第4章 程序设计

在μ'nSP™单片机的汇编程序设计中，用户可以不用考虑程序代码在实际物理存储器中的存储地址，而是通过伪指令（如“.CODE”、“.TEXT”、“.RAM”等）来通知编译器把程序代码定位在什么类型的存储空间即可.至于具体的存储地址则由编译器管理。对于数据存储器的管理同样由 IDE 的编译器来完成。当用户想在数据存储区内定义一个变量时，只需通过伪指令（如“.RAM”、“.IRAM”等）来通知编译器在数据存储区内建立一个变量即可。

μ'nSP™单片机的汇编指令针对 C 语言进行了优化，所以其汇编的指令格式很多地方直接类似于 C 语言。另外其开发仿真环境 IDE 也直接提供了 C 语言的开发环境，C 函数和汇编函数可以方便地进行相互调用，详细方法在本章节中将详细介绍。

4.1 μ'nSP™ IDE 的项目组织结构

项目提供用户程序及资源文档的编辑和管理，并提供各项环境要素的设置途径。因此，用户从编程到调程之前实际上都是围绕着项目的操作。

新建项目包括三类文件：源文件(Source files)、头文件(Head files)和用来存放文档或项目说明的文件(External Dependencies)其组织结构如表 4.1 所示。这种项目管理的方式，会把与项目相关的代码模块组织为一个有机的整体，便于开发人员对其代码以及相关文件文档的管理。在表 4.1中，详细描述了一个新建项目后自动产生的各种文件。

在这里，不详细叙述如何对 IDE 进行全面的设置，相关内容可以参阅 IDE 章节。但是从编写调试代码的角度来看，需要反复提出的有如下一些重要的设置：

1) 路径的设置：菜单 tools>>option…>>Directiories，可以进行路径的设置。当项目中的文件或函数库不与项目文件在同一个目录时，需要对此进行设置。

2) 链接库函数的加载：菜单 Project>>Setting….>>Link，可以加载应用函数库。例如，在语音应用时，需要加载凌阳音频算法库 SACM25.lib。

另外，尽管在项目中的 Head File 文件夹下面加入了所需要的头文件，但是在汇编文件和 C 文件中仍然需要用伪指令将其包含到自己的文件中。

μ' nSP™ IDE 开发系统提供了 SPCE061A 的寄存器定义的汇编头文件 hardware.inc 以及 C 语言的头文件 hardware.h。当我们需要对芯片设置时，需要将这些头文件加入项目中。开发系统还提供了对芯片进行设置的一些子函数，这些子函数都放在汇编文件 hardware.asm 中，提供开发人员使用。在凌阳的语音算法函数库中所提供的 API 函

数，也将用到 hardware.asm 中的函数。

表4.1 μ'nSP™ IDE 新建项目的结果

自动生成文件			File 视窗建立元组	
名称	文件名或 文件扩展名	包含信息	Source Files	用于存放源文件，经编译生成扩展名为 obj 的目标文件
项目文件	.scs	当前项目中源文件的信息	Head Files	用于存放头文件，通常是一些要包含在源文件中的接口
资源文件	.rc	当前项目中资源文件的信息		
资源表[注]	Resource.asm		External Dependencies	用来存放文档记录或项目说明等文件
资源表头文件	Resource.inc		Resource 视窗建立 Resource 元组	
MAKE 文件	Makefile	当前项目中重新编辑的文件信息	用来存放项目的资源文件	

注：μ'nSP™ IDE 根据 Resource 视窗里的资源树结构建立起一个资源表。而此资源树的结构可通过用鼠标对各资源文件名的拖拽来改变。

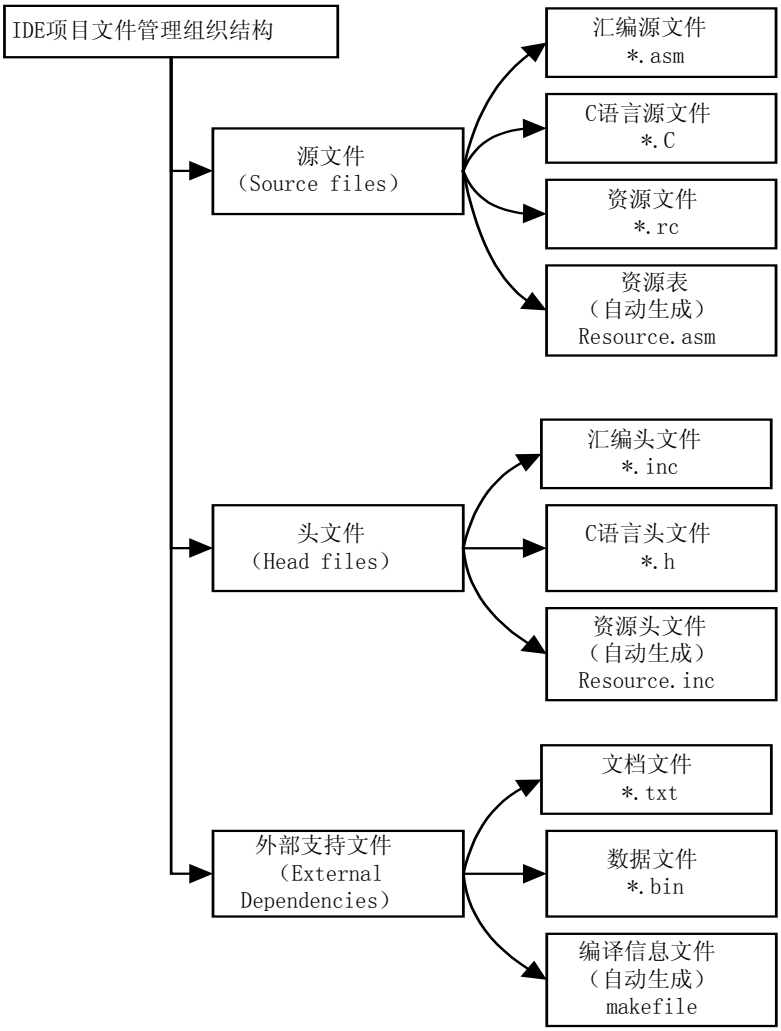


图4.1 IDE 项目文件管理的组织结构

4.2 汇编语言程序设计

C 的编译器 GCC 把 C 语言代码编译为汇编代码。汇编编译器 Xasm16 对汇编代码进行编译成为目标文件。链接器将目标文件、库函数模块、资源文件连接为整体形成一个可在芯片上运行的可执行文件。这样的一个代码流动过程见图 4.2所示。

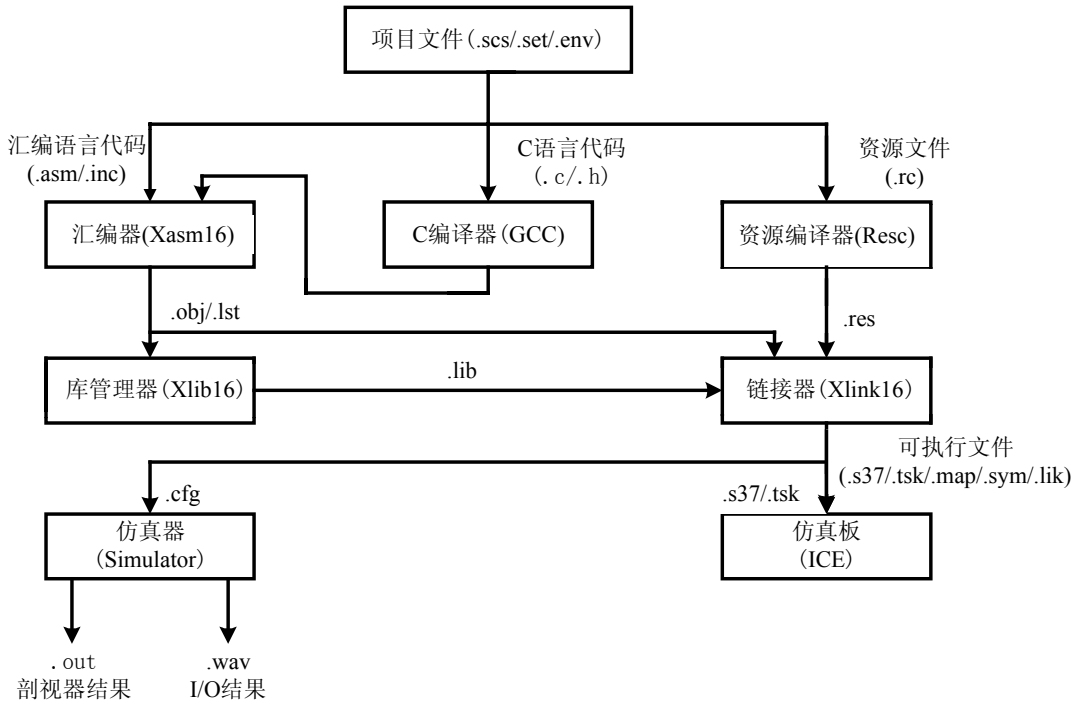


图4.2 代码流动结构示意

μ'nSP™的汇编指令只有单字和双字两种，其结构紧凑，且最大限度地考虑了对高级语言中 C 语言的支持。另外，在需要寻址的各类指令中的每一个指令都可通过与 6 种寻址方式的组合而形成一个指令子集，目的是为增强指令应用的灵活性和实用性。而算逻运算类指令中的 16 位×16 位的乘法运算指令（Mul）和内积运算指令（Muls），又提供了对数字信号处理应用的支持。此外，复合式的「移位算逻操作」指令允许操作数在经过 ALU 的算逻操作前可先由移位器进行各种移位处理，然后再经 ALU 的算逻运算操作。灵活、高效是μ'nSP™指令系统的显著特点。

4.2.2 一个简单的汇编代码

程序4-1 是 IDE 开发环境中所提供的一个 1 到 100 累加的范例。

```
//*********************************************************************************************/
```

```
// 名称：  4_1
// 描述：  计算 1 to 100 累加值
// 日期：  2002/12/10
//*************************************************************************************/
.RAM                            // 定义预定义 RAM 段
.VAR     I_Sum;                 // 定义变量
.CODE                           //定义代码段
//=========================================================================
// 函数：  main()
// 描述：主函数
//=========================================================================

.PUBLIC  _main;                 // 对 main 程序段声明
_main:                          //主程序开始
         R1  =  0x0001;         // r1=[1..100]
         R2  =  0x0000;         // 寄存器清零
L_SumLoop:
         R2  +=  R1;            // 累计值存到寄存器 r2
         R1  +=  1;             // 下一个数值
         CMPR1,100;             // 判断是否加到 100
         JNA L_SumLoop;         // 如果 r1 <= 100 跳到 L_SumLoop
         [I_Sum]  =  R2;        // 在 I_Sum 中保存最终结果
L_ProgramEndLoop:               // 程序死循环
         JMP L_ProgramEndLoop;

//*************************************************************************************/
// main.c 结束
//*************************************************************************************/
```

在此程序中，可以看到：

1) 汇编必须有一个主函数的标号“_main”，而且必须声明此“_main”为全局型标号：“.PUBLIC　_main”

2) 程序代码没有定义实际的物理地址，而是以伪指令“.CODE”声明此程序代码可以定位在任何一个程序存储区内。汇编代码在程序存储区中的定位则由 IDE 负责管理。

3) 程序用伪指令“.RAM”在数据存储区内声明了一个变量“R_Sum”，我们无需关心“R_Sum”的实际物理地址，IDE 将负责安排和管理数据变量在数据存储区的地址安排。

4) 变量名 R_Sum 实际上代表了变量的地址，在汇编中我们对变量进行读写操作时，则需要用[R_Sum]来表示变量中的实际内容。此部分的详细内容可以阅读第三章中的数据寻址方式一节。

通过这段汇编代码，是让初学 SPCE061A 的读者对程序设计建立一个基本概念，下面我们将详细介绍汇编语言的编程方法。

4.2.3 汇编的语法格式

用μ'nSP™的汇编指令编写程序需按一定的语法规则和格式进行。汇编指令就像是语言中的单词。这些单词，如何组织成能让μ'nSP™汇编器识别的汇编语言，并由此被编译成 CPU 所能识别和执行的机器码？很简单，只要遵循汇编器规定的规则和格式即可。

4.2.3.1 数制、数据类型与参数

μ'nSP™的汇编器将十进制作为缺省数制。十六进制数可用符号“0x”或“$”作为前缀，或用符号“H”作为后缀。对于其它数制的后缀可见表 4.2中所列。

表4.2 μ'nSP™的数制及其后缀规定

数制	后缀
二进制	B
八进制	O 或 Q
十进制	D 或不写
十六进制	H
ASCII 字符串	用双引号或单引号括起，如：“5”或‘5’

μ'nSP™汇编指令中所用的基本数据类型为字型，在此基础之上发展的一些数据类型与字型一起列在表 4.3中。

表4.3 μ'nSP™汇编指令中的数据类型

数据类型	字长度（位数）	无符号数值域	有符号数值域
字型（DW）	16	0~65535	-32768~+32767
双字型（DD）	32	0~4294967295	-2147483648~+2147483647
单精度浮点型（FLOAT）	32	无	以 IEEE 格式表示的 32 位浮点数
双精度浮点型（DOUBLE）	64	无	以 IEEE 格式表示的 64 位浮点数

汇编指令中的参数可以是常数或表达式。常数参数基本有数值型和字符串型两种。数值型参数将按当前数值的数制进行处理（缺省为 10 进制）。如果用户强调参数用某一种数制，则必须给数值加必要的前缀或后缀来表示清楚。

4.2.3.2 连接运算符及其优先次序

在μ'nSP™的汇编指令中可用一些运算符来连接常量数值，或者用一些修饰符对常量数值进行修饰；以便于程序员灵活编程以及μ'nSP™汇编器的辨认或操作。表 4.4将这些运算符或修饰符列出。

表4.4 μ'nSP™汇编指令中的连接运算符及修饰符

运算符、修饰符	操作内容描述
!，&&，||	逻辑非，逻辑与，逻辑或
^或 XOR，&或 AND，|或 OR	按位非，按位与，按位或
+、-	(一元操作修饰符) 任意指定一个正、负操作数或表达式
*、/、+、-	无符号数乘法、除法、加法、减法
>>、<<[1]	把移位操作符前的数值或表达式向右、左移位；右移时最高位用 0 填充，左移时最低位用 0 填充
==，!=，>，<，>=，<=[2]	等于，不等于，大于，小于，大于等于，小于等于

IM6、A6 SEG、OFFSET[3]	(一元修饰符)引入一个 6 位(第 0~5 位)的数字表达式、地址表达式 引入 22 位地址表达式中的 6 位（第 16-21 位）、16 位（第 0-15 位）常量数值，用来修饰一个可再分配的地址值的页域或偏移量域

注：

[1]:移位操作符（>>、<<）后的数值表示的是移位位数。

[3]:修饰操作必须用空格起始及结束。修饰符与芯片的地址容量有关。

上表中的连接运算符必须依照约定的优先次序操作。表 4.5将此优先次序列出，同一行运算符具有相同的优先次序。约定的优先次序可以用圆括弧强制改变。

表4.5 运算符的优先次序

优先级别	运算符
最高 ↓ 最低	‘!’ ‘-’，‘+’ //一元操作符，用来指定正操作数和负操作数 ‘%’，‘/’，‘*’ ‘-’，‘+’ //减/加符号 ‘>>’，‘<<’ ‘<’，‘<=’，‘>’，‘>=’ ‘!=’，‘==’ ‘^’或者 ‘xor’，‘&’或者‘and’，‘|’或者‘or’ ‘||’，‘&&’

4.2.3.3 地址表达式与标号

修饰符 SEG 与 OFFSET 常常应用在计算表的地址。例如有如下一张 1 到 10 的平方表：

```
.CODE
Square_Table:
    .DW   1,4,9,16,25,36,49,64,81,100
```

在编译链接过程中，链接器将自动在将以上 10 个数据放在程序存储区内，并且 Square_Table 就代表了此 10 个连续数据的起始 22 位的起始地址。那么“SEG Square_Table”就代表了 22 位地址的高 6 位，而“OFFSET Square_Table”则代表了 22 位地址的低 16 位。

SPCE061A 只有 32kword 的程序存储空间，所以其高 6 位的地址一定为 0。如果上面的 1 到 10 的平方表应用在 SPCE061A 中，那么常量 Square_Table 与常量 OFFSET Square_Table 的值是相等的。

我们想通过标号要得到一段程序代码或表的实际的物理地址时，往往需要 SEG 和 OFFSET 这样的修饰符。

μ’nSP™汇编语言程序中所有标号的定义都是字母大小写区分的。全局标号原则上可以由任意数量的字母和数字字符组成，但只有前 32 位是有效的。它可以写在文件中的任何一列上，但必须以字母字符或下划线（_）开头，且标号名后须以冒号（：）来结束。在下面的程序例子中 LABEL1，LABEL2 和 LABEL3 都是全局标号。

局部标号只有在局部区域内定义才有效。正是这种约束才使得局部标号的定义可以安全地重复使用在整个程序各处而不致产生混乱错误。

局部标号应当注意以下几点：

1) 局部标号也像全局标号那样最多可有 32 个字母或数字字符，且必须以字母字

符或问号（？）开头。μ'nSP™的汇编器通常规定用问号（？）作为局部标号的前缀或后缀。除此之外，局部标号最好也遵循全局标号的使用规则。

2) 在不同的局部区域里所定义的局部标号都有不同的含义，且标号?a 是不同于标号 a?的。

3) 切忌将诸如“+、-、*、/”这类运算符用在局部标号中。

伪指令 VAR，SECTION 或 ENDS 是不可以用在局部标号结尾处的（见下面μ'nSP™汇编器的伪指令内容）。

```
LABEL1：                或者          LABEL1：
?a：    NOP                           a?：    NOP
?b：    JMP     ?a                    b?：    JMP     a?
JMP     ?b                            JMP     b?
```

4.2.3.4 程序注释与符号规定

程序注释行必须用双斜线（//）或分号（；）起始，它可与程序指令在同一行，或跟在指令后，亦可在指令的前一行或后一行。

μ'nSP™的汇编器规定伪指令不必区分字母的大小写，亦即书写伪指令时既可全用大写，也可全用小写，甚至可以大小写混用。但所有定义的标号包括宏名、结构名、结构变量名、段名及程序名则一律区分其字母的大小写。

4.2.4 汇编语言的程序结构

程序最基本的结构形式有顺序、循环、分支、子程序四种结构。顺序结构在这里不作讨论，在此章节中将从分支、循环、子程序出发，向读者介绍嵌套递归与中断程序的设计方法。

4.2.4.1 分支程序设计

分支结构可分为双分支结构和多分支结构两种如图 4.3所示。在程序体中，根据不同的条件执行不同的动作,在某一确定的条件下，只能执行多个分支中的一个分支。

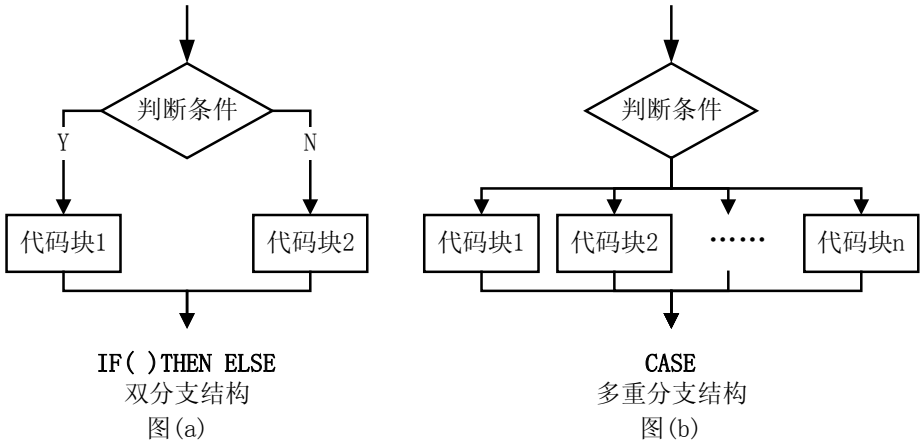


图4.3 分支结构的两种形式

由于高级语言提供了 IF….ELSE 或 SWITCH….CASE…CASE 的语句，使得分支结构的层次清晰，分支路径明确。然而在汇编语言中，只能依靠跳转语句实现这样的结构，那么遵循这单入口单出口的程序设计方法，显得尤其重要。图 4.5表示了一个两分支结构的汇编语言实现方式。图 4.5给出了汇编语言实现多重分支的一种方式。

在实际的程序开发过程中，我们不仅仅追求功能的实现，还要保证代码的稳定性、通用性、可读性等等。汇编语言不具备高级语言的指令，所以在代码编写的过程中尽量使得结构清晰，功能明确。

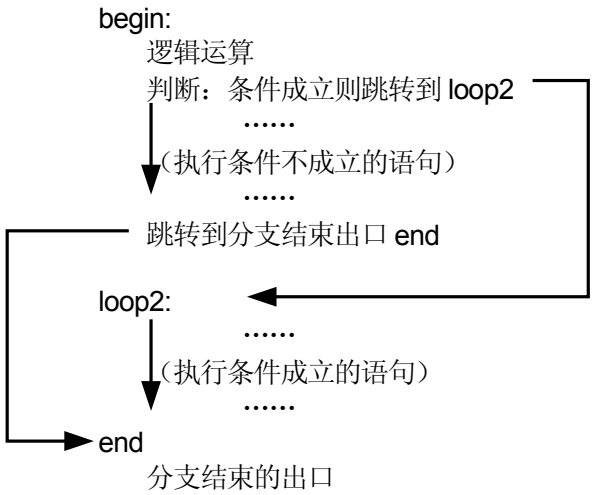


图4.4 汇编语言实现分两分支路径的方式

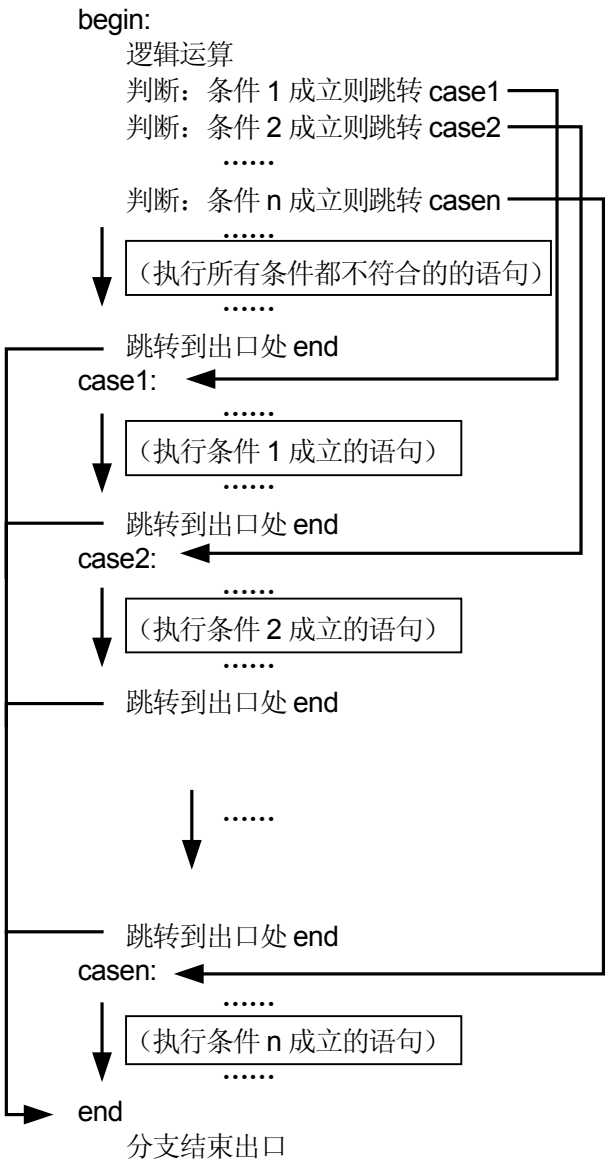


图4.5 汇编语言多重分支方式

下面我们用相应的例子来详细的说明这两种分支结构。

程序4-2 阶跃函数

说明：这是一个典型的双分支结构，输入值大于等于 0 时则返回 1，输入值小于 0 时返回 0。

入口参数：R1;（有符号数）

出口参数：R1

子程序名：F_Step

流程图如图 4.6所示。

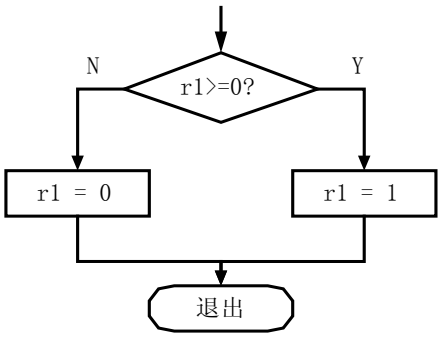


图4.6 阶跃函数流程图

程序的代码如下：

```
    .PUBLIC F_Step;
    .CODE
    F_Step:    .proc
     CMP R1,0;          //与 0 比较
     JGE ?negtive;      //大于等于 0 则跳转到非负数处理
     R1 = 0;            //小于 0 则返回 0
     JMP ?Step_end;     //跳转到程序结束处
?negtive:
     R1 = 1;            //大于 0，则返回 1
?Step_end:
     RETF;
     .ENDP
```

下面的例子是 IDE 开发环境提供的语音应用程序的范例“A2000”中断服务子程序。由于产生 FIQ 中断的中断源有三种：TimerA、TimerB 和 PWM。所以在中断中要进行判断，根据判断的结果跳转到相应的代码中。其流程图见图 4.7所示。

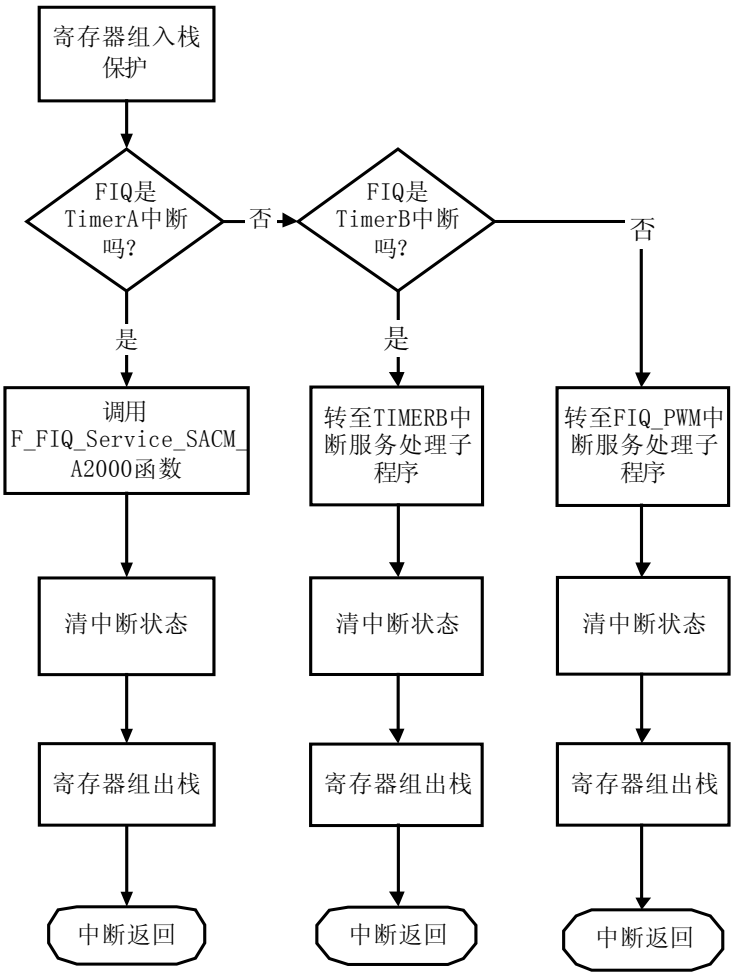


图4.7 A2000 中断服务程序中采用的分支结构的程序设计

程序4-3 A2000 的中断服务程序

```
//=============================================================================
//函数: FIQ()
//语法：void FIQ(void)
//描述：FIQ 中服务断函数
//参数：无
//返回：无
//=============================================================================
.PUBLIC _FIQ;
_FIQ:
    PUSH    R1,R4 TO [sp];
    R1=0x2000;
    TEST R1,[P_INT_Ctrl];
    JNZ L_FIQ_TimerA;
```

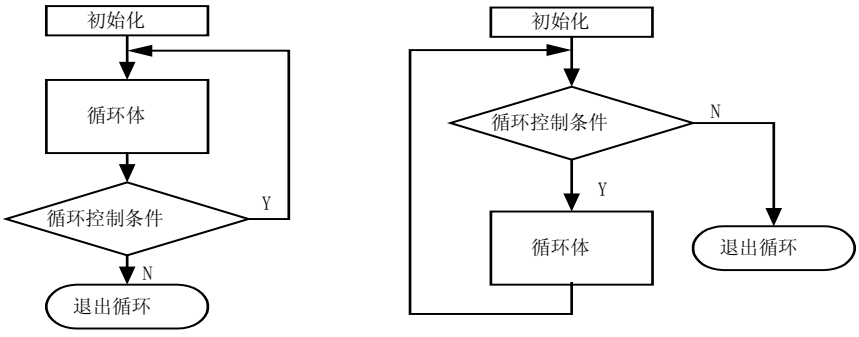
```
    R1=0x0800;
    TEST  R1,[P_INT_Ctrl];
    JNZ  L_FIQ_TimerB;
L_FIQ_PWM:
    R1=C_FIQ_PWM;
    [P_INT_Clear]=R1;
    POP  R1,R4  from[sp];
    RETI;
L_FIQ_TimerA:
    [P_INT_Clear]=R1;
    CALL  F_FIQ_Service_SACM_A2000;          //调用 A2000 中断服务函数
    POP  R1,R4  FROM  [sp];
    RETI;
L_FIQ_TimerB:
    [P_INT_Clear]=R1;
    POP  R1,R4  FROM  [sp];
    RETI;
//*************************************************************************************************/
void  F_FIQ_Service_SACM_A2000（）；来自 sacmv25.lib,API 接口函数。
//*************************************************************************************************/
```

4.2.4.2 循环程序设计

汇编语言中没有专用的循环指令，但是可以使用条件转移指令通过条件判断来控制循环是继续还是结束。

4.2.4.2.1 循环程序的结构形式。

在一些实际应用系统中，往往同一组操作要重复许多次，这种强制 CPU 多次重复执行一串指令的基本程序结构称为循环程序结构。循环程序可以有两种结构形式，一种是 WHILE_DO 结构形式；另一种是 DO_UNTIL 结构形式。如图 4.8所示：



（a）DO_UNTIL结构　　（b）WHILE_DO结构

图4.8 循环结构的两种方式

WHILE_DO 结构把对循环控制条件的判断放在循环的入口，先判断条件，满足条件就执行循环体，否则退出循环。DO_UNTIL 结构则先执行循环体然后再判断条件，满足则继续执行循环操作，一旦不满足条件则退出循环。这两种结构可以根据具体情况选择使用。一般来说，如果有循环次数等于 0 的可能则应选择 WHILE_DO 结构。不论哪一种结构形式，循环程序一般由三个主要部分组成：

1) 初始化部分：为循环程序做准备，如规定循环次数、给各个变量和地址指针预置初值。

2) 循环体：每次都要执行的程序段，是循环程序的实体，也是循环程序的主体。

3) 循环控制部分：这部分的作用是修改循环变量和控制变量，并判断循环是否结束，直到符合结束条件时，跳出循环为止。

下面是这两种循环结构的举例：

程序4-4 数据搬运

把内存中地址为 0x0000～0x0006 中的数据，移到地址为 0x0010～0x0016 中。流程图如图 4.9所示。

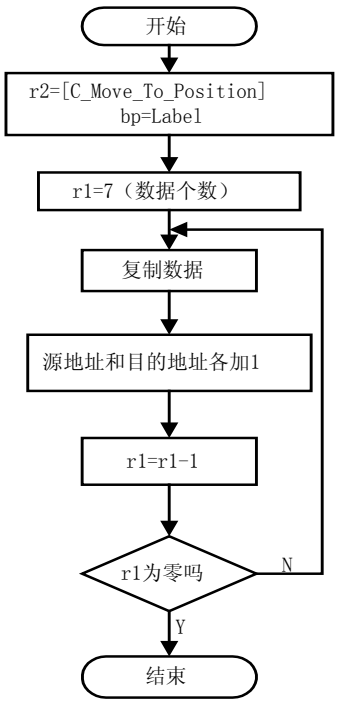


图4.9 程序流程图

```
.//*************************************************************************/
// 描述：  把内存中地址为 0x0000～0x0006 中的数据，移到地址为 0x0010～0x0016 中。
// 日期：  2002/12/6
//*************************************************************************************/
.IRAM
Label:
.DW  0x0001,0x0002,0x0003,0x0004,0x0005,0x0006,0x0007;
.VAR  C_Move_To_Position=0x0010;     //定义起始地址；
.CODE
//=========================================================================
// 函数：  main()
// 描述：主函数
//=========================================================================
.PUBLIC  _main;
_main:

      R1=7;                          //设置要移动的数据的个数
      R2=[C_Move_To_Position];
      BP=Label;
L_Loop:
      R3=[BP];                       //被移动的数据送入 r3
      [R2]=R3;                       // 被移动的数据送往目的地址
      BP+=1;                         //源地址加 1
      R2=R2+1;                       //目的地址加 1
      R1-=1;                         //计数减 1
      JNZ  L_Loop;

MainLoop:
      jmp  MainLoop;

//*************************************************************************************/
// main.c 结束
//*************************************************************************************/
```

程序4-5 延时程序

向 B 口送 0xffff 数据，点亮 LED 灯，延时 1 秒后，再向 B 口送 0x0000 数据，熄灭 LED 灯。程序代码如下，其中延时子程序的流程图如图 4.10所示。

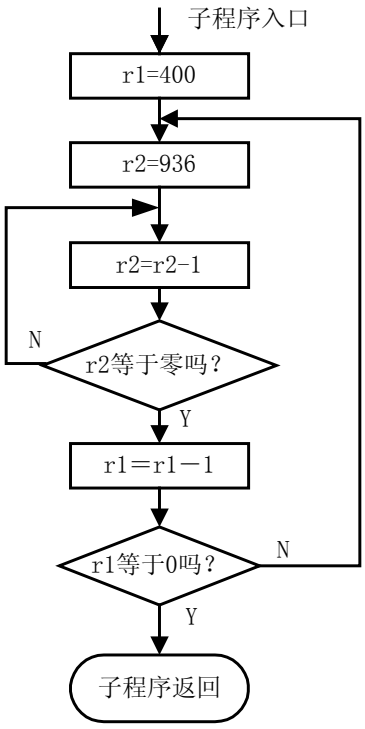


图4.10 时间延时子程序流程图

```
//*****************************************************************************************/
// 描述：  延时程序，向 B 口送 0xffff 数据，点亮 LED 灯，延时 1 秒后，再向 B 口送 0x0000 数据，
//         熄灭 LED 灯。
// 日期：  2002/12/10
//*****************************************************************************************/
.DEFINE  P_IOB_DATA             0x7005;
.DEFINE  P_IOB_DIR              0x7007;
.DEFINE  P_IOB_ATTRI            0x7008;
.CODE
//=========================================================================
// 函数：  main()
// 描述：主函数
//=========================================================================
.PUBLIC  _main;
_main:
    R1=0xffff;
    [P_IOB_DIR]=R1;
    [P_IOB_ATTRI]=R1;
    R1=0x0000;
    [P_IOB_DATA]=R1;            //设 B 口为同相的低电平输出
```

```
L_MainLoop:
      R2=0xffff;
     [P_IOB_DATA]=R2                    //向 B 口送 0xffff；
     CALL  L_Delay;                     //调用 1 秒的延时子程序
     R2=0x0000;
     [P_IOB_DATA]=R2;                   //向 B 口送 0x0000；
     CALL  L_Delay;                     //调用 1 秒的延时子程序
     JMP  L_MainLoop;

//=============================================================================
//函数: L_Delay()
//语法：void L_Delay(int A,int B，int C)
//描述：延时子程序
//参数：无
//返回：无
//=============================================================================
L_Delay: .PROC                          //延时 1 秒的子程序
      loop:
      R1=200;
L_Loop1:
      R2=1248;
      nop;
     nop;
L_Loop2:
     R2-=1;
     JNZ  L_Loop2;
     R1-=1;
     JNZ  L_Loop1;
 RETF;
 .ENDP

//*****************************************************************************************/
// main.c 结束
//*****************************************************************************************/
```

下面分析一下如何进行时间延时，延时时间主要与两个因素有关：其一是循环体（内循环）中指令执行的时间；其二是外循环变量（时间常数）的设置。上例选用系统默认的 FOSC,CPUCLK，CPUCLK=FOSC/8=24M/8=3M（Hz），所以一个 CPU 周期为 1/3M(s)。执行一条 r1=400 指令的时间为 4 个 CPU 周期，执行一条 nop 指令的时间为 2 个 CPU 周期，执行 r2－＝1 指令的时间为 2 个 CPU 周期，执行 jnz Loop2 指令的时间为 2 或 4 个 CPU 周期（当条件满足时为 5 个 CPU 周期,条件不满足时 3 个 CPU 周期）。所以上例子中的时间延时子程序的时间为：(4+4+6*1248+2+4-2)*200+4-2=1500002 个 CPU 周期，约为 0.5 秒，有点不精确。故在进行精确的时间延时，一般不采用这种方法，而是采用中断来延时，因为 SPCE061A

单片机有丰富的定时中断源，如：2Hz ，4Hz，128Hz 等。当然在一般的延时程序也可以采用指令延时，它也挺方便的，在上例的延时程序中只要改变 r1 的值就可以很方便地改变延时时间，比如：r1＝4，那么它的延时时间为 10ms。

4.2.4.2.2 多重循环

在某些问题的处理中，仅采用单循环往往不够，还必须采用多重循环才能解决。所谓多重循环是指在循环程序中嵌套有其它循环程序。多重循环程序设计的基本方法和单重循环程序设计是一致的，应分别考虑各重循环的控制条件及其程序实现，相互之间不能混淆。另外应该注意在每次通过外层循环再次进入内层循环时，初始化条件必须重新设置。多重循环体的结构如图 4.11所示。

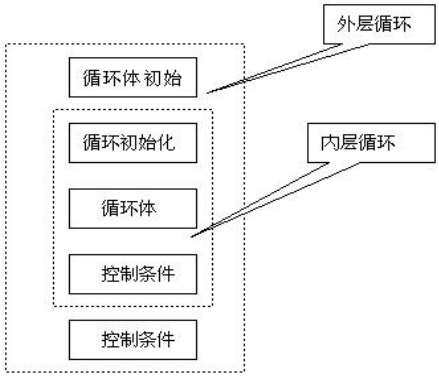


图4.11 多重循环结构

程序4-6 冒泡排序

在首地址为 0x0000 的 RAM 中依次存有数据 40，6，32，12，9，24，28，要求使用冒泡法编写对这些数进行按升序排序的程序。

算法说明：这个算法类似水中气泡上浮,俗称冒泡法 。执行时从前向后进行相邻数比较，如果数据的大小次序与要求顺序不符时（升序），就将两个数互换，否则为正序不互换。为进行升序排序应通过这种将相邻数互换方法，使小数向前移，大数向后移。如此从前向后进一次冒泡，就会把最大数换到最后；再进行一次冒泡，就会把次大数排队在倒数第二的位置；••••••。

原始数据顺序为：40，6，32，12，9，24，28。按升序排列，它们的冒泡过程如图 4.12所示。

原数据	第一轮	第二轮	第三轮	第四轮	第五轮	第六轮
40	6	6	6	6	6	6
6	32	12	9	9	9	9
32	12	9	12	12	12	12
12	9	24	24	24	24	24
9	24	28	28	28	28	28
24	28	32	32	32	32	32
28	40	40	40	40	40	40

图4.12 冒泡过程

针对上述的排序过程，有两个问题需要说明：

1) 由于每次冒泡排序都从上向下排定了一个大数（升序），因此每次冒泡所需进行的比较次数都递减 1。例如有 n 个数排序，则第一次冒泡需比较（n－1）次，第二次比较则需（n－2）次，・・・・・・。但实际编程时，有时为了简化程序，往往把各次的比较次数都固定为（n－1）次。

2) 对于 n 个数，理论上说应该进行（n－1）次冒泡才能完成排序，但实际上有时不到（n－1）次就已经排完。在上述排序过程中，共进行 3 次冒泡就已经完成排序。因此我们需要设置一个标志变量，来判断排序是否完成，如果完成，则不进行下一轮的冒泡，退出循环，冒泡结束。

冒泡排序的程序流程图如图 4.13所示。

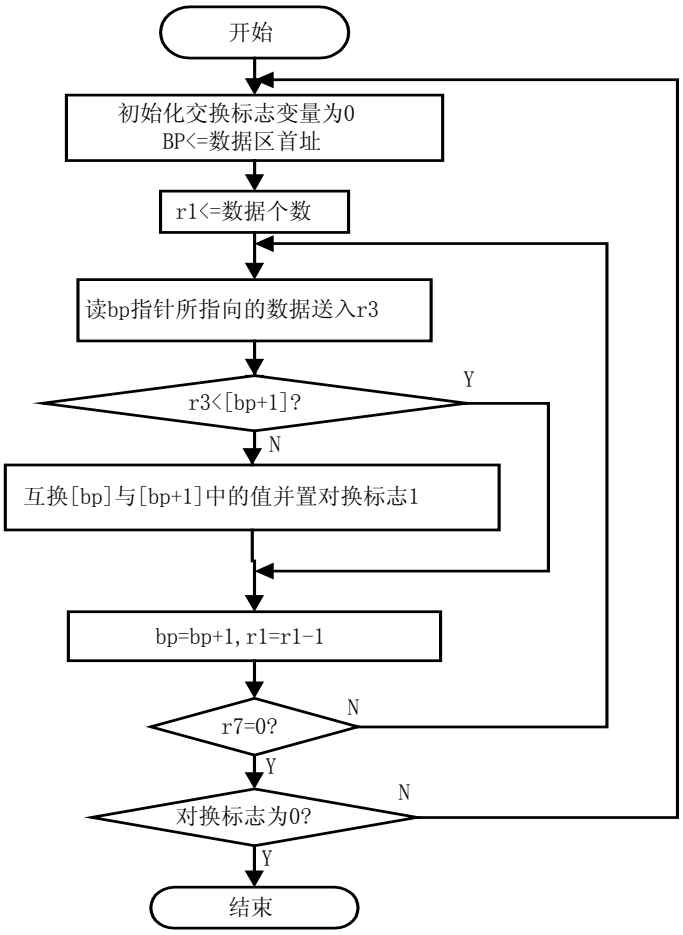


图4.13 冒泡程序流程图

源代码如下。

```
//*****************************************************************************************/
// 描述：  冒泡排序,在首地址为 0x0000 的 RAM 中依次存有数据 40，6，32，12，9，24，28，使
//         用冒泡法编写对这些数进行按升序排序的程序。
// 日期：  2002/12/6
//*****************************************************************************************/
.IRAM
Array: .DW  40,6,32,12 ,9,24,28;
.VAR  C_Flag;                              //定义数据交换标志
.CODE
//=========================================================================
// 函数：  main()
// 描述：主函数
//=========================================================================
.PUBLIC  _main;
_main:
      L_Sort:
      BP=Array;                            //数据的首地址送 bp
      R1=0x0006;                           //数据的个数送 r1
R4=0x0000;
[C_Flag]=R4;                               //清交换标志
L_Loop:
     R3=[BP];
     CMP  R3,[BP+1];                       //两数比较
     JB  L_Next;                           //如果第一个小于第二个数则跳转
     R2=[BP+1];
     [BP]=R2;                              //第二个数移到存第一数的单元中
     [BP+1]=R3;                            //第一个数移到存第二数的单元中
     R3=0x0001;
     [C_Flag]=R3;                          //交换标志置 1
L_Next:
      BP=BP+1;                             //地址增 1
      R1-=1;                               //计数为 0 吗？，否则跳转
      JNZ  L_Loop;
      R4=[C_Flag]
      JNZ  L_Sort;                         //交换标志为 0 吗？否则跳到 SORT
L_MainLoop:
      JMP  L_MainLoop;

//*****************************************************************************************/
// main.c 结束
```

//***/

4.2.4.3 子程序

在实际应用中，经常会遇到在同一程序中，需要多次进行一些相同的计算和操作，例如：延时，算术运算等。如果每次使用时都再从头开始编写这些程序，则程序不仅繁琐，而且浪费内存空间，也给程序的调试增加难度。因此，可以采用子程序的概念，将一些重复使用的程序标准化，使之成为一个独立的程序段，需要时调用即可。我们就把这些程序段称作为子程序。一般来说子程序的结构包括三个部分：1.子程序的定义声明和开始标号部分；2.子程序的实体内容部分，表明程序将进行怎样的操作；3.子程序的结束标号部分。子程序的结构可以用图 4.14所示。

图4.14 子程序结构

程序的调用包括主程序调用子程序，子程序调用子程序等。程序调用是通过调用指令“CALL ……”来实现的。程序执行的过程中，当遇到调用子程序指令，CPU 便会将下一条指令的地址压入堆栈暂时保护起来，然后转到被调用的子程序入口去执行子程序，当执行到 RETF 时返回，CPU 又将堆栈中的返回地址弹出送到 PC，继续执行原来的程序。其过程如图 4.15所示。

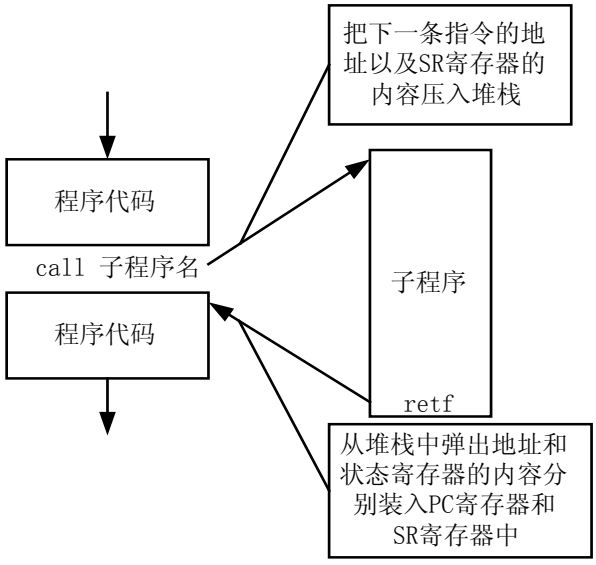


图4.15 子程序调用过程

在程序调用的过程中，需要注意到的问题是断点的现场保护。就是说，子程序将占用的资源是否与主程序冲突，子程序将会破坏什么寄存器的内容，而这些寄存器是否是主程序持续使用的等等。通常的做法是用堆栈对现场进行保护，在子程序开始就把子程序要破坏掉的寄存器的内容压栈保护，当子程序结束的时候，再弹栈恢复现场。

程序调用的过程都伴随着参数的传递，正确的参数传递要满足入口和出口条件。入口条件指执行子程序时所必需的有关寄存器内容或源程序的存储器的存储地址等，主程序调用子程序时必须先满足入口条件，换句话说就是满足子程序对输入参数的约定。出口参数就是指子程序执行完了之后运算结果所存放的寄存器或存储器地址等，也就是说，必须确定主程序对输出参数的约定。

通常来说,参数的传递有以下几种情况：

1) 通过寄存器传递

2) 通过变量传递

3) 通过堆栈传递

下面我们针对每一种情况进行具体讲解，分析。

4.2.4.3.1 通过寄存器传递参数

寄存器传递参数，是最常用的一种参数传递的方式。我们常用到的传递参数的寄存器有 4 个，分别为 R1~R4；在程序调用的过程中，寄存器中的值也会被带到被调用的子程序中供子程序使用。以主程序调用子程序为例：在调用子程序前 R1~R4 这 4 个寄存器中可能暂存一些值，发生调用子程序以后，这些值仍被带到相应的子程序中继续参加子程序的运算，子程序运算结束后返回主程序，这些寄存器的新值也会被带到主程序中继续参加主程序的运算。这个过程也可以用图 4.16来表示。实线表示参数的传递方向是由主程序到子程序，虚线表示参数的传递方向是由子程序到主程序。

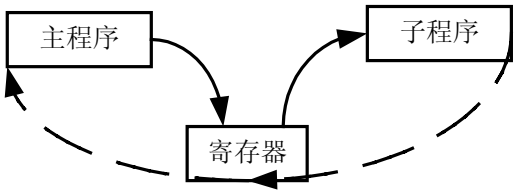


图4.16 通过寄存器传递参数

下面的范例程序，就是利用寄存器传递变量。

程序4-7 求 32 位有符号数的绝对值

```
//===========================================================================
```

```
//函数: F_Abs_32（）
//语法：void F_Abs_32（int A,int B）
//描述：求 32 位有符号数具对值
//参数：r3 有符号数低 16 位，r4 有符号数高 16 位
//返回：r1 绝对值结果的低 16 位，r2 绝对值结果的高 16 位
//=============================================================================
.CODE
.PUBLIC F_Abs_32
F_Abs_32:
    R1 = R3;            //传送低 16 位
    R2 = R4;            //传送高 16 位
    JMI ?neg;           //如果为负则跳转到负数处理
    RETF;               //为正数则无需任何处理，返回
?neg:                   //负数处理
    R1 ^= 0xFFFF;       //低 16 位去反
    R2 ^= 0xFFFF;       //高 16 位取反
    R1 += 1;            //低 16 位加 1
    R2 += 0,Carry;      //高 16 位加进位
    RETF;
```

4.2.4.3.2 通过变量传递参数

通过变量进行的参数传递，主要是通过全局型变量实现的。在汇编中，一个变量名，就代表了一个实际的寄存器的物理地址。可以直接对物理地址进行赋值和读取，但这种的方法会带来很多麻烦。用变量名去代表一个实际的物理地址，就涉及到某部分汇编代码是否认识该变量名的问题。

如果在某个汇编文件中定义了一个全局变量（.PUBLIC），那么此汇编文件中的所有汇编代码都能够使用这个变量。但是在其他的汇编文件中，仍不能直接使用这个变量。在这种情况下，需要在使用这个变量的汇编文件中将该变量声明成外部变量（.external），既可使用这个变量，同时该变量也起到了参数传递的作用。

如图 4.17所示。实线表示参数的传递方向是由主程序到子程序，虚线表示参数的传递方向是由子程序到主程序。

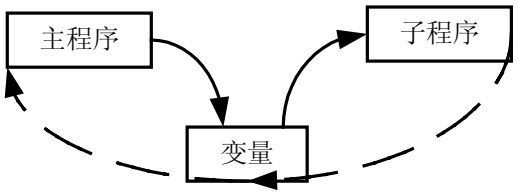


图4.17 利用变量传递参数

4.2.4.3.3 通过堆栈传递参数

在 C 函数与汇编函数的相互调用过程中，主要通过堆栈来传递参数，而在函数返回时，则采用寄存器来传递返回值。在主程序把要传递的参数压入堆栈，然后调用子程序。子程序从堆栈中寻找需要的参数进行处理。当子程序返回后，主程序需要进行弹栈处理，以恢复参数压入堆栈前的堆栈状态，如图 4.18所示。事实上，IDE 开发环境中的 C 语言与汇编语言的相互调用，就是采用堆栈传递参数，寄存器返回参数的方式。SPCE061A 使用 BP 寄存器，可以实现变址寻址方式，可以简洁地实现堆栈传递参数的过程。

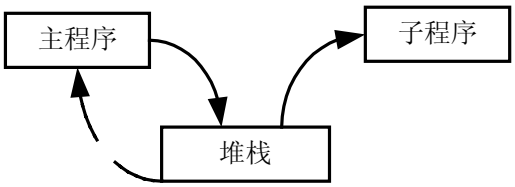


图4.18 堆栈传递参数

在4.3.3.3节中的范例程序 4-14就是一个典型的利用堆栈来传递参数的过程。

4.2.5 嵌套与递归

4.2.5.1 子程序的嵌套

子程序嵌套就是指子程序调用子程序。其中嵌套的层数称为嵌套深度。图 4.19表示了三重嵌套的过程。

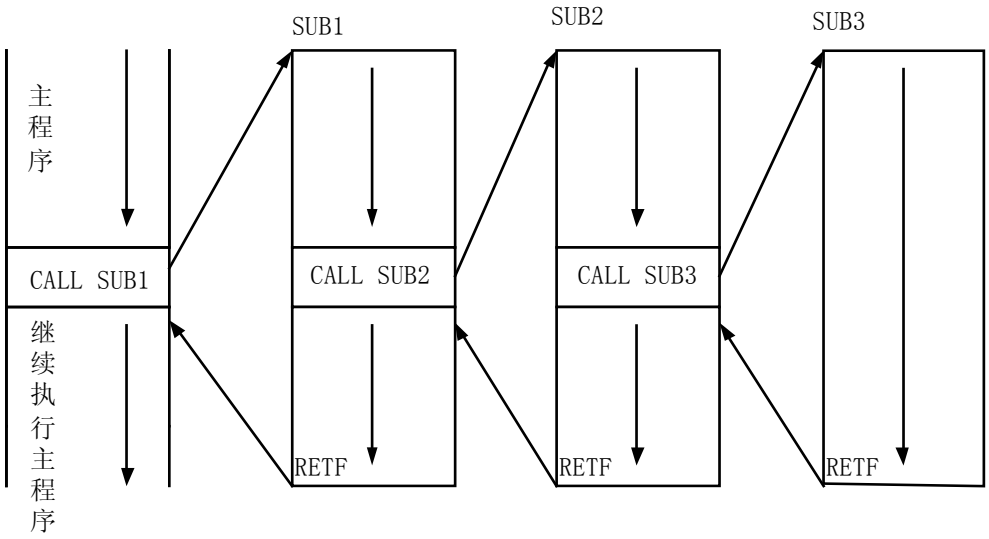


图4.19 三重程序嵌套过程

子程序嵌套要注意以下几个方面：

1) 寄存器的保护和恢复，以避免各层子程序之间发生因寄存器冲突而出错的情况。

2) 程序中如果使用了堆栈来传递参数，应对堆栈小心操作，避免堆栈使用不当造成子程序不能正确返回的出错情况。

3) 子程序的嵌套层数不是无限的。堆栈是在数据存储区内开辟的空间，而由于SPCE061A 单片机的数据存储的空间为 2k WORD。

4.2.5.2 递归子程序

递归调用是指子程序调用自身子程序。

进行递归调用时需注意的地方是，一个递归程序必须有一个能够退出递归调用的测试语句。也就是说，递归调用是有条件的，满足了条件后，才可以进行递归调用；如果无条件地进行递归调用，那么会使堆栈空间溢出，导致严重的错误。下面的一段代码没有退出条件，运行的结果，必然是错误的。

程序4-8 一个错误的递归代码

```
.PUBLIC _Recursion;                          //无结束递归条件的的代码
_Recursion: .PROC
     R2=R2-1;
     CALL _Recursion;                        //无条件调用递归程序，会产生堆栈溢出
Loop:                                        //下面的程序永远不会被执行
    R1=R1+1;
```

```
CMP  R2,0;                          //r2 如果等于零，则 N 阶计算结束
     JNZ  Loop;
     [Sum]=R1;                      //将计算结果存入变量 Sum
     RETF
```

下面的例子是计算 n 的阶乘，从中可以看到使用递归结构，会使得程序变的非常简洁。

程序4-9 N！（N>=0）的程序

N！＝N*(N-1)*(N-2)*……*1

N 的阶乘递推公式如下：

$$\begin{cases} 0! = 1 \\ N!=N*(N-1)! \qquad (N>0) \end{cases}$$

编程分析：

求 N！本身是一个子程序，由于 N！是 N 和（N－1）！的乘积，所以求（N－1）！必须递归调用求 N！的子程序。过程如图 4.20所示。

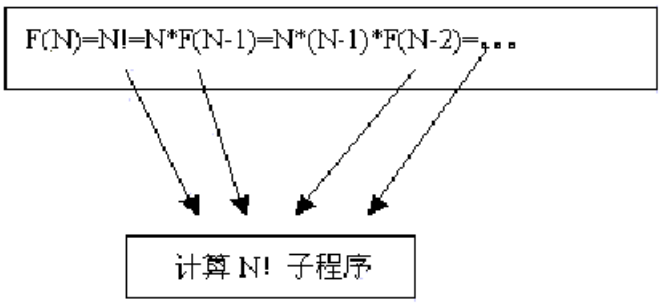


图4.20 n 的阶乘可以用一个递推公式来表示

程序说明：r1 存的是阶乘数 N，r2 存的是 N!的值。该程序可以计算的最大阶乘数 N 等于 8，当 N 大于 8 的时候，会溢出，这时 R2 被赋值为 0xffff。

程序代码如下，程序流程图如图 4.21所示。

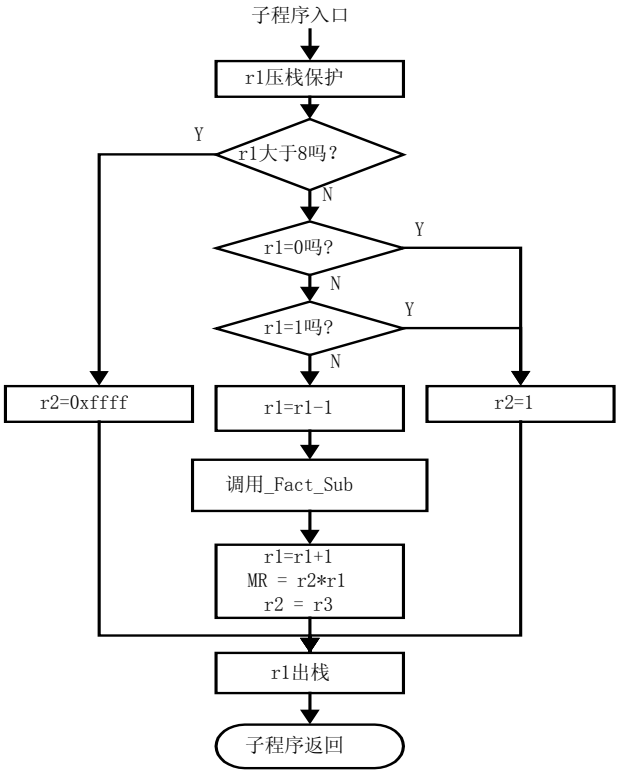


图4.21　n 阶乘程序流程图

```
//*****************************************************************************************/
// 描述：  N 的阶乘递推
// 日期：  2002/12/10
//*****************************************************************************************/
.CODE
//=========================================================================
// 函数：  main()
// 描述：主函数
//=========================================================================
.PUBLIC  _main;
_main:
      R1  =  0x005;
      CALL  L_Fact_Sub;
 MainLoop:
      JMP  MainLoop;
L_Fact_Sub:
      PUSH  R1 , R1  TO  [SP];
      CMP  R1,0x0008;                    //判断阶乘数是否大于 8
      JA  L_Overflow;                    //如果大于 8 则溢出，R2<-0xffff;
```

```
    CMP  R1 ,0x0000;
    JNE   L_eq1;
    R2  =  0x01;
    JMP   L_Fend                          //退出
  L_eq1:
    CMP   R1,  0x0001;
    JNE   L_eq2;
    R2  =  0x01;
    JMP   L_Fend;                         //退出
 L_eq2:
    R1-  =0x01;
    CALL  L_Fact_Sub;                     //调用递归子程序
    R1+  =0x01;
    MR  =  R2*R1;
    R2  =  R3;
    JMP  L_Fend;
L_Overflow:
     R2=0xFFFF
L_Fend:
    POP   R1,R1  FROM  [SP];
    RETF

//*************************************************************************************************/
// main.c 结束
//*************************************************************************************************/
```

递归的目的是简化程序设计，使程序易读。但递归增加了系统开销。时间上，执行调用与返回的额外工作要占有 CPU 时间。空间上，随着每递归一次，就要占用一定的栈空间。在实际应用的开发中应该根据实际情况折中考虑。

4.3 C 语言程序设计

是否具有对高级语言 HLL（High Level Language）的支持已成为衡量微控制器性能的标准之一。显然，在 HLL 平台上要比在汇编级上编程具有诸多优势：代码清晰易读、易维护，易形成模块化，便于重复使用从而增加代码的开发效率。

HLL 中又因 C 语言的可移植性最佳而成为首选。因此，支持 C 语言几乎是所有微控制器设计的一项基本要求。μ'nSP™指令结构的设计就着重考虑了对 C 语言的支持。GCC 是一种针对μ'nSP™操作平台的 ANSI-C 编译器，

4.3.1 μ'nSP™支持的 C 语言算逻操作符（#）

在μ'nSP™的指令系统算逻操作符与 ANSI-C 算符大同小异，见表 4.6。

表4.6 μ'nSP™ 指令的算逻操作符

算逻操作符	作用
+、-、*、/、%	加、减、乘、除、求余运算
&&、||	逻辑与、或
&、|、^、<<、>>	按位与、或、异或、左移、右移
>，>=，<，<=，= =，!=	大于、大于或等于、小于、小于或等于、等于、不等于
=	赋值运算符
?　:	条件运算符
,	逗号运算符
*、&	指针运算符
.	分量运算符
sizeof	求字节数运算符
[　]	下标运算符

4.3.2 C 语言支持的数据类型

μ'nSP™支持 ANSI-C 中使用的基本数据类型如表 4.7所示。

表4.7 μ'nSP™对 ANSI-C 中基本数据类型的支持

数据类型	数据长度(位数)	值　域
char	16	-32768~32767
short	16	-32768~32767
int	16	-32768~32767
long int	32	-2147483648~2147483647
unsigned char	32	0~65535
unsigned short	16	0~65535
unsigned int	16	0~65535
unsigned long int	32	0~4294967295
float	32	以 IEEE 格式表示的 32 位浮点数
double	64	以 IEEE 格式表示的 64 位浮点数

4.3.3 程序调用协议

由于 C 编译器产生的所有标号都以下划线（_）为前缀，而 C 程序在调用汇编程序时要求汇编程序名也以下划线(_)为前缀。

模块代码间的调用，是遵循μ'nSP™体系的调用协议(Calling Convention)。所谓调用协议，是指用于标准子程序之间一个模块与另一模块的通讯约定；即使两个模块是以不同的语言编写而成，亦是如此。

调用协议是指这样一套法则：它使不同的子程序代码之间形成一种握手通讯接口，并完成由一个子程序到另一个子程序的参数传递与控制，以及定义出子程序调用与子程序返回值的常规规则。

调用协议包括以下一些相关要素：

1)　调用子程序间的参数传递；

2)　子程序返回值；

3)　调用子程序过程中所用堆栈；

4)　用于暂存数据的中间寄存器。

μ'nSP™体系的调用协议的内容如下：

1.　参数传递

参数以相反的顺序（从右到左）被压入栈中。必要时所有的参数都被转换成其在函数原型中被声明过的数据类型。但如果函数的调用发生在其声明之前，则传递在调用函数里的参数是不会被进行任何数据类型转换的。

2.　堆栈维护及排列

函数调用者应切记在程序返回时将调用程序压入栈中的参数弹出。

各参数和局部变量在堆栈中的排列如图 4.22所示。

3.　返回值

16 位的返回值存放在寄存器 R1 中。32 位的返回值存入寄存器对 R1、R2 中；其中低字在 R1 中，高字在 R2 中。若要返回结构则需在 R1 中存放一个指向结构的指针。

4.　寄存器数据暂存方式

编译器会产生 prolog/epilog 过程动作来暂存或恢复 PC、SR 及 BP 寄存器。汇编器则通过‘CALL’指令可将 PC 和 SR 自动压入栈中，而通过‘RETF’或‘RETI’指令将其自动弹出栈来。

5.　指针

编译器所认可的指针是 16 位的。函数的指针实际上并非指向函数的入口地址，而是一个段地址向量_function_entry，在该向量里由 2 个连续的 word 的数据单元存放的值才是函数的入口地址。

下面以具体实例来说明μ'nSP™体系的调用协议。

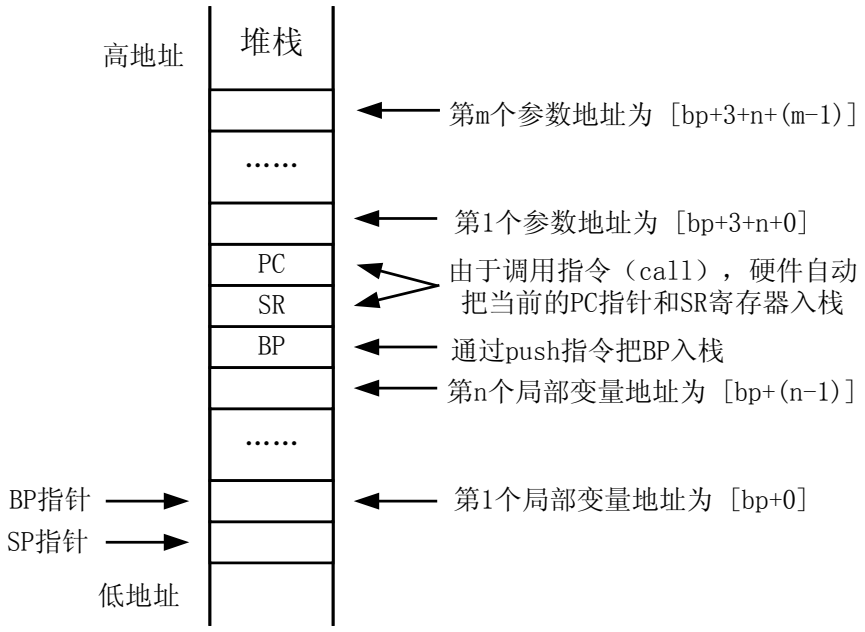


图4.22 程序调用参数传递的堆栈调用

4.3.3.2 在 C 程序中调用汇编函数

在 C 中要调用一个汇编编写的函数，需要首先在 C 语言中声明此函数的函数原型。尽管不作声明也能通过编译并能执行代码，但是会带来很多的潜在的 bug。

下面首先观察最简单的 C 调用汇编的堆栈过程：

程序4-10 无参数传递的 C 语言调用汇编函数

```
//*************************************************************************************/
// 描述：  无参数传递的 C 语言调用汇编函数
// 日期：  2002/12/10
//*************************************************************************************/
void  F_Sub_Asm(void);      //声明要调用的函数的函数原型，此函数没有任何参数的传递
//=====================================================================
// 函数：  main()
// 描述：主函数
//=====================================================================
int  main(void){
      while(1)
            F_Sub_Asm();
      return  0;
}
//*************************************************************************************/
//void  F_Sub_Asm(void); 来自于 asm.asm。延时程序，无入口出口参数。
```

```
// main.c 结束
//*****************************************************************************/
```

汇编函数如下：

```
//=============================================================================
//函数: F_Sub_Asm()
//语法：void F_Sub_Asm（void）
//描述：延时程序
//参数：无
//返回：无
//=============================================================================
.CODE
.PUBLIC _F_Sub_Asm
_F_Sub_Asm:
    NOP;
    RETF;
```

在 IDE 开发环境下运行可以看到调用过程堆栈变化十分简单，如图 4.23所示。

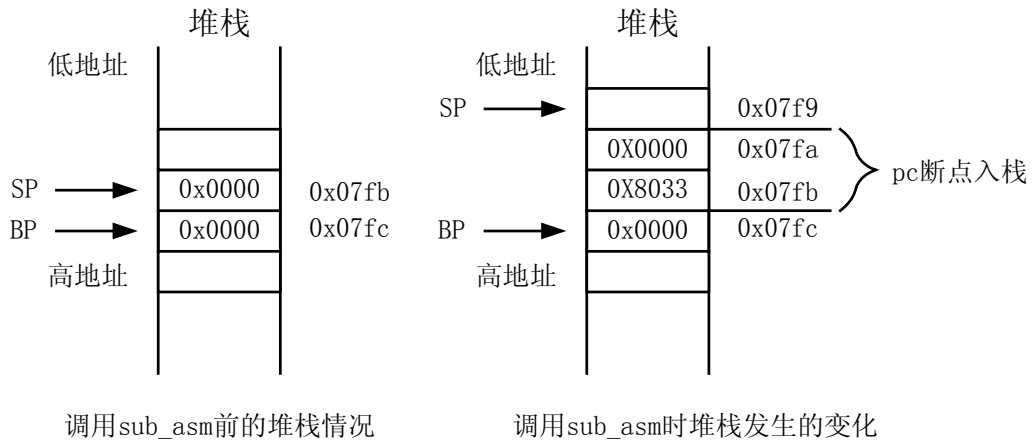


图4.23 最简单的程序调用的堆栈变化

现在在 C 语言中加入局部变量来观察调用过程：

程序4-11 C 语言中具有局部变量

```
//*****************************************************************************/
// 描述：  局部变量调用示意
// 日期：  2002/12/10
//*****************************************************************************/
void F_F_Sub_Asm(void);     //声明要调用的函数的函数原型，此函数没有任何参数的传递

//=============================================================================
```

```
// 函数：  main()
// 描述：主函数
//============================================================================
int main(){
    int i = 1, j = 2, k = 3;
    while(1){
        F_F_Sub_Asm();
        i = 0;
        i++;
        j = 0;
        j++;
        k = 0;
        k++;
    }
    return 0;
}

//*****************************************************************************************/
// void F_F_Sub_Asm(void)；来自于 asm.asm,延时子程序。无入口出口参数。
// void F_Show(int A,int B)；点亮 LED;A,LED 的位数（C_Dig）,B,LED 的显示值
// main.c 结束
//*****************************************************************************************/
```

汇编函数如下：

```
    .CODE
//=========================================================================
//函数: F_F_Sub_Asm（）
//语法：void F_F_Sub_Asm(void)
//描述：延时子程序
//参数：无
//返回：无
//=========================================================================
    .PUBLIC _F_F_Sub_Asm
    _F_F_Sub_Asm:
    NOP;
    RETF;
```

在图 4.24中表示出了，C 语言中的局部变量（i,j,k）在堆栈中存放的位置。

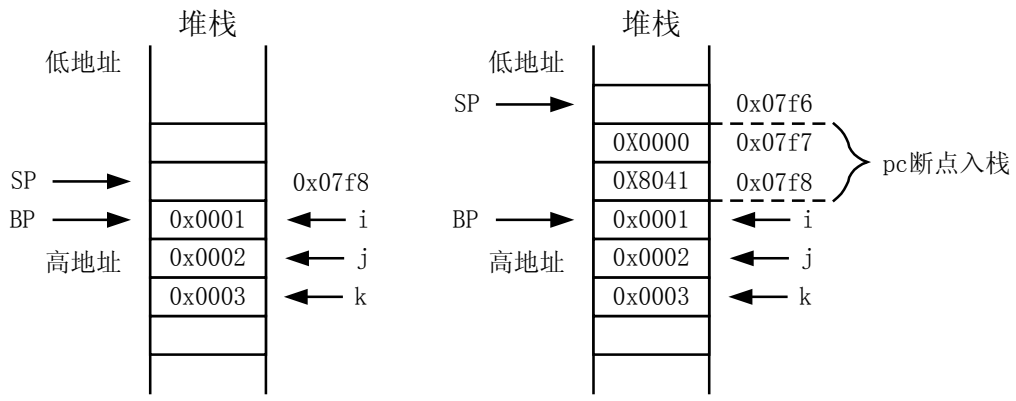


图4.24 具有局部变量的 C 程序调用时的堆栈变化

进一步，我们为函数 sub_asm 传递三个参数 i,j,k。同样来观察堆栈的变化，来理解调用协议。

程序4-12 C 向汇编函数传递参数

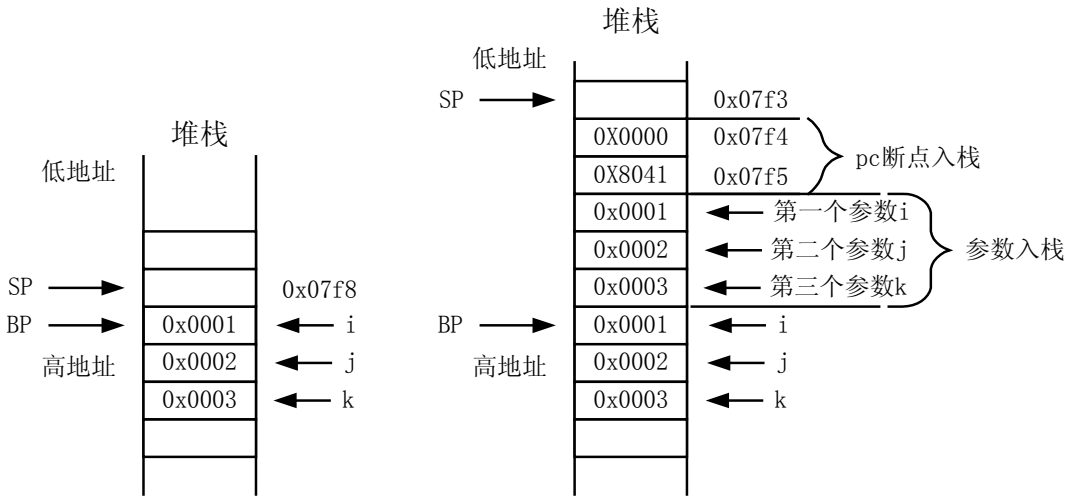
```
//*************************************************************************************/
// 描述： C 向汇编函数传递参数
// 日期： 2002/12/11
//*************************************************************************************/
void F_Sub_Asm(int a,int b,int c);      //声明要调用的函数的函数原型

//=============================================================================
// 函数： main()
// 描述：主函数
//=============================================================================
int main(){
    int i = 1, j = 2, k = 3;
    while(1){
        F_Sub_Asm(i,j,k);
        i = 0;
        i++;
        j = 0;
        j++;
        k = 0;
        k++;
    }
    return 0;
}
```

```
//*************************************************************************************/
//void F_Sub_Asm(int a,int b,int c); 来自于 asm.asm。测试传递参数，a,b,c 所传递的参数，无出口参数。
// main.c 结束
//*************************************************************************************/
```

汇编函数如下：

```
//=========================================================================
//函数: F_Key_Scan（）
//语法：void F_Key_Scan（int a,int b,int c）
//描述：测试传递参数
//参数：a,b,c 所传递的参数
//返回：无
//=========================================================================.
CODE
.PUBLIC _F_Sub_Asm
_F_Sub_Asm:
    NOP;
    RETF;;
```



调用sub_asm前的堆栈情况　　调用sub_asm时堆栈发生的变化

图4.25 C 程序调用时的利用堆栈的参数传递

通过以上三个例子，我们了解到 C 调用函数时是如何进行参数传递的。另外的一个问题就是关于函数的返回值，是怎样实现的。

函数的返回相对简单，在汇编子函数中，返回时寄存器 R1 里的内容，就是此函数 16 位数据宽度的返回值。当要返回一个 32 位数据宽度的返回值时，则利用的是 R1 和 R2 里的内容：R1 为低 16 位内容，R2 为高 16 位的内容。下面的代码说明了这一过程。

程序4-13 函数的返回值

```
//*************************************************************************************************/
// 描述：  测试函数的返回值
// 日期：  2002/12/11
//*************************************************************************************************/
int F_Sub_Asm1(void);         //声明要调用的函数的函数原型
long int F_Sub_Asm2(void);   //声明要调用的函数的函数原型

//=========================================================================
// 函数：  main()
// 描述：主函数
//=========================================================================
int main(){
      int i;
      long int j;
      while(1){
            i = F_Sub_Asm1();
            j = F_Sub_Asm2();
      }
      return 0;
}

//*************************************************************************************************/
//void F_Sub_Asm1(void); 来自于 asm.asm。此函数没有任何参数的传递，但返回整形值。
//void F_Sub_Asm2(void); 来自于 asm.asm。此函数没有任何参数的传递，但返回一个长整型值
// main.c 结束
//*************************************************************************************************/
被调用的汇编代码如下：
.code

//=========================================================================
//函数: F_Sub_Asm1()
//语法：void F_Sub_Asm1（void）
//描述：整形返回值测试
//参数：无
//返回：整形值。
//=========================================================================
.PUBLIC _F_Sub_Asm1
_F_Sub_Asm1:
      R1 = 0xaabb;
      R2 = 0x5555;
      RETF;
```

```
//=============================================================================
//函数: F_Sub_Asm2()
//语法：void F_Sub_Asm2（void）
//描述：长整型值返回值测试
//参数：无
//返回：一个长整型值
//=============================================================================
.PUBLIC _F_Sub_Asm2
_F_Sub_Asm2:
    R1 = 0xaabb;
    R2 = 0xffcc;
    RETF;
```

程序调用的结果，i=0xaabb；j=0xffccaabb。

4.3.3.3 在汇编程序中调用 C 函数

在汇编函数中要调用 C 语言的子函数，那么应该根据 C 的函数原型所要求的参数类型，分别把参数压入堆栈后，再调用 C 函数。调用结束后还需再进行弹栈，以恢复调用 C 函数前的堆栈指针。此过程很容易产生 bug，所以需要程序员细心处理。下面的例子给出了汇编调用 C 函数的过程。

程序4-14 汇编调用 C 的函数

```
//*****************************************************************************************/
// 描述：  汇编调用 C 的函数
// 日期：  2002/12/10
//*****************************************************************************************/
.EXTERNAL _F_Sub_C
.CODE
.PUBLIC _main;

//=============================================================================
// 函数：  main()
// 描述：主函数
//=============================================================================
_main:
    R1 = 1;
    PUSH R1 TO [SP];          //第 3 个参数入栈
    R1 = 2;
    PUSH R1 TO [SP];          //第 2 个参数入栈
    R1 = 3;
    PUSH R1 TO [SP];          //第 1 个参数入栈
    CALL _F_Sub_C;
    POP R1,R3 FROM [SP];      //弹出参数回复 SP 指针
```

```
    GOTO _main;
    RETF;

//*************************************************************************************************/
//void F_Sub_C(int i,int j,int k); 来自于 asm.c。延时程序，入口参数 i,j,k;返回 i
// main.asm 结束
//*************************************************************************************************/
```

C 语言子函数如下：

```
//=========================================================================
//函数: F_Sub_C()
//语法：void F_Sub_C(int i,int j,int k)
//描述：延时程序
//参数：i,j,k
//返回：i
//=========================================================================
int F_Sub_C(int i,int j,int k)
{
    i++;
    j++;
    k++;
    return i;
}
```

4.3.3.4 编程举例

下面举一个 C 语言和汇编混合编程的例子。汇编中利用 2Hz 中断进行计数，C 程序判断时间，在 IOA 口上以 2 秒的速率闪烁。

程序4-15 C 语言与汇编混合编程举例

```
//*************************************************************************************************/
// 描述： C 语言与汇编混合编程举例
// 日期： 2002/12/11
//*************************************************************************************************/
unsigned int TimeCount = 0;
//=========================================================================/
// 函数： main()
// 描述：主函数
//=========================================================================
int main()
{
    TimeCount = 0;
    F_InitIOA(0xFFFF,0xFFFF,0x0000);          //初始化 IOA 口
    SystemInit();                             //系统初始化
 while(1)
    {
```

```
        if(TimeCount<=4)
              LightOff();                          //IOA 口 LED 熄灭
        else  if(TimeCount<=7)
          LightOn();                               //IOA 口 LED 亮
     else
          TimeCount=0;
           }
 }

//*************************************************************************************************/
// void SP_InitIOA(int A,int B，int C); 来自于 System.asm,IOA 初始化。A,方向向量单元，
//                                      B 数据单元,C 属性向量单元
// void SystemInit(); 来自于 System.asm,IOA 初始化。无入口出口参数。
// void LightOff(); 来自于 System.asm。无入口出口参数。
// void LightOff(); 来自于 System.asm。无入口出口参数。
// main.c 结束
//*************************************************************************************************/
//System.asm 汇编程序
.INCLUDE hardware.inc
.CODE
//=========================================================================
//函数: SystemInit()
//语法：void SystemInit（void）
//描述：系统初始化
//参数：无
//返回：无
//=========================================================================
.PUBLIC _SystemInit;                               //系统初始化
_SystemInit: .PROC
      R1=0x0004                                    //开 2Hz 中断
      [P_INT_Ctrl]=R1
      IRQ ON
      RETF;
     .ENDP;

//=========================================================================
//函数: F_InitIOA()
//语法：void F_InitIOA（void）
//描述：IO 口初始化
//参数：无
//返回：无
//=========================================================================
.PUBLIC _F_InitIOA;                                //初始化 IOA 口
_F_InitIOA: .PROC
```

```
    PUSH BP TO [SP];
    BP=SP+1;
    R1=[BP+3];
    [P_IOA_Dir]=R1;
    R1=[BP+4];
    [P_IOA_Attrib]=R1;
    R1=[BP+5];
    [P_IOA_Data]=R1;
    POP BP FROM [SP];
    RETF;
   .ENDP;

//=============================================================================
//函数: LightOn()
//语法：void LightOn（void）
//描述：点亮 led
//参数：无
//返回：无
//=============================================================================
.PUBLIC _LightOn;                          //IOA 口 LED 点亮
_LightOn: .PROC
            R1= 0xFFFF;
            [P_IOA_Data] = R1;
            RETF;
            .ENDP

//=============================================================================
//函数: LightOff()
//语法：void LightOff（void）
//描述：熄灭 led
//参数：无
//返回：无
//=============================================================================
.PUBLIC _LightOff;                         //IOA 口 LED 熄灭
_LightOff: .PROC
            R1= 0x0000;
            [P_IOA_Data] = R1;
            RETF;
            .ENDP
;
//中断程序 ISR.ASM
.PUBLIC _IRQ5
.INCLUDE hardware.inc
.EXTERNAL   _TimeCount;                          //计时
```

```
.TEXT
//=============================================================================
//函数: IRQ5()
//语法：void IRQ5（void）
//描述：IRQ5 中断服务程序
//参数：无
//返回：无
//=============================================================================
_IRQ5:
      PUSH R1,R5 TO [SP]
      R1 = 0x0008;
      TEST R1,[P_INT_Ctrl];
      JNZ L_IRQ5_4Hz;
L_IRQ5_2Hz:                                    //2Hz 中断
     R1=0x0004
     [P_INT_Clear] = R1;                       //清中断
     R1=[_TimeCount]                           //计数器+1
     R1+=1
     [_TimeCount]=R1
      POP R1,R5 FROM [SP];
      RETI;
L_IRQ5_4Hz:                                    //4Hz 中断
     [P_INT_Clear] = R1;
      POP R1,R5 FROM [SP];
      RETI;
```

源程序共包含 C 主程序 main.c、汇编程序 System.asm、中断程序 ISR.ASM 三个程序文件，完成硬件接口的子程序：系统初始化_SystemInit、初始化 IOA 口_F_InitIOA、IOA 口 LED 点亮_LightOn、IOA 口 LED 熄灭_LightOff、清看门狗_Clear_WatchDog 都定义为过程，写在 CODE 段，由 C 主程序调用；中断服务程序写在 TEXT 段。

4.3.4 C 语言的嵌入式汇编

为了使 C 语言程序具有更高的效率和更多的功能，需在 C 语言程序里嵌入用汇编语言编写的子程序。一方面是为提高子程序的执行速度和效率；另一方面，可解决某些用 C 语言程序无法实现的机器语言操作。而 C 语言代码与汇编语言代码的接口是任何 C 编译器毋庸置疑要解决的问题。

通常，有两种方法可将汇编语言代码与 C 语言代码联合在一起。一种是把独立的汇编语言程序用 C 函数连接起来，通过 API (Application Program Interface) 的方式调用；另一种就是我们下面要讲的在线汇编方法，即将直接插入式汇编指令嵌入到 C 函数中。

编译器 GCC 认可的基本数据类型及其值域列示在表 4.8中。

表4.8 GCC 的基本数据类型

数据类型	数据长度(位数)	值　域
Int	16	-32768~32767
long int	32	-2147483648~2147483647
unsigned int	16	0~65535
Unsigned long	32	0~4294967295
float	32	以 IEEE 格式表示的 32 位浮点数
double	64	以 IEEE 格式表示的 64 位浮点数

采用 GCC 规定的在线汇编指令格式进行指令的输入，是 GCC 实现将μ'nSP™汇编指令嵌入 C 函数中的方法。GCC 在线汇编指令格式规定如下：

asm (“汇编指令模板”：输出参数：输入参数：clobbers 参数);

若无 clobber 参数，则在线汇编指令格式可简化为：

asm (“汇编指令模板”：输出参数：输入参数);

下面，将对在线汇编指令格式中的各种成分之内容进行介绍。

1)　汇编指令模板

模板是在线汇编指令中的主要成分，GCC 据此可在当前位置产生汇编指令输出。例如，下面一条在线汇编指令：

asm ("%0 += %1" : "+r" (foo) : "r" (bar));

此处，"%0 += %1"就是模板。其中，操作数"%0"、"%1"作为一种形式参数，分别会由第一个冒号后面实际的输出、输入参数取代。带百分号的数字表示的是第一个冒号后参数的序号。

如下例：

asm ("%0 = %1 + %2" : "=r" (foo) : "r" (bar), "i" (10));

"%0"会由参数 foo 取代，"%1"会由参数 bar 取代，而"%2"则会由数值 10 取代。

在汇编输出中，一个汇编指令模板里可以挂接多条汇编指令。其方法是用换行符'\n'来结束每一条指令，并用 Tab 键符'\t'将同一模板产生在汇编输出中的各条指令在换行显示时缩进到同一列，以使汇编指令显示清晰。如下例：

asm ("%0 += %1\n\t%0 += %1" : "+r" (foo) : "r" (bar));

2)　操作数

在线汇编指令格式中，第一冒号后的参数为输出操作数，第二冒号后的参数为输入操作数，第三冒号后跟着的则是 clobber 操作数。在各类操作数中，引号里的字符代表的是其存储类型约束符；括弧里面的字符串表示的是实际操作数。

如果输出参数有若干个，可用逗号“，”将每个参数隔开。同样，该法则适用于输入参数或 clobber 参数。

3)　操作数约束符

约束符的作用在于指示 GCC，使用在汇编指令模板中的操作数的存储类型。表 4.9 列出了一些约束符和它们分别代表的操作数不同的存储类型，也列出了用在操作数约束符之前的两个约束符前缀。

表4.9 操作数存储类型约束符及约束符前缀

约束符	操作数存储类型	约束符前缀及含义解释	
r	寄存器中的数值	=	+
m	存储器内的数值	为操作数赋值	操作数在被赋值前先要参加运算
i	立即数		
p	全局变量操作数		

4)　GCC 在线汇编指令举例

例 1：asm ("%0 = %1 + %2" : "=m" (foo) : "r" (bar), "i" (10))；

操作数 foo 和 bar 都是局部变量。bar 的值会分配给寄存器（此例中寄存器为 R1），而 foo 的值会置入存储器中，其地址在此由 BP 寄存器指出。GCC 对此会产生如下代码：

```
                    // GCC 在线汇编起始
[BP] = R1 + 10
                    // GCC 在线汇编结束
```

注意，本在线汇编指令产生的汇编代码不能被正确汇编。正确的在线汇编指令应当是：

asm ("%0 = %1 + %2" : "=r" (foo) : "r" (bar), "i" (10))；

它产生如下的汇编代码：

```
                    // GCC 在线汇编起始
R1 = R4 + 10
                    // GCC 在线汇编结束
```

例 2：

```
int a;
int b;
#define SEG(A,B) asm("%0 = seg %1" : "=r" (A) : "p" (&B));
int main(void)
{
  int foo;
  int bar;
  SEG(foo, a);
  SEG(bar, b);
  return foo;
}
```

例 3：asm ("%0 += %1" : "+r" (foo) : "r" (bar))；

操作数 foo 在被赋值前先要参加运算，故其约束符为"+r"，而非"=r"。

4.3.5 利用嵌入式汇编实现对端口寄存器的操作

在 C 的嵌入式汇编中，当使用端口寄存器名称时，需要在 C 文件中加入汇编的包含文件，如下所示：

asm(“.include hardware.inc”);

那么，我们就可以使用端口寄存器的名称，而不必去使用端口的实际的地址。

1)　写端口寄存器

现举例说明：要设定 PortA 端口为输出端口，需要对 P_IOA_Dir 赋值 0xffff。那么在 C 中的嵌入式汇编的实现方式如下：

在 C 中有一个 int 型的变量 i，传送到 P_IOA_Dir 中，则嵌入汇编的实现方式如下：

```
….
asm(“.include  hareware.inc”);
….
int  main(void){
      int  i;
      ….
      asm("[P_IOA_Dir]  =  %0"
            :
                                          //没有输出参数
            :"r"(i)
                                          //只有输入参数,通过寄存器传递变量 i 的内容
            );
      …
}
```

如果需要对端口寄存器直接赋值一个立即数（比如对 P_IOA_Dir 赋值 0x1234），那么内嵌式汇编为：

```
….
asm(“.include  hareware.inc”);
….
int  main(void){
      ….
      asm("[P_IOA_Dir]  =  %0"
            :
                                  //没有输出参数
            :"r"(0x1234)
                                  //只有输入参数，通过寄存器传递立即数 0x1234
            );
      …
}
```

2)　读端口寄存器

对端口寄存器进行读操作的方法，与写类似，下面仍然以 P_IOA_Dir 为例，进行

说明。

如果要实现把端口的寄存器 P_IOA_Dir 的值读出并保存在 C 中的一个 int 变量 j 里，那么可以通过下面的方法来实现。

```
....
asm(".include hareware.inc");
....
int main(void){
    int j;
    ....
    asm("%0 = [P_IOA_Dir]"
        :"=r"(j)
                                //只有输出参数，而无输入参数
        );
    …
}
```

3)　利用 GCC 编程举例

下面是一段 GCC 的代码，实现对 A 口的初始化：设定 A 口为同向输出高电平。

```
asm("[P_IOA_Attrib] = %0\n\t"
    "[P_IOA_Data] = %0\n\t"
    "[P_IOA_Dir] = %0\n\t"
    :
    :
    "r"(0xffff)
    );
```

上面代码通过 GCC 编译后的代码为：

```
R1=(-1)                 // QImode move
                        // GCC inline ASM start
[P_IOA_Attrib] = R1
[P_IOA_Data] = R1
[P_IOA_Dir] = R1

                        // GCC inline ASM end
```

下面是一段 GCC 的代码，实现对 B 口的初始化：设定 B 口为具有上拉电阻的输入。

```
asm("[P_IOB_Attrib] = %0\n\t"
    "[P_IOB_Data] = %1\n\t"
    "[P_IOB_Dir] = %0\n\t"
    :
    :
    "r"(0),
```

```
    "r"(0xffff)
    );
```

上面一段代码通过 GCC 编译后的汇编代码为：

```
R2=(-1)
            // QImode move
R1=0
            // QImode move
            // GCC inline ASM start
[P_IOB_Attrib] = R2
[P_IOB_Data] = R1
[P_IOB_Dir] = R2

            // GCC inline ASM end
```

通过上述两断代码，使得 SPCE061A 的 B 口为输入，A 口为输出，如果我们要实现把 B 口得到的数据从 A 口输出，这样的 GCC 编程需要在 C 中先建立个 int 型的中间变量，通过这个中间变量，写出两个 GCC 的代码来实现。

```
…
int temp;
…
asm("%0 = [P_IOB_Data]"
    :"=r"(temp)
    );
asm("[P_IOA_Buffer] = %0"
    :
    :"r"(temp)
    );
```

通过 GCC 后的代码如下所示。这里将看不到 temp 的影子，GCC 会进行优化处理。

```
R1 = [P_IOB_Data]
[P_IOA_Buffer] = R1
```

通过上述方法的介绍，我们就可以在 C 语言中直接对 SPCE061A 的硬件进行操作。在对硬件读写语句较少的情况下，如果采用 C 调用汇编函数的方法显得有些臃肿，而使用嵌入式汇编会使得代码高效简洁！

4.4 应用程序设计

4.4.1 查表程序

查表，就是在以 $y=f(x_1,x_2,\cdots,x_n)$ 为关系建立的表格中，根据变量 $x_1,x_2,\cdots,x_n$，查找 y 值。由于 SPCE061A 具有寄存器间接寻址和变址寻址方式，所以查表的基本方法有如下两种：

方法一：寄存器间接寻址方式

```
Rn = 表首地址
Rn += 偏移地址
Rd = [Rn]          //取得表中的数据
```

方法二：变址寻址方式

```
BP = 表首地址
Rn = [BP + 偏移地址]//取得表中的数据
```

方法二要比前一种方法来得更简洁些，但是方法二中的偏移地址只能是 6 位的立即数。就是说，方法二查表范围限制在 64word 以内。这一点在使用时，需要提醒读者注意。

由于 SPCE061A 有 32kwordFlash 的程序存储区，所以仅有零页的程序存储区（每页的存储区为 64kword），不涉及到程序区寻址的段选的内容。方法一和方法二，仅仅适用于表放在零页的程序存储区的情形，

4.4.1.1 一维数组查表程序

程序4-16 查 8 位十六进制数平方表

```
//=============================================================================
//函数: F_Square()
//语法：void F_Square（void）
//描述：查 0~0xFF 的平方表。
//参数：R1 = 待查的数，低 8 位有效。（仅仅查 0~0xFF 的平方值）
//返回：R1 = 查表后的平方值结果
//=============================================================================
F_Square: .PROC
    R1 &= 0x00FF;                  //屏蔽高 8 位，仅使低 8 位有效
    R1 += Square_Table;            //计算元素地址
    R1 = [R1];                     //取得数据
    RETF;
    .ENDP
.CODE
Square_Table:
```

```
                                                    //0~255 平方表
.DW   0,      1,      4,      9,      16
                                                    //0～4 平方表
.DW  25,     36,     49,     64,     81
                                                    //5~10 平方表 . . .
.DW 62500,  63001,  63504,  64009,  64516           //250~255 平方表
```

4.4.1.2 二维数组查表程序

设 *mxn* 维单字矩阵如下：

$$\begin{matrix} x_{0,0} & x_{0,1} & \cdots & x_{0,j} & \cdots & x_{0,n-1} \\ x_{1,0} & x_{1,1} & \cdots & x_{1,j} & \cdots & x_{1,n-1} \\ \vdots & & \vdots & & \vdots & \\ x_{i,0} & x_{i,1} & \cdots & x_{i,j} & \vdots & x_{i,n-1} \\ \vdots & & \vdots & & \vdots & \\ x_{m-1,0} & x_{m-1,1} & \cdots & x_{m-1,j} & \cdots & x_{m-1,n-1} \end{matrix}$$

每个元素为单字，矩阵在存储器中存放，从低地址到高地址为 $x_{0,0}, x_{0,1}, \cdots, x_{0,n-1}$；$x_{1,0}, x_{1,1}, \cdots, x_{1,n-1}$；……；$x_{m-1,0}, x_{m-1,1}, \cdots, x_{m-1,n-1}$。共计占用 *mxn* 个字。对于其中某个元素 $x_{i,j}$ 的地址有如下算法：

元素 $x_{i,j}$ 的地址 ＝ 矩阵首址 ＋ $(i \times n + j)$

程序流程图如图 4.26所示：

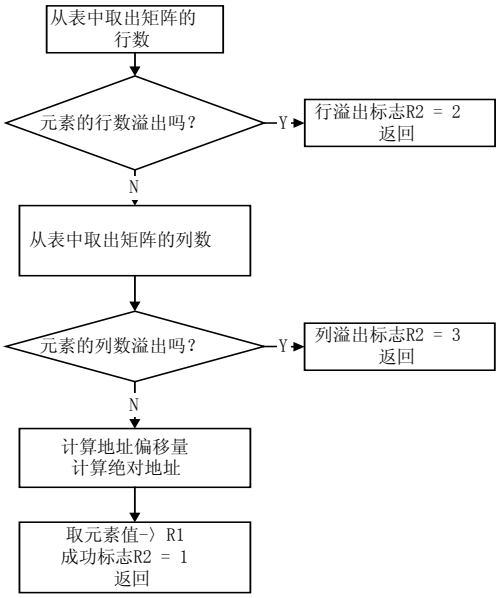


图4.26 矩阵表元素读取流程图

程序4-17 二维数组查表

```
//=============================================================================
//函数: F_Get_Array()
//语法：void F_Get_Array（void）
//描述：实现查表
//参数：R1 = 元素下标 I（矩阵行号）,R2 = 元素下标 j（矩阵列号）
//返回：R1 = 查找到的元素 ,R2 = 查找成功标志（1 为查找成功，2 为行号溢出，3 为列号溢出）
//=============================================================================
F_Get_Array:  .proc
      R3 = [Array_Table];          //取矩阵行数
      CMP R3,R1;                   //比较行是否出界
      JBE ?row;                    //如果出界，则跳到行溢出处理
      R4 = [Array_Table+1];        //取矩阵列数
      CMP R4,R2                    //比较列是否出界
      JBE ?colum;                  //如果出界，则跳到列溢出处理
      MR = R1 * R4;
      R2 = R2 + R3;                //计算元素地址的偏移量
      R2 += Array_Table+2;         //计算元素地址的绝对地址
      R1 = [R2];                   //取元素值
      R2 = 1;                      //成功标示
      RETF;
?row:
      R2 = 2;                      //行溢出标志
      RETF;
?colum:
      R2 = 3;                      //列溢出标志
      RETF;
      .ENDP
      .CODE
Array_Table:
      .DW 3,5                      //定义矩阵行数为 3，列数为 5
      .DW 1,  2,  3,  4,  5        //矩阵 0 行
      .DW 6,  7,  8,  9,  10       //矩阵 1 行
      .DW 11, 12, 13, 14, 15       //矩阵 2 行
      .ENDP
```

4.4.1.3 查表散转程序

在程序 4-18介绍了一种散转程序的方法，这里我们通过查表的方法来实现散转。这两种方法尽管从形式上是不同的，但他们的实现机理是相同的：通过改变 PC 寄存器的值来实现的。

这里，根据某个寄存器的内容为 0，1，2，……，分别转向处理程序 0，1，2，……。把转向的地址组成一个表，通过查表的方式决定程序的跳转。其流程图见图 4.27。

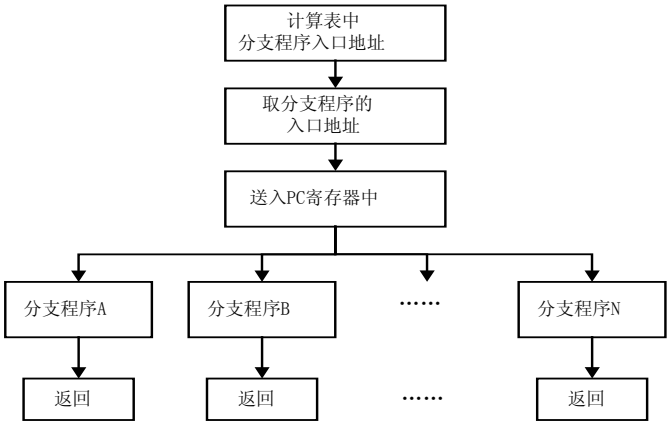


图4.27 查表散转程序结构

程序4-18 查表散转程序

```
//=============================================================================
//函数: F_Swich()
//语法：void F_Swich（void）
//描述：实现查表
//参数：R1 = 要转向的子程序的序号
//返回：无
//=============================================================================
F_Swich: .PROC
      R1 += Switch_Table;
      PC = [R1];
L_SubA:
      NOP;
      RETF;
L_SubB:
      NOP;
      RETF;
    .ENDP
Switch_Table:                    //表定义为：
      .DW L_SubA
      .DW L_SubB
```

4.4.2 数制转换程序

4.4.2.1 二进制码到 BCD 码的转换

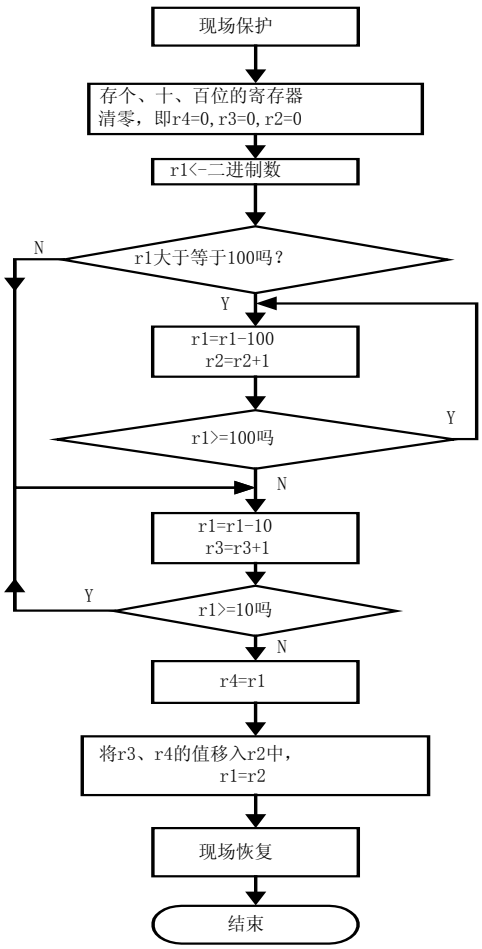


图4.28 二进制到 BCD 码转换流程

程序4-19 二进制码到 BCD 码的转换的子程序：

```
//=============================================================================
//函数: F_Binary_BCD()
//语法：void F_Binary_BCD（void）
//描述：R1 作为入口参数时，低字节存放的是待转换的二进制码。R1 作为出口参数时。其中 bo~b3 存的是个位，
//        b4～b7 存的是十位，b8～b11 存的是百位。参考代码如下，程序流程如图 4.28。
//参数：R1(存放一个字节的二进制数)
//返回：R1(存放 BCD 码)
//=============================================================================
_F_Binary_BCD: .proc
     PUSH R2,R4 TO [SP];
     R2=0;                          //清零，准备存百位数
     R3=0;                          //清零，准备存十位数
     R4=0;                          //清零，准备存个位数
     R1&=0x00ff;                    //屏蔽高字节
```

```
    CMP  R1,100;            //与 100 比较
    JB  L_ShiWei              //小于 100，则跳转
L_BaiWei:
    R1-=100;
    R2+=1;                    //百位数加 1
    CMP  R1,100;
    JAE  L_BaiWei;            //大于等于 100 继续求百位数
L_ShiWei:
    CMP  R1,10;               //与 10 比较
    JB  L_GeWei;              //小于 10，则跳
    R1-=10;
    R3+=1;                    //十位数加 1
    CMP  R1,10;
    JAE  L_ShiWei;            //大于 10，则跳
L_GeWei:
    R4=R1;
    R1=0x0000;
    R1=R1  ROL  4;            //移位寄存器清零
    R3=R3  ROL  4;            //将十位数移出
    R2=R2  ROL  4;            //将十位数移入 R2 寄存器(b4~b7)
    R4=R4  ROL  4;            //将个位数移出
    R2=R2  ROL  4;            //将个位数移入 R2 寄存器（b0~b3）
    R1=R2;                    //其中 b8~b11 存的是百位
    POP  R2,R4  FROM  [SP];
    RETF
  .ENDP
```

4.4.2.2 BCD 码到二进制码的转换

程序4-20 BCD 码到二进制码的转换的子程序

说明：R1 作为入口参数时，存的是 BCD 码，其中 b0～b3 存个位数的 BCD 码 a，b4～b7 存十位数的 BCD 码，b8~b11 存百位数的 BCD 码，b12~b15 存千位数的 BCD 码。R1 作为出口参数时，存的是转换后的二进制数。

算法：设 BCD 码为 a,b,c,d,则相应的二进制数为 1000a+100b+10c+d＝((a*10+b)*10+c)*10+d，将各位 BCD 码分离出之后，即可根据此式转换为二进制数。

子程序代码如下，程序流程图见图 4.29。

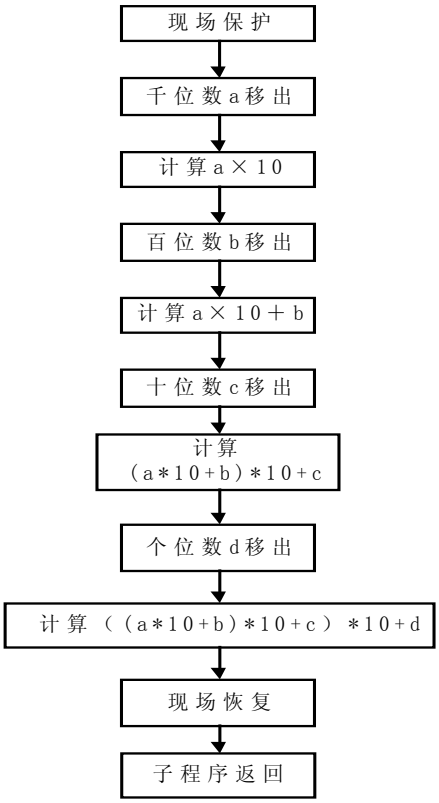


图4.29 BCD 码到二进制数转换流程

```
//=========================================================================
//语法：void F_BCD_Binary（int A）
//参数：R1(存放一个字节的二进制数)
//返回：R1(存放 BCD 码)
//=========================================================================
_F_BCD_Binary: .proc
    PUSH R2,R5 TO [SP];
     R5=R1;
     R5=R5 LSL 4                  //将千位数 a 移出
     R1=R1 ROL 4;
     R1&=0x000f;                  //将千位数 a 存入 R1；
     R2=10;
     MR=R1*R2                     //a*10
     R5=R5 LSL 4;                 //将百位数 b 移出
     R1=R1 ROL 4;
     R1&=0x000f;                  //将百位数 b 存入 R1
     R1+=R3;                      //R1=(a*10+b)
     MR=R1*R2
     R5=R5 lsl 4;                 //将十位数 c 移出
```

```
    R1=R1 Rol 4;
    R1&=0x000f;                    //将十位数 c 存入 R1
    R1+=R3;                        //((a*10+b)*10+c)
    MR=R1*R2                       //((a*10+b)*10+c)*10
    R5=R5 lsl 4;                   //将十位数 d 移出
    R1=R1 Rol 4;
    R1&=0x000f;
    R1+=R3;                        //((a*10+b)*10+c)*10+d
    POP R2,R5 FROM [SP];
    RETF
.ENDP
```

4.4.2.3 二进制码到 ASCII 码的转换

程序4-21 二进制码到 ASCII 码的转换

说明： 对于小于等于 9 的 4 位二进制数加 0x30 得到相应 ASCII 码,对于大于 9 的 4 位二进制数加 0x37 得相应 ASCII 代码。

子程序代码如下：

```
//=============================================================================
//函数: F_BinaRy_BCD()
//语法：void F_BinaRy_BCD（int A）
//描述：将一个 4 位的二进制数转化为 ASCII 码程序
//参数：R1(存放一个 4 位的二进制数)
//返回：R1(存放转换后的 ASCII 码)
//=============================================================================
_F_BinaRy_BCD: .PROC
PUSH R2,R5 TO [SP];
    R1&=0x000f ;                   //屏蔽高十二位
    CMP R1,0x0009;                 //与 9 比较
    JA L_To_F;
    R1+=0x0030;                    //小于、等于 9，则加 0x0030
    JMP L_Over;
L_To_F:
    R1+=0x0037;                    //大于 9 则加 0x0037
L_Over:
    POP R2,R5 FROM [SP];
    RETF
.ENDP
```

4.4.2.4 十六进制数到 ASCII 的转换

程序4-22 十六进制数到 ASCII 的转换

子程序参考代码如下：

```
//函数: F_Hex_ASCII()
```

```
//语法：void F_Hex_ASCII（int A）
//描述：将 1 位十六进制数转换为 ASCII 的程序
//参数：R1(存放 1 位十六进制数)
//返回：R1(存放转换后的 ASCII 码)
//=============================================================================
.PUBLIC _F_Hex_ASCII;
 _F_Hex_ASCII: PROC
     PUSH R2,R5 TO [SP];
      BP=Table;
      BP=BP+R1;
     BP=[BP];
     R1=BP;
     POP R2,R5 FROM [SP];
     RETF
.ENDP
.DATA
Table: .DW 0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39;
       .DW 0x41,0x42,0x43,0x44,0x45,0x46;.endp
.DATA
Table: .DW 0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39;
       .DW 0x41,0x42,0x43,0x44,0x45,0x46;
```

4.4.2.5 ASCII 码转换为二进制数

程序4-23 ASCII 码转换为二进制数的子程序

```
//=============================================================================
//语法：void F_ASCII_Binary（int A）
//描述：延时程序
//参数：R1(存放 1 位十六进制数的 ASCII 码)
//返回：R1(低四位存放转换后的 4 位二进制数)
//=============================================================================
.PUBLIC _F_ASCII_Binary;
_F_ASCII_Binary: .proc
     PUSH R2,R5 TO [SP];
     R1-=0x0030;
     CMP R1,9;
     JA L_To_F;
     JMP L_Exit;
L_To_F:
     R1-=0x0007;
L_Exit:
     POP R2,R5 FROM [SP];
     RETF;
.ENDP
```