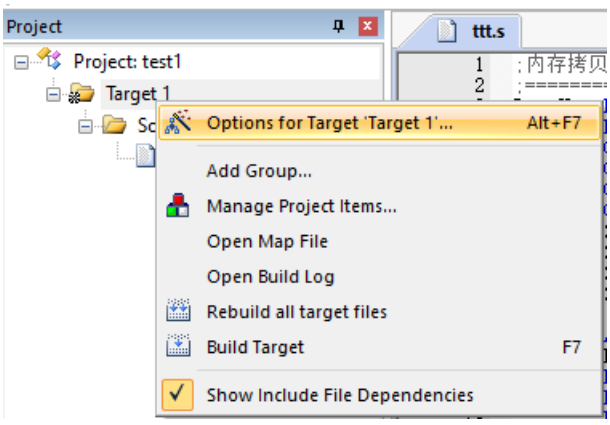


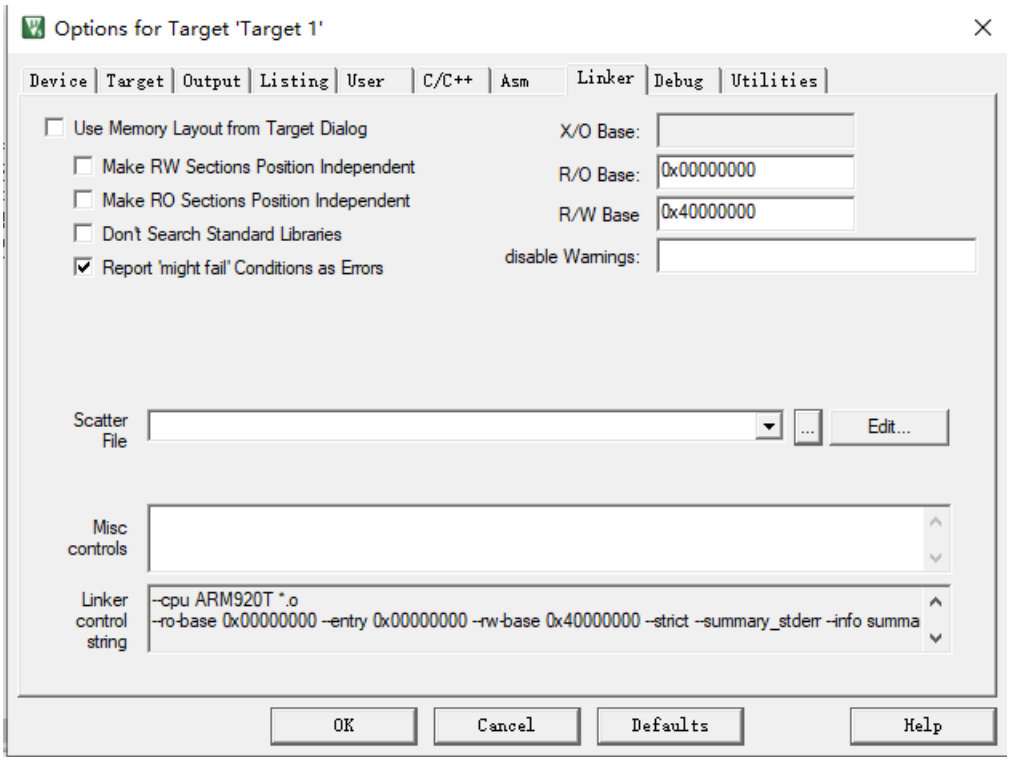




## 5. 关于目标选项“Options for Target”相关说明：

“Options for Target”主要用于配置与目标系统相关的软硬件环境，包括使用的处理器、存储配置、编译选项等。通过点击“Options for Target”菜单（或者使用 Alt+F7 快捷键）就可以进入“Options for Target”配置窗口。



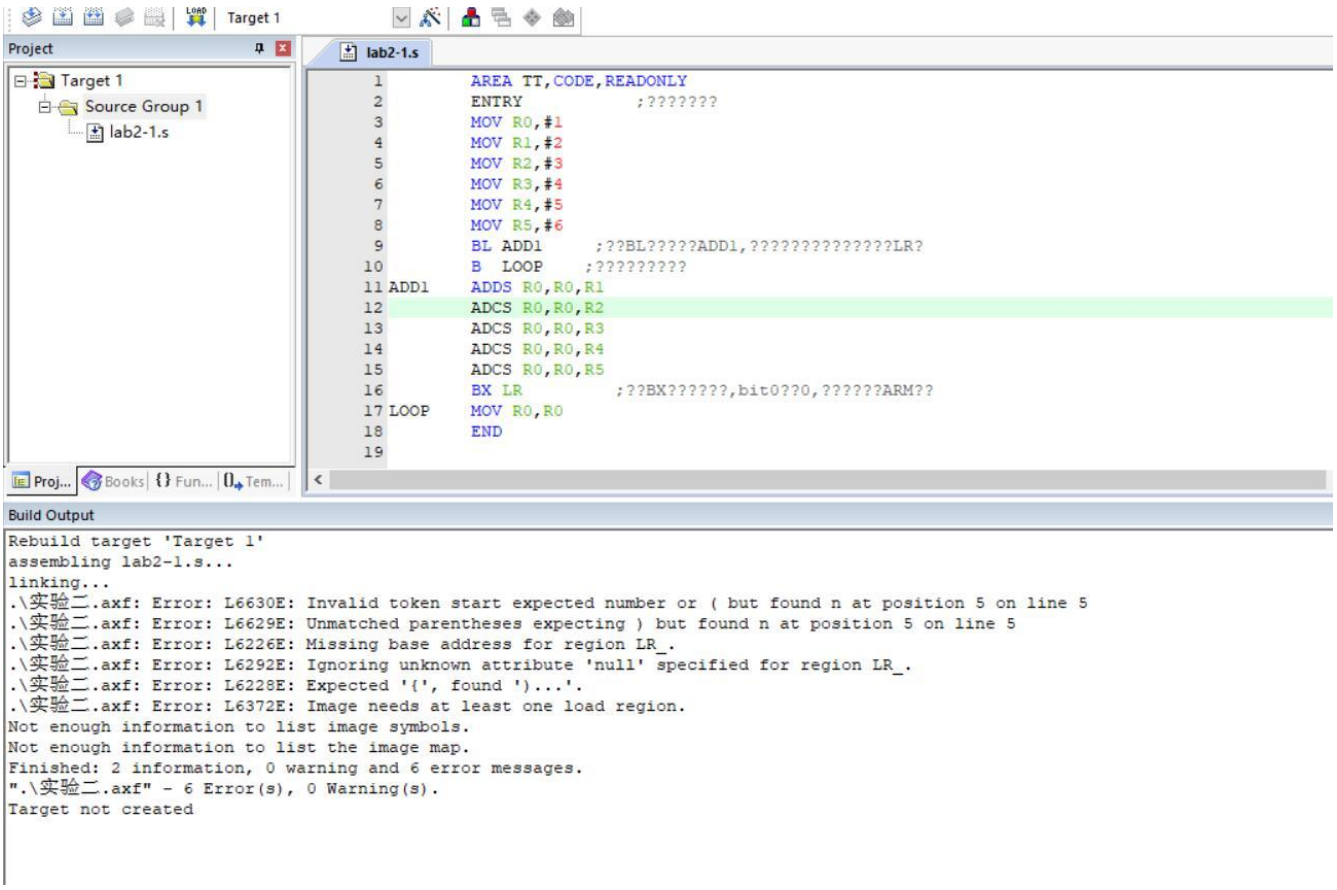
相关的配置界面如下：



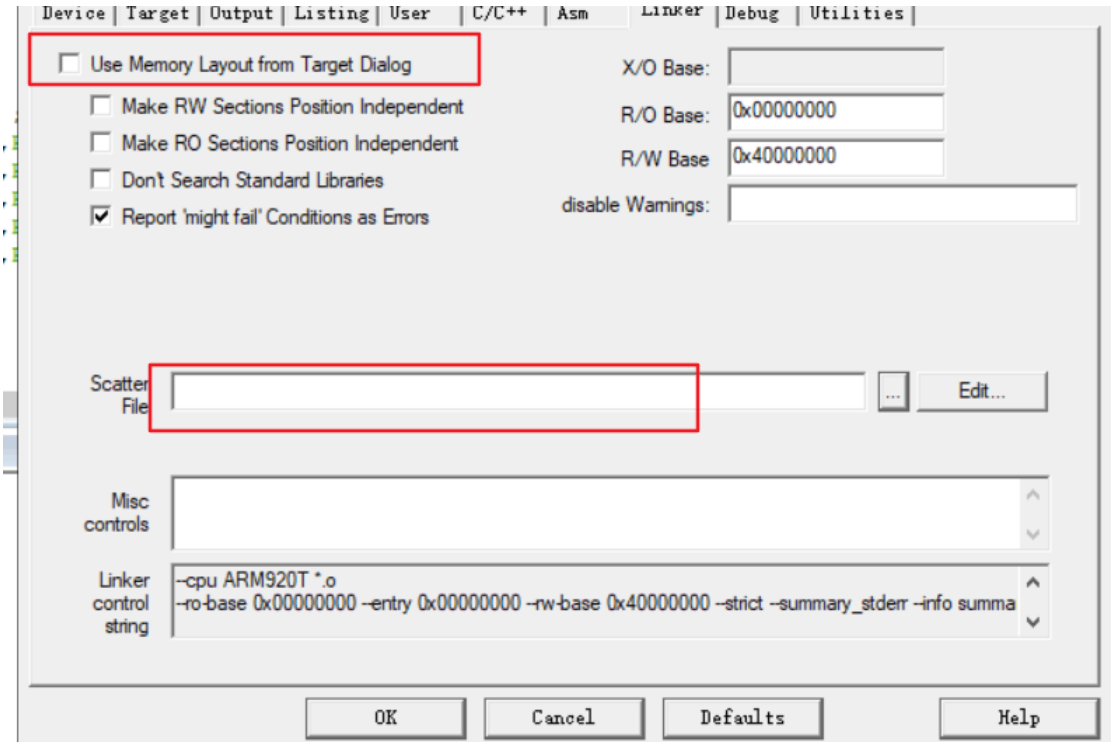

**对于实验一，使用缺省的配置，不需要对相关配置做任何更改！**

## 6. 相关错误提示及更正

做实验一是，有时会报以下错误：下面的程序报了这样一堆错误，重新新建工程后还是如此，解决方法是：




把 Linker 中的这两项删除后，就可以了




## 7. 指令系统寻址方式

(1) **立即数寻址：**也叫立即寻址，操作数本身就在指令中给出，取出指令也就取到了操作数。这个操作数被称为立即数，对应的寻址方式也就叫做立即数寻址。

(2) **寄存器寻址：**就是利用寄存器中的数值作为操作数，这种寻址方式是各类微处理器经常采用的一种方式，也是一种执行效率较高的寻址方式。

(3) **寄存器移位寻址：**当第二操作数为寄存器型时，在执行寄存器寻址操作时，也可以对第二操作数寄存器进行移位，此时第二操作数形式为。

(4) **寄存器间接寻址：**就是以寄存器中的值作为操作数的地址，而操作数本身存放在存储器中。

(5) **基址变址寻址：**将基址寄存器的内容与指令中给出的地址偏移量相加，得到操作数所在的存储器的有效地址。变址寻址方式常用于访问某基地址附近的地址单元。（4K 范围的偏移）。有三种加偏址的方式：前变址、自动变址和后变址寻址方式。

a) 前变址模式：先基址+偏址，生成操作数地址，再做指令指定的操作。也叫前索引偏移。

b) 自动变址模式：先基址+偏移，生成操作数地址，做指令指定的操作。

c) 后变址模式：即先用基地址传数，然后修改基地址（基址+偏移），也叫后索引偏移。

(6) **多寄存器寻址：**采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送。这种寻址方式是多寄存器传送指令 LDM/STM 的寻址方式，这种寻址方式中用一条指令最多可传送 16 个通用寄存器的值。连续的寄存器间用“-”连接，否则用“,”分隔。然后自动修改基址寄存器。

(7) **堆栈寻址：**是对堆栈进行操作的寻址方式堆栈寻址是隐含的，它使用一个专门的寄存器（堆栈指针 SP）指向一块存储区域(堆栈)。四种类型的堆栈工作方式：

a) 满递增堆栈 FA(Full Ascending)：堆栈指针指向最后压入的数据，且由低地址向高地址生长。

b) 满递减堆栈 FD(Full Descending ）：堆栈指针指向最后压入的数据，且由高地址向低地址生长。

c) 空递增堆栈 EA(Empty Ascending)：堆栈指针指向下一个将要放入数据的空位置，且由低地址向高地址生长。

d) 空递减堆栈 ED(Empty Descending)：堆栈指针指向下一个将要放入数据的空位置，且由高地址向低地址生长。

(8) **相对寻址：**以程序计数器 PC 的当前值为基地址，指令中的地址标号作为偏移量，将两者相加之后得到操作数的有效地址。

**【实验报告】**

用立即数寻址、寄存器寻址、寄存器间接寻址、寄存器移位寻址、基址变址寻址、堆栈寻址、多寄存器寻址、相对寻址，加 S 标志，加条件来使用以下指令，并通过查看寄存器、存储器、程序状态寄存器的内容，检查是否与期望一致。(注：不是每条指令都需要使用所有的寻址方式进行验证，每条指令可以选择 2-3 种寻址方式进行验证，但是所有的指令都需要验证，所有的寻址方式，都需要有指令对其进行验证。)

（1）MOV 指令；（2）MVN 指令；（3）AND 指令；（4）EOR 指令；（5）SUB 指令；（6）RSB 指令；（7）ADD 指令；（8）ADC 指令；（9）SBC 指令；（10）RSC 指令；（11）TST 指令；（12）TEQ 指令；（13）CMP 指令；（14）CMN 指令；（15）ORR 指令；（16）BIC 指令；（17）MUL 指令；（18）MLA 指令；（19）UMULL 指令；（20）UMLAL 指令；（21）SMULL 指令；（22）SMLAL 指令；（23）LDR 指令；（24）STR 指令；（25）LDRB 指令；（26）STRB 指令；（27）LDRH 指令；（28）STRH 指令；（29）LDMIA 指令；（30）LDMIB 指令；（31）LDMDA 指令；（32）LDMDB 指令；（33）LDMFA 指令；（34）STMFA 指令；（35）LDMFD 指令；（36）STMFD 指令；（37）LDMEA 指令；（38）STMEA 指令；（39）LDMED 指令；（40）STMED 指令；（41）SWP 指令；（42）B 指令；（43）BL 指令；（44）BX 指令；（45）BLX 指令；（46）MRS 指令；（47）MSR 指令。（注：其中有一条指令指令集中有，但是在 MDK 中不支持，大家测试找出。）

试验完成，填写下表

| 序号 | 指令 | 寻址方式或后缀 | 具体测试代码（不是完整程序，只需包含测试相关的代码） | 结果说明 |
|---|---|---|---|---|
| 1 | MOV | 立即数寻址+S 条件后缀 | MOVS R2，#0x0 | R2 ← #0x0<br>并影响标志位：nZcv(注：字母大写，表示相关标志位为 1) |
| | | 寄存器寻址 | CMP R1，R0 | 如果 R1=R0 |

| | | | | |
|---|---|---|---|---|
| | | +EQ 条件后缀 | MOVEQ R2，R0 | 则执行 R2 ← R0，否则不执行 |
| | | 寄存器移位寻址<br>寄存器移位寻址 | MOV R0，#0x01<br>MOV R1，R0，LSL#3<br>MOVS R1，R0，LSL#3 | 将寄存器 R0 左移 3 位传送到寄存器 R1。本例执行后，R1=0x08 |
| | | | MOV R0, #0x0f<br>MOV R6, R0, LSR #0x01<br>MOVS R6, R0, LSR #0x01<br>MOVS R6, R0, LSR #0x05 | 比较两条 MOV 位移指令，看看带 S 标记和不带 S 标记的不同指令,对标志位的影响。 |
| | | | MOV R0，#0x0F<br>MOV R0，R0，RRX<br>MOVS R0，R0，RRX | 比较两条 MOV 位移指令，看看带 S 标记和不带 S 标记的不同指令,对标志位的影响。 |
| 2 | MVN | | | |
| | | | | |
| | | | | |

# 实验 2　数字滤波实验

实验所属系列:《ARM 处理器体系结构及应用》课内实验　　实验对象：本科

相关课程及专业：嵌入式软件　　实验时数（学分）：4 学时

实验类别　课内上机

实验开发教师：　兰刚

## 【实验目的】

(1) 通过工程中常用的数字滤波编程实现，验培养学生分析问题、解决问题能力。

(2) 通过该编程，进一步巩固和强化学生 ARM 汇编编程的能。

## 【实验内容】

(1) 中值滤波及编程实验，要求如下：

- ✧ 关于 N（N 为奇数）个数的值在程序中能任意、方便设置，并且放在 R0 中；
- ✧ 原始数据放在内存 0X40000000 开始的地址空间；
- ✧ 中值滤波的结果放在寄存器 R1 中；

(2) 均值滤波编程实验，要求如下：

- ✧ 关于 N（N 为偶数）个数的值在程序中能任意、方便设置，如 4、6、10、18 等，该值放在 R0 中；
- ✧ 原始数据放在内存中 0X40000000 开始的地址空间；
- ✧ 均值滤波的结果放在寄存器 R1 中；

## 【实验环境】

Keil MDK-ARM uVision5 开发工具。

## 【实验设备】

PC 机一台。

## 【实验原理】

**（1）中值滤波**

数字图像在其形成、传输记录的过程中往往会受到很多噪声的的污染，比如：椒盐噪声、高斯噪声等，为了抑制和消除这些随即产生的噪声而改善图像的质量，就需要去、对图像进行去滤波噪处理。

中值滤波是图像平滑的一种方法 它是一种非线性平滑滤波技术，在一定条件下可以克服线性滤波带来的图像细节的模糊问题，特别是针对被椒盐噪声污染的图像。中值滤波对脉冲噪声有良好的滤除作用，特别是在滤除噪声的同时，能够保护信号的边缘，使之不被模糊。这些优良特性是线性滤波方法所不具有的。此外，中值滤波的算法比较简单，也易于用硬件实现。所以，中值滤波方法一经提出后，便在数字信号处理领得到重要的应用。

中值滤波方法：对一个数字信号序列 xj(-∞<j<∞)进行滤波处理时，首先要定义一个长度为奇数的 L 长窗口，L=2N+1，N 为正整数。设在某一个时刻，窗口内的信号样本为 x(i-N)，…，x(i)，…，x(i+N)，其中 x(i)为位于窗口中心的信号样本值。对这 L 个信号样本值按从小到大的顺序排列后，其中值，在 i 处的样值，便定义为中值滤波的输出值。

**（2）均值滤波**

均值滤波也用于图像处理，均值滤波也称为线性滤波，其采用的主要方法为邻域平均法。线性滤波的基本原理是用均值代替原图像中的各个像素值，即对待处理的当前像素点（x，y），选择一个模板，该模板由其近邻的若干像素组成，求模板中所有像素的均值，再把该均值赋予当前像素点（x，y），作为处理后图像在该点上的灰度 g（x，y），即 g（x，y）=∑f（x，y）/m m 为该模板中包含当前像素在内的像素总个数。

本事实验要求，将 N 个无符号去掉最大值和最小值，剩余的数求平均值，平均值即为本次滤波所得的结果。

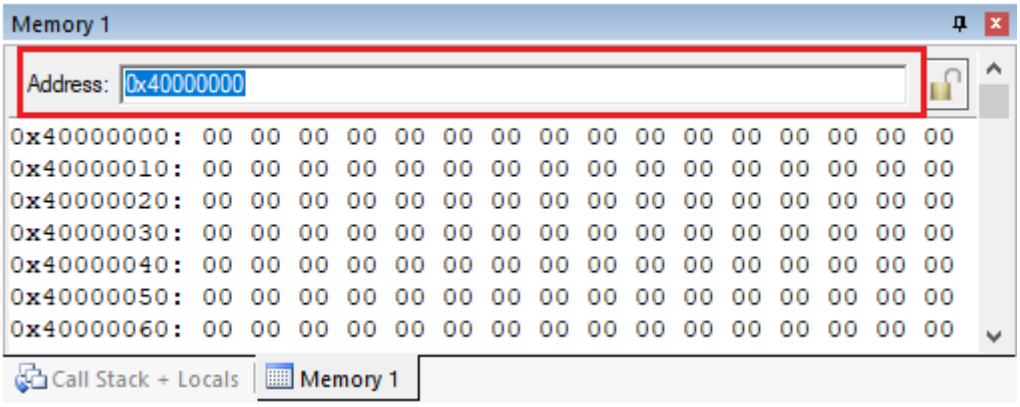
**（3）实验注意事项**

实验要求原始数据放在内存中 0X40000000 开始的地址空间，该地址空间是 S3C2440 片内的 RAM 区，RAM 区的特性就是，在系统启动后，其值不确定（对于仿真实验来说，RAM 单元上电后的初始值为 0），不能在程序中通过 DCD 等分配存储单元的方式来对其设置初始值。在实验时，可以通过以下两种方法解决：

**方法一：**

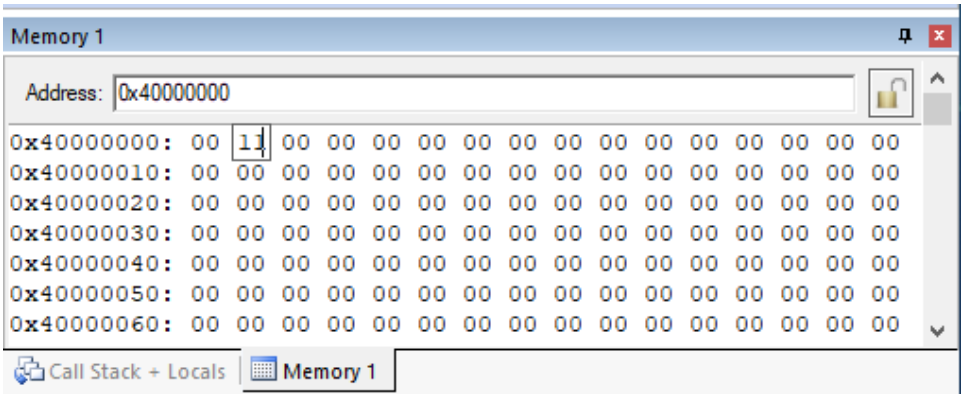
在进入调试状态，正式运行程序前，手动设置从 0X40000000 开始的若干存储单元的值。方法如下：

“Memory Windows”观察窗打开后，界面如下（一般在整个界面的右下角）：




可以在“address”地址栏输入要观察的内存单元的起始地址，以便对指定地址单元进行观察。

也可以双击某单元内容，在该观察窗内直接手动修改某地址单元的内容。如下图所示。




这种方式的缺点就是每次都需要手动输入。

**方法二：**

就是在代码段定义一个连续的数据区，程序开始运行后，把该代码区的内容拷贝到从从 0X40000000 开始的若干 RAM 存储单元中。

**（4）中值滤波程序流程图**

中值滤波的流程如图 1 所示：

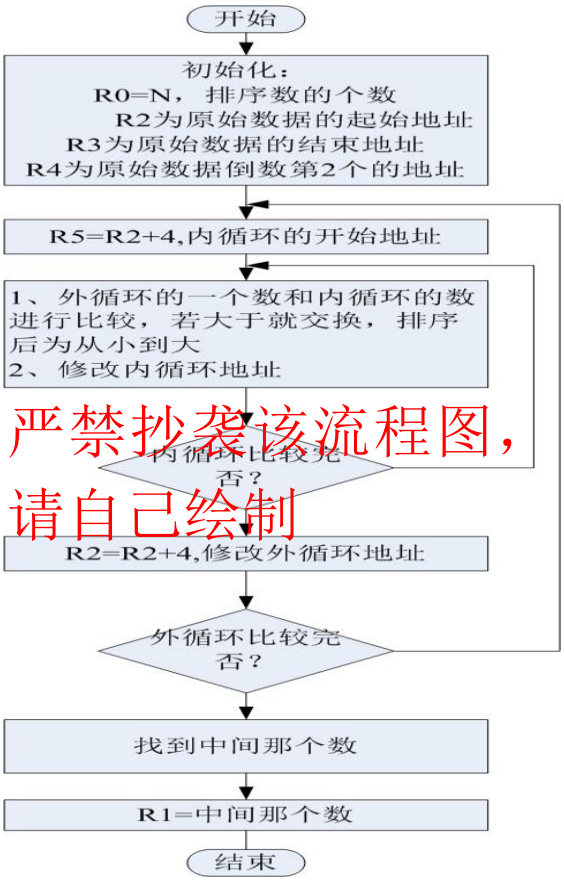





图 1 中值滤波流程图

**（4）均值滤波程序流程图**

均值滤波流程图如图 2 所示：

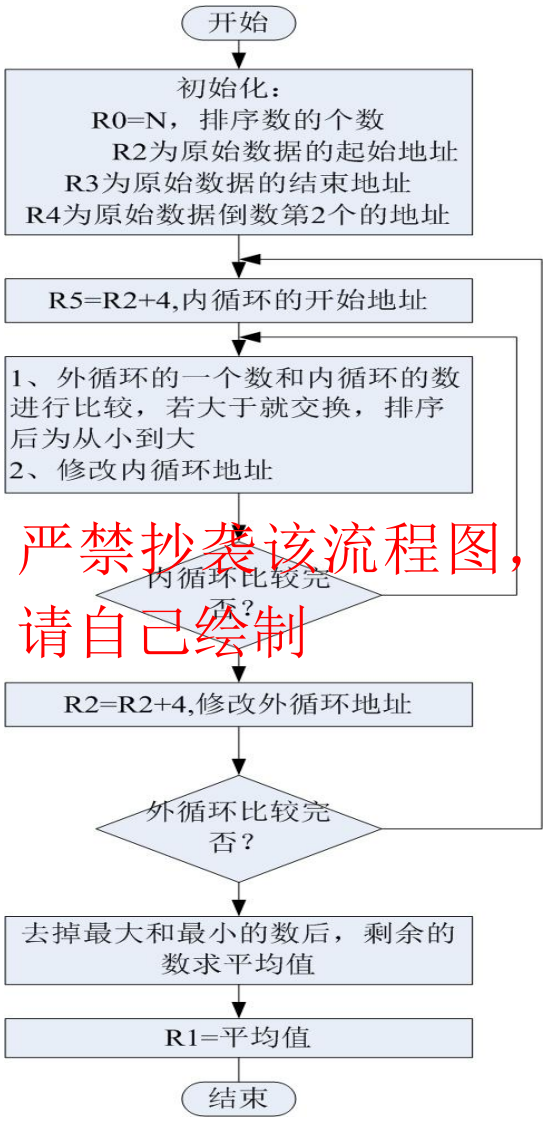





图 2 均值滤波流程图

**【实验报告】**

1. 撰写设计过程并展示源代码，代码需要进行注释。
2. 展示运行结果并论述。

# 实验3　ARM主程序调用ARM/C语言子程序

实验所属系列:《ARM 处理器体系结构及应用》课内实验　　实验对象：本科

相关课程及专业：嵌入式软件　　实验时数（学分）：4 学时

实验类别　课内上机

实验开发教师：　兰刚

## 【实验目的】

(1) 了解 ARM 应用程序框架。

(2) 了解 ARM 汇编程序函数和 C 语言程序函数相互调用时，遵循的 ATPCS 标准；

(3) 了解和掌握 ARM 汇编程序调用 C 语言程序函数的基本方法；

(4) 了解和掌握 ARM 汇编程序调用 C 语言程序函数的参数传递过程。

## 【实验内容】

编写一个 ARM 汇编语言程序，该程序具备以下功能：ARM 指令主程序调用 ARM 指令子程序；

(1) 编写一个宏 strcopy 实现字符串拷贝功能（字符串以 0 做为结束符），该宏具有两个个输入参数：$DEST、$SRC，$SRC 是源字符串的在内存中的首地址，$DEST 是目的字符串在内存中的首地址。

(2) 编写一个 C 语言子程序 Proc_C，C 语言子程序的参数个数为 6 个分别是 i1、i2、i3、i4、i5、i6，C 言子程序实现的功能为:（i1+i2+i3+i4）*i5-i6，观察参数是如何传递的。

(3) 编写一个 ARM 汇编语言子程序 Proc_Arm，通过 R0 传入一个参数 i，在汇编子程序 Proc_Arm 中调用 C 函数 Proc_C()，以实现下面的功能： int Proc_C(int i) {return －Proc_C(i, 2*i, 3*i, 4*i,5*i, 6*i)}。返回结果送至标号为 val 的内存字单元中保存。

(4) 在汇编程序中，使用 strcopy 宏实现两次字符串的拷贝功能，要求在代码段给两个字符串分配存储空间，拷贝到 RAM 区指定的地址单元。

(5) 在汇编主程序中，调用 Proc_Arm 子程序，传入参数为 1。

(6) 分析通过反汇编得到的 C 程序的 ARM 指令代码段，了解参数传递过程。

**【实验环境】**

Keil MDK-ARM uVision5 开发工具。

**【实验设备】**

PC 机一台。

**【实验原理】**

**1. ARM 工程**

由于 C 语言便于理解，有大量的支持库，所以它是当前 ARM 程序设计所使用的主要编程语言。

对硬件系统的初始化、CPU 状态设定、中断使能、主频设定 以及 RAM 控制参数初始化等 C 程序力所不能及的底层操作，还是要由汇编语言程序 来完成。

ARM 工程 的各种源文件之间的关系，以及最后形成可执行文件的过程如下图 1 所示：

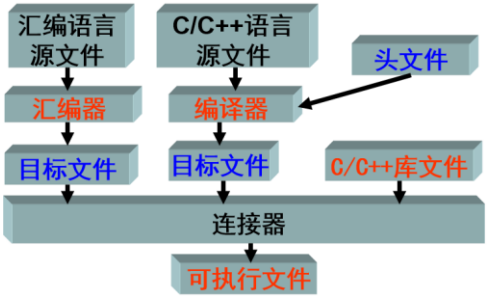



图 1 汇编语言和 C 语言混合编译链接示意图

在应用系统的程序设计中，若所有的编程任务均用汇编语言 来完成，其工作量是可想而知的，这样做也不利于系统升级或应用软件移植。

通常汇编语言部分完成系统硬件的初始化；高级语言部分完成用户的应用。

执行时，首先执行初始化部分，然后再跳转到 C/C++部分。整个程序结构显得清晰明了，容易理解。为方便工程开发，ARM 公司的开发环境 ARM ADS 为用户提供了一个可以选用的应用程序框架。该框架把为用户程序做准备工作的程序分成了： 启动代码 和 应用程序初始化 两部分。

用于硬件初始化的汇编语言部分叫做 启动代码；用于应用程序初始化的 C 部分叫做初始化部分。整个程序如下图 2 所示：

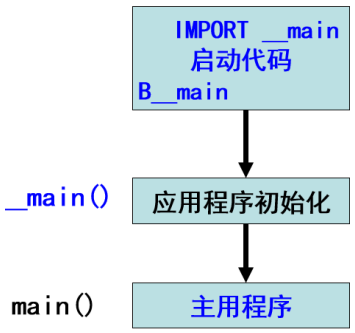



图 2 ARM 应用程序框架

### 2. 过程调用标准 ATPCS

在 ARM 工程中，C 程序调用汇编函数和汇编程序调用 C 函数是经常发生的事情。为此人们制定了 ARM-Thumb 过程调用标准 ATPCS（ARM-Thumb Procedure Call Standard）。

(1) ATPCS 规定，ARM 的数据堆栈为 FD 型堆栈，即递减满堆栈。

(2) ATPCS 标准规定，对于参数个数不多于 4 的函数，编译器必须按参数在列表中的顺序，自左向右为它们分配寄存器 R0~R3。其中函数返回时，R0 还被用来存放函数的返回值。

(3) 如果函数的参数多于 4 个，那么多余的参数则按自右向左的顺序压入数据堆栈，即参数入栈顺序与参数顺序相反。

(4) 根据 ATPCS 的 C 语言程序调用汇编函数，参数由左向右依次传递给寄存器 R0~R3 的规则。

### 3. 子程序的调用与返回

人们把可以多次反复调用的、能完成指定功能的程序段称为“子程序”。把调用子程序的程序称为“主程序”。

为进行识别，子程序的 第 1 条指令 之前必须赋予一个 标号，以便其他程序可以用这个标号调用子程序。

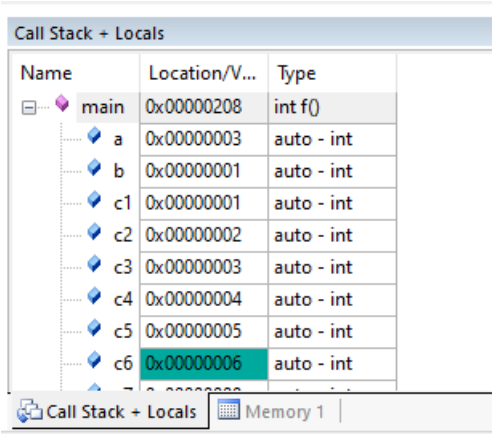
在 ARM 汇编语言程序中，主程序一般通过 BL 指令来调用子程序。该指令在执行时完成如下操作：将子程序的返回地址存放在连接寄存器 LR 中，同时将程序计数器 PC 指向子程序的入口点。

为使子程序执行完毕能 返回 主程序的调用处，子程序末尾 处应有 MOV、LDMFD 等指令，并在指令中将返回地址重新复制到 PC 中。

在调用子程序的同时，也可以使用 R0~R3 来进行 参数的传递 和从子程序返回 运算结果。

## 4. 本实验需要注意的事项和错误解决方法

**(1) 务必注意：在 C 语言函数中，不能使用 printf 函数来输出，否则程序可以正确编译通过，但是无法正常调试，因为系统没有定义 SWI 相对应的软中断。相关变量的值，在 Call Stack-locals 窗口中查看。**




**(2) 该实验，不要拷贝启动文件。**

**(3) 关于实验代码无法停止，不停的执行 C 函数的问题**

大家需要注意：ARM 汇编语言的 END 伪指令的作用，仅仅是告诉汇编器，汇编程序到此结束，后面的内容不需要继续向下汇编了。该伪指令不是程序停止的指令，程序运行完 END 伪指令前的一条指令后，不会自动停止，会继续向后取指运行。

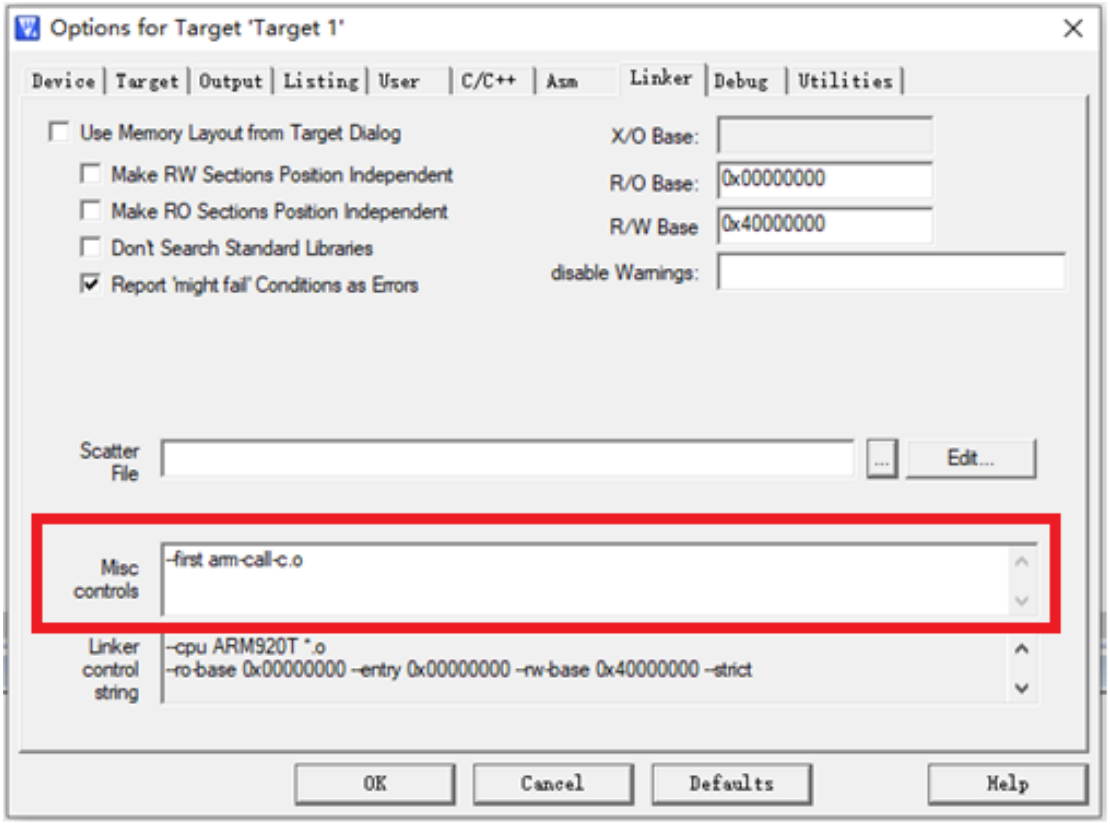
而在 ARM 调用 C 函数的实验程序中，C 函数的程序一般就紧跟在 ARM 汇编程序段的后面，因此，程序运行完 END 伪指令前的最后一条指令后，会继续取后面一条指令执行，即 C 函数的第一条指令执行，导致的效果就是，程序不停的调用 C 函数执行。

对于一个嵌入式系统来说，其主程序往往是一个无限循环的过程，主程序往往一遍一遍的不停的执行某些操作，而不会有停止的指令停止系统运行，最多是系统没有任务时，进入休眠的状态，即类似 S3C2440 的掉电（SLEEP）状态。

**(4) 文件顺序**

在汇编主程序调用 C 子程序的实验中，有时会出现，编译后 C 语言始终位于汇编语言前面的情况。这个时候可以使用以下方法解决：

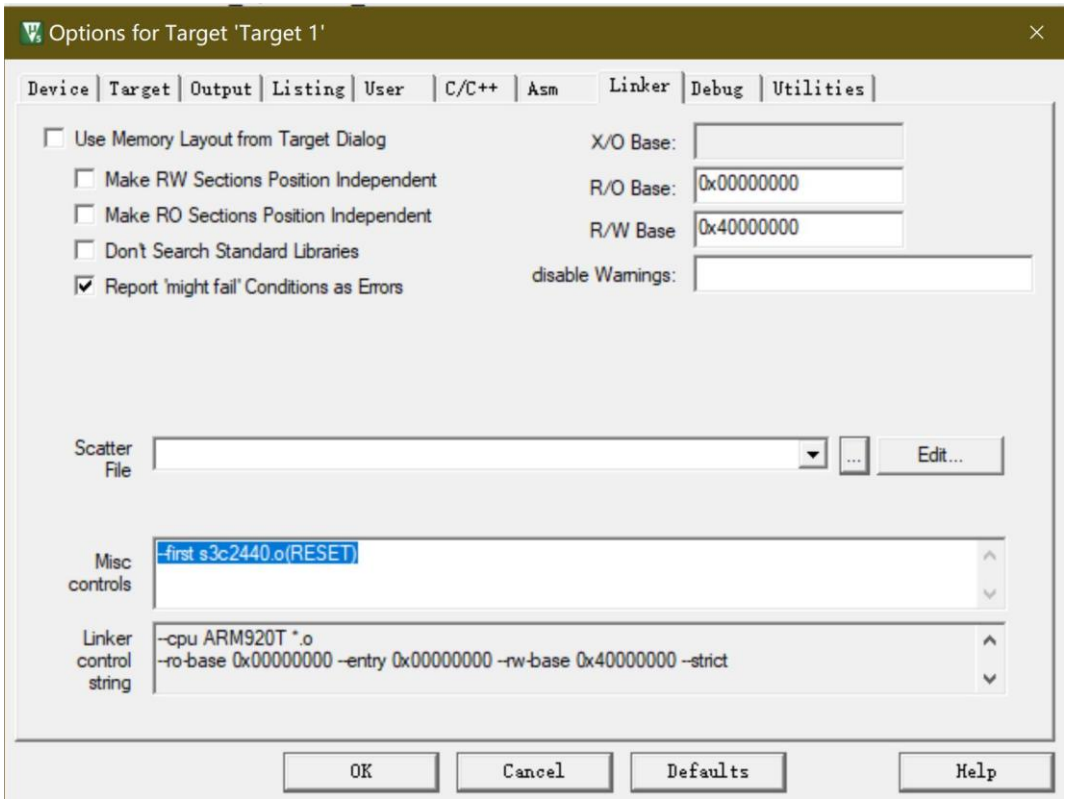
在 linker 页面中，输入 –first arm-call-c.o(这个.o 文件是 arm-call-c.s 编译后生成的目标文件，你需要根据你自己定义的文件名，设置该选项)，该选项告诉链接器，把该目标文件放置到最前面。相关设置界面如下图所示：




**(5) Error: L6211E: Ambiguous section selection. Object s3c2440.o contains more than one section.**

该错误产生的原因是，指定了 3c2440.o 目标文件放在链接后的二进制文件的最前面，但是由于 3c2440.s 中定义了不止一个代码段，所有还必须指明，3c2440.s 中那个代码段放在最前面。因此需要把在 linker 页面中“Mac controls”设置为：“－first s3c2440.o(RESET)”指明 3c2440.s 中的名称为“RESET”的代码段位于程序的最前面，即 0x000000000 地址单元开始放置 RESET 代码段。（注意：－first 在 first 前面是两根短横线）。

还有需要注意：.s 文件和.c 文件在点前面部分的文件名不要取得一样。因为如果前面部分取得一样，则输出的.o 目标文件的文件名就是一样的，会导致该设置不起作用。

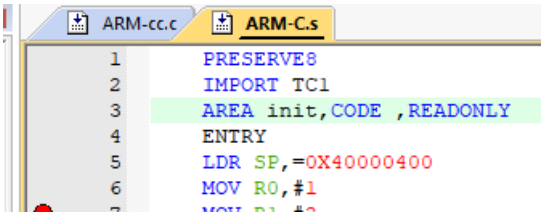



**(6) Error: L6238E: arm-c.o(AAA) contains invalid call from '~PRES8 (The user did not requir**

程序顶部加个

PRESERVE8

就行了，没有其他需要修改了。




**【实验报告】**

1. 撰写设计过程并展示源代码，代码需要进行注释。
2. 展示运行结果并论述。
3. 回答一下问题：

说明 ARM 汇编程序中，参数多余 4 个（比如为 6 个）时，参数是如何压栈传递的？通过 C 语言对应的汇编程序，分析 C 语言子程序是如何取得相关传入参数的？

# 实验 4 C 语言主程序调用 ARM 子程序

实验所属系列:《ARM 处理器体系结构及应用》课内实验　　实验对象：本科

相关课程及专业：嵌入式软件　　实验时数（学分）：4 学时

实验类别　课内上机

实验开发教师：　兰刚

## 【实验目的】

(1) 了解 ARM 应用程序框架。

(2) 了解 ARM 汇编程序函数和 C 语言程序函数相互调用时，遵循的 ATPCS 标准；

(3) 了解和掌握 C 语言程序调用 ARM 语言程序函数的基本方法；

(4) 了解和掌握 C 语言程序调用 ARM 汇编程序函数的参数传递过程；

(5) 掌握内联汇编和嵌入式汇编的编程方法。

## 【实验内容】

(1) C 语言主程序调用 ARM 指令子程序；

(2) 程序的参数个数要求至少 6 个，ARM 子程序实现的功能为:（i1+i2+i3+i4）*i5-i6，并返回计算结果。观察参数是如何传递的。

(3) 在 C 语言的 main 函数中，使用内联汇编编写一段程序，使得 c 语言中定义的 3 个 int 类型的变量(a,b,tmp)有如下的关系： tmp =(a+b)*8。

(4) 以嵌入式汇编的方式，编写一个子程序，实现字节序的转换。该子程序的函数的格式要求如下：int convertNum(int i);要求把输入的整数 i（32 位，四个字节顺序为 AABBCCDD），转换为另一种字节序（转换后的字节序为 DDCCBBAA），返回结果为转换后的 int 值。在主程序中调用该子程序，并验证其结果的正确性。

## 【实验环境】

Keil MDK-ARM uVision5 开发工具。

## 【实验设备】

PC 机一台。

【实验原理】

**1. ARM 工程**

由于 C 语言便于理解，有大量的支持库，所以它是当前 ARM 程序设计所使用的主要编程语言。

对硬件系统的初始化、CPU 状态设定、中断使能、主频设定 以及 RAM 控制参数初始化等 C 程序力所不能及的底层操作，还是要由汇编语言程序 来完成。

ARM 工程 的各种源文件之间的关系，以及最后形成可执行文件的过程如下图 1 所示：

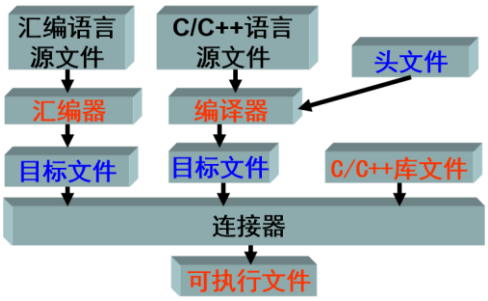



图 1 汇编语言和 C 语言混合编译链接示意图

在应用系统的程序设计中，若所有的编程任务均用汇编语言 来完成，其工作量是可想而知的，这样做也不利于系统升级或应用软件移植。

通常汇编语言部分完成系统硬件的初始化；高级语言部分完成用户的应用。

执行时，首先执行初始化部分，然后再跳转到 C/C++部分。整个程序结构显得清晰明了，容易理解。为方便工程开发，ARM 公司的开发环境 ARM ADS 为用户提供了一个可以选用的应用程序框架。该框架把为用户程序做准备工作的程序分成了： 启动代码 和 应用程序初始化 两部分。

用于硬件初始化的汇编语言部分叫做 启动代码；用于应用程序初始化的 C 部分叫做初始化部分。整个程序如下图 2 所示：

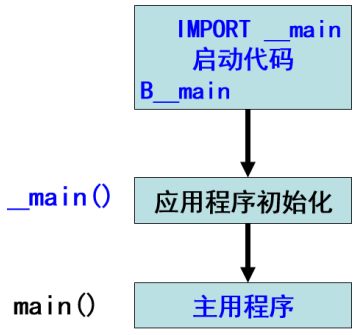


图 2 ARM 应用程序框架

### 2. 过程调用标准 ATPCS

在 ARM 工程中，C 程序调用汇编函数和汇编程序调用 C 函数是经常发生的事情。为此人们制定了 ARM-Thumb 过程调用标准 ATPCS（ARM-Thumb Procedure Call Standard）。

(1) ATPCS 规定，ARM 的数据堆栈为 FD 型堆栈，即递减满堆栈。

(2) ATPCS 标准规定，对于参数个数不多于 4 的函数，编译器必须按参数在列表中的顺序，自左向右为它们分配寄存器 R0~R3。其中函数返回时，R0 还被用来存放函数的返回值。

(3) 如果函数的参数多于 4 个，那么多余的参数则按自右向左的顺序压入数据堆栈，即参数入栈顺序与参数顺序相反。

(4) 根据 ATPCS 的 C 语言程序调用汇编函数，参数由左向右依次传递给寄存器 R0~R3 的规则。

### 3. C/C++语言和汇编语言的混合编程之：内联汇编和嵌入式汇编

除了上面介绍的函数调用方法之外，ARM 编译器 armcc 中含有的内嵌汇编器还允许在 C 程序中 内联或嵌入式汇编代码，以提高程序的效率。

所谓内联汇编程序，就是在 C 程序中直接编写汇编程序段而形成一个语句块，这个语句块可以使用除了 BX 和 BLX 之外的全部 ARM 指令来编写，从而可以使程序实现一些不能从 C 获得的底层功能。

其格式为：

```
__asm
{
    汇编语句块
}
```

内联汇编 与 真实汇编 之间有很大区别，会受到很多限制。

（1）它不支持 Thumb 指令；除了程序状态寄存器 PSR 之外，不能直接访问其他任何物理寄存器等；

（2）如果在内联汇编程序指令中出现了以某个寄存器名称命名的操作数，那么它被叫做虚拟寄存器，而不是实际的物理寄存器。编译器在生成和优化代码的过程中，会给每个虚拟寄存器分配实际的物理寄存器，但这个物理寄存器可能

与在指令中指定的不同。唯一的一个例外就是状态寄存器 PSR ，任何对 PSR 的引用总是执行指向物理 PSR；

（3）在内联汇编代码中不能使用寄存器 PC（R15）、LR（R14）和 SP（R13），任何试图使用这些寄存器的操作都会导致出现错误消息；

（4）虽然内联汇编代码可以更改处理器模式，但更改处理器模式会禁止使用 C 操作数或对已编译 C 代码的调用，直到将处理器模式恢复为原设置之后。

嵌入式汇编

嵌入式汇编程序是一个编写在 C 程序外的单独汇编程序，该程序段可以像函数那样被 C 程序调用。

与内联汇编不同，嵌入式汇编具有真实汇编的所有特性，数据交换符合 ATPCS 标准，同时支持 ARM 和 Thumb，所以它可以对目标处理器进行不受限制的低级访问。但是不能直接引用 C/C++ 的变量。

用 __asm 声明的嵌入式汇编程序，像 C 函数那样，可以有参数和返回值。定义一个嵌入式汇编函数的语法格式为：

```
__asm return–type function–name(parameter-list)
  {
    汇编程序段
  }
```

灵活地使用内联汇编和嵌入式汇编，有助于提高程序效率。

**4. 本实验需要注意的事项和错误解决方法**

(1) 该实验在新建工程时，需要拷贝启动文件：s3c2440.s。

(2) 在 C 调 ARM 的 ARM 子程序中，**需要有保护现场和会恢复现场的代码**。即程序中，除 R0-R3 外的所有其它被使用的寄存器，都必须入栈保存，返回前出栈恢复原值。 还有一点，大家务必注意，**在进入子程序前的 SP 寄存器的值和子程序返回后 SP 的值要保持不变。**这个是考核重点。

(3) 使用堆栈 SP 之前，必须对 SP 初始化，给它分配地址，否则不能使用。切记，如果不初始化 SP，调用子程序或使用堆栈后，会出现程序跑飞的情况。但是对于本实验，在启动文件 s3c2440.s 中已经对堆栈做了初始化操作，不用自己写代码初始化 SP 指针。

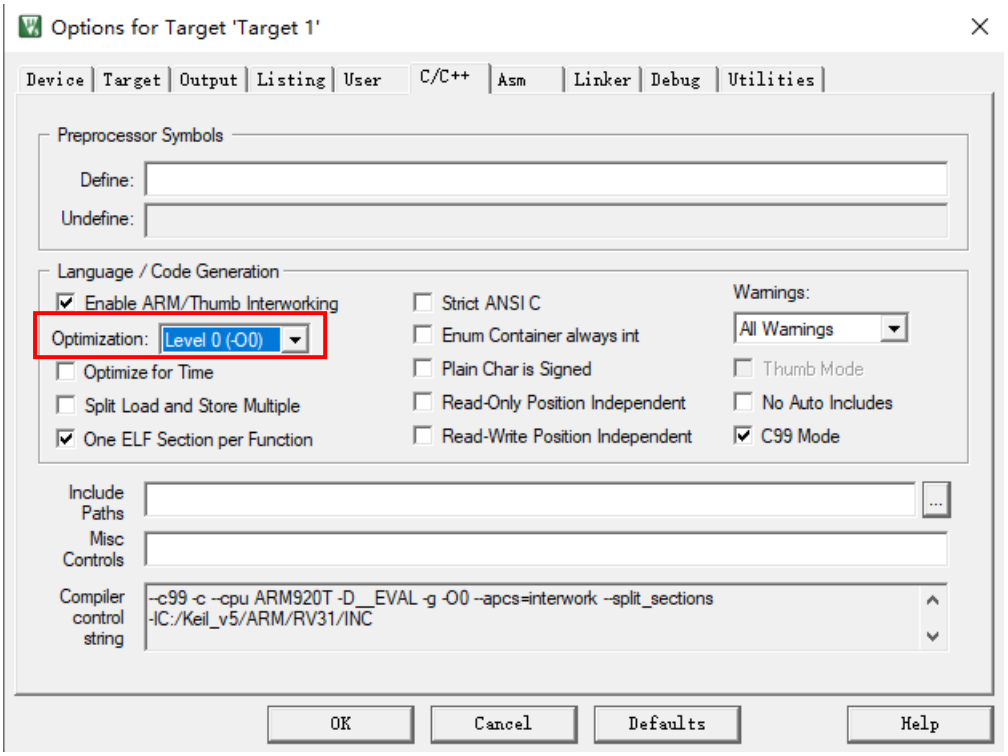
(4) 对于 C 语言调用汇编子程序的情况，C 语言是从 main 函数开始执行，这种情况，需要加载启动代码，也就是当你新建一个工程，并选择了处理器型号后，在系统提示“是否拷贝启动文件”界面中选“是”，拷贝启动文件。自己新建两个源程序文件，分别对应.c 的 c 语言源程序文件和.s 的汇编语言源程序文件。

(5) 需要注意：如果编译时提示有些变量没有使用的警告，最好把未使用的变量屏蔽掉，否则 debug 的时候，有时可能会出现异常。

[1] 比如：如果内联函数最后给 tmp 变量赋值后，而 tmp 在内联函数以后，其值就不再使用，则调试时会跳过内联汇编，因为编译器会进行优化。

[2] 还比如：在 c 中 main 函数定义了各种变量，而且对变量进行了赋值，但是这些变量值进行了赋值后就不再使用这些变量，这编译器在编译进行优化时，就会把只赋值而不使用的变量当做无效变量而优化掉，因此在 **Call Stack-locals 窗口中就观察不到对应变量的赋值变化。**

[3] 这个问题的解决方法就是在其后使用该变量；还有一种解决方法，就是把 C 语言的代码优化程度降低，选择 0 级优化：设置如下：

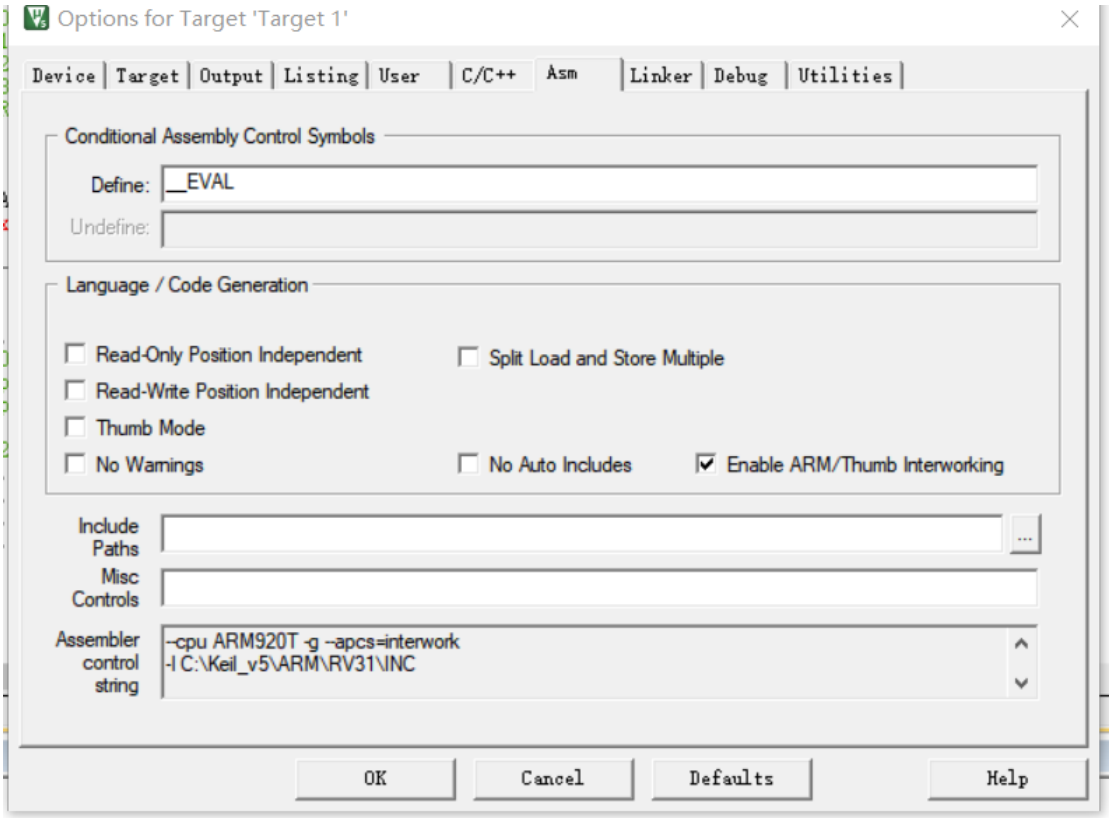



**注：对于内联汇编程序，其结果最好在内联汇编结束后，再次使用。如果不使用，编译器还是可能对其进行优化（哪怕优化级别被设置为最低），相关的**

**内联汇编的指令全部被编译为 NOP 伪指令。因此最好在内联汇编后，添加一行代码：tmp = tmp +1，再次使用 tmp 变量。**

**(5) Error: L6218E: Undefined symbol Image$$ER_ROM1$$RO$$Length (referred from s3c2440.o).**

asm 中定义__EVAL 即可（注是两个下换线）




**【实验报告】**

1. 撰写设计过程并展示源代码，代码需要进行注释。
2. 展示运行结果并论述。
3. 回答一下问题：

通过实验，查看 C 语言对应的汇编程序段，分析 C 语言调用 ARM 子程序时，在输入参数多余 4 个（比如是 6 个）的情况下，C 语言程序汇编后对应的汇编程序段是如何实现参数传递的？结果是如何返回的？ARM 汇编子程序是如何取得相关传入参数的？

# 实验 5　流水灯仿真实验

实验所属系列:《ARM 处理器体系结构及应用》课内实验　　实验对象：本科

相关课程及专业：嵌入式软件　　实验时数（学分）：4 学时

实验类别　课内上机

实验开发教师：　兰刚

## 【实验目的】

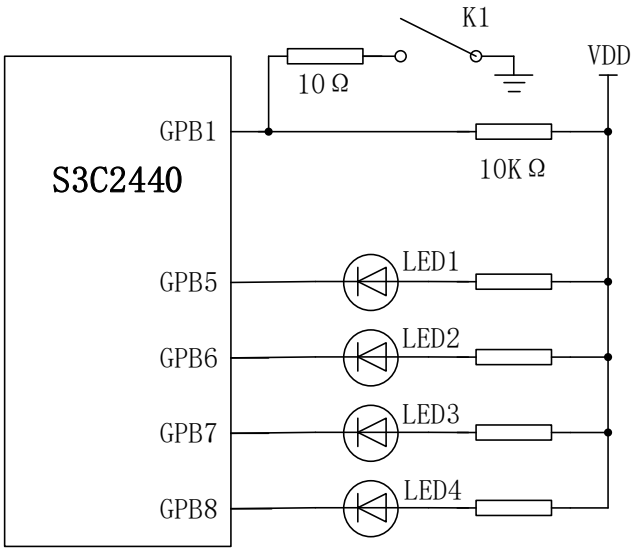
(1) 掌握 ARM 处理器的输入输出接口。

(2) 掌握通过 MDK 提供的仿真功能，实现系统的仿真运行。

(3) 通过该编程实验，进一步巩固和强化学生 ARM 汇编编程的能力，ARM 应用程序框架，培养学生实际应用的能力。

## 【实验内容】

按下面电路图，编写一个流水灯程序，并通过 MDK 的仿真功能进行验证。




实验要求：

(1) 有 1 个拨码开关 K1(接 GPB1 端口)作为输入；

(2) 有 4 个指示灯作为输出(接 GPB5-GPB5 端口)；

(3) 拨码开关 K1 输入高电平时，指示灯从上到下（LED1 到 LED4）循环显示，每次只有一个灯亮；

(4) 拨码开关 K1 输入低电平时，指示灯从下到上（LED4 到 LED1）循环显示，每次只有一个灯亮；

(5) 使用 C 语言编写程序，给完整程序并加注释。

(6) 通过 MDK 的仿真功能验证程序的正确性。

**【实验环境】**

Keil MDK-ARM uVision5 开发工具。

**【实验设备】**

PC 机一台。

**【实验原理】**

**1、端口 B 相关寄存器说明**

(1) 端口 B 对应的寄存器

端口 B 控制寄存器包括：端口 B 配置寄存器 GPBCON、数据寄存器 GPBDAT 和上拉寄存器 GPBUP，具体如下所示：

| 寄存器 | 地址 | R/W | 描述 | 复位值 |
|---|---|---|---|---|
| GPBCON | 0X56000010 | R/W | 配置端口B的引脚 | 0X0 |
| GPBDAT | 0X56000014 | R/W | 端口B的数据寄存器 | 不确定 |
| GPBUP | 0X56000018 | R/W | 端口B的上拉寄存器 | 0X0 |
| Reserve | 0X5600001C | - | 保留 | 不确定 |

(2) GPBCON 各位的描述如下：

| PBCON | Bit | Description |
|---|---|---|
| GPB10 | [21:20] | 00 = Input 01 = Output<br>10 = nXDREQ0 11 = reserved |
| GPB9 | [19:18] | 00 = Input 01 = Output<br>10 = nXDACK0 11 = reserved |
| GPB8 | [17:16] | 00 = Input 01 = Output<br>10 = nXDREQ1 11 = Reserved |
| GPB7 | [15:14] | 00 = Input 01 = Output<br>10 = nXDACK1 11 = Reserved |
| GPB6 | [13:12] | 00 = Input 01 = Output<br>10 = nXBREQ 11 = reserved |
| GPB5 | [11:10] | 00 = Input 01 = Output<br>10 = nXBACK 11 = reserved |
| GPB4 | [9:8] | 00 = Input 01 = Output<br>10 = TCLK [0] 11 = reserved |
| GPB3 | [7:6] | 00 = Input 01 = Output<br>10 = TOUT3 11 = reserved |
| GPB2 | [5:4] | 00 = Input 01 = Output<br>10 = TOUT2 11 = reserved] |
| GPB1 | [3:2] | 00 = Input 01 = Output<br>10 = TOUT1 11 = reserved |
| GPB0 | [1:0] | 00 = Input 01 = Output<br>10 = TOUT0 11 = reserved |

(3) 端口 B 数据寄存器 GPBDAT 中各位的描述如下：

GPBDAT 为准备输出或输入的数据，其值为 11 位[10:0]。

| GPBDAT | 位 | 描述 |
|---|---|---|
| GPB[10:0] | [10:0] | 当端口配置为输入端口时，外部源的数据可以从对应引脚读出；<br>当端口配置为输出端口时，写入寄存器的数据会被发送到对应的引脚上；<br>当端口配置为功能引脚时，如果读该位的值将是一个不确定的值。 |

(4) 端口 B 上拉寄存器 GPBUP 中各位的描述如下：

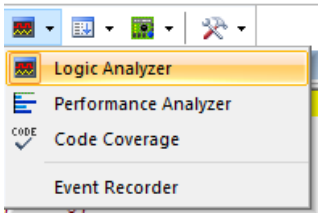
| GPBUP | 位 | 描述 |
|---|---|---|
| GPB[10:0] | [10:0] | 0：允许上拉电阻连接到对应脚；<br>1：不允许上拉电阻连接到对应脚。 |

**2、仿真操作步骤说明**

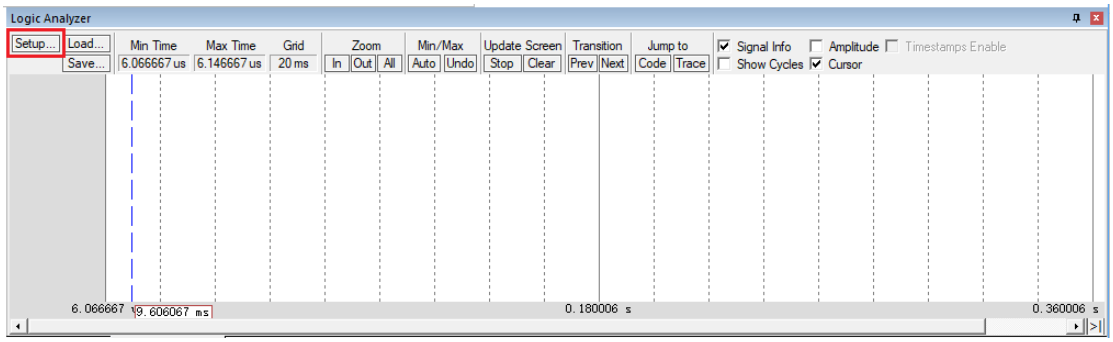
(1) 新建工程（需要拷贝 S3C2440.s 初始化程序），录入代码，并编译通过。

(2) 在菜单中点击“debug”菜单中的“start/stop debug Session(Ctrl+F5)”，进入调试界面。

(3) 在调试界面中，选择“Analysis Windows”→“Logic Analyzer”

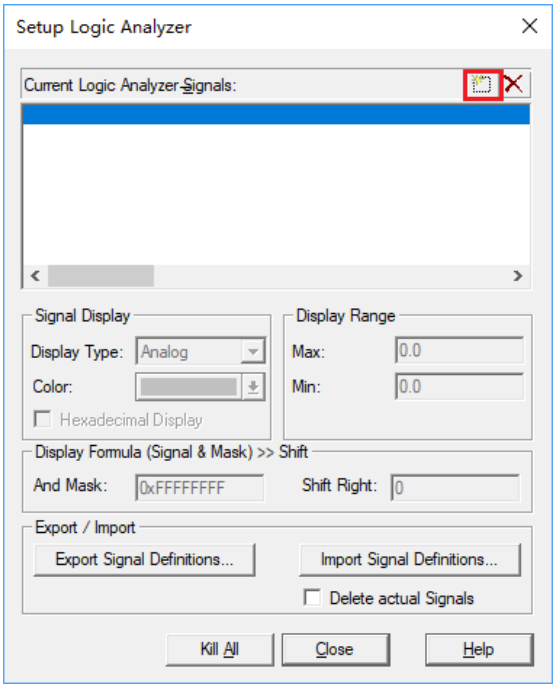



(4) 进入“Logic Analyzer”界面后，点击“Setup”按键，设置需要观察的输出端口。

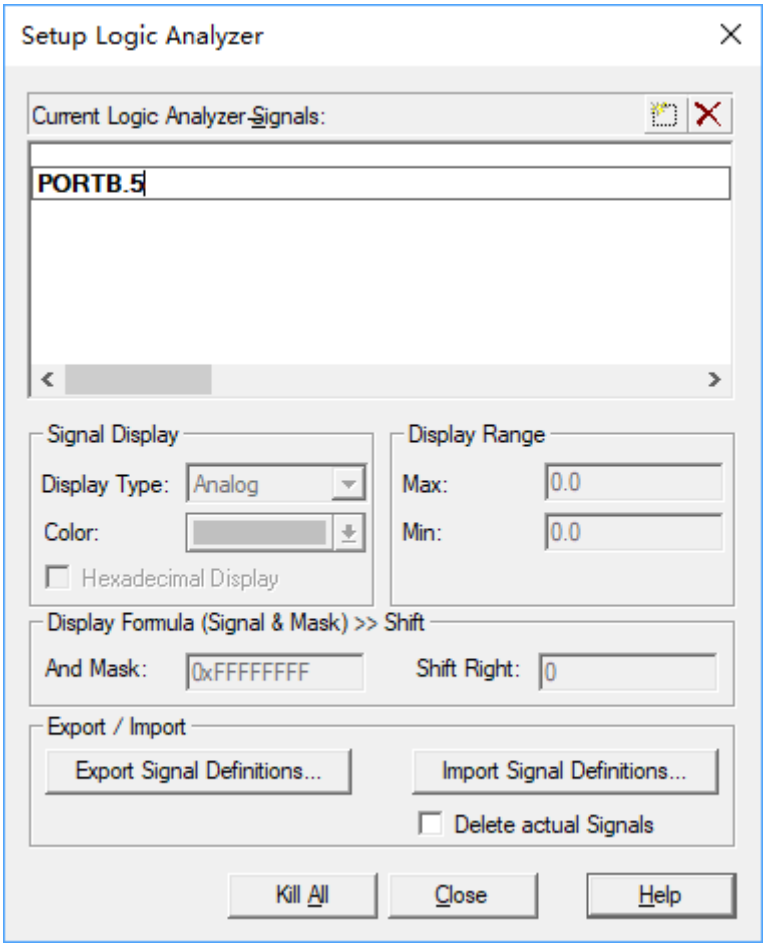



(5) 进入“Setup Logic Analyzer”窗口后，就可以对需要观察的输出端口引脚信号进行设置。由于本实验主要涉及到 GPB5-8 共 4 个输出端口的输出信号进行观察，因此需要添加这 4 个信号。具体添加的方法如下：

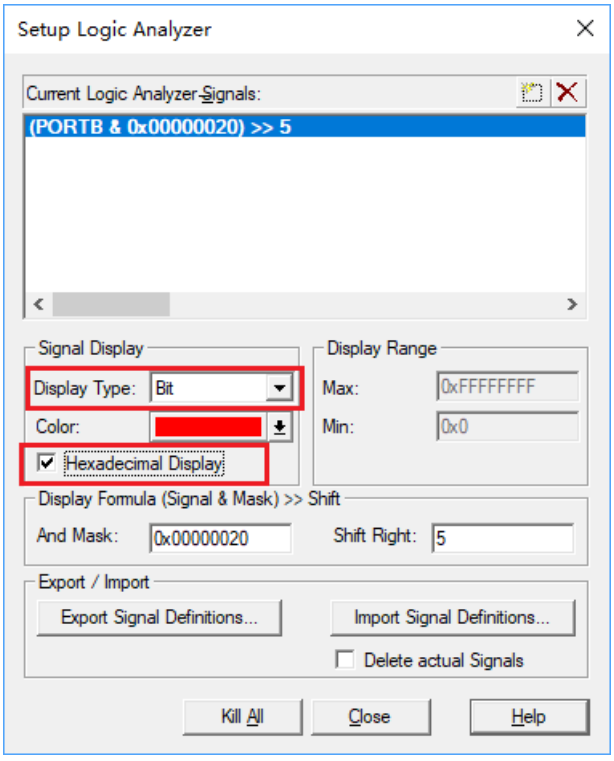
[1]. 点击“添加”按键。



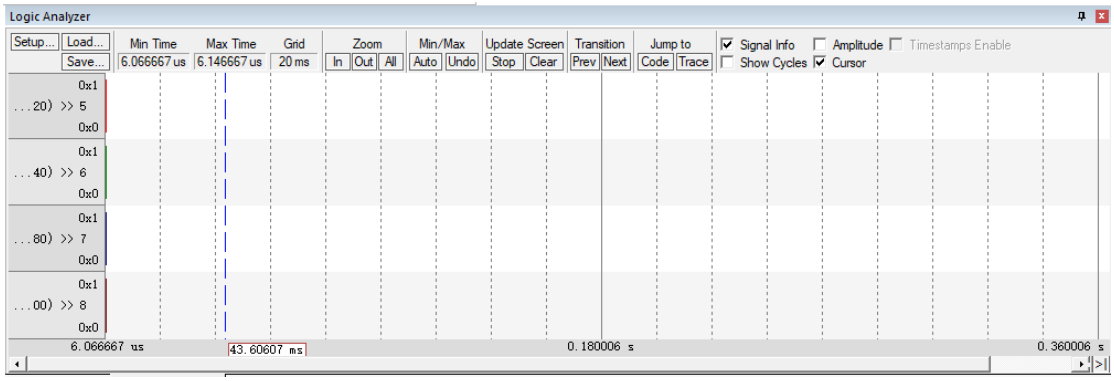

[2]. 输入要查看的引脚，例如 GPIOB_Pin_5 引脚，则输入 PORTB.5，**注意，B 后面有一个小数点，而且都是大写，不能是小写**



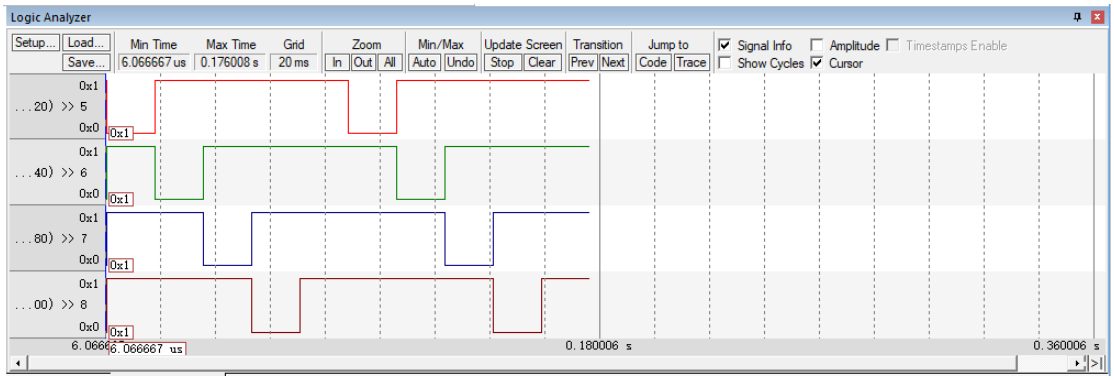

[3]. 点击空白处，输入的引脚显示如下，修改相关参数，使之如图中红框所示。



[4]. 按以上方法，输入 4 个需要观察的端口，然后点击“Close”关闭窗口。设置完成后，“Logic Analyzer”窗口显示如下，里面多了 4 个引脚的显示。

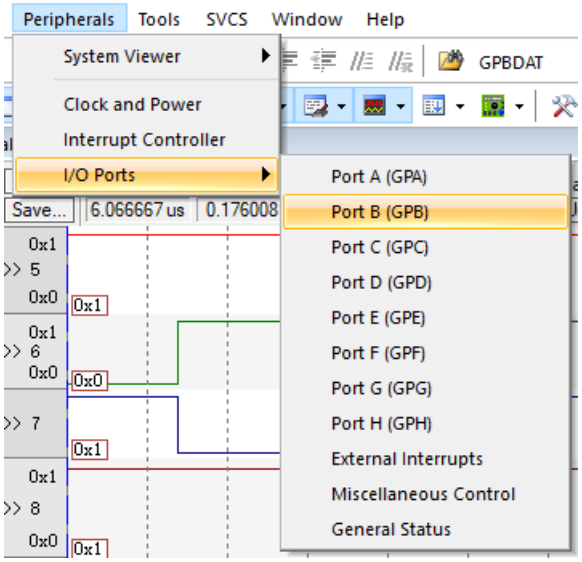



(6) 设置好相关引脚信号后，就可以进行回到“Debug”界面，调试运行。建议设置好断点，让程序循环一次就停止在断点处，再进行下一次循环。下图就是循环运行两次后（每次循环依次点亮 4 个等，然后全部熄灭），输出的波形。下图是拨码开关 K1（GPB1）输入为高电平的时候情况，指示灯从上到下（LED1 到 LED4）循环显示，相关端口输出低电平时，对应的 LED 灯被点亮。

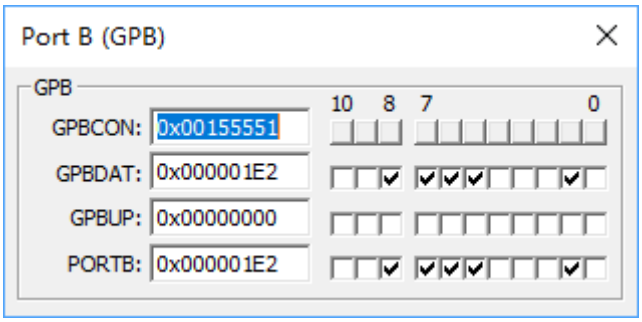



(7) 然后测试拨码开关 K1（GPB1）输入为低电平的时候情况，这时需要设置 GPB1 的输入电平为低电平，具体方法如下：

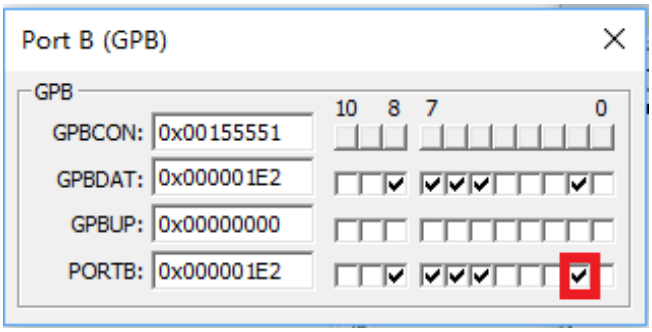
[1]. 选择“Peripherals”→“I/O Ports”→“PortB(GPB)”。

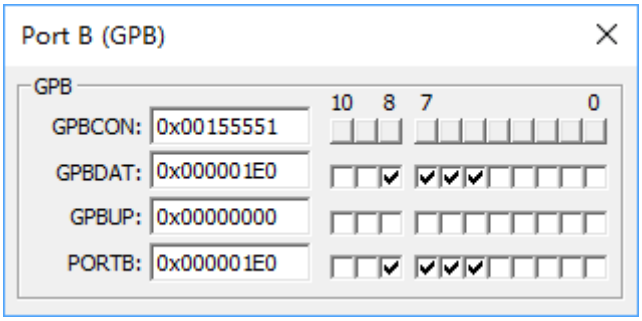



[2]. 出现以下的设置界面，图中有√的引脚当前为高电平。



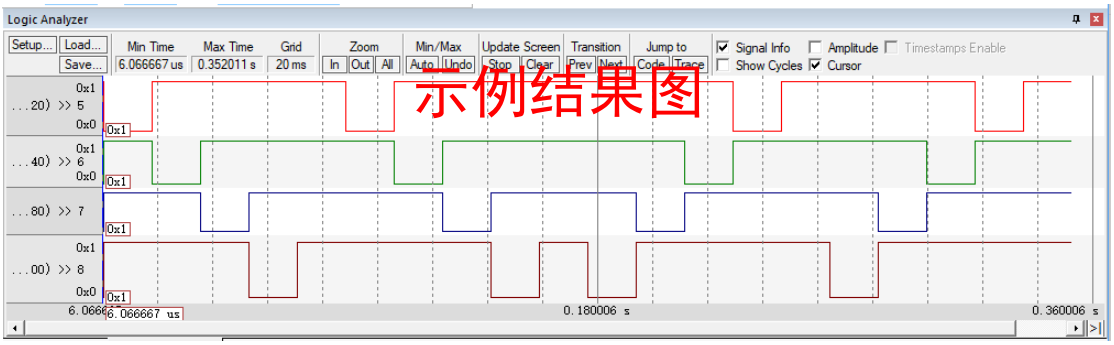

[3]. 把下图中红框中的√去掉后，则对应引脚 GPB1 就设置为了低电平。



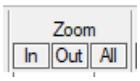




(8) 再次回到“Debug”界面，继续调试运行，得到的完整的仿真输出波形如下

图，图中后半部分就是 GPB1 为低电平时，指示灯从下到上（LED4 到 LED1）循环显示。



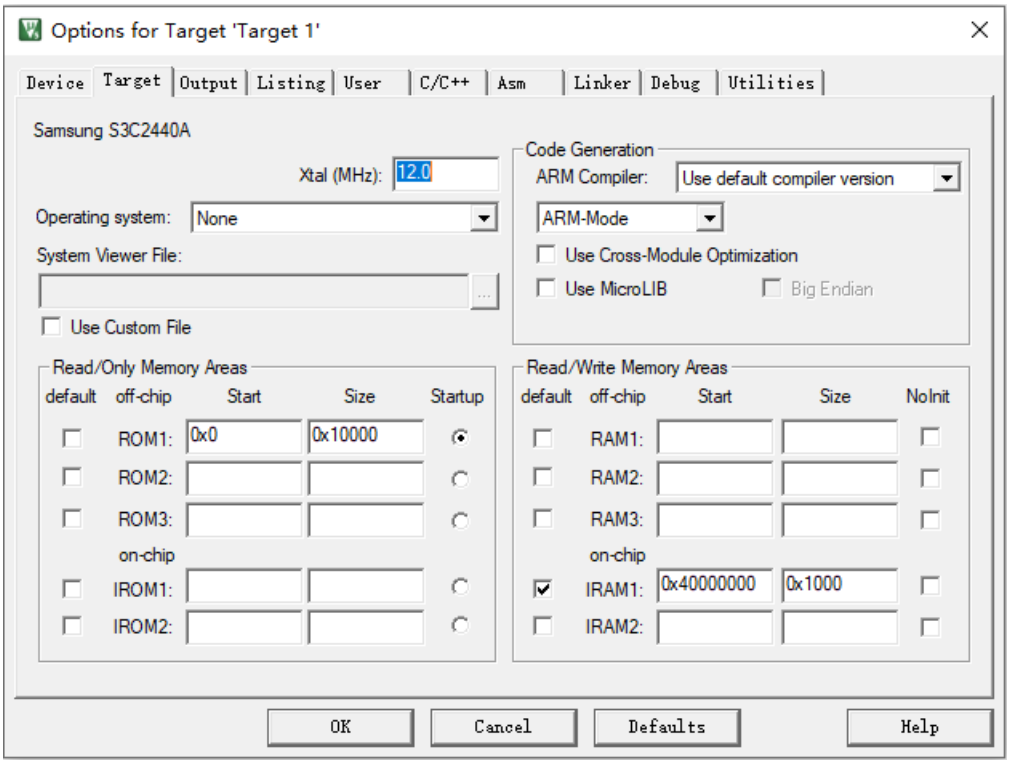

(9) 可以使用图中的“Zoom”中的按键放大或缩小波形，便于观察。

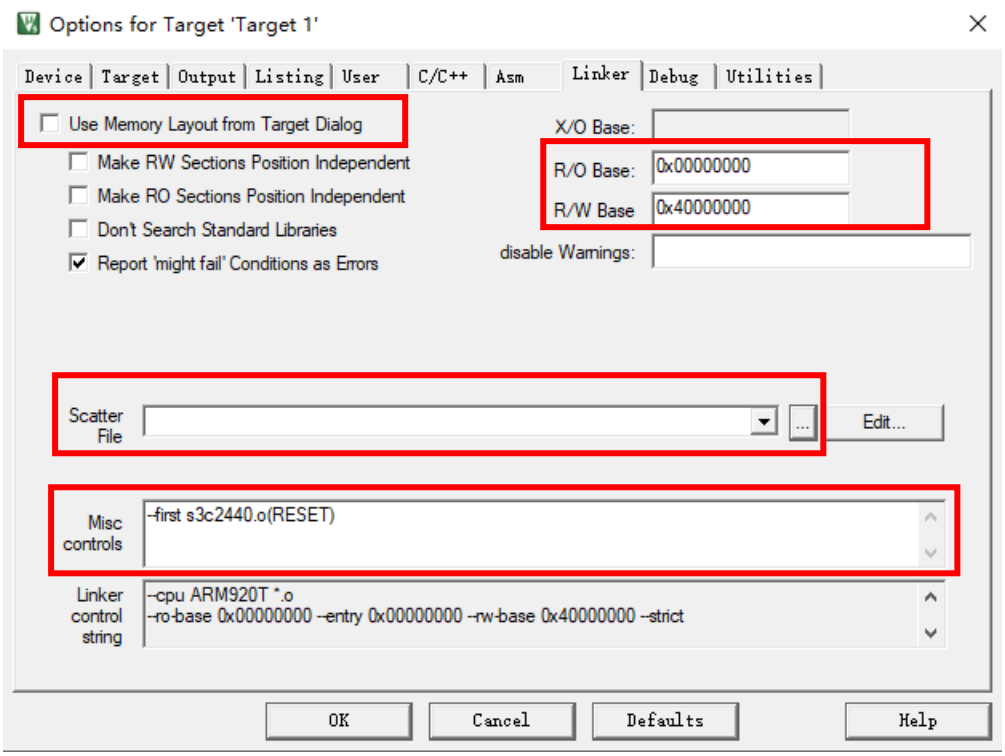


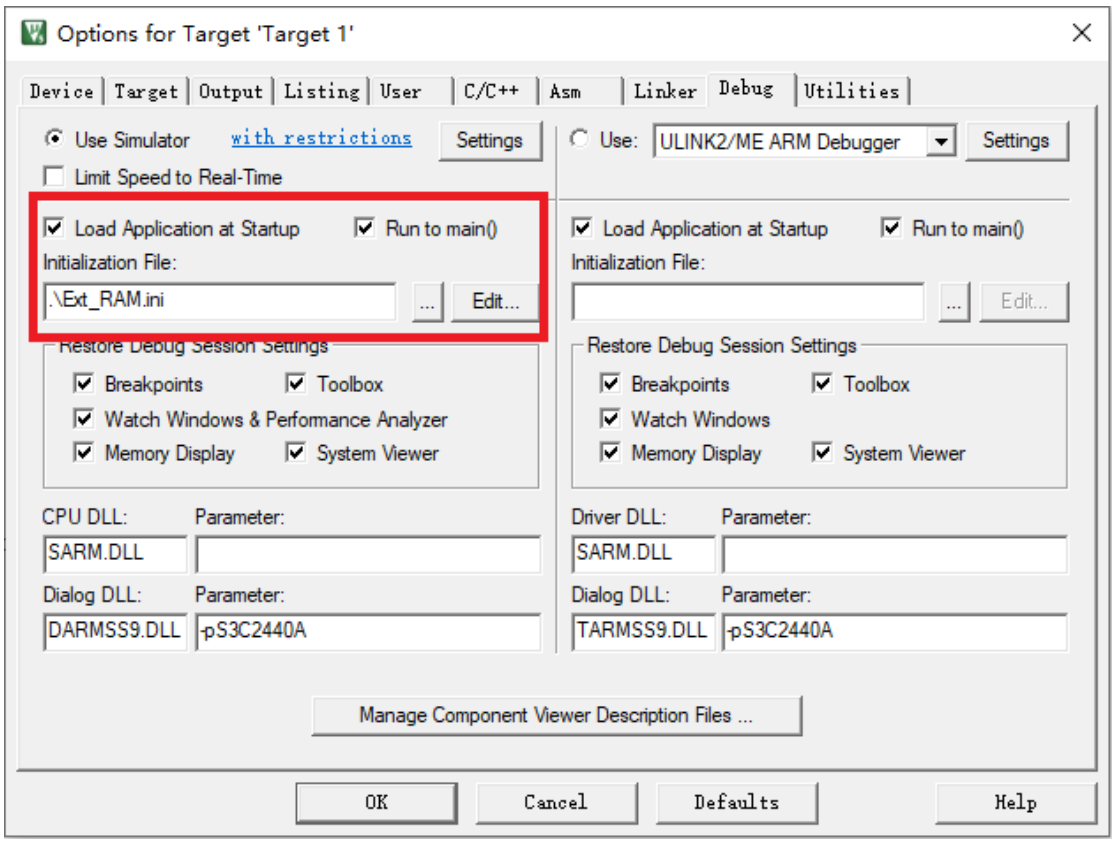

3、工程文件相关配置

可以使用以下两种方式中的一种来配置工程文件：

（1）方式一（推荐配置方式）：



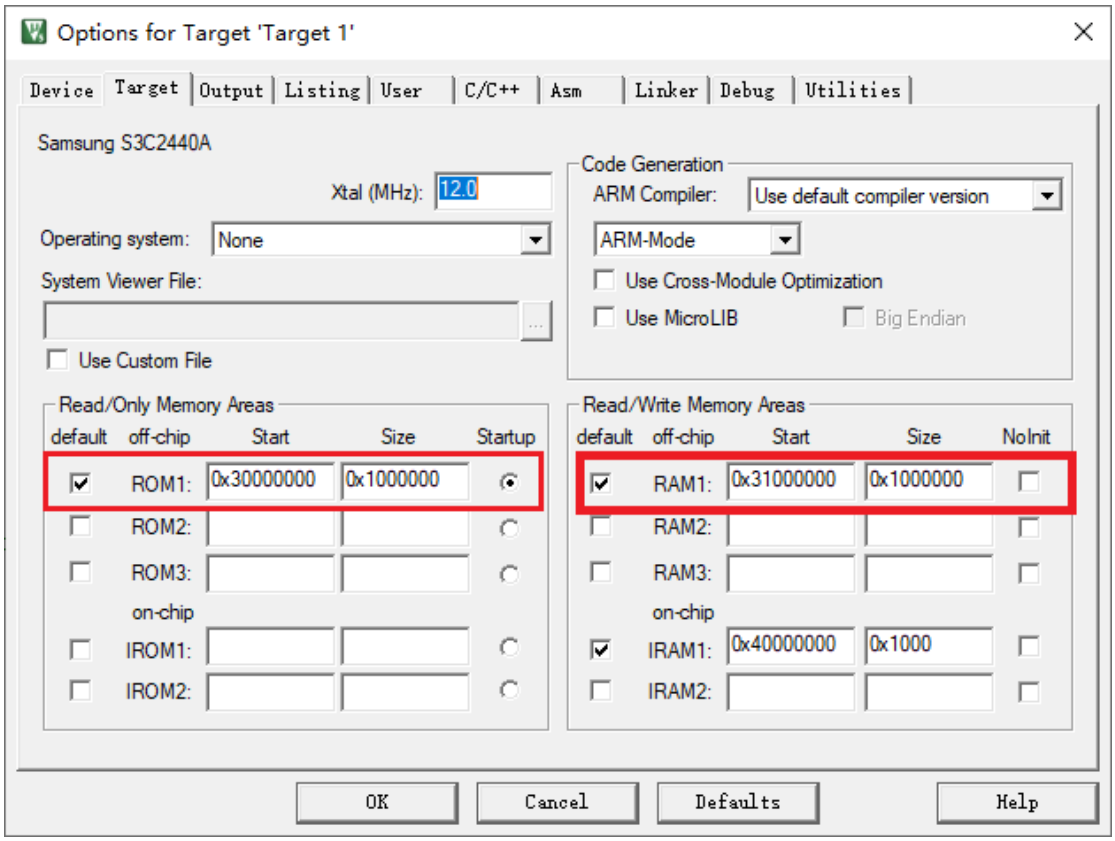


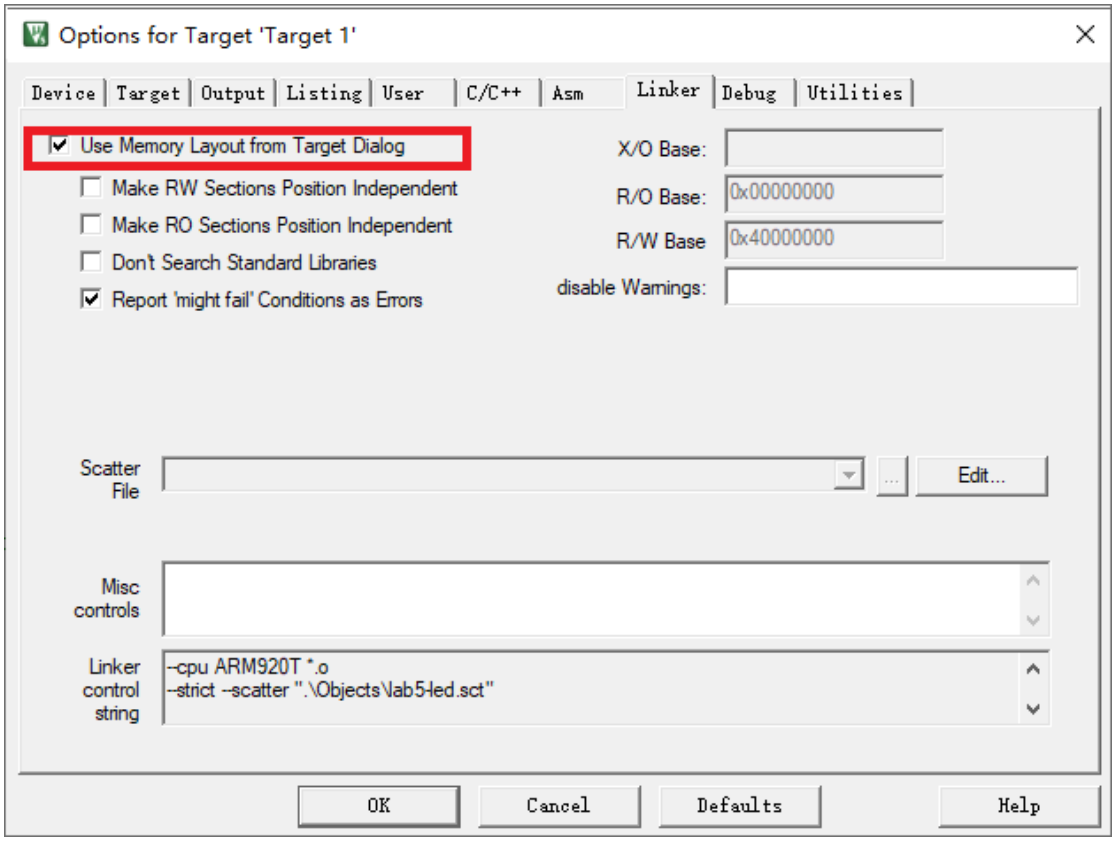



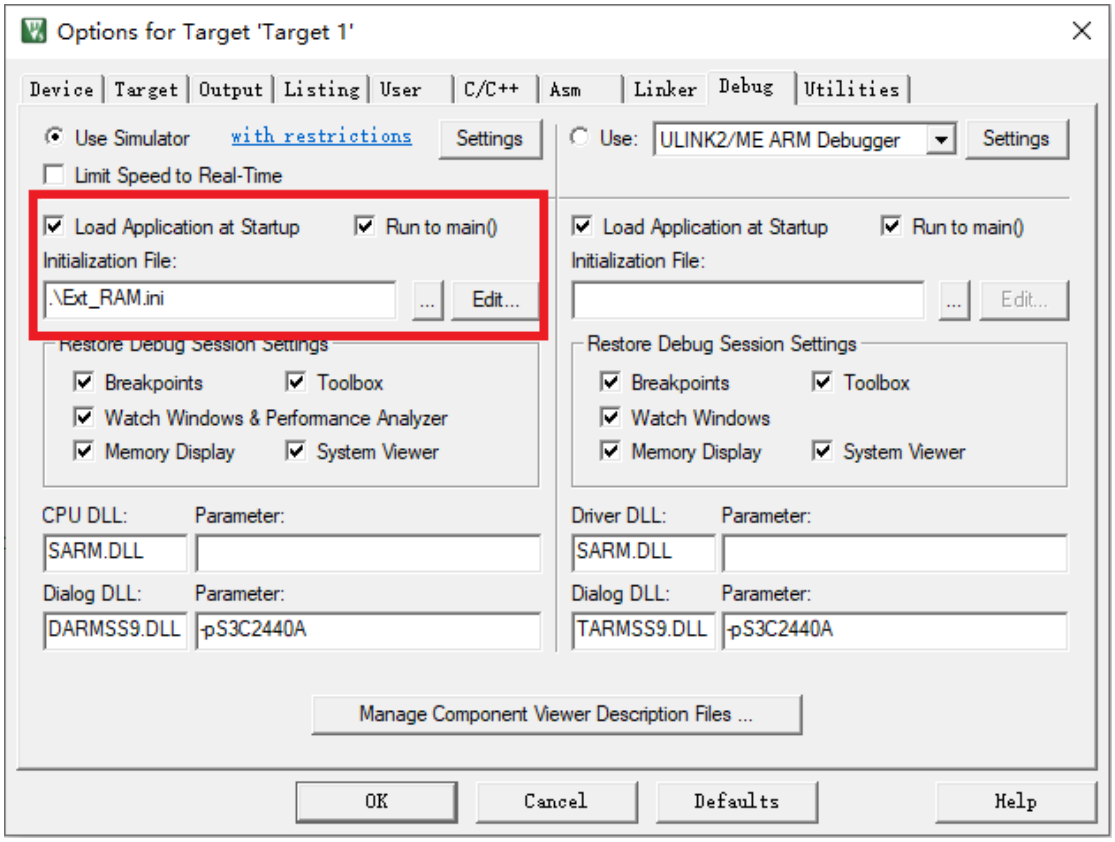

Ext_RAM.ini 的内容如下：

```
map 0x53000000,0x53000100 read write
map 0x48000000,0x48000100 read write
```

（2）方式二：


Options for Target 'Target 1'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Samsung S3C2440A
Xtal (MHz): 12.0
Operating system: None
System Viewer File:
Use Custom File
Code Generation
ARM Compiler: Use default compiler version
ARM-Mode
Use Cross-Module Optimization
Use MicroLIB
Big Endian
Read/Only Memory Areas
default off-chip Start Size Startup
ROM1: 0x30000000 0x1000000
ROM2:
ROM3:
on-chip
IROM1:
IROM2:
Read/Write Memory Areas
default off-chip Start Size NoInit
RAM1: 0x31000000 0x1000000
RAM2:
RAM3:
on-chip
IRAM1: 0x40000000 0x1000
IRAM2:
OK
Cancel
Defaults
Help




Ext_RAM.ini 的内容如下：

```
/*********************************************************/
/*Ext_RAM.INI: External RAM(SDRAM)Initialization File                */
/*********************************************************************/
// <<< Use Configuration Wizard in Context Menu >>>
/*********************************************************************/
/* This file is part of the uVision/ARM development tools.            */
/* Copyright (c) 2005-2008 Keil Software. All rights reserved.        */
/* This software may only be used under the terms of a valid, current,  */
/* end user licence from KEIL for a compatible version of KEIL software  */
/* development tools. Nothing else gives you the right to use this software.  */
/*********************************************************************/

map 0x53000000,0x53000100 read write
map 0x48000000,0x48000100 read write

FUNC void SetupForStart (void) {

// <o> Program Entry Point
   PC = 0x30000000;
}

FUNC void Init (void) {

   _WDWORD(0x4A000008, 0xFFFFFFFF);          // Disable All Interrupts

   _WDWORD(0x53000000, 0x00000000);          // Disable Watchdog Timer

                                             // Clock Setup
                          // FCLK = 300 MHz, HCLK = 100 MHz, PCLK = 50 MHz
   _WDWORD(0x4C000000, 0x0FFF0FFF);          // LOCKTIME
   _WDWORD(0x4C000014, 0x0000000F);          // CLKDIVN
   _WDWORD(0x4C000004, 0x00043011);          // MPLLCON
   _WDWORD(0x4C000008, 0x00038021);          // UPLLCON
   _WDWORD(0x4C00000C, 0x001FFFF0);          // CLKCON

                                          // Memory Controller Setup for
SDRAM
   _WDWORD(0x48000000, 0x22000000);          // BWSCON
   _WDWORD(0x4800001C, 0x00018005);          // BANKCON6
   _WDWORD(0x48000020, 0x00018005);          // BANKCON7
   _WDWORD(0x48000024, 0x008404F3);          // REFRESH
   _WDWORD(0x48000028, 0x00000032);          // BANKSIZE
   _WDWORD(0x4800002C, 0x00000020);          // MRSRB6
   _WDWORD(0x48000030, 0x00000020);          // MRSRB7
```

```
   _WDWORD(0x56000000, 0x000003FF);          // GPACON: Enable Address lines
for SDRAM
}

// Reset chip with watchdog, because nRST line is routed on hardware in a way
// that it can not be pulled low with ULINK

_WDWORD(0x40000000, 0xEAFFFFFE);             // Load RAM addr 0 with branch
to itself
CPSR = 0x000000D3;                            // Disable interrupts
PC      = 0x40000000;                         // Position PC to start of RAM
_WDWORD(0x53000000, 0x00000021);              // Enable Watchdog
g, 0                                          // Wait for Watchdog to reset chip

Init();                                       // Initialize memory
//LOAD Objects\test.axf INCREMENTAL           // Download program
SetupForStart();                              // Setup for Running
g, main                                       // Goto Main
```

**【实验报告】**

1. 撰写设计过程并展示源代码，代码需要进行注释。
2. 展示运行结果并论述。

**注意事项：**

- **给接口寄存器赋值，建议使用与或操作，只修改需要配置的对应 bit 位，其余位都保持其原值不变。**
- **SP 指针不用再自己定义，在 s3c2440.s 启动文件中，已经给出了相关定义。相关代码如下：**

```
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   USR_Stack_Size
__initial_sp    SPACE   ISR_Stack_Size
Stack_Top
      ……
                LDR     R0, =Stack_Top
                MOV     SP, R0
```

- **可以直接使用标准实验报告中给出的模板，在 C 编程时可以直接使用寄存器的名称对其寻址。相关定义如下：**

```
#define GPBCON              (*(volatile unsigned long *) 0x56000010)
#define GPBDAT              (*(volatile unsigned long *) 0x56000014)
```

比如：

```
GPBCON = 0x155551;          //GBP1 是输入端口，
                            //其余端口都是输出端口
```

- **注：有时仿真后，看不见图形的变化，这个可能是波形太大，窗口只显示了波形的一部分。可以尝试点击如下图所示的 Zoom→ALL，使得仿真波形可以完整显示在窗口中。**

# 实验 6　基于中断的按键处理程序实验

实验所属系列:《ARM 处理器体系结构及应用》课内实验　　实验对象：本科

相关课程及专业：嵌入式软件　　实验时数（学分）：4 学时

实验类别　课内上机

实验开发教师：　兰刚

## 【实验目的】

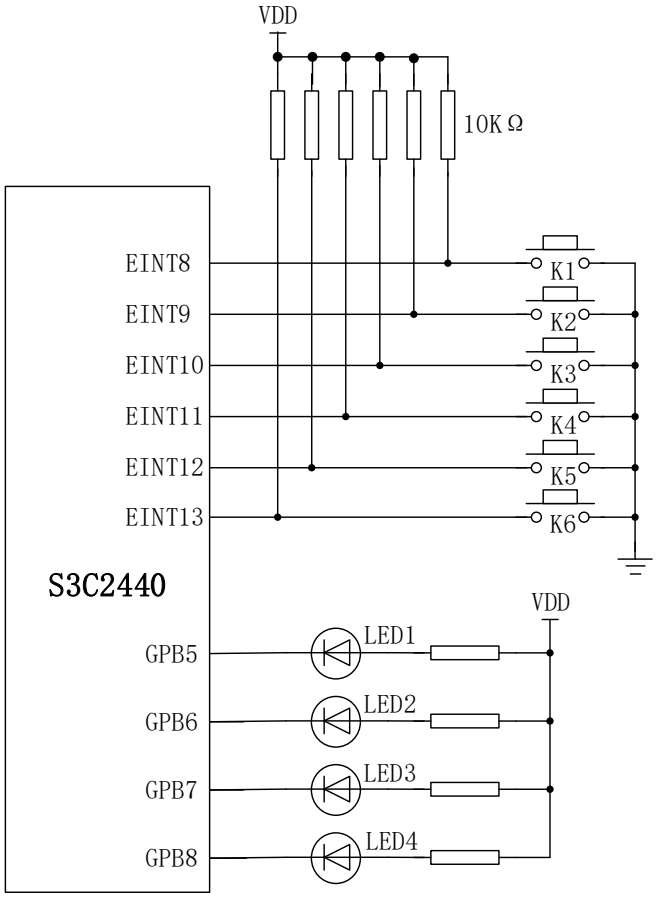
(1) 掌握 ARM 处理器的中断处理过程。

(2) 掌握 ARM 处理器中断服务程序的编写方法。

(3) 通过该编程实验，进一步巩固和强化学生 ARM 汇编编程的能力，ARM 应用程序框架，培养学生实际应用的能力。

## 【实验内容】

按下面电路图，编写一个基于中断的按键处理程序，并通过 MDK 的仿真功能进行验证。



实验要求：

(1) 系统有 6 个按键作为输入，这 6 个按键分别连接到 EINT8-EINT13 这 6 个外中断输入端，系统以中断的方式处理这些按键输入；

(2) 有 4 个指示灯作为输出(接 GPB5-GPB8 端口)，端口输出低电平时，对应的 LED 等被点亮；

(3) K1 按键按下后，指示灯 LED1 灯亮；K2 按键按下后，指示灯 LED2 灯亮；K3 按键按下后，指示灯 LED3 灯亮；K4 按键按下后，指示灯 LED4 灯亮；K5 按键按下后，所有的指示灯都被灯亮；K6 按键按下后，熄灭所有的指示灯；（注：本实验只率。某时刻只有一个按键按下的情况。）；

(4) 主程序 C 语言编写程序，给出完整程序并添加注释。

(5) 通过 MDK 的仿真功能验证程序的正确性。

**【实验环境】**

Keil MDK –ARM uVision5 开发工具。

**【实验设备】**

PC 机一台。

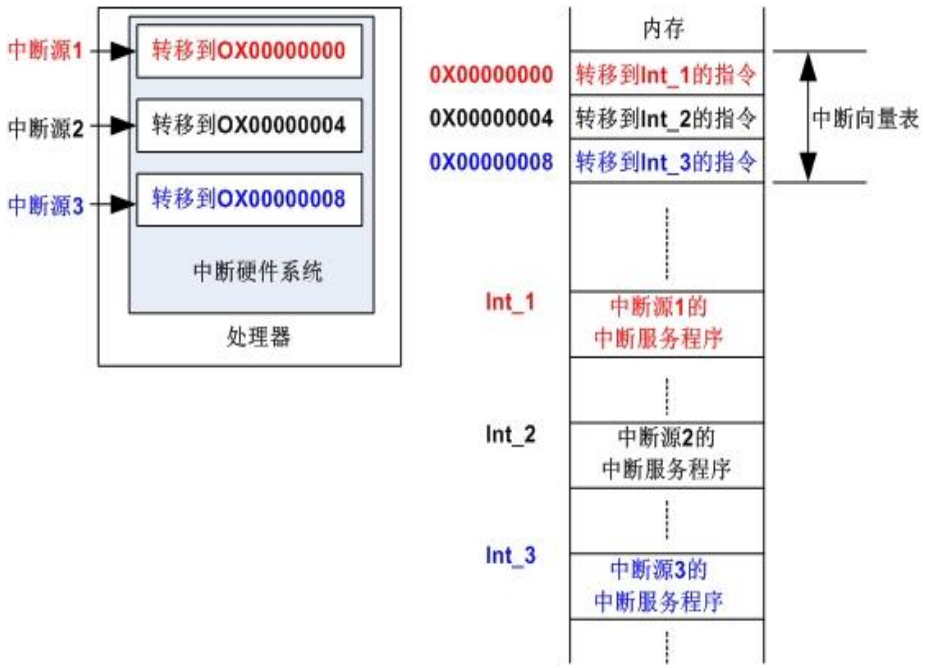
**【实验原理】**

**1. 中断及中断向量**

为了与普通子程序的首地址进行区分，中断服务程序的首地址（入口地址）通常被叫做中断向量，或中断矢量。

在处理器收到中断请求之后，它们都需要获得中断服务程序首地址——中断向量。所有的中断向量都按一定规律存放在一个固定的存储区域，这个集中存放了中断向量或与中断向量相关信息的存储区域就叫做中断向量表。

S3C2440 的中断向量安排如下：




## 2. ARM 的中断（异常）向量表

ARM 中断（异常）的各个向量在向量表中的分配如下：

| 中断（异常） | 向量在低端向量表的地址 | 向量在高端向量表的地址 |
|---|---|---|
| 复位（RESET） | 0x00000000 | 0xFFFF0000 |
| 未定义指令（UNDEF） | 0x00000004 | 0xFFFF0004 |
| 软中断（SWI） | 0x00000008 | 0xFFFF0008 |
| 预取指令中止（PABT） | 0x0000000C | 0xFFFF000C |
| 数据中止（DABT） | 0x00000010 | 0xFFFF0010 |
| 保留 | 0x00000014 | 0xFFFF0014 |
| 中断（IRQ） | 0x00000018 | 0xFFFF0018 |
| 快中断（FIQ） | 0x0000001C | 0xFFFF001C |

## 3. S3C2440X 中断控制器

S3C2440X 中断控制器有 60 个中断源，对外提供 24 个外中断输入引脚，内部所有设备都有中断请求信号，例如：DMA 控制器、UART、IIC 等。中断控制器中包括 5 个控制寄存器：源挂起寄存器、中断模式寄存器、屏蔽寄存器、优先级寄存器、中断挂起寄存器。

(1) 中断屏蔽寄存器 INTMSK

中断屏蔽寄存器 INTMSK 包括了 32 位，每一个比特位均与相应的一个中断源相对应。如果某位被设置为 1，那么 CPU 不会执行相应中断源提出的中断请求（即使源挂起寄存器的相应位被设置为 1）。如果屏蔽位为 0，那么中断请求会被正常执行。

(2) INTOFFSET 寄存器

INTOFFSET 寄存器的值说明了 INTPND 寄存器中哪一个 IRQ 模式的中断请求有效。这个比特位可以通过清除 SRCPND 和 INTPND 寄存器来自动清除。

| INTOFFSET | 偏移量 | INTOFFSET | 偏移量 |
|---|---|---|---|
| INT_ADC | 31 | INT_UART2 | 15 |
| INT_RTC | 30 | INT_TIMER4 | 14 |
| INT_SPI1 | 29 | INT_TIMER3 | 13 |
| INT_UART0 | 28 | INT_TIMER2 | 12 |
| INT_IIC | 27 | INT_TIMER1 | 11 |
| INT_USBH | 26 | INT_TIMER0 | 10 |
| INT_USBD | 25 | INT_WDT_AC97 | 9 |
| INT_NFCON | 24 | INT_TICK | 8 |
| INT_UART1 | 23 | nBATT_FLT | 7 |
| INT_SPI0 | 22 | INT_CAM | 6 |
| INT_SDI | 21 | EINT8_23 | 5 |
| INT_DMA3 | 20 | EINT4_7 | 4 |
| INT_DMA2 | 19 | EINT3 | 3 |
| INT_DMA1 | 18 | EINT2 | 2 |
| INT_DMA0 | 17 | EINT1 | 1 |
| INT_LCD | 16 | EINT0 | 0 |

(3) 中断挂起寄存器 INTPND（C 语言程序中使用 rINTPND 进行访问）

| INTPND | Bit | Description | Initial State |
|---|---|---|---|
| INT_ADC | [31] | 0 = Not requested, 1 = Requested | 0 |
| INT_RTC | [30] | 0 = Not requested, 1 = Requested | 0 |
| INT_SPI1 | [29] | 0 = Not requested, 1 = Requested | 0 |
| INT_UART0 | [28] | 0 = Not requested, 1 = Requested | 0 |
| INT_IIC | [27] | 0 = Not requested, 1 = Requested | 0 |
| INT_USBH | [26] | 0 = Not requested, 1 = Requested | 0 |
| INT_USBD | [25] | 0 = Not requested, 1 = Requested | 0 |
| INT_NFCON | [24] | 0 = Not requested, 1 = Requested | 0 |
| INT_UART1 | [23] | 0 = Not requested, 1 = Requested | 0 |
| INT_SPI0 | [22] | 0 = Not requested, 1 = Requested | 0 |
| INT_SDI | [21] | 0 = Not requested, 1 = Requested | 0 |
| INT_DMA3 | [20] | 0 = Not requested, 1 = Requested | 0 |
| INT_DMA2 | [19] | 0 = Not requested, 1 = Requested | 0 |
| INT_DMA1 | [18] | 0 = Not requested, 1 = Requested | 0 |
| INT_DMA0 | [17] | 0 = Not requested, 1 = Requested | 0 |
| INT_LCD | [16] | 0 = Not requested, 1 = Requested | 0 |
| INT_UART2 | [15] | 0 = Not requested, 1 = Requested | 0 |
| INT_TIMER4 | [14] | 0 = Not requested, 1 = Requested | 0 |
| INT_TIMER3 | [13] | 0 = Not requested, 1 = Requested | 0 |
| INT_TIMER2 | [12] | 0 = Not requested, 1 = Requested | 0 |
| INT_TIMER1 | [11] | 0 = Not requested, 1 = Requested | 0 |
| INT_TIMER0 | [10] | 0 = Not requested, 1 = Requested | 0 |
| INT_WDT_AC97 | [9] | 0 = Not requested, 1 = Requested | 0 |
| INT_TICK | [8] | 0 = Not requested, 1 = Requested | 0 |
| nBATT_FLT | [7] | 0 = Not requested, 1 = Requested | 0 |
| INT_CAM | [6] | 0 = Not requested, 1 = Requested | 0 |
| EINT8_23 | [5] | 0 = Not requested, 1 = Requested | 0 |
| EINT4_7 | [4] | 0 = Not requested, 1 = Requested | 0 |
| EINT3 | [3] | 0 = Not requested, 1 = Requested | 0 |
| EINT2 | [2] | 0 = Not requested, 1 = Requested | 0 |
| EINT1 | [1] | 0 = Not requested, 1 = Requested | 0 |
| EINT0 | [0] | 0 = Not requested, 1 = Requested | 0 |

(4) 外部中断挂起寄存器 EINTPEND（C 语言程序中使用 rEINTPEND 进行访问）

| EINTPEND | Bit | Description | Reset Value |
|---|---|---|---|
| EINT23 | [23] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT22 | [22] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT21 | [21] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT20 | [20] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT19 | [19] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT18 | [18] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT17 | [17] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT16 | [16] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT15 | [15] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT14 | [14] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT13 | [13] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT12 | [12] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT11 | [11] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT10 | [10] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT9 | [9] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT8 | [8] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT7 | [7] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT6 | [6] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT5 | [5] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| EINT4 | [4] | It is cleard by writing "1"<br>0 = Not occur 1 = Occur interrupt | 0 |
| Reserved | [3:0] | Reserved | 0000 |

## 4. S3C2440X 外中断相关控制器

在具体执行中断之前，要初始化好要用的中断。2440 的外部中断引脚 EINT 与通用 IO 引脚 F 和 G 复用（本实验用的 EINT8-23 使用的是 PORTG 对应的引脚），要想使用中断功能，就要把相应的引脚配置成中断模式，如我们想把端口 G0 设置成外部中断（EINT8），而其他引脚功能不变，则使用以下语句

rGPGCON = rGPGCON & (~(0x03)) | (0x02)。

配置完引脚后，还需要配置具体的中断功能。我们要打开某一中断的屏蔽，这样才能响应该中断，相对应的寄存器为 INTMSK；

还要设置外部中断的触发方式，如低电平、高电平、上升沿、下降沿等，相对应的寄存器为 EXTINTn。比如设置 EINT8 使用下降沿触发：

rEXTINT1 &= ~(7<<0);

rEXTINT1 |= (3<<0);　　//设置 EINT[8]下降沿触发

另外由于 EINT4 到 EINT7 共用一个中断向量，EINT8 到 EINT23 也共用一个中断向量，而 INTMSK 只负责总的中断向量的屏蔽，要具体打开某一具体的中断屏蔽，还需要设置 EINTMASK。比如使能 EINT8：

rEINTMASK &= ~(1<<8)

相关寄存器如下：

(1) 端口 G 的控制寄存器各位含义如下：

| GPGCON | Bit | Description |
|---|---|---|
| GPG15* | [31:30] | 00 = Input　01 = Output<br>10 = EINT[23]　11 = Reserved |
| GPG14* | [29:28] | 00 = Input　01 = Output<br>10 = EINT[22]　11 = Reserved |
| GPG13* | [27:26] | 00 = Input　01 = Output<br>10 = EINT[21]　11 = Reserved |
| GPG12 | [25:24] | 00 = Input　01 = Output<br>10 = EINT[20]　11 = Reserved |
| GPG11 | [23:22] | 00 = Input　01 = Output<br>10 = EINT[19]　11 = TCLK[1] |
| GPG10 | [21:20] | 00 = Input　01 = Output<br>10 = EINT[18]　11 = nCTS1 |
| GPG9 | [19:18] | 00 = Input　01 = Output<br>10 = EINT[17]　11 = nRTS1 |
| GPG8 | [17:16] | 00 = Input　01 = Output<br>10 = EINT[16]　11 = Reserved |
| GPG7 | [15:14] | 00 = Input　01 = Output<br>10 = EINT[15]　11 = SPICLK1 |
| GPG6 | [13:12] | 00 = Input　01 = Output<br>10 = EINT[14]　11 = SPIMOSI1 |
| GPG5 | [11:10] | 00 = Input　01 = Output<br>10 = EINT[13]　11 = SPIMISO1 |
| GPG4 | [9:8] | 00 = Input　01 = Output<br>10 = EINT[12]　11 = LCD_PWRDN |
| GPG3 | [7:6] | 00 = Input　01 = Output<br>10 = EINT[11]　11 = nSS1 |
| GPG2 | [5:4] | 00 = Input　01 = Output<br>10 = EINT[10]　11 = nSS0 |
| GPG1 | [3:2] | 00 = Input　01 = Output<br>10 = EINT[9]　11 = Reserved |
| GPG0 | [1:0] | 00 = Input　01 = Output<br>10 = EINT[8]　11 = Reserved |

(2) 外部中断控制 EINT1 的部分位含义如下：

| EXTINT1 | Bit | Description |
|---|---|---|
| FLTEN15 | [31] | Filter enable for EINT15<br>0 = Filter Disable 1 = Filter Enable |
| EINT15 | [30:28] | Setting the signaling method of the EINT15.<br>000 = Low level 001 = High level 01x = Falling edge triggered<br>10x = Rising edge triggered 11x = Both edge triggered |
| FLTEN14 | [27] | Filter enable for EINT14<br>0 = Filter Disable 1 = Filter Enable |
| EINT14 | [26:24] | Setting the signaling method of the EINT14.<br>000 = Low level 001 = High level 01x = Falling edge triggered<br>10x = Rising edge triggered 11x = Both edge triggered |
| FLTEN13 | [23] | Filter enable for EINT13<br>0 = Filter Disable 1 = Filter Enable |
| EINT13 | [22:20] | Setting the signaling method of the EINT13.<br>000 = Low level 001 = High level 01x = Falling edge triggered<br>10x = Rising edge triggered 11x = Both edge triggered |
| FLTEN12 | [19] | Filter enable for EINT12<br>0 = Filter Disable 1 = Filter Enable |
| EINT12 | [18:16] | Setting the signaling method of the EINT12.<br>000 = Low level 001 = High level 01x = Falling edge triggered<br>10x = Rising edge triggered 11x = Both edge triggered |
| FLTEN11 | [15] | Filter enable for EINT11<br>0 = Filter Disable 1 = Filter Enable |
| EINT11 | [14:12] | Setting the signaling method of the EINT11.<br>000 = Low level 001 = High level 01x = Falling edge triggered<br>10x = Rising edge triggered 11x = Both edge triggered |
| FLTEN10 | [11] | Filter enable for EINT10<br>0 = Filter Disable 1 = Filter Enable |
| EINT10 | [10:8] | Setting the signaling method of the EINT10.<br>000 = Low level 001 = High level 01x = Falling edge triggered<br>10x = Rising edge triggered 11x = Both edge triggered |
| FLTEN9 | [7] | Filter enable for EINT9<br>0 = Filter Disable 1 = Filter Enable |
| EINT9 | [6:4] | Setting the signaling method of the EINT9.<br>000 = Low level 001 = High level 01x = Falling edge triggered<br>10x = Rising edge triggered 11x = Both edge triggered |
| FLTEN8 | [3] | Filter enable for EINT8<br>0 = Filter Disable 1 = Filter Enable |
| EINT8 | [2:0] | Setting the signaling method of the EINT8.<br>000 = Low level 001 = High level 01x = Falling edge triggered<br>10x = Rising edge triggered 11x = Both edge triggered |

(3) 外部中断屏蔽寄存器（EINTMASK）

| EINTMASK | Bit | Description |
|---|---|---|
| EINT23 | [23] | 0 = enable interrupt 1= masked |
| EINT22 | [22] | 0 = enable interrupt 1= masked |
| EINT21 | [21] | 0 = enable interrupt 1= masked |
| EINT20 | [20] | 0 = enable interrupt 1= masked |
| EINT19 | [19] | 0 = enable interrupt 1= masked |
| EINT18 | [18] | 0 = enable interrupt 1= masked |
| EINT17 | [17] | 0 = enable interrupt 1= masked |
| EINT16 | [16] | 0 = enable interrupt 1= masked |
| EINT15 | [15] | 0 = enable interrupt 1= masked |
| EINT14 | [14] | 0 = enable interrupt 1= masked |
| EINT13 | [13] | 0 = enable interrupt 1= masked |
| EINT12 | [12] | 0 = enable interrupt 1= masked |
| EINT11 | [11] | 0 = enable interrupt 1= masked |
| EINT10 | [10] | 0 = enable interrupt 1= masked |
| EINT9 | [9] | 0 = enable interrupt 1= masked |
| EINT8 | [8] | 0 = enable interrupt 1= masked |
| EINT7 | [7] | 0 = enable interrupt 1= masked |
| EINT6 | [6] | 0 = enable interrupt 1= masked |
| EINT5 | [5] | 0 = enable interrupt 1= masked |
| EINT4 | [4] | 0 = enable interrupt 1= masked |
| Reserved | [3:0] | Reserved |

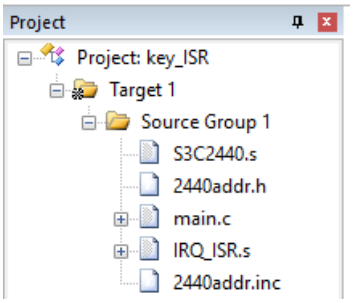
## 5. 实验相关提示和说明

(1) 工程文件相关说明

本次实验在群文件里提供一个实验压缩包 Lab6-stu.rar，里面包含了以下几个文件：

- ✧ S3C2440.s：系统初始化程序，这个可以使用系统自己生成的同名文件。
- ✧ 2440addr.inc：这个是用于 ARM 汇编的 S3C2440 寄存器地址定义文件，使用这个文件，可以在 ARM 汇编程序中直接用各寄存器的名称作为地址来编程，而不必使用 32 位的二进制地址。
- ✧ IRQ_ISR.s：我重新定义的用于 IRQ 的中断服务程序。
- ✧ 2440addr.h：个是用于 C 语言的 S3C2440 寄存器地址定义文件，使用这个文件，可以在 C 语言程序中直接用各寄存器的名称作为地址来编程，而不必使用 32 位的二进制地址。
- ✧ main.c：主程序，这个需要你们自己编写的部分，只给出了程序的运行框架，其余部分需要你们自己编写。
- ✧ Ext_RAM.ini：这个是系统调试需要的配置文件。

把以上几个文件 copy 到新建的工程目录下，然后再把相关文件添加到工程里。做实验只需要编写 main.c 里的程序，其余的直接引用就可以了。




调试时，Options for Target 界面相关配置见“6.工程文件相关配置”。

(2) 给接口寄存器赋值，要求使用与或操作，只修改需要配置的对应 bit 位，其余位都保留其原值不变。即，把要设置的 bit 位先用与的方式全部清 0，再用或的方式设置相关 bit 位。

(3) 中断服务程序里，首先判断是何种中断产生了：

[1] 先判断一级中断是否是 EIN8_23；

[2] 再判断二级中断，即到底是哪个外中断产生了。

(4) 在确定了中断类型后，在进行正式的中断服务处理前，先清除相关的中断标志位，清除顺序如下：

[1] 先清除 EINTPEND 中的标志位；

[2] 再清除中断挂起寄存器 SRCPND；

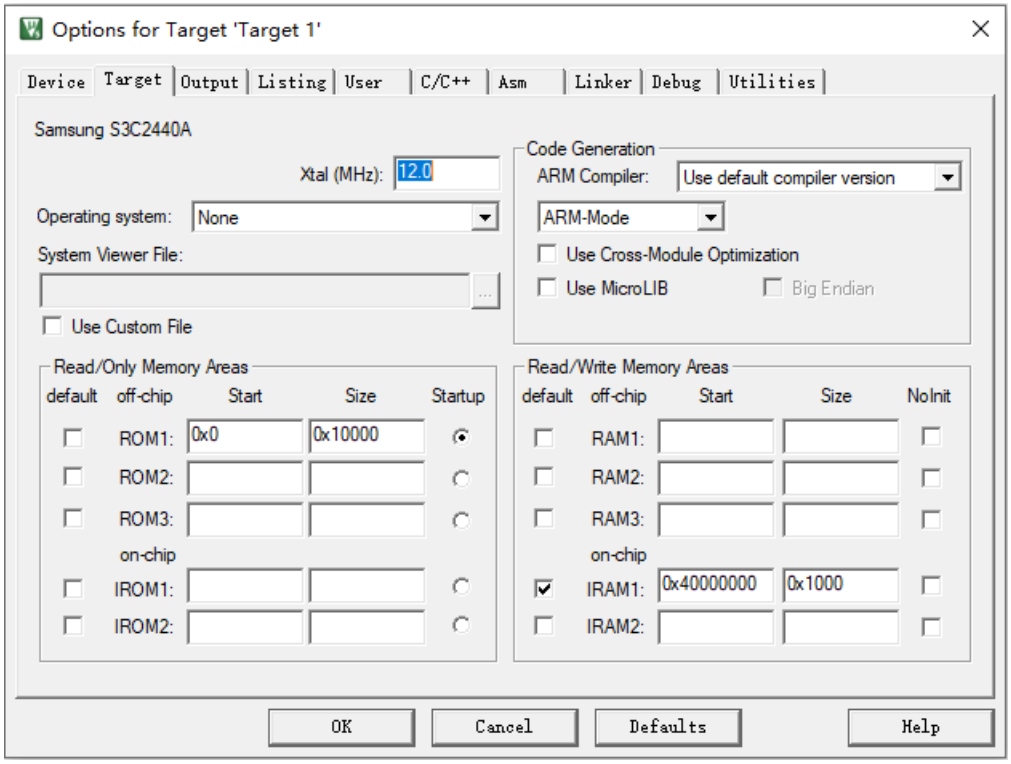
[3] 最后清除源挂起寄存器 INTPND；

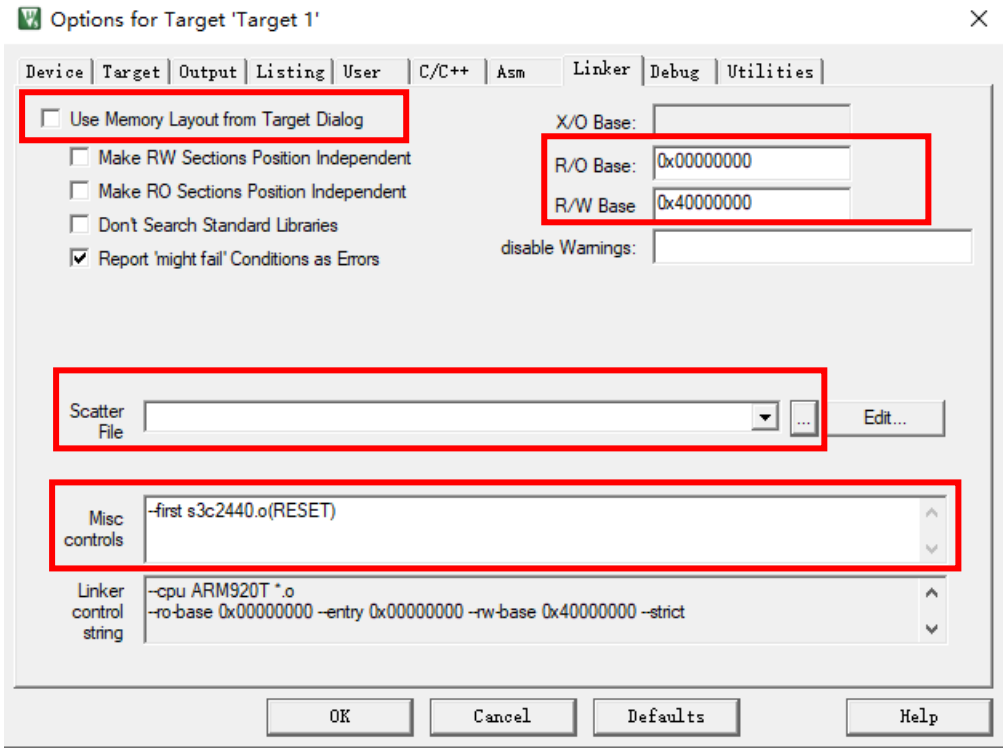
[4] 以上寄存器的相关标志位通过写 1 清 0（使用赋值方法写 1）。

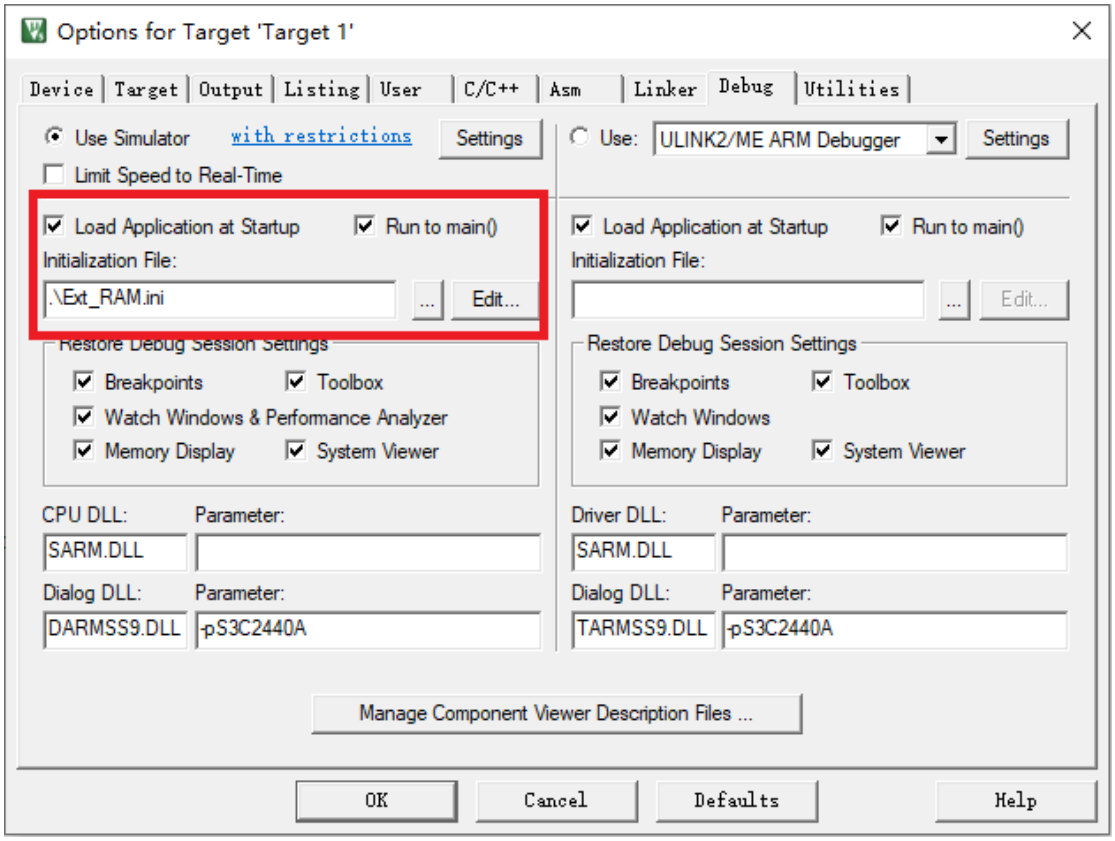
## 6. 工程文件相关配置

可以使用以下两种方式中的一种来配置工程文件：

（1）方式一（推荐配置方式）：


Options for Target 'Target 1'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Samsung S3C2440A
Xtal (MHz): 12.0
Operating system: None
System Viewer File:
Use Custom File
Code Generation
ARM Compiler: Use default compiler version
ARM-Mode
Use Cross-Module Optimization
Use MicroLIB
Big Endian
Read/Only Memory Areas
default off-chip Start Size Startup
ROM1: 0x0 0x10000
ROM2:
ROM3:
on-chip
IROM1:
IROM2:
Read/Write Memory Areas
default off-chip Start Size NoInit
RAM1:
RAM2:
RAM3:
on-chip
IRAM1: 0x40000000 0x1000
IRAM2:
OK
Cancel
Defaults
Help

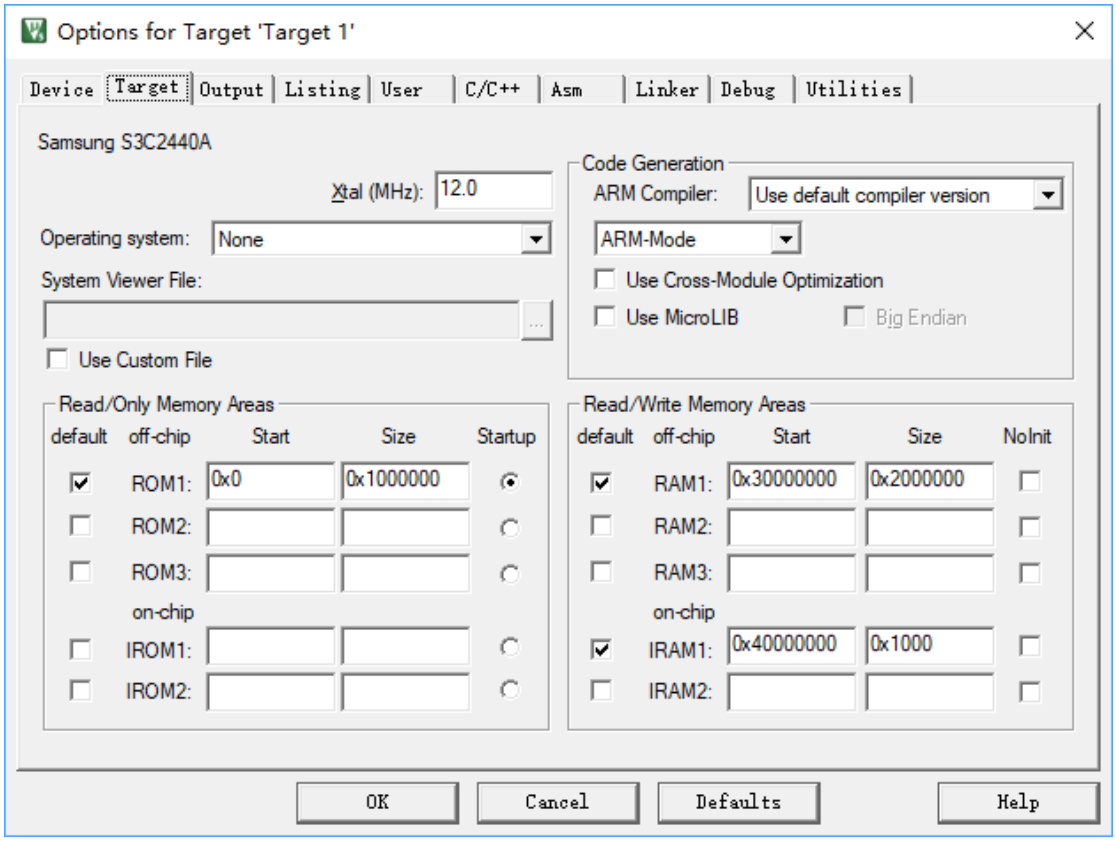

Options for Target 'Target 1'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Use Memory Layout from Target Dialog
Make RW Sections Position Independent
Make RO Sections Position Independent
Don't Search Standard Libraries
Report 'might fail' Conditions as Errors
X/O Base:
R/O Base: 0x00000000
R/W Base 0x40000000
disable Warnings:
Scatter File
Edit...
Misc controls
--first s3c2440.o(RESET)
Linker control string
--cpu ARM920T *.o
--ro-base 0x00000000 --entry 0x00000000 --rw-base 0x40000000 --strict
OK
Cancel
Defaults
Help

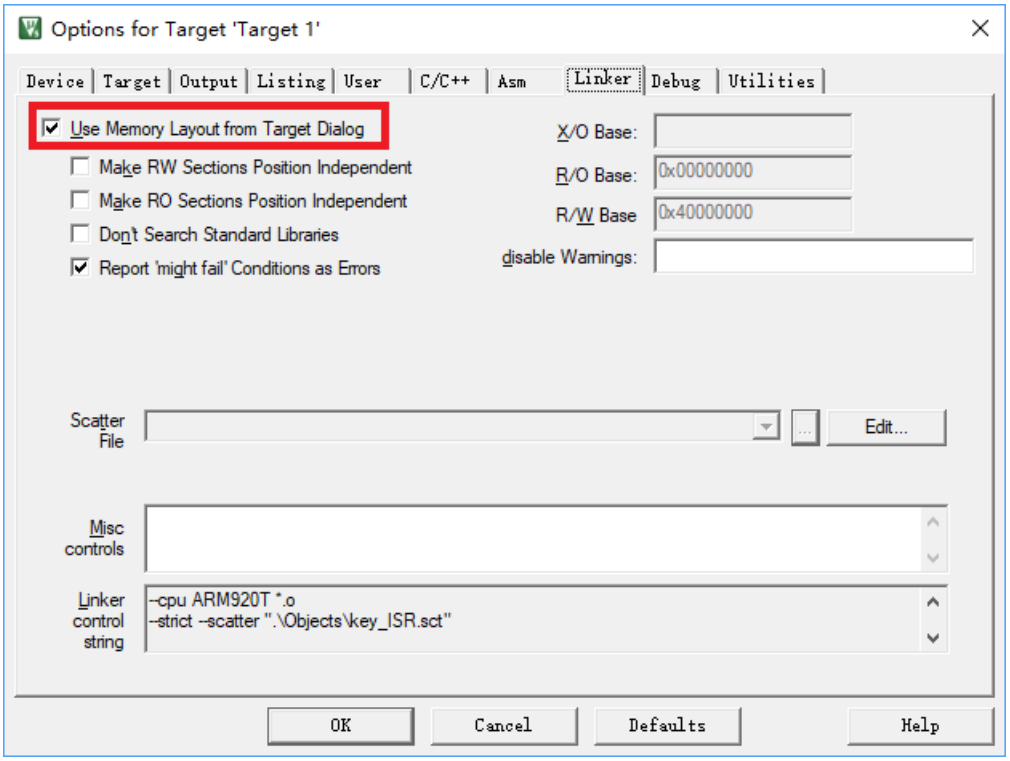


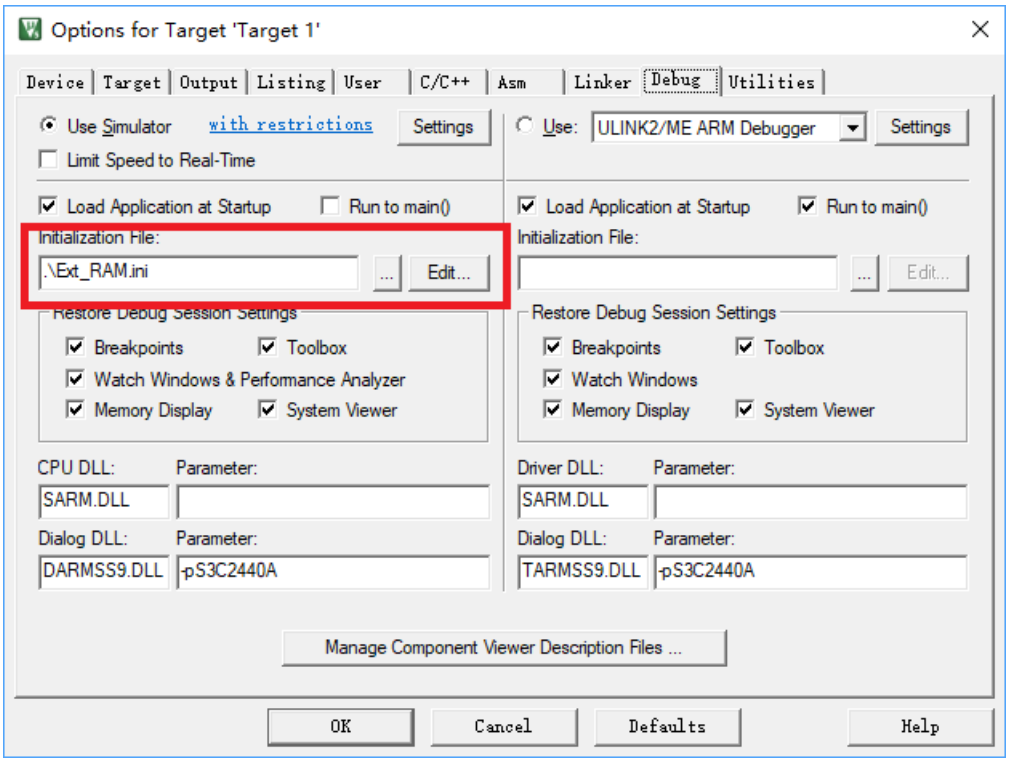

Ext_RAM.ini 的内容如下：

```
map 0x30000000,0x31000000 read write
map 0x53000000,0x53010000 read write
map 0x48000000,0x48010000 read write
```

（1）方式二：


Options for Target 'Target 1'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Samsung S3C2440A
Xtal (MHz): 12.0
Operating system: None
System Viewer File:
Use Custom File
Code Generation
ARM Compiler: Use default compiler version
ARM-Mode
Use Cross-Module Optimization
Use MicroLIB
Big Endian
Read/Only Memory Areas
default off-chip Start Size Startup
ROM1: 0x0 0x1000000
ROM2:
ROM3:
on-chip
IROM1:
IROM2:
Read/Write Memory Areas
default off-chip Start Size NoInit
RAM1: 0x30000000 0x2000000
RAM2:
RAM3:
on-chip
IRAM1: 0x40000000 0x1000
IRAM2:
OK
Cancel
Defaults
Help


Options for Target 'Target 1'
Device
Target
Output
Listing
User
C/C++
Asm
Linker
Debug
Utilities
Use Memory Layout from Target Dialog
Make RW Sections Position Independent
Make RO Sections Position Independent
Don't Search Standard Libraries
Report 'might fail' Conditions as Errors
X/O Base:
R/O Base: 0x00000000
R/W Base 0x40000000
disable Warnings:
Scatter File
Edit...
Misc controls
Linker control string
--cpu ARM920T *.o
--strict --scatter ".\Objects\key_ISR.sct"
OK
Cancel
Defaults
Help




## 7. 仿真操作

系统的仿真和实验 5 的仿真过程类似：参看输出波形和实验 5 查看输出波形的方式完全一样。而中断信号的产生，则和实验 5 按键输入信号的产生类似，设置外中对应端口的输入为低电平即可。**注：本次实验建议中断采用下降沿触发方式。**

**【实验报告】**

1. 撰写设计过程并展示源代码，代码需要进行注释。
2. 展示运行结果并论述。