前　言

本教材是结合《μ'nSP™系列 SPCE061A 单片机基础与应用技术》一书而设计实验，与课堂教学内容结合紧密。比如实验 1－4，内容浅显易懂，并附有例子程序，让读者们能尽快了解，并掌握凌阳单片机的语法结构和基本编程方法。通过由浅入深的一系列训练，可以使读者全面了解并掌握凌阳 μ'nSP™系列单片机的各个功能模块的实现方法，突出地了解 μ'nSP™系列单片机在广泛应用实例中的优势。

本书共分三章，内容如下：

第一章　SPCE061A 基础实验，共有 23 个实验。

第二章　语音实验，共有 6 个实验。

第三章　综合应用实验，共有 8 个实验。

本书语音实验和综合应用实验中的每一个例子都有一定的代表性，也有一定的深度，建议读者从基础实验学起。本书中的所有实验均以 μ'nSP™十六位单片机实验箱为实验设备。

本书所附的光盘中的所有源程序代码，都是经过调试，并可直接运行。读者在学习的时候，可以直接使用它来调试、验证程序，以达到节省时间、快速入门的目的；另外，本书中一些例子的程序代码具有一定的实际应用价值，读者只要稍作修改，就可直接引用。

由于编者水平有限，编写时间仓促，书中错漏在所难免，敬请读者及专家赐正。

北阳电子 应用推广部

二零零二年八月

第一章 SPCE061A 基础应用实验

实验一 熟悉 μ'nSP™ IDE 环境下的汇编程序的编写

【实验目的】

1)熟悉 μ'nSP™ IDE 环境及在该环境下用汇编语言编写的应用程序。

2)熟悉简单的 μ'nSP™汇编语言指令。

【实验设备】

装有 WINDOWS 系统和 μ'nSP™ IDE 仿真环境的 PC 机一台。

【实验步骤】

1)将 μ'nSP™ IDE 打开后,建立一个新工程。

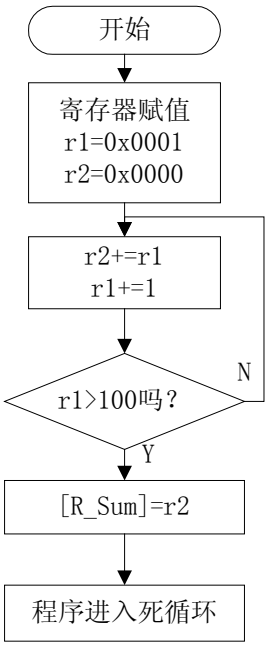
2)在该项目的源文件夹(SOURCE FILES)下建立一个新的汇编语言文件。

3)编写汇编代码。

4)编译程序，软件调试，观察并跟踪其结果,查看各个寄存器状态,等等。

【程序流程图】

主程序流程图：



【程序范例】

```
//=====================================================================//
// Program:  计算 1 to 100 累加值
// Output: [sum] = 5050(十进制)  或  13BA(十六进制)
//=====================================================================//

.RAM                                    // 定义预定义 RAM 段
.var  R_Sum;                            // 定义变量
```

```
.CODE                                   //定义代码段
.public _main;                          // 对 main 程序段声明
_main:
        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
        cmp r1,100;                     // 加到 100 否
        jna   L_SumLoop;                // 如果 r1 <= 100 跳到 L_SumLoop
        [R_Sum] = r2;                   // 在 R_Sum 中保存最终结果
L_ProgramEndLoop:                       // 程序死循环
        jmp  L_ProgramEndLoop;
```

【程序练习】

在 μ'nSP™ IDE 下用汇编语言使用冒泡法编写一个排序程序。

```
.iram
array：  .dw    5,89,40,12,55,32,18,46,77,21
    //=============================================================================//
                EX1     END
//=============================================================================//
```

实验二　熟悉 µ'nSP™ IDE 环境下的 C 语言的编写

【实验目的】

熟悉 µ'nSP™ IDE 环境及在该环境下用 C 语言编写的应用程序。

【实验设备】

装有 WINDOWS 系统和 µ'nSP™ IDE 仿真环境的 PC 机一台。

【实验步骤】

1)将 µ'nSP™ IDE 打开后,建立一个新工程。

2)在该项目的源文件夹(SOURCE FILES)下建立一个新的 C 语言文件。

3)编写 C 语言代码。

4)编译程序，软件调试，观察并跟踪其结果,查看各个寄存器状态,等等。

【程序范例】

```
//=================================================================//
// Program:  计算 1 to 100 累加值
// Output: [sum] = 5050(十进制)  或  13BA(十六进制)
//=================================================================//

int main()
{
      int i, Sum=0;
      for   (i=0;i<=100;i++)
            Sum = Sum + i;                          // Sum  是累加的结果
      while(1){};                                   //  程序死循环
                                                    //  用变量  Watch  窗口看  Sum  的值
}
```

【程序练习】

在 µ'nSP™ IDE 下用 C 语言使用冒泡法编写一个排序程序。

Int Array[] = {{5,89,40,12,55,32,18,46,77,21}

```
/  / ————————————————————————————
                        EX2   END
/  / ————————————————————————————
```

实验三　使用汇编语言实现 A 口的输出实验

【实验目的】

1)通过实验了解 A 口作为输出口时的使用方法。

2)使用汇编语言来实现 A 口作为输出口的方法。

【实验设备】

1)装有 WINDOWS 系统和 μ'nSP™ IDE 仿真环境的 PC 机一台。

2) μ'nSP™ 十六位单片机实验箱一个。

【实验原理】

通过点亮不同发光二极管来显示 A 口输出的数值不同。

【实验步骤】

1)根据硬件连接图连接好硬件。

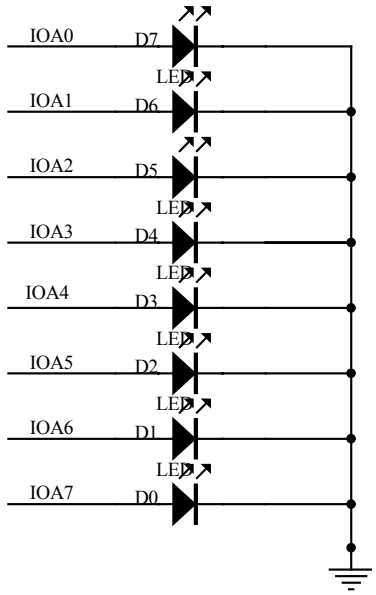
2)将 μ'nSP™ IDE 打开后,建立一个新工程。

3)在该项目的源文件夹(SOURCE FILES)下建立一个新的汇编语言文件。

4)编写汇编代码。

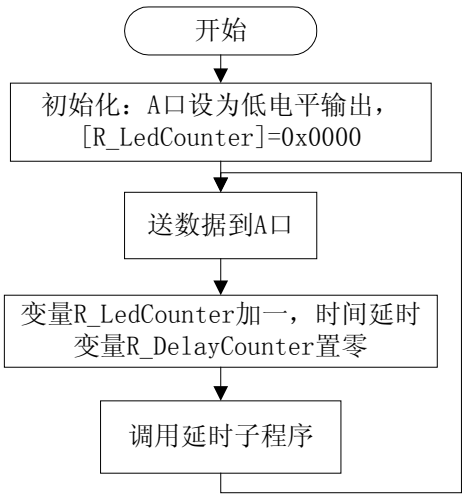
5)编译程序，软件调试，跟踪结果，观察 LED 及各个寄存器状态等等。

【硬件连接图】



【程序流程图】

主程序流程图：



【程序范例】

```
//*****************************************************************
.INCLUDE hardware.inc;                          // 包括 hardware.inc 文件

.RAM
    .VAR R_LedCounter;                              //定义变量
.VAR R_DelayCounter;
.CODE
.public _main;                                  //对 _main 程序声明
_main:
        R1 = 0xFFFF;                            //设置 A 口为同相低电平输出口
        [P_IOA_Dir] = R1;
        [P_IOA_Attrib] = R1;
        R1 = 0x0000;
        [P_IOA_Data] = R1;
        [R_LedCounter] = R1;                    //对 LED counter 清零
L_MainLoop:
        R1 = [R_LedCounter];
        [P_IOA_Data] = R1;                      //送数据到 A 口
        R1 = [R_LedCounter];
        R1 += 1;                                // LED Counter 加 1
        [R_LedCounter] = R1;
        R1 = 0x0000;                            // Delay Counter 清零
        [R_DelayCounter] = R1;
        call F_Delay;                           // 延时
        jmp L_MainLoop;                         // 跳到 L_MainLoop 循环点亮 LED

////////////////////////////////////////
//功能: 延时子函数
////////////////////////////////////////
F_Delay:
L_DelayLoop:
        R1=0x0001;                              // 清看门狗
        [P_Watchdog_Clear]=R1;
        R1 = [R_DelayCounter];
        R1 += 1;                                // Delay counter 加 1
        [R_DelayCounter] = R1;
        jnz L_DelayLoop;                        // 加到 65536 吗?
        retf;
```

【程序练习】

使用汇编语言实现 B 口的输出实验

```
//=============================================================================//
//          EX3       END
//=============================================================================//
```

实验四　使用 C 语言实现 A 口的输出实验

【实验目的】

1)通过实验了解 A 口作为输出口时的使用方法。

2)使用 C 语言来实现 A 口作为输出口的方法。

【实验设备】

1)装有 WINDOWS 系统和 μ'nSP™ IDE 仿真环境的 PC 机一台。

2) μ'nSP™ 十六位单片机实验箱一个。

【实验原理】

通过点亮不同发光二极管来显示 A 口输出的数值不同。

【实验步骤】

1)根据硬件连接图连接好硬件。

2)将 μ'nSP™ IDE 打开后,建立一个新工程。

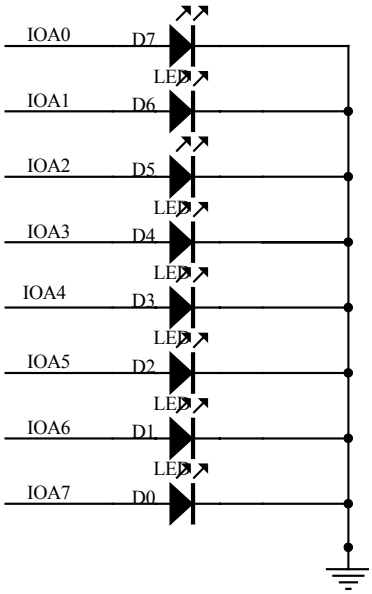
3)在该项目的源文件夹(SOURCE FILES)下建立一个新的汇编语言文件。

4)编写 C 语言代码。

5)编译程序，软件调试。

6)跟踪结果，观察 LED 及各个寄存器状态等等。

【硬件连接图】



【程序范例】

```
/*********************************************************************/
#include "hardware.h"

void Delay()
{                                                    //延时子程序
    unsigned int i;
    for(i=0; i<32768; i++){}
}

int main()
```

```
{
    int LedCounter=0;
    int Temp;
    //设置  A 口  的  Dir, Data, Attrib
    SP_Init_IOA(0xffff,0xffff,0xffff);
    //  设置 A 口为无数据反相功能的高电平输出端
    // SP_Init_IOA(xx,xx,xx)在 hardware.h 文件中已被定义
    while(1)
    {
        SP_Export(Port_IOA_Data,LedCounter);
        //  送数据到 A 口
        // SP_Export (xx,xx)在 hardware.h 文件中已被定义
        LedCounter ++;
        Delay();
    }
}
```

【程序练习】

使用 C 语言实现 B 口的输出实验

```
//=============================================================//
            EX4     END
//=============================================================//
```

实验五　使用汇编语言实现 A 口为输入 B 口为输出实验

【实验目的】

1)通过实验了解 A 口作为输入、B 口为输出口时的使用方法。

2)使用汇编语言来实现 A 口作为输入、B 口为输出口的方法。

【实验设备】

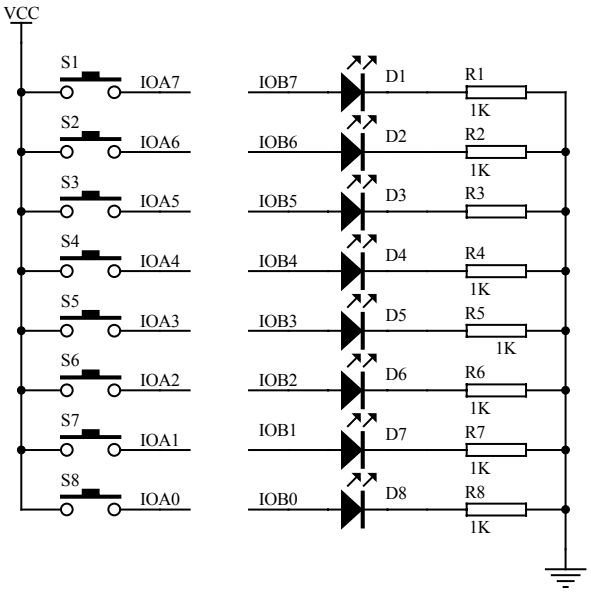
1)装有 WINDOWS 系统和 μ'nSP™ IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

根据按键的不同,A 口的数据就不同,则传送到 B 口的数据就不同,相应的发光二极管就被点亮。

【硬件连接图】



【实验步骤】

1)根据硬件连接图连接好硬件。

2)将 μ'nSP™ IDE 打开后,建立一个新工程。

3)在该项目的源文件夹(SOURCE FILES)下建立一个新的汇编语言文件。

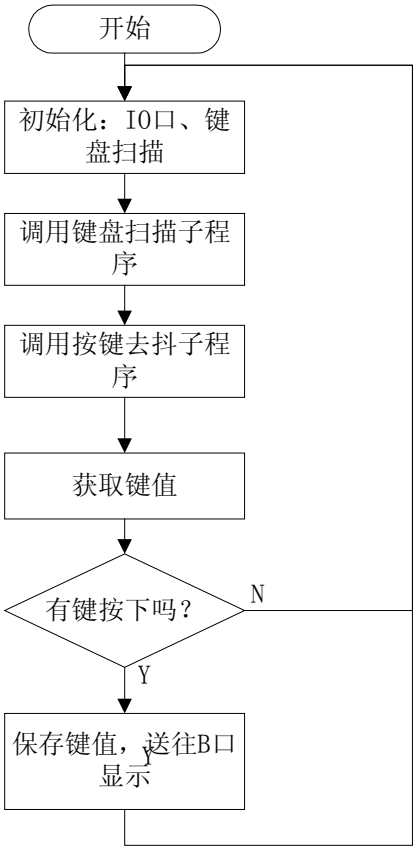
4)编写程序代码。

5)编译程序，软件调试。

6)按键并观察 LED 及各个寄存器的状态。

【程序流程图】

主程序流程图：



【程序范例】

```
/*********************************************************************/
.INCLUDE hardware.inc;
.INCLUDE key.inc;                       // 包括 key.inc 文件

.RAM
.VAR R_KeyCode;                         //定义变量保存键值

.CODE
    .public _main;                          // 对 main 程序声明
_main:
        call F_User_Init_IO;            // 初始化 IO 口
        call F_Key_Scan_Initial;        //初始化键盘扫描

L_MainLoop:
        call F_Key_Scan_ServiceLoop;    // 扫描键盘
        call F_Key_DebounceCnt_Down;    // Key debounce

        call F_SP_GetCh;                // 取键值
        cmp R1, 0x0000;                 // 是否有键按下
        je   L_MainLoop;                // 没有，继续检测

        [R_KeyCode] = R1;               //有，保存键值
```

```
        [P_IOB_Buffer] = R1;                    // 送到 B 口显示
        jmp  L_MainLoop;

F_User_Init_IO:
        r1 = 0x0000;                            // 设置 A 口为带下拉电阻的输入端
        [P_IOA_Dir] = r1;
        [P_IOA_Attrib] = r1;
        [P_IOA_Data] = r1;

        r1 = 0xffff;                            // 设置 B 口为无数据反相功能的低电
        [P_IOB_Dir] = r1;                       // 平输出
        [P_IOB_Attrib] = r1;
        r1 = 0x0000;
        [P_IOB_Data] = r1;
        retf;
```

【程序练习】

使用汇编语言实现 A 口的输出 B 口为输入实验

```
//=====================================================================//
// EX5      END
//=====================================================================//
```

实验六　使用 C 语言实现 A 口为输入 B 口为输出实验

【实验目的】

1)通过实验了解 A 口作为输入、B 口为输出口时的使用方法。

2)使用 C 语言来实现 A 口作为输入、B 口为输出口的方法。

【实验设备】

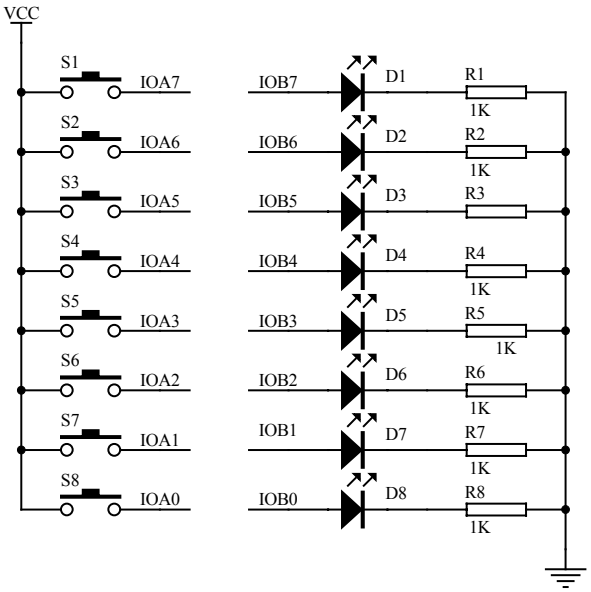
1)装有 WINDOWS 系统和 μ'nSP™ IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

根据按键的不同,A 口的数据就不同,则传送到 B 口的数据就不同,相应的发光二极管就被点亮。

【硬件连接图】



【实验步骤】

1)根据硬件连接图连接好硬件。

2)将 μ 'nSP™ IDE 打开后,建立一个新工程。

3)在该项目的源文件夹(SOURCE FILES)下建立一个新的汇编语言文件。

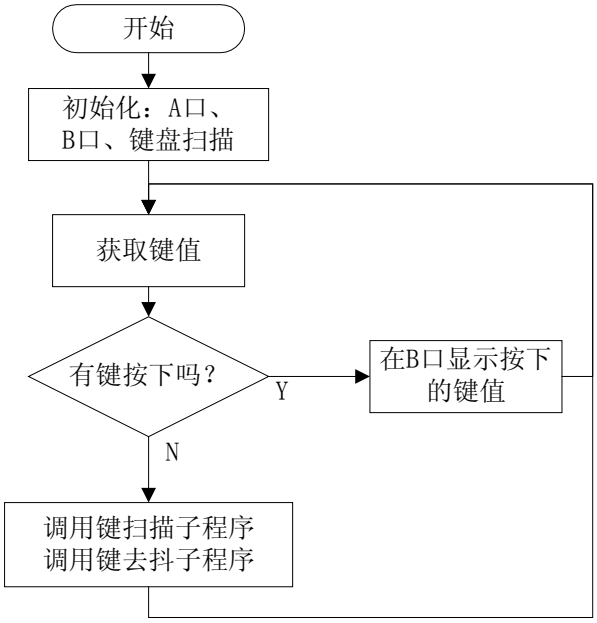
4)编写程序代码。

5)编译程序，软件调试。

6)按键并观察 LED 及各个寄存器状态。

【程序流程图】

主程序流程图：



【程序范例】

```
/*********************************************************************/
#include "hardware.h"

int main()
{
    int Key;
    // 设置 Dir,  Data,  Attrib
    SP_Init_IOA(0x0000,0x0000,0x0000);        // 设置 A 口为带下拉电阻的输入
    SP_Init_IOB(0xffff,0x0000,0xffff);        // 设置 B 口为无数据反相功能的低电平
                                              // 输出
    System_Initial();
    while(1)
    {
        Key = SP_GetCh();                     //取键值

        switch(Key)
        {
            case 0x0000:                      //没有键按下
                break;

            case 0x0001:                      // Key 1
            case 0x0002:                      // Key 2
            case 0x0004:                      // Key 3
            case 0x0008:                      // Key 4
            case 0x0010:                      // Key 5
            case 0x0020:                      // Key 6
            case 0x0040:                      // Key 7
```

```
                case 0x0080:                        // Key 8
                        SP_Export(Port_IOB_Buffer,Key);
                        // 在 B 口显示按下的键值
                    break;
                default:
                    break;
            }
            System_ServiceLoop();
        }
}
```

【程序练习】

使用 C 语言实现 A 口的输出 B 口为输入实验

```
//=====================================================================//
//          EX6       END
//=====================================================================//
```

实验七　定时器 Timer A/B 实验

【实验目的】

1)通过实验了解定时器 Timer A/B 的结构及使用方法 。

2)掌握预置数单元 P_TimerA/B_Data 和定时控制单元 P_ TimerA/B_Ctrl 的设置方法。

3)熟悉定时器 Timer A/B 的编程方法。

【实验设备】

1)装有 μ'nSP™ IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

3)示波器一台。

【实验原理】

TimerA 和 TimerB 定时器启动后，在预置数单元 P_TimerA_Data 或 P_TimerB_Data 内置入一个计数初值 N 后，定时器/计数器会在选择的时钟源频率下开始向计数增加的方向计数 N+1，N+2，…… FFFEH，当计数到 FFFFH 后定时器/计数器溢出。

一方面，产生一个中断请求信号 TA_TimeOut_INT 或 TB_TimeOut_INT，被 CPU 响应后执行相应的中断服务程序。与此同时，计数初值 N 会被自动重新置入定时器/计数器内，并重复上述加计数的过程。

另一方面，该溢出信号会作为脉宽调制输出计数器的时钟源输入，使其输出一个具有四位可调的脉宽调制占空比输出信号 APWMO 或 BPWMO，其中 IOB8，IOB9 分别为 APWM，BPWM 的输出端。IOB8 接一个发光二极管，同时接示波器，可以通过观察二极管亮灭的快慢来对比频率的变化，通过示波器观察信号的频率，如果示波器测量到的频率为 P，则 f=16*P，f 即为 CLK Source 时钟基准。

【实验步骤】

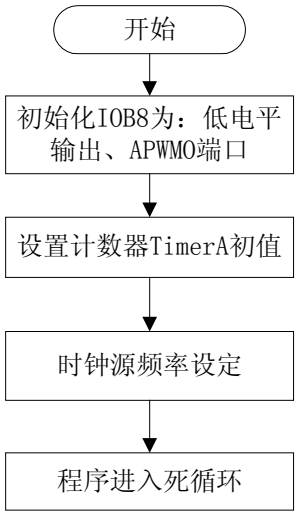
1）根据实验内容连接硬件电路并检查。

2）画流程图并编写程序代码。

3）编译程序，软件调试。

4）通过示波器观察输出频率并记录比较是否与实际相符。

【程序流程图】



【程序范例】

```
//*************************************************************
//Note: 1)选择不同的时钟源,同一占空比 8/16，观察示波器中频率的变化
```

```
//      2)选择同一时钟源，改变占空比,观察示波器，比较脉冲宽度
// Date: 2002/06/19
//************************************************************
.define     timefosc_2      0x0230;     //clkA 选择 fosc/2Hz
.define     timefosc_256    0x0231;     //clkA 选择 fosc/256Hz
.define     timeclk_32768   0x0232;     //clkA 选择 32768Hz
.define     timeclk_8192    0x0233;     //clkA 选择 8192Hz
.define     timeclk_4096    0x0234;     //clkA 选择 4096Hz
.define     timeclk_2048    0x0205;     //clkB 选择 2048Hz
.define     timeclk_1024    0x020d;     //clkB 选择 1024Hz
.define     timeclk_256     0x0215;     //clkB 选择 256Hz
.define     timeclk_4       0x0225;     //clkB 选择 4Hz
.define     timeclk_2       0x0065;     //clkB 选择 2HzHz

//频率选择 fosc/2Hz;
//使用者也可以自己设置不同频率下的占空比；

.define     timepwm_1       0x0070;     //脉宽选择 1/6
.define     timepwm_2       0x00b0;     //脉宽选择 2/6
.define     timepwm_3       0x00f0;     //脉宽选择 3/6
.define     timepwm_4       0x0130;     //脉宽选择 4/6
.define     timepwm_5       0x0170;     //脉宽选择 5/6
.define     timepwm_6       0x01b0;     //脉宽选择 6/6
.define     timepwm_7       0x01f0;     //脉宽选择 7/6
.define     timepwm_8       0x0230;     //脉宽选择 8/6
.define     timepwm_9       0x0270;     //脉宽选择 9/6
.define     timepwm_10      0x02b0;     //脉宽选择 10/6
.define     timepwm_11      0x02f0;     //脉宽选择 11/6
.define     timepwm_12      0x0330;     //脉宽选择 12/6
.define     timepwm_13      0x0370;     //脉宽选择 13/6
.define     timepwm_14      0x03b0;     //脉宽选择 14/6
.define     time_clk        timefosc_2;     //频率选择
.define     time_pwm        timepwm_4;      //脉宽选择

.define     P_TimerA_Data       0x700A;
.define     P_TimerA_Ctrl       0x700B;
.define     P_IOB_DATA          0x7005;
.define     P_IOB_DIR           0x7007;
.define     P_IOB_ATTRI         0x7008;
.define     P_Feedback          0x7009;

.code
.public _main;
_main:
```

```
    r1=0x0100;                                  //IOB8 设置为输出口
    [P_IOB_DIR]=r1;
    [P_IOB_ATTRI]=r1;
    r1=0x0000;
    [P_IOB_DATA ]=r1;
    r1=0x0000;
    [P_Feedback]=r1;                            //设置 IOB8 口为 APWMO 端口
    r1=0xff9f ;
        //设定 TA_TIMEOUT/16=(time_clk/96)/16=8Hz
    [P_TimerA_Data]=r1;
    r1=time_clk;
    //r1=time_pwm;
    [P_TimerA_Ctrl]=r1;
loop:
    nop;
    nop;
    nop;
    jmp loop;
```

【程序练习】

1)固定时钟如：CLK 选为 Fosc/2Hz，设置不同的计数初值，观察溢出时 IOB8 输出的频率是否正确。

2)使用汇编语言实现 TimerB 定时器实验：当输入时钟 CLKA=FOSC/2、CLKA=FOSC/256、CLKA=32768HZ、CLKA=8192HZ、CLKA=4096HZ 时观察输出频率。

```
//============================================================//
// EX7     END
//============================================================//
```

实验八　系统时钟实验

【实验目的】

1)了解 SPCE061 PLL 振荡器的功能及其应用。

2)掌握系统时钟单元 P_SystemClock 的设置方法。

3)熟悉系统时钟和 CPU 时钟频率的编程方法。

【实验设备】

1)装有 μ'nSP™ IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

3)示波器一台。

【实验原理】

在 SPCE061A 内，P_SystemClock(写)($7013H)单元控制着系统时钟和 CPU 时钟。通过设置该单元的 B5-B7 位可以改变系统时钟的频率(Fosc=20/24/32/40/49MHz)；将第 0-2 位置为“111”可以使 CPU 时钟停止工作，系统切换至低功耗的备用状态。在备用状态下，通过设置该单元的 B4 位可以接通或关闭 32KHz 实时时钟。而且通过设置该单元的 B3 位可以使 32768Hz 时钟处自动弱振或强振状态。本实验通过选择不同 Fosc 信号频率或改变 CPUClk 频率来观察发光二极管亮灭的快慢。

【实验步骤】

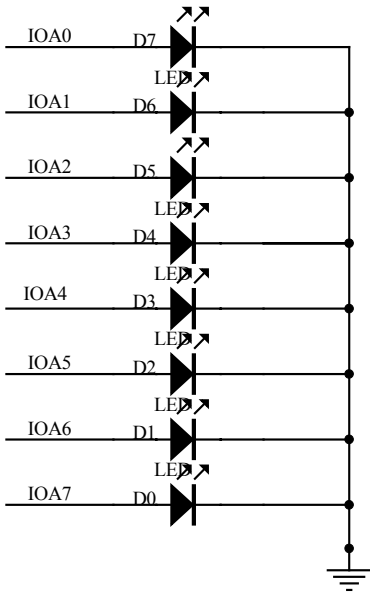
1）根据实验内容连接硬件电路并检查。

2）画流程图并编写程序代码。

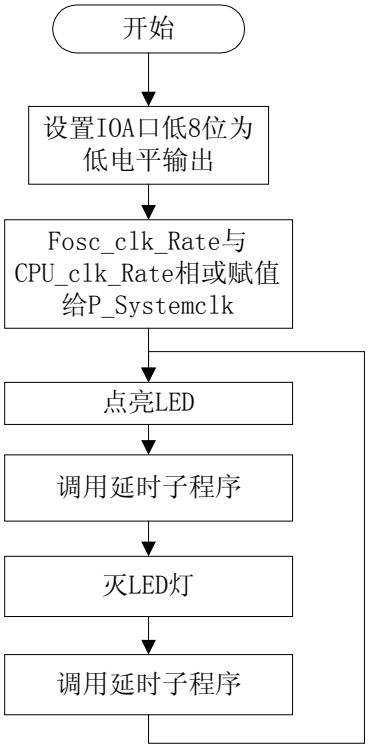
3）编译程序，软件调试。

4）观察 LED 亮灭的快慢，并通过示波器观察波形。

【硬件连接图】



【程序流程图】



【程序范例】

```
//*************************************************************
// Note: 1)选择不同 Fosc 信号频率,观察发光二极管亮灭快慢
//       2)改变 CPUClk 频率,观察发光二极管亮灭快慢
// Date: 2002/06/19
//*************************************************************

.include Hardware.inc;

.define     FoscCLK_20          0x00;          //Fosc=20.480MHz
.define     FoscCLK_24          0x20;          //Fosc=24.576MHz
.define     FoscCLK_32          0x40;          //Fosc=32.768MHz
.define     FoscCLK_40          0x60;          //Fosc=40.960MHz
.define     FoscCLK_49          0x80;          //Fosc=49.152MHz

.define    CPUCLK_Fosc          0x00;          //CPUClk 选 Fosc
.define    CPUCLK_Fosc2         0x01;          //CPUClk 选 Fosc/2
.define    CPUCLK_Fosc4         0x02;          //CPUClk 选 Fosc/4
.define    CPUCLK_Fosc8         0x03;          //CPUClk 选 Fosc/8
.define    CPUCLK_Fosc16        0x04;          //CPUClk 选 Fosc/16
.define    CPUCLK_Fosc32        0x05;          //CPUClk 选 Fosc/32
.define    CPUCLK_Fosc64        0x06;          //CPUClk 选 Fosc/64

.define    Fosc_CLK_RATE     FoscCLK_40;      //选择不同 Fosc 信号频率
```

```
.define    CPU_CLK_RATE     CPUCLK_Fosc2; //改变 CPUClk 频率
.CODE
.public _main
_main:
          R1=0x00FF;
           [P_IOA_Dir]=R1;                    // IOA: [7..0] output
           [P_IOA_Attr]=R1;
           r1=0;
           [P_IOA_Data]=R1;
           R1=Fosc_CLK_RATE;                  //Fosc
          R1|=CPU_CLK_RATE;                   //CPUClk
           //系统时钟选择设置,32768Hz 时钟默认为自动弱振模式即 B30 为 0
          [P_SystemClock]=R1;
MainLoop:
          R1=0x00FF;                         //LED 亮
          [P_IOA_Data]=R1;
          CALL Delay;
          R1=0x00;                           //LED 灭
          [P_IOA_Data]=R1;
          CALL Delay;
          JMP MainLoop;

     Delay:                                       //延时
          R3=0x4;
  DelayLoop2:
          R4=0xFFFF;
  DelayLoop1:
               R4-=1;
               JNZ DelayLoop1;
               R3-=1;
               JNZ DelayLoop2;
               RETF;

//=====================================================================//
// EX8   END
//=====================================================================//
```

实验九　FIQ 中断实验

【实验目的】

1）了解 FIQ 的中断向量和中断源。

2）掌握中断控制单元 P_INT_Ctrl, P_INT_Clear 的设置方法。

3）熟悉中断的编程方法。

【实验设备】

1)装有 μ'nSP™ IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

FIQ 中断对应 PWM、TMA、TMB 中断源，通过写 P_INT_Ctrl 来设置中断允许，FIQ_TMA,FIQ_TMB 中断源分别是通过定时器 A、定时器 B 产生的，当计满溢出时产生中断请求信号 TA_TIMEOUT_INT 或 TA_TIMEOUT_INT,CPU 响应后进入中断执行相应的子程序，中断程序里可以通过读取 P_INT_Ctrl 单元,判断是哪个中断源，并进入相应的子程序控制发光二极管点亮 。

【实验步骤】

1）根据实验内容自行设计，连接硬件电路并检查。

2）画流程图并编写程序代码。

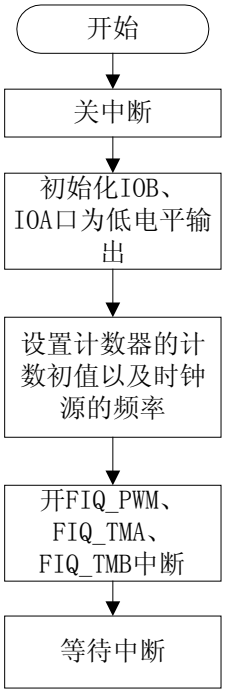
3）编译程序，软件调试。

4) 观察 LED、各个寄存器状态等。

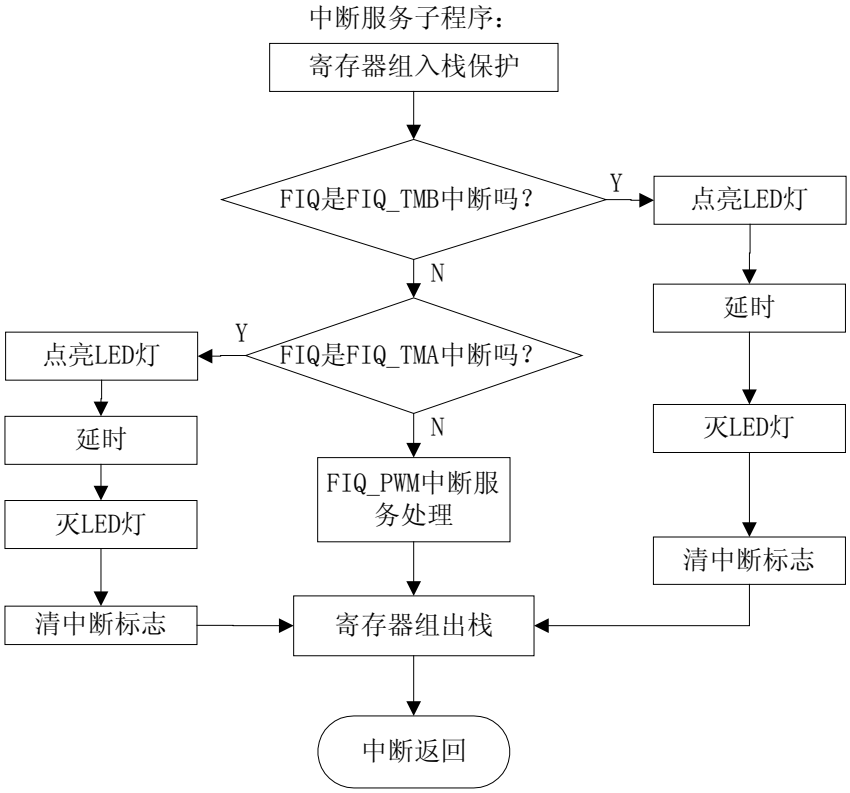
【程序流程图】

主程序流程图：

主程序：



FIQ 中断服务子程序流程图 ：



【程序范例】

```
//*********************************************************
// Note: FIQ 有 FIQ_PWM、FIQ_TMA 和 FIQ_TMB 三个中断源,当定时器 A 或 B
//计满溢出时产生中断请求信号 TA_TIMEOUT_INT 或 TA_TIMEOUT_INT,
//CPU 响应后进入中断执行相应的子程序控制二极管发光。
//A 口的低四位接 LED 灯，B 口的低四位接 LED 灯.
// Date: 2002/06/19
//*********************************************************
.define     P_IOA_DATA      0x7000;
.define     P_IOA_DIR       0x7002;
.define     P_IOA_ATTRI     0x7003;
.define     P_IOB_DATA      0x7005;
.define     P_IOB_DIR       0x7007;
.define     P_IOB_ATTRI     0x7008;
.define     P_INT_CTRL      0x7010;
.define     P_INT_CLEAR     0x7011;
.define     P_TimerA_Data   0x700A;
.define     P_TimerA_Ctrl   0x700B;
.define     P_TimerB_Data   0x700C;
.define     P_TimerB_Ctrl   0x700D;
.define     timea_clk       0x000d;
.define     timeb_clk       0x0004;
```

```
.code
.public _main
_main:
    int off
    r1=0xffff                          //IOA 口设为同相高电平输出口
    [P_IOA_ATTRI]=r1
    [P_IOA_DIR]=r1
    [P_IOA_DATA]=r1

    r1=0xffff                          //B 口的低 8 位设置为同相高电平输出
    [P_IOB_DIR]=r1
    [P_IOB_ATTRI]=r1
    [P_IOB_DATA]=r1

    r1=0xf09f;                         //设置计数初值
    [P_TimerA_Data]=r1;
    [P_TimerB_Data]=r1;
    r1=timea_clk;
    [P_TimerA_Ctrl]=r1;
    r1=timeb_clk;
    [P_TimerB_Ctrl]=r1;

    r1=0xa800                          //开中断 FIQ_PWM、FIQ_TMA、FIQ_TMB
    [P_INT_CTRL]=r1
    int FIQ

loop:
    nop
    nop
    nop
    jmp loop

.text
.public _FIQ
_FIQ:
    push r1,r5 to [sp]                 //压栈保护
    r1=0x0800
    test r1,[P_INT_CTRL]               //比较是否为 FIQ_TMB
    jnz   FIQ_TMB                      //是，则转至对应程序段
    r1=0x2000
    test r1,[P_INT_CTRL]               //否，则比较是否为 FIQ_TMA
    jnz   FIQ_TMA                      //是，则转至对应程序段
```

```
FIQ_PWM:                                    //否，则进入 FIQ_PWM 中断
    [P_INT_CLEAR]=r1                        //r1=0x8000
    pop r1,r5 from [sp]
    reti

FIQ_TMA:
        r1=0xfff0
        //点亮接在 A 口低四位的四个 LED 灯（低电平点亮）
    [P_IOA_DATA]=r1;


    r2=0xffff;                              //延时
delay1:
    r2-=1;
    jnz delay1;

    r1=0xffff;                              //熄灭 LED
    [P_IOA_DATA]=r1;

    r1=0x2000;
    [P_INT_CLEAR]=r1;
    pop r1,r5 from [sp];
    reti;

FIQ_TMB:
    r1=0xfff0;
    //点亮接在 B 口低四位的四个 LED 灯（低电平点亮）
    [P_IOB_DATA]=r1;


    r2=0xffff;                              //延时
delay2:
    r2-=1;
    jnz delay2;

    r1=0xffff;                              //熄灭 LED
    [P_IOB_DATA]=r1;

    r1=0x0800;
    [P_INT_CLEAR]=r1;
    pop r1,r5 from [sp];
    reti;
```

实验十　IRQ0/IRQ1/IRQ2 中断实验

【实验目的】

1）了解 IRQ0、IRQ1、IRQ2 的中断向量和中断源。

2）掌握中断控制单元 P_INT_Ctrl、P_INT_Clear 的设置方法。

3）熟悉中断的编程方法。

【实验设备】

1)装有 μ'nSP™ IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

IRQ0_TWM、IRQ1_TMA、IRQ2_TMB　中断源,通过写 P_INT_Ctrl 来设置中断允许，IRQ_TMA 和 IRQ_TMB 中断源分别是通过定时器 A、定时器 B 产生的，当计满溢出时产生中断请求信号 TA_TIMEOUT_INT 或 TA_TIMEOUT_INT,CPU 响应后进入对应的中断，并在相应的中断程序里执行控制发光二极管亮灭程序，从而了解这三个中断及各自的编程。

【实验步骤】

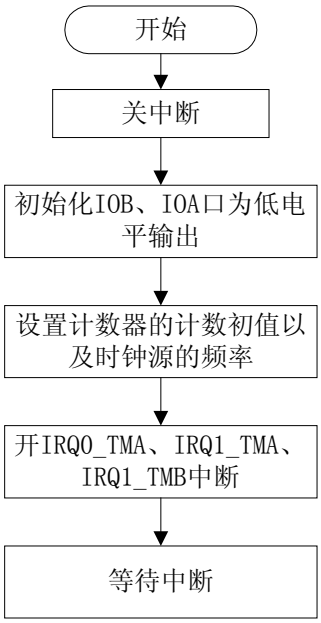
1）根据实验内容自行设计，连接硬件电路并检查。

2）画流程图并编写程序代码。

3）编译程序，软件调试。

4) 观察 LED、各个寄存器状态,等等。

【程序流程图】



【程序范例】

```
//*************************************************************
// Note:IRQ0_PWM、IRQ1_TMA、IRQ2_TMB 三个中断,当定时器 A 或 B
//计满溢出时产生中断请求信号 TA_TIMEOUT_INT 或 TA_TIMEOUT_INT,
//CPU 响应后执行相应的中断服务子程序
//A 口的低四位接 LED 灯，B 口的低四位接 LED 灯.
// Date: 2002/06/19
//*************************************************************
.define        P_IOA_DATA              0x7000;
.define        P_IOA_DIR               0x7002;
```

```
.define     P_IOA_ATTRI         0x7003;
.define     P_IOB_DATA          0x7005;
.define     P_IOB_DIR           0x7007;
.define     P_IOB_ATTRI         0x7008;
.define     P_INT_CTRL          0x7010;
.define     P_INT_CLEAR         0x7011;
.define     P_TimerA_Data       0x700A;
.define     P_TimerA_Ctrl       0x700B;
.define     P_TimerB_Data       0x700C;
.define     P_TimerB_Ctrl       0x700D;
.define     timea_clk           0x020d;
.define     timeb_clk           0x0004;

.code
.public _main;
_main:
    int off;
    r1=0xffff;                          // IOA 为无数据反相功能的低电平输出口
    [P_IOA_ATTRI]=r1;
    [P_IOA_DIR]=r1;
    r1=0x0000;
    [P_IOA_DATA]=r1;

    r1=0xffff;                          // IOB 为无数据反相功能的低电平输出口
    [P_IOB_DIR]=r1;
    [P_IOB_ATTRI]=r1;
    r1=0x0000;
    [P_IOB_DATA]=r1;

    r1=0xff9f;
    [P_TimerA_Data]=r1;
    [P_TimerB_Data]=r1;
    r1=timea_clk;
    [P_TimerA_Ctrl]=r1;
    r1=timeb_clk;
    [P_TimerB_Ctrl]=r1;

    r1=0x5400;                          //开中断 IRQ0_PWM、IRQ1_TMA、IRQ2_TMB
    [P_INT_CTRL]=r1;
    int IRQ;
loop:
    nop;
    nop;
    nop;
```

```
    jmp loop;

.text
.public _IRQ0;
.public _IRQ1;
.public _IRQ2;
_IRQ0:
    push r1,r5 to [sp]                          //压栈保护
    r1=0x4000;
    [P_INT_CLEAR]=r1;
    pop r1,r5 from [sp];
    reti;

_IRQ1:
    push r1,r5 to [sp];                         //压栈保护
    r1=0xfff0 ;                                 //点亮与 A 口低四位相连的 LED 灯
[P_IOA_DATA]=r1;
    r1=0x1000;
    [P_INT_CLEAR]=r1;
    pop r1,r5 from [sp];
    reti;

_IRQ2:
    push r1,r5 to [sp];                         //压栈保护
    r1=0xfff0;                                  //点亮与 B 口低四位相连的 LED 灯
    [P_IOB_DATA]=r1;
    r1=0x0400;
    [P_INT_CLEAR]=r1;
    pop r1,r5 from [sp];
    reti;

//=============================================================//
// EX10    END
//=============================================================//
```

实验十一　　IRQ4 中断实验

【实验目的】

1)了解 IRQ4 的中断向量和中断源。

2)掌握中断控制单元 P_INT_Ctrl, P_INT_Clear 的设置方法。

3)熟悉中断的编程方法。

【实验设备】

1)装有 μ'nSP™ IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

IRQ4 中断对应 4096Hz、2048Hz、1024Hz 中断源,通过写 P_INT_Ctrl 来设置中断允许，CPU 响应后进入中断， 我们编写一个用中断方式控制发光二极管亮灭程序, 中断程序里读取 P_INT_Ctrl 单元, 判断是哪个中断源，转到相应中断程序控制对应发光二极管亮或灭.从而了解 IRQ4 中断的组成及编程。

【实验步骤】

1）根据实验内容自行设计，连接硬件电路并检查。

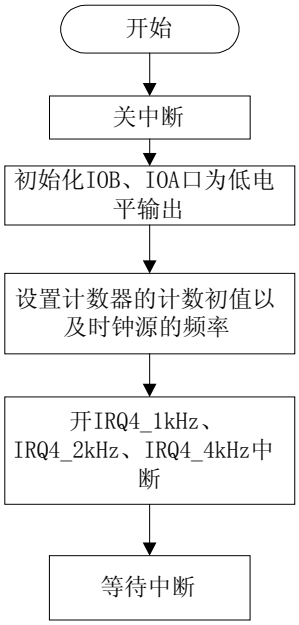
2）画流程图并编写程序代码。

3）编译程序，软件调试。

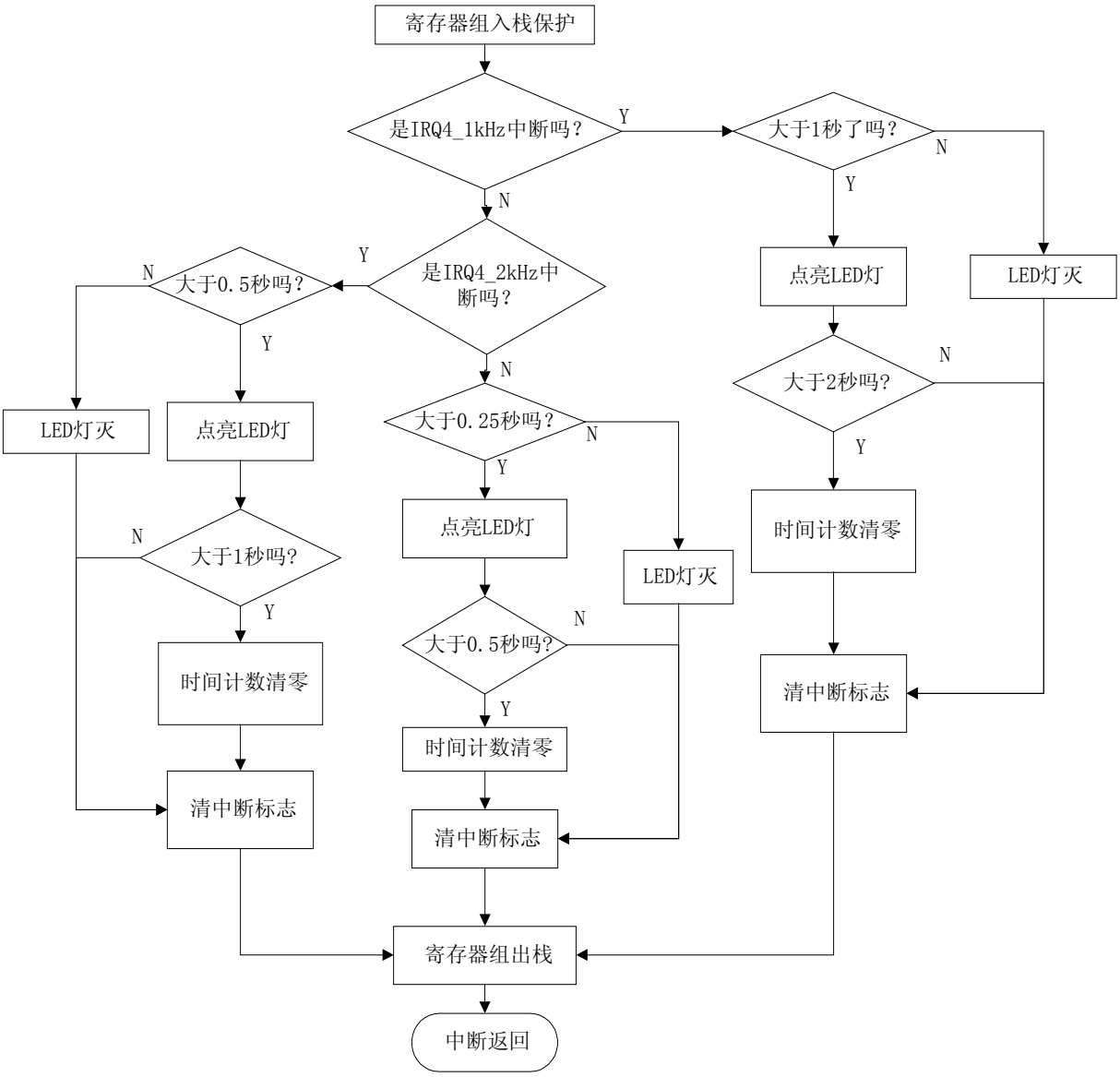
4) 观察 LED、各个寄存器状态等。

【程序流程图】

主程序流程图：



中断服务子程序：



【程序范例】

```
//*************************************************************
// Note: IRQ5 中断有三个中断源：1KHz、2Kz 和 4KHz,每一个中断分别控制与 IOA0-IOA1
//、IOA2-IOA3 和 IOA4-IOA7 相连的 LED 灯。
// Date: 2002/06/19
//*************************************************************
.define         P_IOA_DATA      0x7000;
.define         P_IOA_DIR       0x7002;
.define         P_IOA_ATTRI     0x7003;
.define         P_IOB_DATA      0x7005;
.define         P_IOB_DIR       0x7007;
.define         P_IOB_ATTRI     0x7008;
.define         P_INT_CTRL      0x7010;
```

```
.define          P_INT_CLEAR     0x7011;

.RAM
.VAR TIME1
.VAR TIME2
.VAR TIME4

.code
.public _main
_main:
      int off;
      r1=0xffff;                          //IOA 口为同相高电平输出口；
      [P_IOA_ATTRI]=r1;
      [P_IOA_DIR]=r1;
      [P_IOA_DATA]=r1;
      r1=0xffff;                          //IOB 口为同相低电平输出口；
      [P_IOB_ATTRI]=r1;
      [P_IOB_DIR]=r1;
      [P_IOB_DATA]=r1;
      r1=0x0070;                          //开中断 IRQ4_4KHz、IRQ4_2KHz 和 IRQ4_1KHz;
      [P_INT_CTRL]=r1;
      R1=0;
      [TIME1]=r1;
      [TIME2]=r1;
      [TIME4]=r1;
      int   irq;

loop:
      nop;
      nop;
      nop;
      nop;
      jmp loop;

.text
.public _IRQ4
_IRQ4:
      push r1,r5 to [sp]                  //压栈保护；
      r1=0x0010;
      test r1,[P_INT_CTRL];               //比较是否为 1KHz 的中断源；
      jnz l_irq4_1k;                      //是，则转至对应程序段；
      r1=0x0020;
      test r1,[P_INT_CTRL]                //否，则比较是否为 2KHz 的中断源；
      jnz l_irq4_2k;                      //是，则转至对应程序段；
```

```
l_irq4_4k:                          //否，则进入 4KHz 程序段；
    r2=[TIME4];
    r2+=0x0001;
    [TIME4]=R2;
    cmp r2,0x03ff                   //比较是否为 0.25 秒；
    jbe LED4kHZ_OFF                 //小于等于则 LED 灭；
    r1=0xff0f;                      //大于则 LED 亮；
    [P_IOA_DATA]=r1;
    cmp r2,0x07ff                   //比较是否为 0.5 秒；
    jbe LED4kHz_RET                 //小于等于则 LED 继续亮；
    r2=0x000;                       //否则，TIME4 单元清零，返回中断；
    [TIME4]=R2;
    jmp LED4kHz_RET
LED4kHZ_OFF:
    r1=0xffff;
    [P_IOA_DATA]=r1;
LED4kHz_RET:
    r1=0x0040;
    [P_INT_CLEAR]=r1;
    pop r1,r5 from [sp]
    RETI;
l_irq4_2k:
    r2=[TIME2];
    r2+=0x0001;
    [TIME2]=R2;
    cmp r2,0x03ff                   //比较是否为 0.5 秒；
    jbe LED2kHZ_OFF                 //小于等于则 LED 灭；
    r1^=0xfff3;                     //大于则 LED 亮；
    [P_IOA_DATA]=r1;
    cmp r2,0x07ff                   //比较是否为 1 秒；
    jbe LED2kHz_RET                 //小于等于则 LED 继续亮；
    r2=0x000;                       //否则，TIME2 单元清零，返回中断；
    [TIME2]=R2;
    jmp LED2kHz_RET
LED2kHZ_OFF:
    r1=0xffff;
    [P_IOA_DATA]=r1;
LED2kHz_RET:
    r1=0x0020;
    [P_INT_CLEAR]=r1;
    pop r1,r5 from [sp]
    RETI
```

```
l_irq4_1k:
    r2=[TIME1];
    r2+=0x0001;
    [TIME1]=R2;
    cmp r2,0x03ff                    //比较是否为 1 秒；
    jbe LED1kHZ_OFF                  //小于等于则 LED 灭；
    r1=0xfffc;                       //大于则 LED 亮；
    [P_IOA_DATA]=r1;
    cmp r2,0x07ff                    //比较是否为两秒；
    jbe LED1kHz_RET                  //小于等于则 LED 继续亮；
    r2=0x000;                        //否则，TIME1 单元清零，返回中断；
    [TIME1]=R2;
    jmp LED1kHz_RET
LED1kHZ_OFF:
    r1=0xffff;
    [P_IOA_DATA]=r1;
LED1kHz_RET:
    r1=0x0010;
    [P_INT_CLEAR]=r1;
    pop r1,r5 from [sp]
    RETI
```

实验十二　　IRQ5 中断实验

【实验目的】

1)了解 IRQ5 的中断向量和中断源。

2)掌握中断控制单元 P_INT_Ctrl, P_INT_Clear 的设置方法。

3)熟悉中断的编程方法。

【实验设备】

1)装有 μ'nSP™ IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

IRQ5 中断对应 4Hz、2Hz 中断源,通过写 P_INT_Ctrl 来设置中断允许，程序运行后就会产生相应的中断. 我们编写一个用中断方式控制发光二极管亮灭程序, 中断程序里读取 P_INT_Ctrl 单元,判断是哪个中断源，转到相应中断程序控制对应发光二极管亮或灭.从而了解 IRQ5 中断的组成及编程。

【实验步骤】

1）根据实验内容自行设计，连接硬件电路并检查。

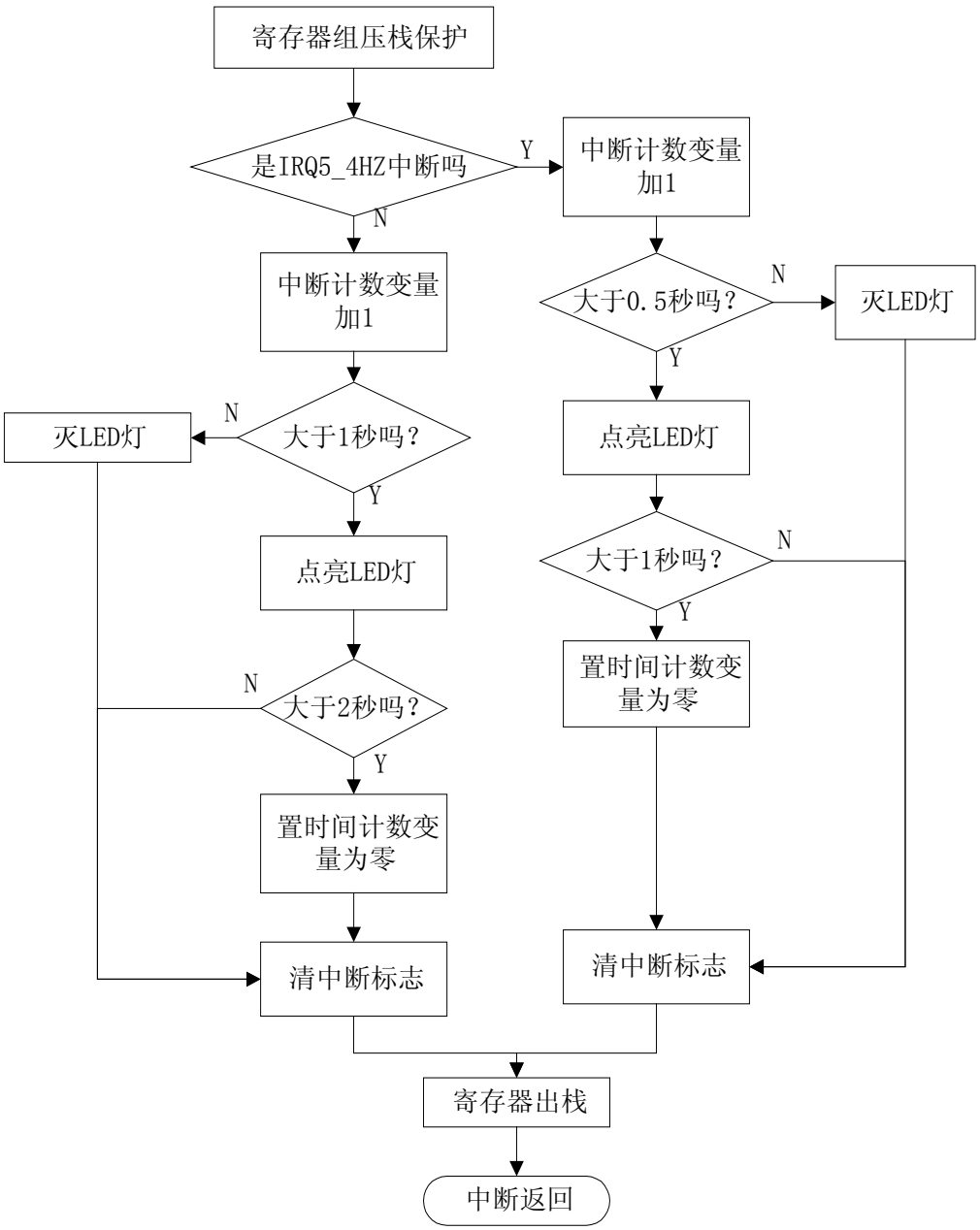
2）画流程图并编写程序代码。

3）编译程序，软件调试。

4) 观察 LED、各个寄存器状态,等等。

【程序流程图】

中断服务子程序流程图：



【程序范例】

```
//*********************************************************************
// Note: IRQ5 中断有两个中断源：2Hz 和 4Hz,每个中断分别控制二极管的亮灭//即 2Hz 中断控制与 A
口的 IOA0-IOA3 位相连的 LED，4Hz 中断控制与 IOB 口
//的 IOB0-IOB3 位相连的 LED 灯。
// Date: 2002/06/19
//*********************************************************************
.define         P_IOA_DATA      0x7000;
.define         P_IOA_DIR       0x7002;
```

```
.define         P_IOA_ATTRI     0x7003;
.define         P_IOB_DATA      0x7005;
.define         P_IOB_DIR       0x7007;
.define         P_IOB_ATTRI     0x7008;
.define         P_INT_CTRL      0x7010;
.define    P_INT_CTRL_NEW       0x702D;
.define         P_INT_CLEAR     0x7011;

.RAM
.VAR TIME2
.VAR TIME4

.code
.public _main
_main:
    int off
    r1=0xffff                       //IOA 口为同相高电平输出口；
    [P_IOA_ATTRI]=r1
    [P_IOA_DIR]=r1
    [P_IOA_DATA]=r1
    r1=0xffff                       //IOB 口为同相高电平输出口；
    [P_IOB_ATTRI]=r1
    [P_IOB_DIR]=r1
    [P_IOB_DATA]=r1
    r1=0x000c                       //开中断 IRQ5_4Hz 和 IRQ5_2Hz
    [P_INT_CTRL]=r1
    [P_INT_CTRL_NEW]=r1
    R1=0
    [TIME2]=r1
    [TIME4]=r1
    int   irq
loop:
    nop
    nop
    nop
    nop
    jmp loop
.text
.public _IRQ5
_IRQ5:
    push r1,r5 to [sp]              //压栈保护
    r1=0x0008
    test r1,[P_INT_CTRL]            //比较是否为 4Hz 的中断源
    jnz l_irq5_4                    //是，则转至对应程序段
```

```
l_irq5_2:                          //否，则进入 2Hz 程序段
    r1=0x0004
    [P_INT_CLEAR]=r1
    r2=[TIME2]
    r2+=0x0001
    [TIME2]=R2
    cmp r2,2                       //比较是否为 1 秒
    jbe LED2HZ_OFF                 //小于等于则 LED 灭
    r1=0xfff0                      //大于则 LED 亮
    [P_IOA_DATA]=r1
    cmp r2,3                       //比较是否为两秒
    jbe LED2Hz_RET                 //小于等于则 LED 继续亮
    r2=0x000                       //否则，TIME1 单元清零，返回中断
    [TIME2]=R2
    jmp LED2Hz_RET
LED2HZ_OFF:
    r1=0xffff
    [P_IOA_DATA]=r1
LED2Hz_RET:

    pop r1,r5 from [sp]
    RETI
l_irq5_4:
    r1=0x0008
    [P_INT_CLEAR]=r1
    r2=[TIME4]
    r2+=0x0001
    [TIME4]=R2
    cmp r2,3                       //比较是否为 0.5 秒
    jbe LED4HZ_OFF                 //小于等于则 LED 灭
    r1=0xfff0                      //大于则 LED 亮
    [P_IOB_DATA]=r1
    cmp r2,7                       //比较是否为 1 秒
    jbe LED4Hz_RET                 //小于等于则 LED 继续亮
    r2=0x000                       //否则，TIME2 单元清零，返回中断
    [TIME4]=R2
    jmp LED4Hz_RET
LED4HZ_OFF:
    r1=0xffff
    [P_IOB_DATA]=r1
LED4Hz_RET:

    pop r1,r5 from [sp]
    RETI
```

实验十三　　IRQ6 中断实验

【实验目的】

1)了解 IRQ6 的中断向量和中断源

2)掌握中断控制单元 P_INT_Ctrl, P_INT_Clear 的设置方法

3)熟悉中断的编程方法.

【实验设备】

1)装有 μ'nSP™ IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个

【实验原理】

IRQ6 中断对应 TMB1、TMB2 中断源, 先设置 P_TimeBase_Setup 通过改变 tmb1_clk 可以选择时基频率 TMB1（8/16/32/64Hz）；同样改变 tmb2_clk 可以选择时基频率 TMB2（128/256/5122/1024Hz），写 P_INT_Ctrl 设置中断允许，CPU 响应中断后可以通过读取 P_INT_Ctrl 单元,判断是哪个中断源，并进入相应的子程序控制发光二极管亮灭。

【实验步骤】

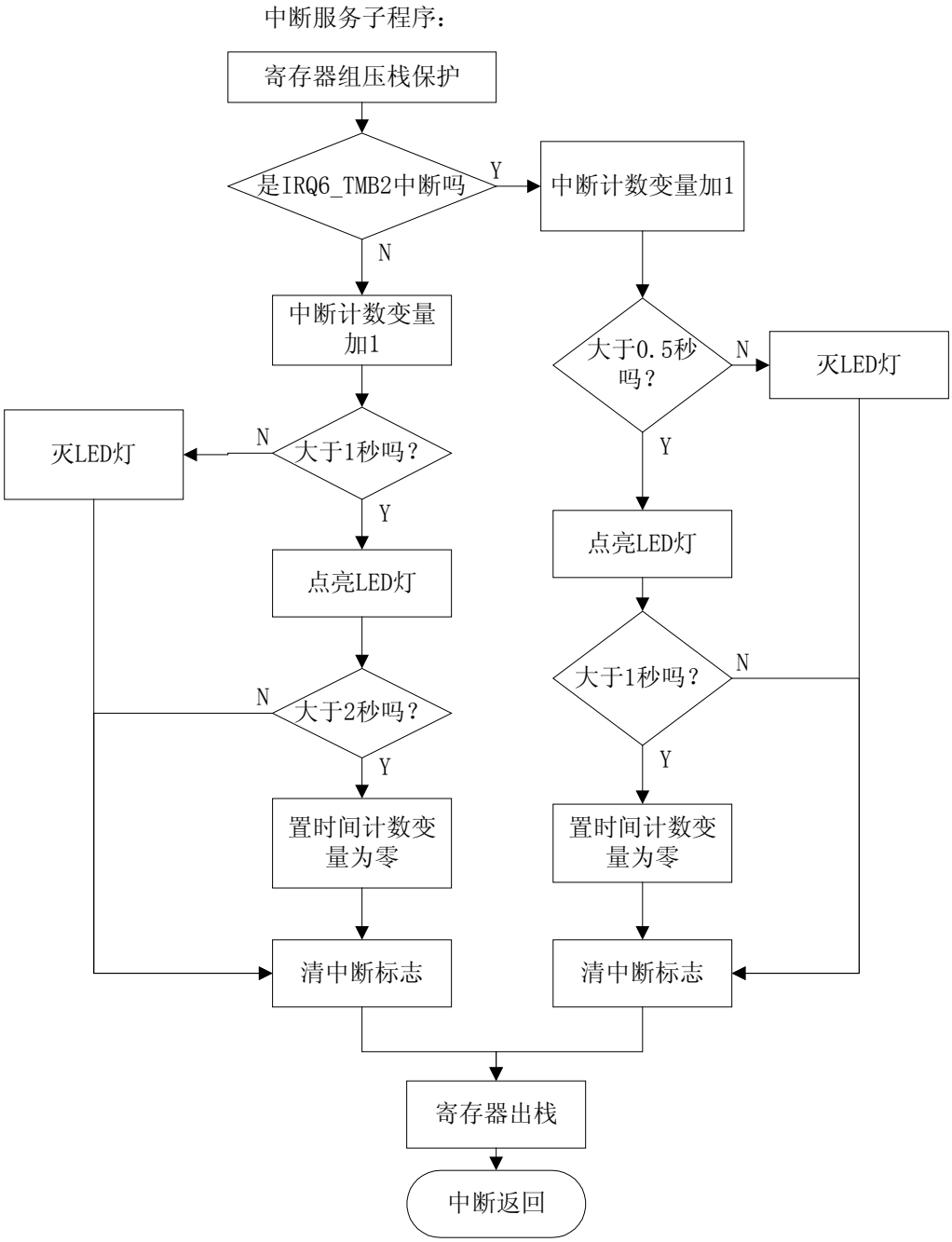
1）根据实验内容自行设计，连接硬件电路并检查。

2）画流程图并编写程序代码。

3）编译程序，软件调试。

4) 观察 LED、各个寄存器状态,等等。

【程序流程图】



【程序范例】

```
//*********************************************************************
// Note: IRQ6 中断有两个中断源：IRQ6_TMB1 和 IRQ6_TMB2。每个中断都控制着二极管
//亮灭，即 IRQ6_TMB2 中断控制与 B 口的低 4 位相连的 LED 的亮灭，IRQ6_TMB1 中断
//控制与 A 口的低 4 位相连的 LED 的亮灭。
//Date: 2002/06/19
//*********************************************************************
.define         tmb1clk_8Hz  0x0000
.define         tmb1clk_16Hz       0x0001
```

```
.define     tmb1clk_32Hz    0x0002
.define     tmb1clk_64Hz    0x0003
.define     tmb2clk_128Hz   0x0000
.define     tmb2clk_256Hz   0x0004
.define     tmb2clk_512Hz   0x0008
.define     tmb2clk_1024Hz  0x000c
.define     tmb1_clk        tmb1clk_64Hz
.define     tmb2_clk        tmb2clk_128Hz



.define     P_IOA_DATA      0x7000;
.define     P_IOA_DIR       0x7002;
.define     P_IOA_ATTRI     0x7003;
.define     P_IOB_DATA      0x7005;
.define     P_IOB_DIR       0x7007;
.define     P_IOB_ATTRI     0x7008;
.define     P_INT_CTRL      0x7010;
.define     P_INT_CLEAR     0x7011;

.define     P_Timebase_setup 0x700e
.define     P_Timebase_clear 0x700f

.RAM
.VAR TIME1
.VAR TIME2

.code
.public _main
_main:
    int off
    r1=0xffff                       //设置 IOA 口为同相高电平输出口
    [P_IOA_ATTRI]=r1
    [P_IOA_DIR]=r1
    [P_IOA_DATA]=r1
    r1=0xffff                       //设置 IOB 口为同相高电平输出口
    [P_IOB_ATTRI]=r1
    [P_IOB_DIR]=r1
    [P_IOB_DATA]=r1

    r1=tmb1_clk
    r1|=tmb2_clk
    [P_Timebase_setup]=r1
    r1=0x0003                       //开中断 IRQ6_TMB1 和 IRQ6_TMB2
```

```
        [P_INT_CTRL]=r1
        R1=0
        [TIME1]=r1
        [TIME2]=r1
        int    irq;
loop:
        nop
        nop
        nop
        nop
        jmp loop

.text
.public _IRQ6
_IRQ6:
        push r1,r5 to [sp]                  //压栈保护
        r1=0x0001
        test r1,[P_INT_CTRL]                //比较是否为 IRQ6_TMB2 的中断源
        jnz l_irq6_tmb2                     //是，则转至对应程序段

l_irq6_tmb1:                                //否，则进入 IRQ6_TMB1 程序段；
        r2=[TIME1]
        r2+=0x0001
        [TIME1]=R2
        cmp r2,64                           //比较是否为 1 秒；
        jbe LED1_OFF                        //小于等于则 LED 灭；
        r1=0xfff0                           //大于则 LED 亮；
        [P_IOA_DATA]=r1
        cmp r2,128                          //比较是否为两秒；
        jbe LED1_RET                        //小于等于则 LED 继续亮；
        r2=0x000                            //否则，TIME1 单元清零，返回中断；
        [TIME1]=R2
        jmp LED1_RET
LED1_OFF:
        r1=0xffff
        [P_IOA_DATA]=r1
LED1_RET:
        r1=0x0002
        [P_INT_CLEAR]=r1
        pop r1,r5 from [sp]
        RETI

l_irq6_tmb2:
        r2=[TIME2]
```

```
        r2+=0x0001
        [TIME2]=R2
        cmp r2,64                       //比较是否为 0.5 秒；
        jbe LED2_OFF                    //小于等于则 LED 灭；
        r1=0xfff0                       //大于则 LED 亮；
        [P_IOB_DATA]=r1
        cmp r2,128                      //比较是否为 1 秒；
        jbe LED2_RET                    //小于等于则 LED 继续亮；
        r2=0x0000                       //否则，TIME2 单元清零，返回中断；
        [TIME2]=R2
        jmp LED2_RET
    LED2_OFF:
        r1=0xffff
        [P_IOB_DATA]=r1
    LED2_RET:
        r1=0x0001
        [P_INT_CLEAR]=r1
        pop r1,r5 from [sp]
        RETI
```

实验十四　外部中断 EXT1,EXT2 实验

【实验目的】

1）了解 IRQ3 的中断向量和中断源及外部时钟的触发方式。

2）掌握中断控制单元 P_INT_Ctrl, P_INT_Clear 的设置方法。

3）熟悉中断的编程方法。

【实验设备】

1)装有 μ'nSP™ IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

IRQ3 中断对应 EXT1、EXT2、IRQ3_KEY（键唤醒 IRQ3_KEY 后面会讲到）三个中断源, 通过写 P_INT_Ctrl 来设置中断允许,程序运行后外部输入信号 EXT1、 EXT2 端输入负跳沿（将 IOB2、IOB3 接地）就会产生触发信号响应中断， 中断程序里通过读取 P_INT_Ctrl 单元, 判断是哪个中断源，转到相应子程序控制对应发光二极管亮灭，从而了解 IRQ3 中断的组成及编程方法。

【实验步骤】

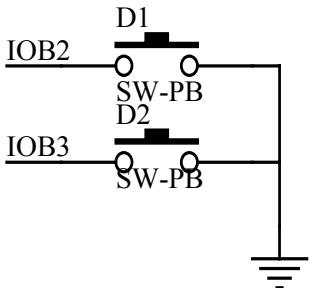
1)根据实验内容连接硬件电路并检查。

2)画流程图并编写程序代码。

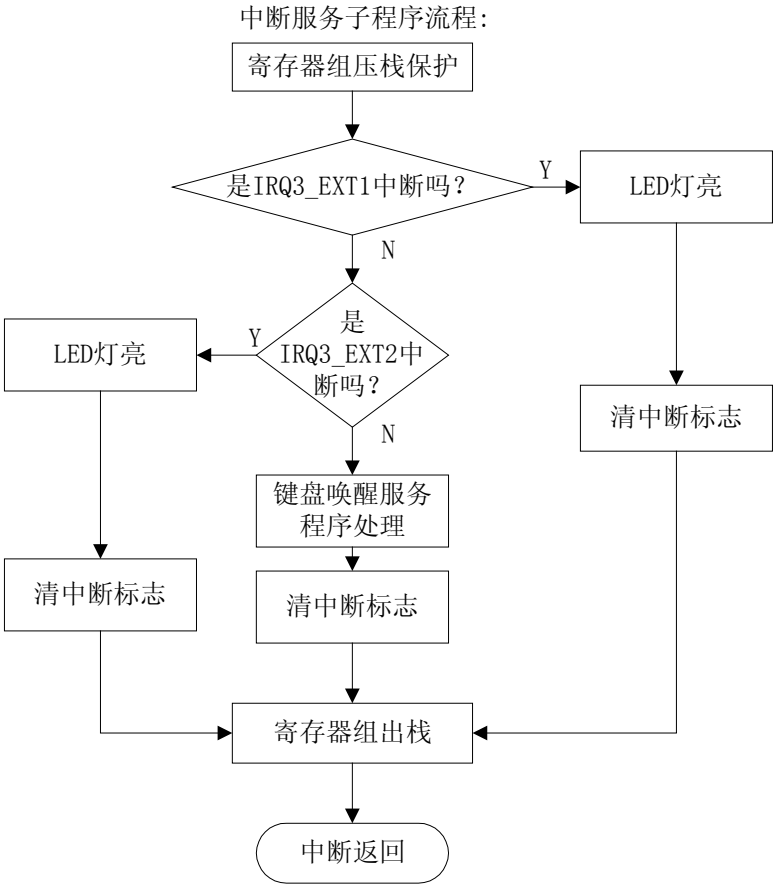
3 编译程序，软件调试。

4)按键(IOB2 或 IOB3 接地)观察 LED 并跟踪其结果。

【硬件连接图】



【程序流程图】



【程序范例】

```
//*************************************************************
//Note: IRQ3 有 IRQ3_EXT2、IRQ3_EXT1 和 IRQ3_KEY 三个中断源, 这里
//只做外部中断，（键唤醒中断后面有专门的实验）通过 IOB2 或 IOB3 产生
//一个负跳沿来触发中断，进入中断点亮与 IOA0-IOA3 相连的 LED 灯，或
//与 IOA4-IOA7 相连的 LED 灯
// Date: 2002/06/19
//*************************************************************
.define     P_IOA_DATA      0x7000;
.define     P_IOA_DIR       0x7002;
.define     P_IOA_ATTRI     0x7003;
.define     P_IOB_DATA      0x7005;
.define     P_IOB_DIR       0x7007;
.define     P_IOB_ATTRI     0x7008;
.define     P_INT_CTRL      0x7010;
.define     P_INT_CLEAR     0x7011;
.define     P_Feedback      0x7009;


.code
.public _main
```

```
_main:
      int off
      r1=0xffff                          //设置 IOA 口为同相高电平输出口；
      [P_IOA_ATTRI]=r1
      [P_IOA_DIR]=r1
      [P_IOA_DATA]=r1

      //设置 IOB2、IOB3 设成带上拉电阻的输入端口
      r1=0x0000;
      [P_IOB_DIR]=r1;
      [P_IOB_ATTRI]=r1;
      r1=0x000c ;
      [P_IOB_DATA]=r1;
      r1=0x0300;                         //开中断 IRQ3_EXT1、IRQ3_EXT2
      [P_INT_CTRL]=r1;
      int irq;
loop:
      nop
      nop
      nop
      jmp loop


.text
.public _IRQ3
_IRQ3:
    push r1,r5 to [sp]                   //压栈保护
    r1=0x0100
    test r1,[P_INT_CTRL]                 //比较是否为 IRQ3_EXT1
    jnz   irq3_ext1                      //是，则转至对应程序段；
    r1=0x0200
    test r1,[P_INT_CTRL]                 //否，则比较是否为 IRQ3_EXT2
    jnz   irq3_ext2                      //是，则转至对应程序段；
    r1=0x0200

irq3_key:                                //否，则进入键唤醒中断
      r1=0x0080
      [P_INT_CLEAR]=r1
      pop r1,r5 from [sp]
      reti

irq3_ext2:
    r1=0xff0f                            //点亮与 IOA4-IOA7 相连的 LED 灯
    [P_IOA_DATA]=r1
```

```
    r1=0x0200
    [P_INT_CLEAR]=r1
    pop r1,r5 from [sp]
    reti

irq3_ext1:
    r1=0xfff0                                   //点亮与 IOA0-IOA3 相连的 LED 灯
    [P_IOA_DATA]=r1
    r1=0x0100
    [P_INT_CLEAR]=r1
    pop r1,r5 from [sp]
    reti
```

实验十五　　键唤醒

【实验目的】

1)了解 SPCE061 睡眠和唤醒的结构原理。

2)熟悉 SPCE061 睡眠和唤醒的编程方法。

【实验设备】

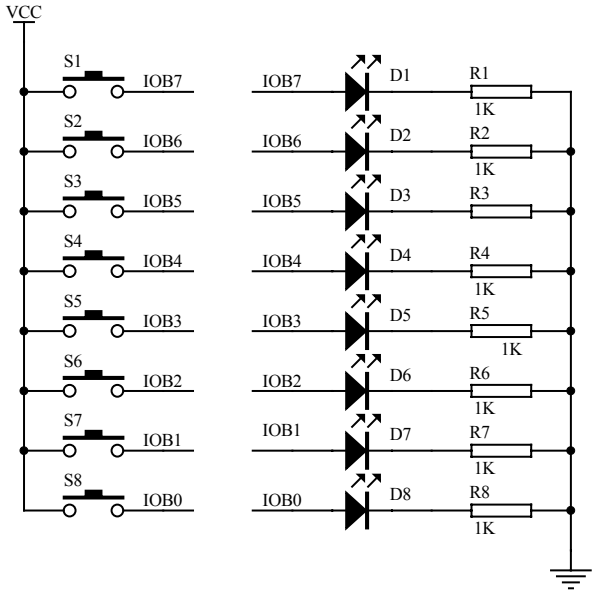
1)装有 μ'nSP™ IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

系统正常运行时点亮 8 个 LED，按 KEY1，系统接收到睡眠信号时，关闭系统时钟(PLL 振荡器)进入睡眠状态，8 个 LED 全部熄灭，系统睡眠指示灯（黄色 LED）被点亮 ；按任意键，收到唤醒信号后接通系统时钟(PLL 振荡器)，同时 CPU 会响应唤醒事件的处理并进行初始化。这时 8 个 LED 被循环熄灭。

【硬件连接图】



【实验步骤】

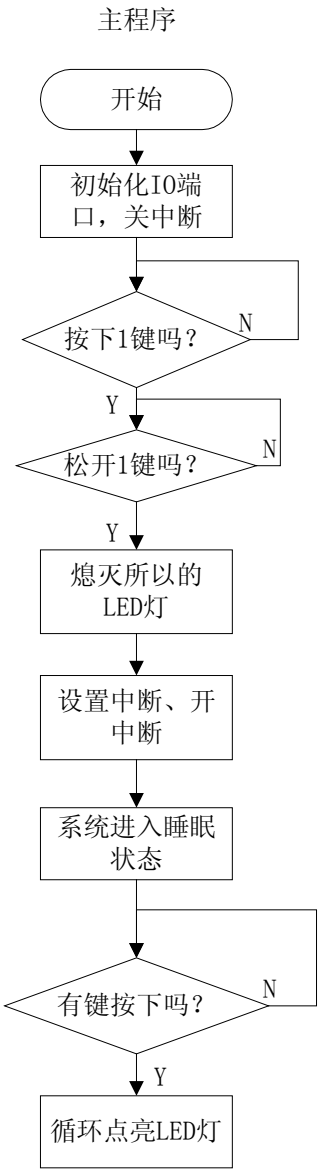
1)根据硬件连接图连接硬件并检查。

2)画流程图并编写程序代码。

3)编译程序，软件调试。

4)观察 LED、各个寄存器状态。

【程序流程图】



【程序范例】

```
//*************************************//
//文件名称：KeyWakeUp.asm
//描述：检测触键唤醒的功能
//A 口接 1*8 键盘
//B 口低八位接 8 个 LED
//按 1 键进入睡眠，熄灭 LED 。按任意键唤醒，循环熄灭 LED。
//无入口出口参数
//*************************************//
.include hardware.inc
.define P_IOA_RL      0x7004;

.code
.public _main;
```

```
_main:

    r1=0x0000;                          //初始化 A 口为带下拉电阻的输入
    [P_IOA_Dir]=r1;
    [P_IOA_Attrib]=r1;
    [P_IOA_Data]=r1;

    r1=0xffff;                          //初始化 B 口
    [P_IOB_Dir]=r1;
    [P_IOB_Attrib]=r1;
    r1=0x0000;                          //初始化为低电平输出
    [P_IOB_Data]=r1;
    INT OFF;                            //关中断


keydown:
    r1 = [P_IOA_Data];
    r2 = r1;
    r1 = 0x0001;
    cmp r2, r1;
    jne keydown;

key1:
keyup:
    r1 = [P_IOA_Data];
    r2 = r1;
    r1 = 0x0000;
    cmp r2, r1;
    jne keyup;
    r1 = 0xffff;
    [P_IOB_Data] = r1;                  //熄灭所有的 LED

    r1=0x0080;                          //中断初始化
    [P_INT_Ctrl]=r1;
    r1=[P_IOA_RL];
    INT IRQ;                            //开中断
    r1=0x0007;
    [P_SystemClock]=r1;                 //进入睡眠状态

    r1=0xfffe;                          //当有键唤醒时继续执行程序
    r4=0xffff;
    r4=r4 lsl 4;                        //清移位寄存器
    r2=0xffff;
L_Loopin:                               //循环熄灭 LED
```

```
        r2 -= 1;
        jnz L_Loopin;
        r1 = r1 rol 1;
        [P_IOB_Data] = r1;
        cmp r1,0xff7f;
        jne L_Loopin;
        r1 = 0xfffe;
        jmp L_Loopin;

//*********************************//

//键唤醒中断

//*********************************//
.text
.public _IRQ3
_IRQ3:
        push r1,r4 to [sp];

        r1=0x0080;
        test r1,[P_INT_Ctrl];                   //是否为键唤醒中断
        jz L_notKeyArouse;                      //否，退出中断程序
        r1=0x0080;
        [P_INT_Clear]=r1;                       //是，清中断
L_notKeyArouse:
        pop r1,r4 from [sp];
        reti;
.end
```

【程序练习】

初始化 A 口为带上拉电阻的输入口，练习触键唤醒程序。

初始化 A 口为悬浮输入口，练习触键唤醒程序。

※注意:只有 A 口的低八位具有触键唤醒功能。

```
//=============================================================//
// EX15 END
//=============================================================//
```

实验十六　UART 实验

【实验目的】

1）了解 SPCE061A 串行口（UART）的结构、与 PC 机串行通讯的原理。

2）了解 UART 的各配置单元 P_UART_BaudScalarLow (7024H)、P_UART_BaudScalarHigh (7025H)、 P_UART_Command1(7021H)和 P_UART_Command2 (7022H)的功能及控制方法。

3）掌握 PC 机与单片机通讯的编程方法。

【实验设备】

1) 装有 μ'nSP™ IDE 仿真环境的 PC 机一台。

2) μ'nSP™十六位单片机实验箱一个。

3）串口线一个。

【实验原理】

UART 模块提供了一个全双工标准接口，用于完成 SPCE061A 与外设之间的串行通讯。借助于 IOB 口的特殊功能和 UART IRQ 中断，可以同时完成 UART 接口的接收发送数据的过程。

硬件电路原理为：因为 SPCE061 在线调试器有 232 接口，PC 机端也有 232 接口，所以使用一个串口线连接起来就可以，或者使用 9 针头串口座焊接一个串口线。

【实验步骤】

1)根据内容自行设计硬件连接图，连接硬件并检查。

2)画流程图并编写程序代码。

3)编译程序，软件调试。

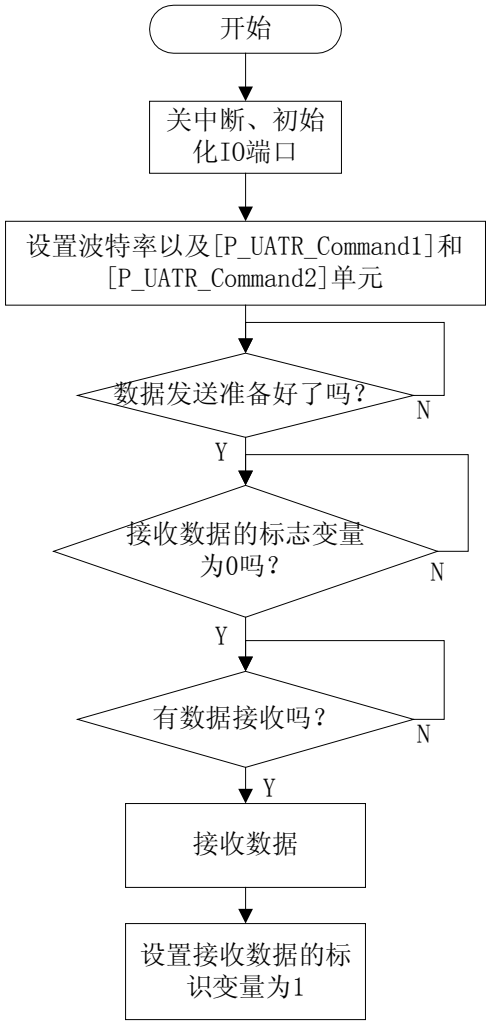
4)观察 LED、各个寄存器状态。

附：超级终端使用方法：

单击〖开始〗/〖程序〗/〖附件〗/〖通讯〗/〖超级终端〗显示超级终端图标。双击超级终端图标即可进入超级终端建立“新连接”界面。在文本框中添入新连接的名称。单击“确定”按钮。在连接时使用：后面选择“直接连接到串口 1”或 2 都可以。单击“确定”按钮。进入 COM1 属性设置界面，选择各参数。注意 SPCE061 无奇偶校验位。单击“确定”则进入主界面。保存新建的连接。在主窗口上可以键入数字键和字符键，而且这些数据都可以通过串口发送出去。在主窗口上也可以显示接收到的数据。

【程序流程图】

主程序流程图：



【程序范例】

```
//*****************************************************************************
//程序名称：UART_PC.asm
//描述：与 PC 机通讯，使用串口总线与 PC 机通讯。
//终端软件：超级终端，
//收到超级终端发送的数据并发送回去，在超级终端界面可以看到接收到的数据。
//*****************************************************************************
.include hardware.inc
.ram
      .var recFlag;                                    //接收数据标识，0 未接收数据 1 接收数据
.code
.public _main;
_main:
_UART_INIT:
F_UART_INIT:
      INT OFF;
```

```
    r1 = 0x0000;
    [recFlag] = r1;
    R1 = 0x0000;                        //未使能任何中断
    [P_INT_Ctrl] = R1;

    R1 = 0x0480;                        //设置 IOB7 为输入 IOB10 为输出
    [P_IOB_Attrib] = R1;
    R1 = 0x0400;
    [P_IOB_Dir] = R1;
    R1 = 0x0000;
    [P_IOB_Data] = R1;

    R1 = 0x006b;                        //设置波特率为 115.2Kbps
    [P_UART_BaudScalarLow] = R1;
    R1 = 0x0000;
    [P_UART_BaudScalarHigh] = R1;

    R1 = 0x0000;
    R4 = 0x00C0;                        //输入和输出使能设置
    [P_UART_Command1] = R1;
    [P_UART_Command2] = R4;

L_ResendData:
    L_Check_TxRDY:
    R2 = [P_UART_Command2];
    R2 &= 0x0040;                       //检测输出是否 READY
    JZ L_Check_RxRDY;

    //发送 8 位十六进制数 cc
    r1 = [recFlag];
    cmp r1,0x0000;                      //是否接收过数据
    jz   L_Check_RxRDY;

    [P_UART_Data] = R4;                 //发送数据
    r1 = 0x0000;
    [recFlag] = r1;

L_Check_RxRDY:
    R2 = [P_UART_Command2];             //检测是否有数据接收
    R2 &= 0x0080;
    JZ L_Check_RxRDY;
    R4 = [P_UART_Data];                 //接收数据
    r1 = 0x0001;
    [recFlag] = r1;                     //设置接收标识符
```

goto L_ResendData;

【程序练习】

自己编程实现双机通讯（用中断，要有奇偶校验等功能）

```
//=========================================================//
// EX16 END
//=========================================================//
```

实验十七　A/D 转换

【实验目的】

1)了解 ADC 输入接口的结构与转换原理

2)熟悉模拟量输入口 LINE_IN1—LINE_IN7 的使用

3)掌握 P_ADC、 P_ADC_CTRL 单元的设置方法

【实验设备】

1)装有 μ'nSP™ IDE 仿真环境的 PC 机一台

2)μ'nSP™十六位单片机实验箱一个

【实验原理】

ADC 工作方式分手动和自动两种当 ADC 工作在自动方式下：

1)外部信号由 LIN_IN[1~7]即 IOA[0~6]输入并直接被送入缓冲器 P_ADC_MUX_Data($702BH)；，在 ADC 自动方式被启用后，会产生出一个启动信号，即 RDY=0。此时，DAC0 的电压模拟量输出值与外部的电压模拟量输入值进行比较，以尽快找出外部电压模拟量的数字量输出值，A/D 转换的结果保存在 SAR 内。

2）当 10 位 A/D 转换完成时，RDY 会被置‘1’。此时，用户通过读取 P_ADC_MUX_Data($702BH) 单元可以获得 10 位 A/D 转换的数据。而从该单元读取数据后，又会使 RDY 自动清‘0’来重新开始进行 A/D 转换。

本实验通过改变 LINE_IN 端口的模拟电压来改变 IOB 口输出的数据，采用自动方式即定时器 A 溢出执行 ADC 转换，可以通过发光二极管的点亮了解转换的数据值。

【实验步骤】

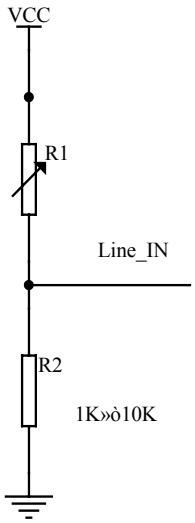
1）根据实验内容连接硬件电路并检查。

2）画流程图并编写程序代码。

3）编译程序，软件调试。

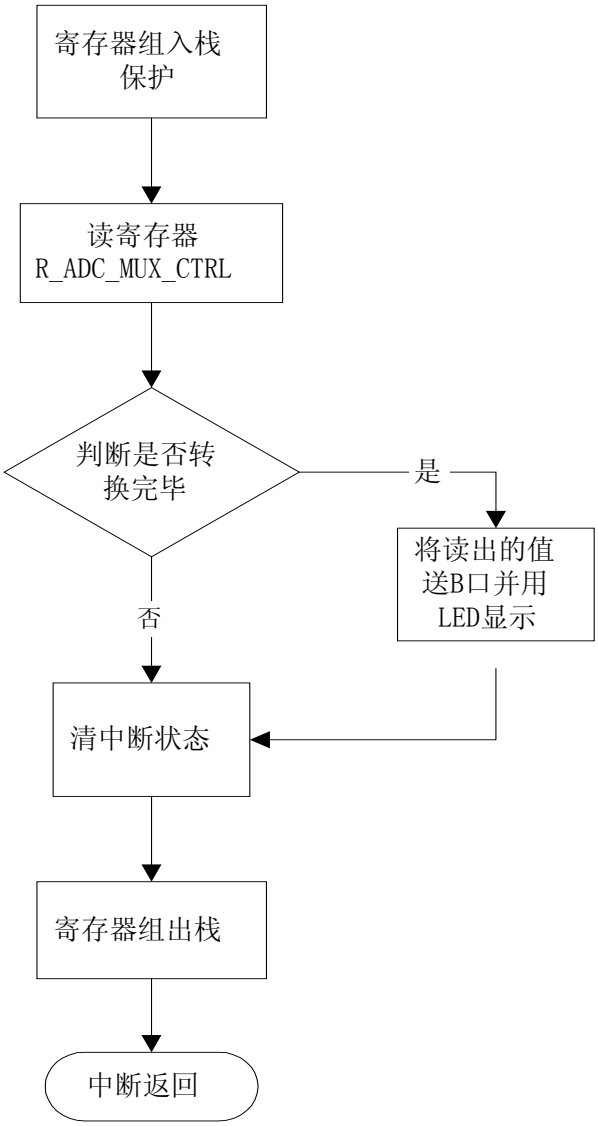
4) 通过改变输入电压，来观察发光二极管的亮灭。

【硬件连接图】



【程序流程图】

IRQ1 中断服务子程序框图：



【程序范例】

```
//*********************************************************************
//Note:通过模拟量输入口 LINE_IN 输入电压值，通过读取 P_ADC_MUX_Data
//单元可以获得 10 位 A/D 转换的数据。而从该单元读取数据后，又会使 RDY 自
//动清'0'来重新开始进行 A/D 转换。若未读取 P_ADC_MUX_Data 单元中的数据
//RDY 仍保持为'1'，则不会启动下一次的 A/D 转换。
//*********************************************************************
.define     P_IOB_DATA      0x7005
.define     P_IOB_DIR       0x7007
.define     P_IOB_ATTRI     0x7008
.define     P_INT_Ctrl      0x7010
.define     P_INT_CLEAR     0x7011
.define     P_ADC_Ctrl      0x7015
```

```
.define         P_ADC_MUX_Ctrl      0x702b
.define         P_ADC_MUX_DATA      0x702C
.define         P_DAC_Ctrl          0x702A

.CODE
.public _main
_main:

        r1=0xffff
        [P_IOB_ATTRI]=r1                    //IOB 口设置为同向输出口
        [P_IOB_DIR]=r1
        r1=0x0000
        [P_IOB_DATA]=r1;

        R1=0x0001                           //选择通道 LINE_IN 为 IOA0
        [P_ADC_MUX_Ctrl]=R1

        R1 = 0x0001                         //允许 A/D 转换
        [P_ADC_Ctrl] = R1

        nop
        nop
        nop
        nop
 _AD:

        r1=[P_ADC_MUX_Ctrl]                 //读寄存器[P_ADC_MUX_Ctrl]的 B15 位
                                            //判断是否转换完毕
        test r1,0x8000
        jz _AD                              //否，继续转换
        r1=[P_ADC_MUX_DATA]                 //是，则读出[P_ADC_MUX_DATA]转换结果
                                            //同时触发 A/D 重新转换
        [P_IOB_DATA]=r1;
        jmp _AD;
```

【程序练习】

练习 ADC 工作在手动方式下的转换。

```
//=============================================================================//
// EX17    END
//=============================================================================//
```

实验十八　双通道 D/A

【实验目的】

1)了解音频输出接口的结构与转换原理。

2)掌握 P_DAC2、P_DAC1、 P_DAC_CTRL 单元的设置方法。

【实验设备】

1)装有 u'nsp IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

3)示波器一台。

【实验原理】

SPCE061A 提供的音频输出方式为双 DAC 方式。在此方式下，DAC1、DAC2 转换输出的模拟量电流信号分别通过 AUD1 和 AUD2 管脚输出，输入的数字量分别写入 P_DAC1(写) ($7017)和 P_DAC2(写) ($7016)单元。

DAC 工作可以直接将 DAR 的数据锁存到 DAC 中，也可以采用定时中断的方法，即定时器溢出时响应中断，在中断中写输入数据到 P_DAC 中。

本实验采取直接方式，通过编程实现一个锯齿波，将实验板的两通道 DAC 输出分别接示波器 CHI、CH2 可以观察到锯齿波形，同时也可以听到 AUD1 和 ADU2 两端的扬声器有持续间断的声音。

【实验步骤】

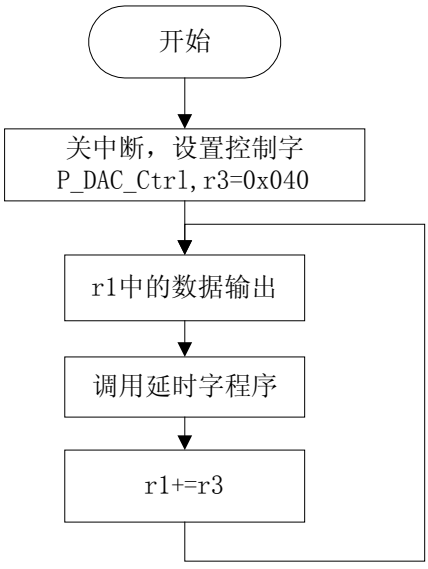
1）根据实验内容连接硬件电路并检查。

2）画流程图并编写程序代码。

3）编程实现 D/A 转换功能。

4) 编译程序，软件调试，通过 DAC1 和 DAC2 来播放语音，同时观察示波器输出波形。

【程序流程图】



【程序范例】

```
//*************************************************************
// Note:本实验采取直接方式，通过编程实现一个锯齿波，用示波器则可以观察到锯齿波
//形,同时也可以听到 AUD1 和 ADU2 两端的扬声器有持续间断的声音。
// Date: 2002/06/19
//*************************************************************
.define          P_DAC_Ctrl          0x702A
```

```
.define         P_DAC1          0x7017
.define         P_DAC2          0x7016

.public _main;
.code

_main:
    INT off;
    r1=0x0000;
    [P_DAC_Ctrl]=r1;
    r3=0x0040;                          //D/A 转换为 10 位，即 B15~B6
    r1=0x0000;

MainLoop1:
    [P_DAC1]=r1;
    [P_DAC2]=r1;
    call Delay                          //调用延时
    r1+=r3
    jmp MainLoop1

Delay:                                  //延时
    r2=0
DelayLoop:
    r2+=2048;
    jnz DelayLoop
    retf
```

【程序练习】

练习采用定时中断即 IRQ1_TMA 的方式进行转换。编程实现三角波、方波、梯形波等各种波形观察实验结果。

```
//=============================================================================//
// EX18    END
//=============================================================================//
```

实验十九　一路输入的录音

【实验目的】

1)了解 ADC 输入接口的结构、转换原理及实时录音并实时播放的功能。

2)熟悉麦克风输入口 MIC_IN 的使用。

3)进一步掌握 P_ADC、P_ADC_CTRL、P_DAC1、P_DAC2、P_DAC_Ctrl 各单元的设置方法。

【实验设备】

1)装有 u'nsp IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

实验 10 讲了 ADC 由通道 LIN_IN[1~7]即 IOA[0~6]输入信号时的工作方法，本实验也是 ADC 实验的一种，但外部信号由通道 MIC_IN 输入，再经过缓冲器和放大器。AGC 功能将通过 MIC_IN 通道输入的模拟信号的放大值控制在一定范围内，然后放大信号经采样-保持模块被送至比较器参与 A/D 转换值的确定，最后送入 P_ADC ($7014H)。

本实验通过编程实现录放音的功能，采用自动方式即定时器 A 溢出执行 ADC 转换，通过 A/D 将 MIC_IN 输入的语音信号转换为数字信号，再通过 D/A 的两个通道 AUD1 和 AUD2 播放。

【实验步骤】

1）根据实验内容连接硬件电路并检查。

2）编写程序代码。

3) 编译程序、软件调试、运行程序，实现实时录音并实时播放的功能 。

【程序范例】

```
// 时钟频率为 Fosc/2,采样率为 8kHz
.define     TIMER_DATA_FOR_8KHZ     0xfa00;
.define     P_TimerA_Ctrl           0x700b;
.define     P_TimerA_Data           0x700a;
.define     P_ADC                   0x7014;
.define     P_ADC_Ctrl              0x7015;
.define     P_DAC1                  0x7017;
.define     P_DAC2                  0x7016;
.define     P_DAC_Ctrl              0x702A;
.define     P_INT_Ctrl              0x7010;
.define     P_INT_Clear             0x7011;


//AD 初始化子程序
.CODE
_InitAD_DA:
    INT OFF;

    R1 = 0x0030;                    // 时钟频率为 CLKA 的 Fosc/2
    [P_TimerA_Ctrl] = R1;
    R1 = TIMER_DATA_FOR_8KHZ;       // 采样率为 8kHz
    [P_TimerA_Data] = R1;
    R1 = 0x003d;                    // 设置 AGC
```

```
    [P_ADC_Ctrl] = R1;
// 采用自动方式、且通过 MIC_IN 通道输入,

    R1 = 0x00A8;
//通过定时器 A 的溢出锁存数据，ADC 为自动方式
    [P_DAC_Ctrl] = R1;
    R1 = 0x1000;
    [P_INT_Ctrl] = R1;                  // 开中断 IRQ1_TM
    INT IRQ;
    retf;

//主程序
.PUBLIC _main;
_main:
    call _InitAD_DA;                    // 调用 AD 初始化子程序

loop:
    nop;
    nop;
    nop;
    nop;
    jmp loop;

.PUBLIC _IRQ1 ;
_IRQ1:
    push R1 to [SP];                    //R1 压栈保护

    R1 = [P_ADC];                       //读出 P_ADC 的值
    [P_DAC1] = R1;                      //将 ADC 采样的数据送给 P_DAC1 播放
    [P_DAC2] = R1;                      //将 ADC 采样的数据送给 P_DAC2 播放
    R1 = 0x1000 ;
    [P_INT_Clear] = R1;                 //清除 P_INT_Clear 单元的值
    pop R1 from [SP];
    reti;

//============================================================//
// EX19 END
//============================================================//
```

实验二十　　片内 2K SRAM 读写

【实验目的】

1)通过实验了解 SPCE061 2K SRAM 的读写原理；

2)熟悉 SPCE061 2K SRAM 的读写编程方法。

【实验设备】

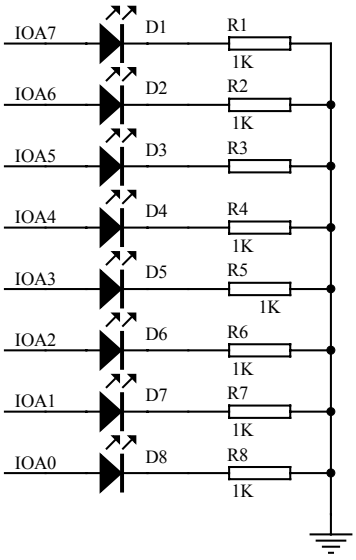
1)有 μ'nSP™ IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

向 2K 的 SRAM（0x0000---0x07ff）地址中写数据，然后再读出来比较是否正确，错误则点亮 LED。

【硬件连接图】



【实验步骤】

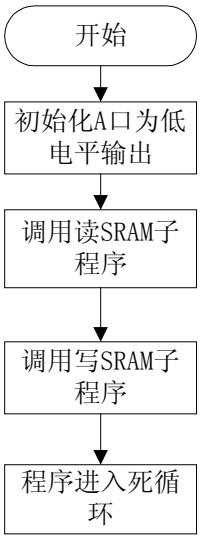
1)根据硬件连接图连接硬件并检查。

2)画流程图并编写程序代码。

3)编译程序，软件调试。

4)观察 LED、各个寄存器状态。

【程序流程图】



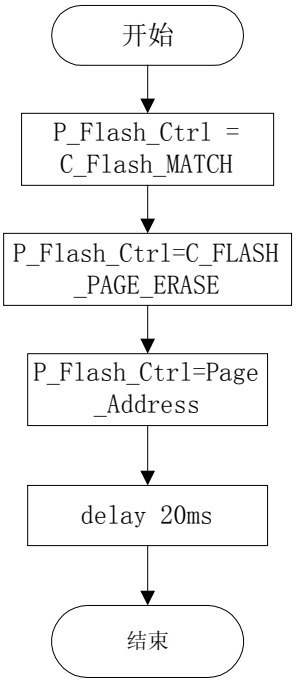
【程序范例】

```
//*****************************************
//程序名称：SRAM_2K_READ_WRITE.asm
//描述:A 口接 LED，当读写错误时，点亮 LED。
//*****************************************
.include hardware.inc
.define SRAM_SIZE    0x0800;
.code
.public    _main;
_main:

      r1=0xffff;                                    //初始化 A 口为同相高电平输出端
      [P_IOA_Dir]=r1;
      [P_IOA_Attrib]=r1;
      [P_IOA_Data]=r1;
      call sramWrite;
      call sramRead;
loop:
      jmp loop;

//*****************************//
//描述:向 0--0x7fa 中写数据 7fa--0
//*****************************//
.public    sramWrite ;
sramWrite: .proc
    r3 = 0x0000;
    r1 = sp;
    r2 = 0x0000;
loopWR:
    [r1] = r2;
    r2 += 1;
    r1 -= 1;
    jnz loopWR;
    retf;
  .endp;
//*****************************************
//描述:从 0--0x7fa 中读数据 0x7fa--0,失败点亮 LED
//*****************************************
sramRead:
    r1 += 1;
    r2 -= 1;
 F_sramRead:
    r3 = [r1];
    cmp r3,r2;
```

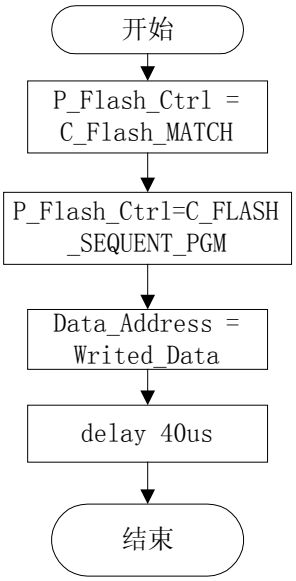
```
    jne Fail_LED;
    r2 -= 1;
    r1 += 1;
    cmp r1,sp;
    jne    F_sramRead;
    retf;

Fail_LED:                                   //点亮与 IOA8-IOA15 相连的 LED 灯
    r1 = 0x00ff;
    [P_IOA_Data] = r1;
     retf;

//=============================================================//
// EX20 END
//=============================================================//
```

实验二十一　32K Flash 读/写

32K 字的内嵌式闪存被划分为 128 个 PAGE(每个 PAGE 存储容量为 256 字)，
第一页 [0x8000—0x80ff]最后一页为[0xff00—0xffff]。它们在 CPU 空闲状态下均可通过编程被设置为只读或读/写工作方式。全部 32K 字闪存均可在 ICE 工作方式下被编程写入或被擦除。
Flash 的控制端的地址为 0x7555　（P_Flash_Ctrl　= 0x7555）
Flash 的匹配数据为 0xAAAA　（C_FlASH_MATCH　= 0xAAAA）
Flash 的页擦除控制字为 0x5511　(C_FLASH_PAGE_ERASE　= 0x5511)
Flash 的字写入控制字为 0x5533　（C_FLASH_1WORD_PGM　= 0x5533）
Flash 的顺序写入多字的控制字为 0x5544（C_FLASH_SEQUENT_PGM = 0x5544）Flash 页擦除的过程为：



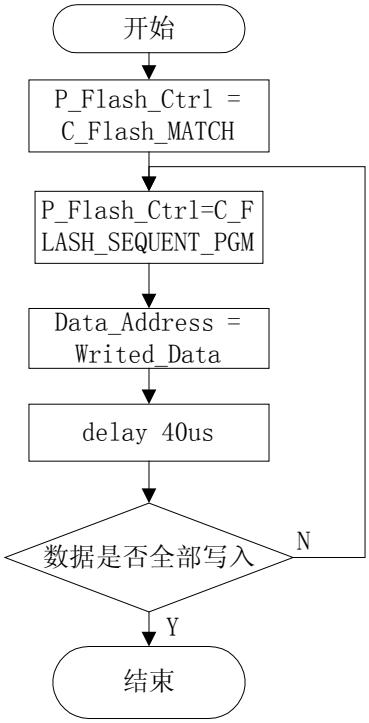
注：　Page_Address 为擦除该页中的任意地址。如:擦除第一页则页地址可为 80XX（X 可为任意数据）。
Flash 字写入的过程：



注：

Data_Address 为被写数据的存储地址（0x8000—0xffff），Writed_Data 为被写数据。

Flash 的顺序写入多字的过程：



【实验目的】

1)了解 SPCE061 的 32 K 闪存读写原理。

2)熟悉 SPCE061 的 32 K 闪存读写编程方法。

【实验设备】

1）装有 μ'nSP™ IDE 仿真环境的 PC 机一台。

2) μ'nSP™十六位单片机实验箱一个。

【实验原理】

在 CPU 空闲时，定义 FLASH 为可读写，擦除全部数据，检测是否擦除成功，失败点亮 LED1。写单字失败时点亮第二个 LED。

【硬件连接图】

与上一实验相同。

【实验步骤】

1)根据硬件连接图连接硬件并检查。

2)画流程图并编写程序代码。

3)编译程序，软件调试。

4)观察 LED、各个寄存器状态。

【程序范例】

```
//////////////////////////////////////////////////////////////////////////////
//程序名称：FLASH_Read_Write_Erase_main.c
//描述:  A 口低 8 位接 8 个 LED，当读写擦除全部成功时，点亮所有的 LED
//擦除失败时点亮第一个 LED
//写单字失败时点亮第二个 LED
//写页失败时点亮第三个 LED
```

```
//=====================================================================
#include "hardware.h"
//#define C_BLANKPAGE          0x8400;
#define C_FAIL_LED4            0x00f7;
#define C_FAIL_LED3            0x00fb;
#define C_FAIL_LED2            0x00fd;
#define C_FAIL_LED             0x00fe;
#define C_SUCCESS_LED          0x0000;
int main()
{
    int * sector, * addr;
    int i;
    const int num[]={
        1,  2,  3,  4,  5,  6,  7,  8,  9,  10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32,
        33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
        49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64,
        65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
        81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,
        97, 98, 99, 100,101,102,103,104,105,106,107,108,109,110,111,112,
        113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128};

    SP_Init_IOA(0xffff,0xffff,0xffff);          //设置 A 口为输出端
    sector = 0xa000;
    F_FlashErase(sector);                       //擦除 256 字节/128 字
    for (i=0;i<0x100;i++)                       //擦除成功与否
    {
        if (*(sector+i)!=0xffff)
        {
            SP_Export(Port_IOA_Buffer, C_FAIL_LED);
            //否，点亮第一个 LED
            return(1);
        }
    }
    addr = 0xA000;
    F_FlashWrite1Word(addr,0x5555);             //在 A000 地址处写 0x5555
    if (* addr!=0x5555)                         //写成功否
    {
        SP_Export(Port_IOA_Buffer, C_FAIL_LED2);
        //否，点亮第二个 LED
        return(1);
    }
    addr = 0xA001;
    F_FlashWrite1Word(addr,0xAAAA);             //在 A001 地址处写 0xaaaa
```

```
    if (* addr!=0xAAAA) //写成功否
    {
        SP_Export(Port_IOA_Buffer, C_FAIL_LED3);
        //否，点亮第三  个 LED
        return(1);
    }
    F_FlashErase(sector);
    F_FlashWrite(sector,&num, 128);              //写 128 个数据到指定的地址中
    for (i=0;i<128;i++)                          //写成功否
    {
        if (*(sector+i) != num[i])
        {
            SP_Export(Port_IOA_Buffer, C_FAIL_LED4);
            //否，点亮第四个 LED
            return(1);
        }
    }
    SP_Export(Port_IOA_Buffer, C_SUCCESS_LED);
        //成功，点亮所有的 LED
    return(0);
}
```

实验二十二　低电压检测实验

低电压监测功能可以提供系统内电源电压的使用情况。

【实验目的】

1）了解 SPCE061 低电压检测原理及功能。

2）熟悉 SPCE061 低电压检测的编程方法。

【实验设备】

1）装有 μ'nSP™ IDE 仿真环境的 PC 机一台。

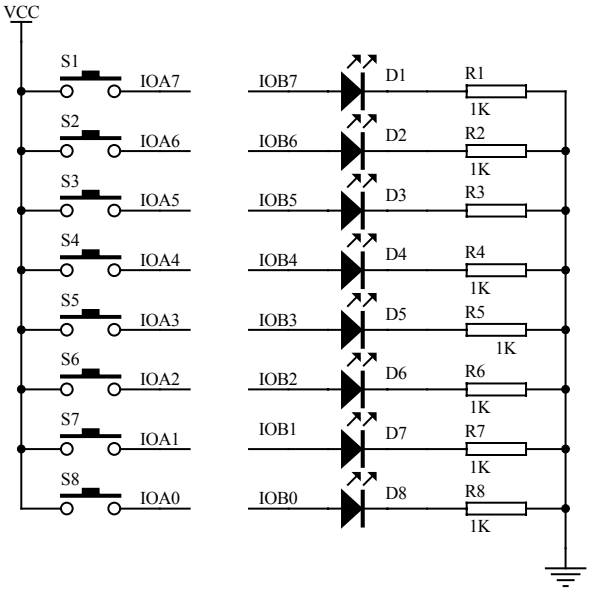
2) μ'nSP™十六位单片机实验箱一个。

【实验原理】

通过可调电压源改变电压，SPCE061A 具有 4 级电压监测低限：2.4V、2.8V、3.2 和 3.6V，可通过对 P_LVD_Ctrl 单元编程进行控制。用按键选择 LVD 电压基准，按下 key1 为选择 2.4V，当电源电压低于 LVD 电压时，LED 被点亮。按下 key2 为选择 2.8V，当电源电压低于 LVD 电压时，LED 被点亮。按下 key3 为选择 3.2V，当电源电压低于 LVD 电压时，LED 被点亮。按下 key4 为选择 3.6V。当电源电压低于 LVD 电压时，LED 被点亮。

注意：应使用 3.3V 电源端。

【硬件连接图】



【实验步骤】

1) 根据硬件连接图连接硬件并检查。

2) 画流程图并编写程序代码。

3) 编译程序，软件调试。

4) 结合硬件调试。

5) 按键 1 调电源电压到低于 2.4V。

6) 观察 LED、各个寄存器状态。

7) 按键 2 调电源电压到低于 2.8V。

8) 观察 LED、各个寄存器状态。

9) 按键 3 调电源电压到低于 3.2V。

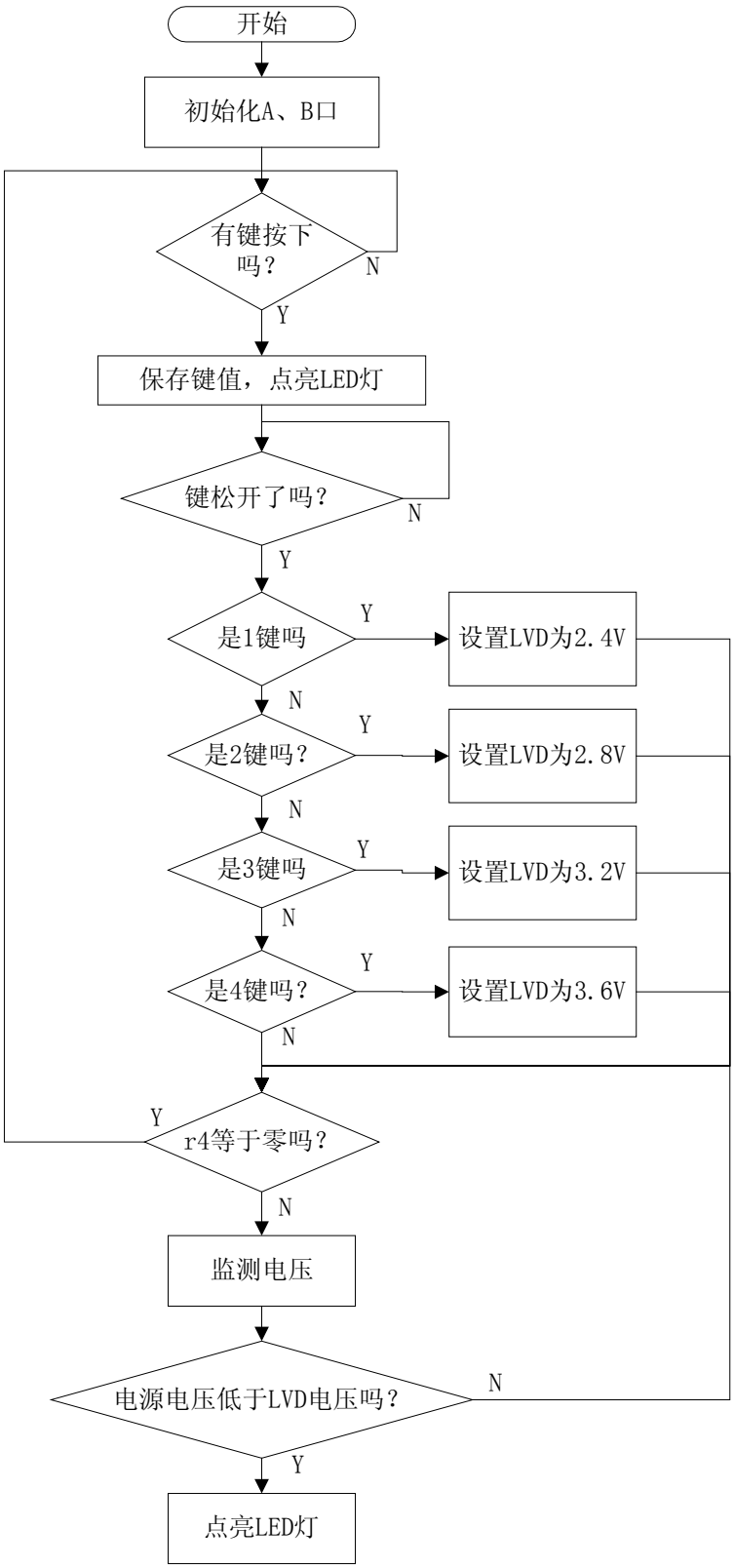
10)观察 LED、各个寄存器状态。

11)按键 4 调电源电压到低于 3.6V。

12)观察 LED、各个寄存器状态。

【程序流程图】

主程序流程图：



【程序范例】

```
//*****************************************//
//文件名称：LVD.asm
//描述：A 口低 8 位接键盘，B 口低 8 位接 LED
//按 1 键      低电压为 2.4v
//按 2 键      低电压为 2.8v
//按 3 键      低电压为 3.2v
//按 4 键      低电压为 3.6v
//*****************************************//
.include hardware.inc
.public _main；
.code
_main:
      r1=0x0000 ;                          //初始化 A 口为带下拉电阻的输入
      [P_IOA_Dir]=r1;
      [P_IOA_Attrib]=r1;
      [P_IOA_Data]=r1;

      r1=0xffff ;                           //设 B 口为无数据反相功能的低电平输出
      [P_IOB_Dir]=r1;
      [P_IOB_Attrib]=r1;
      r1=0x0000;
      [P_IOB_Data]=r1;

keydown:                                   //扫描键盘
      r1 = [P_IOA_Data];
      r1 &= 0x000f;                        //屏蔽高 12 位
      cmp    r1,0x0000;
      jz keydown;
      r2 = r1;                             //保存键值

keyup:
      r1=0x0000;
      [P_IOB_Data]=r1;                     //点亮 led

      r1 = [P_IOA_Data];                   //判断按下松开与否
      cmp r1,0x0000;
      jnz keyup;

key1:                                      //是否为一键
      r1 = 0x0001;
      cmp r2,r1;
      jne key2;
```

```
    r3 = 0x0000;
    [P_LVD_Ctrl] = r3;                  //设置 LVD 为  2.4v

key2:                                   //是否为二键
    r1 = 0x0002;
    cmp r2,r1;
    jne key3;
    r3 = 0x0001;
    [P_LVD_Ctrl] = r3 ;                 //设置 LVD  为 2.8v

key3:                                   //是否为三键
    r1 = 0x0004;
    cmp r2,r1;
    jne key4;
    r3 =    0x0002;
    [P_LVD_Ctrl] = r3;                  //设置 LVD 为 3.2v

key4:                                   //是否为四键
     r1 = 0x0008;
     cmp r2,r1;
     jne check_lvd;
     r3 = 0x0003;
     [P_LVD_Ctrl] = r3;                 //设置为 LVD 为 3.6v

     r4 = 0x0000;
check_lvd:

    r4 -= 1;                            //限时
    cmp r4, 0x0000;
    je keydown;

    r1 = [P_LVD_Ctrl];                  //监测电压
    r3 = 0x8000;
    r1 &= r3;
    cmp r1,0x0000;
    jz check_lvd;

    [P_IOB_Data] = r2;                  //熄灭 LED

    jmp keydown;     //************************************************************//
```

实验二十三　实验 LVR 实验

对于 SPCE061 芯片具有多种复位方式，如上电复位、外部复位、低电压复位。这里只着重介绍低电压复位。

低电压复位原理：当电源电压低于 2.2V 时，系统会变得不稳定且易出故障。系统设置了低电压复位(LVR)功能后，当电源电压低于该值时，会在 4 个时钟周期之后产生一个复位信号，使系统复位。

【实验目的】

1)了解 SPCE061 的低电压复位方式及原理 。

2)熟悉 SPCE061 的低电压复位方式的编程方法。

【实验设备】

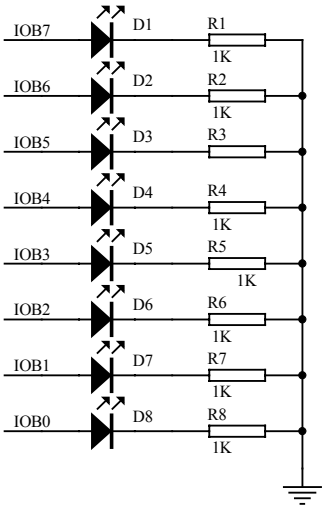
1)装有 μ'nSP™ IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

在低电压复位实验中，在电源端处接可变稳压源，调节可变电压源使电压低于 2.2V，当系统连续复位时， IOB 口的 LED 闪烁。

【硬件连接图】



【实验步骤】

1)根据硬件连接图连接好硬件。

2)将 μ 'nSP™ IDE 打开后，建立一个新工程。

3)在该项目的源文件夹(SOURCE FILES)下建立一个新的汇编语言文件。

4)编写程序代码。

5)编译程序，软件调试。

6)调可变电压源到 2.2V 以下，观察 LED 是否闪烁及各个寄存器状态。

【程序范例】

```
//*************************************//
//程序名称：LVR.asm
//描述：复位一次。LED 闪烁。
//B 口接 8 个 LED。
//无参数
//*************************************//
```

```
.include hardware.inc

.code
.public _main;
_main:
                                        //设 B 口为无数据反相功能的高电平输出
    r1=0xffff ;
    [P_IOB_Dir]=r1;
    [P_IOB_Attrib]=r1;
    r1 = 0xffff;
    [P_IOB_Data]=r1;
    call delay;                         //延时

    r1 = 0x0000 ;                       //点亮 LED
    [P_IOB_Data] = r1;
    call delay ;                        //延时

loop:
    jmp loop;
//*********************************//
//延时程序
//无入口出口参数
//*********************************//
delay:
    r2 = 0x02;
loop_out:
    r1 = 0xff00;
loop_in:
    r1 -=1；
    cmp r1,0x0000；
    jnz loop_in;
    r2 -= 1;
    cmp r2,0x0000；
    jnz loop_out;
   retf;
////////////////////////////////////////////////////////////////////
```

第二章　语音实验部分

常见的几种音频压缩算法

1)不同音频质量等级的编码技术标准 （频响）：
电话质量：200HZ－3.4KHZ
AM 质量：50HZ－7KHZ
FM 质量：20HZ－15KHZ
CD 质量：10HZ－20KHZ
凌阳音频压缩方法一般指的是电话质量标准即频率在 200HZ－3.4KHZ.

2)数据压缩分类：
压缩分无损压缩和有损压缩,无损压缩一般指： 磁盘文件，压缩比低：2：1－4：1,而有损压缩则是指： 音／视频文件，压缩比可高达：100：1
凌阳音频压缩算法根据不同的压缩比分为以下几种:
SACM-A2000：压缩比为 8： 1 ，8：1.25，8：1.5
SACM-S480： 压缩比为：80：3，80：4.5
SACM-S240： 压缩比为：80：1.5
按音质排序：A2000>S480>S240

凌阳音频压缩编码

1)波形编码：**sub-band** 即 SACM-A2000
特点：高质量、高码率，适于高保真语音／音乐。

2)参数编码：声码器（vocoder）模型表达，抽取参数＆激励信号进行编码。线性预测编码(**LPC)**即 SACM-S240
特点：压缩比大，计算量大，音质不高，廉价！

3)混合编码：**CELP** 即 SACM-S480
特点：综合参数和波形编码之优点。
除此之外,还具有 FM 音乐合成方式即 SACM-MS01

音频压缩技术之趋势

1)降低资料率，提高压缩比，用于廉价、低保真场合（如：电话）。

2)追求高保真度，复杂的压缩技术（如：CD）

语音压缩方法

语音压缩分 DOS 和 WINDOWS 两种：

(一)DOS 下的压缩：

A2000：

1) 用 PC 机录制一个 wav 语音文件
2) 用 cool edit pro 软件转换成 8k16 位的文件
3) 用 A2000 压缩生成 16k(或 20k,24k)压缩率的文件
4) 在 ms-dos 下：

e:\>sacm2000.exe 16 *.wav *.out *.16k

或 (e:\>sacm2000.exe 20 *.wav *.out *.20k

e:\>sacm2000.exe 24 *.wav *.out *.24k)

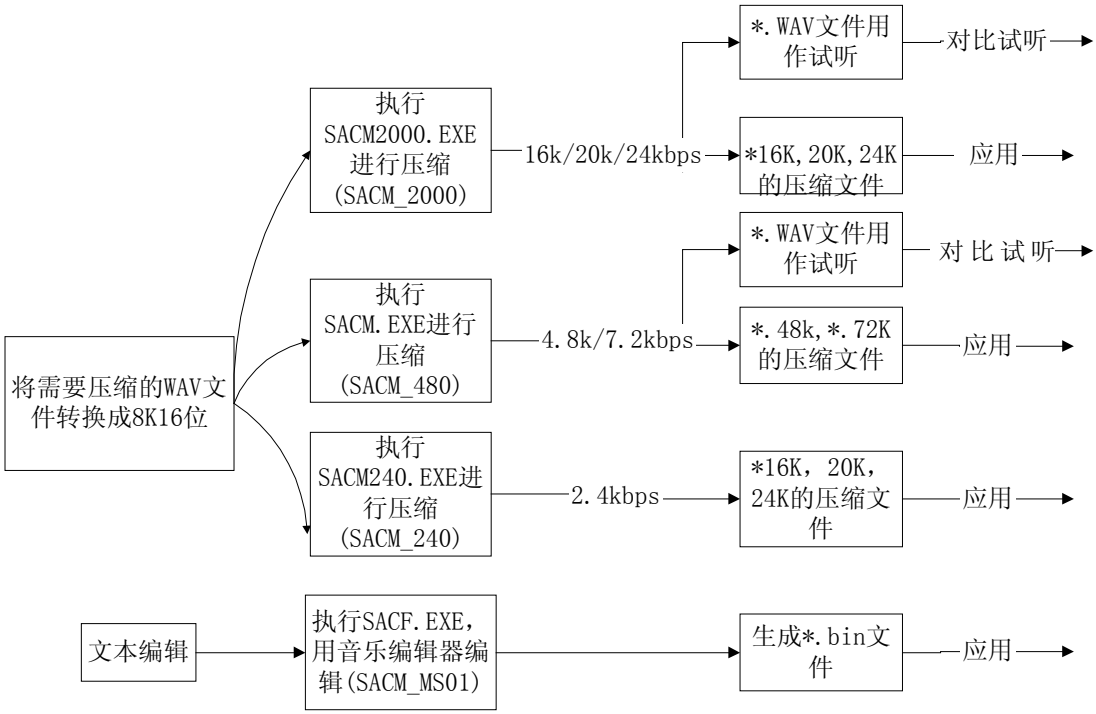
S480：

1) 用 PC 机的录音机录制语音文件生成 8K16 位的 WAV 文件
2) 用 s480 压缩生成 4.8k（或 7.2k）压缩率的文件
3) 在 ms-dos 下：

e:\>sacm.exe *.wav *.48k *.out –s48

或（e:\>sacm.exe *.wav *.72k *.out –s72）

下图是凌阳音频压缩编码 (SACM)方法的流程：



(二)WINDOWS 下的压缩：

如下图所示，可以选择一个或多个 wav 文件进行压缩，具体步骤可根据提示来操作


北阳音频压缩
欢迎使用北阳音频压缩软件：
本工具可提供SACM—A2000 和SACM—S480 两种方法对WAV文件进行压缩.
WAV文件经SACM－A2000算法压缩后变为SUB-BAND的音频格式，由于它的压缩比较小(8：1)所以具有高质量、高码率的特点适用于高保真语音和音乐.
SACM—S480的压缩比较大（80：3），存储容量大,音质介于A2000和S240之间,适用于语音播放：如“文曲星”词库等。
选择转换方式:
压缩多个文件
压缩单个文件
Sunnorth
下一步(N) >
取消

实验一　SACM-A2000

【实验目的】

1)了解凌阳单片机以 SACM_A2000 语音格式播放及程序的编写方法。

2)了解凌阳音频编码算法库(SACM_Lib)。

3)了解 SACM_A2000 的语音文件。

【实验设备】

1)装有 u'nsp IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

将 PCM 音频格式的 8K16 位 WAV 文件经 SACM－A2000 算法压缩后变为 SUB-BAND 的音频格式，压缩前为：*.wav ,8k/16bit；压缩后为：*.16k/*.20k/*.24k,并生成*.out 文件用于对比试听(pc-play)，即在 PC 机上试听*.out 文件并和实验板上的扬声器的声音进行对比。由于它的压缩比较小(8：1)所以具有高质量、高码率的特点适用于高保真语音和音乐。

【实验步骤】

1）用 PC 机录制一个 wav 语音文件。

2）用 Windows 压缩工具将该 wav 文件压缩为 16k(或 20k,24k)的文件。

3）将 μ'nSP™ IDE 打开后,建立一个新工程。

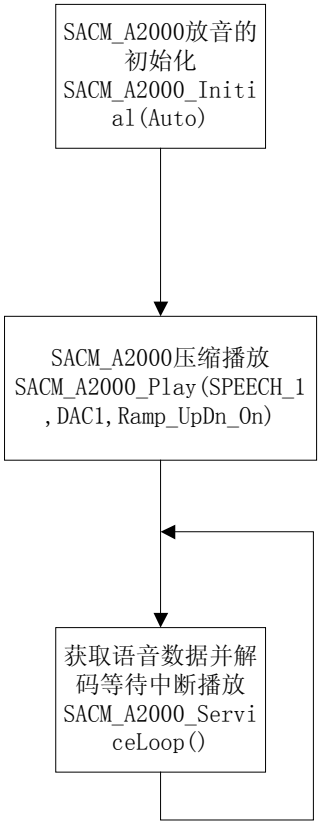
4）添加 SACM_A2000 的语音文件到程序中的资源文件夹（resource）下。

5）添加、编写程序代码。

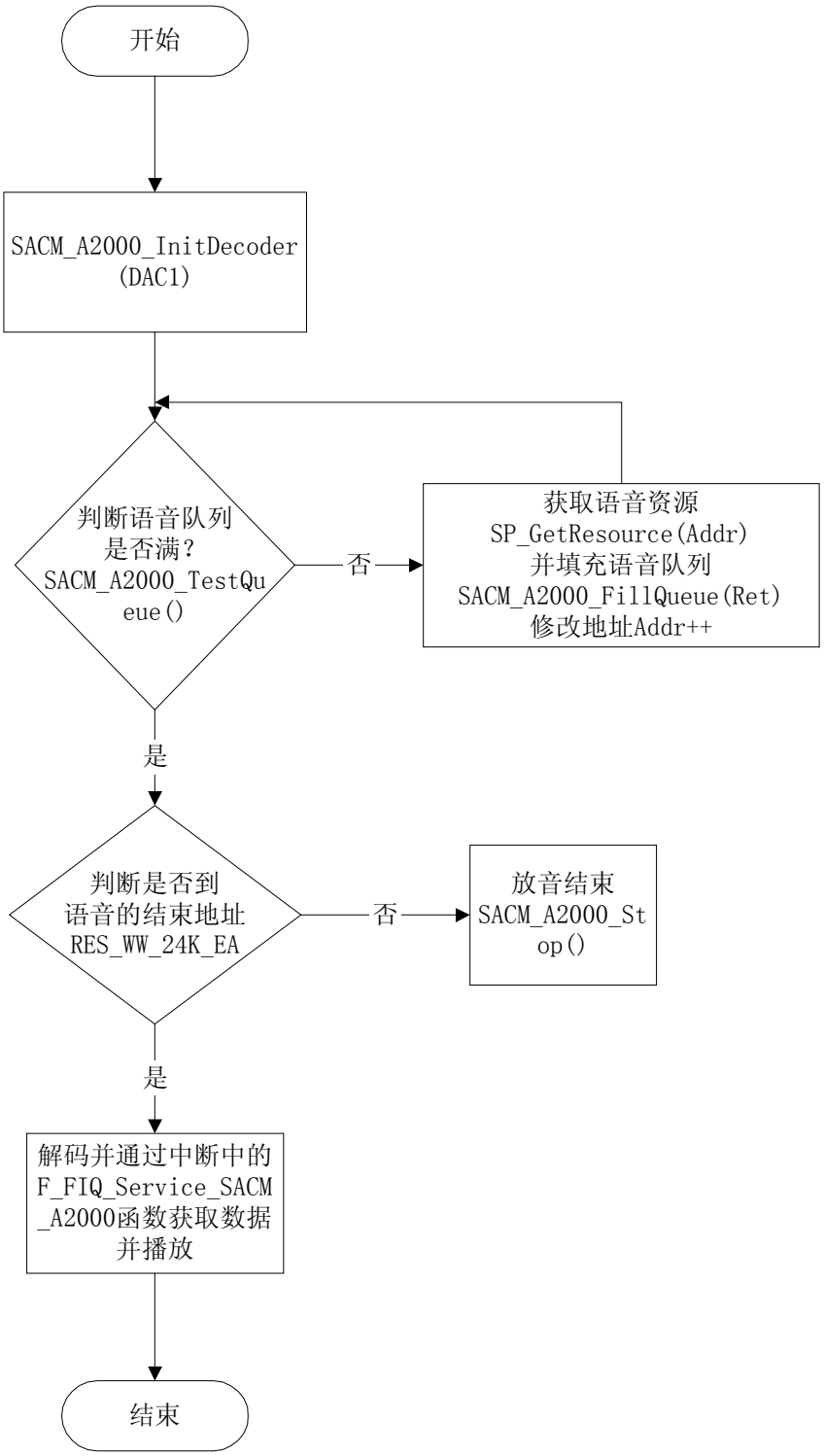
6）编译程序，观察结果。

【程序流程图】

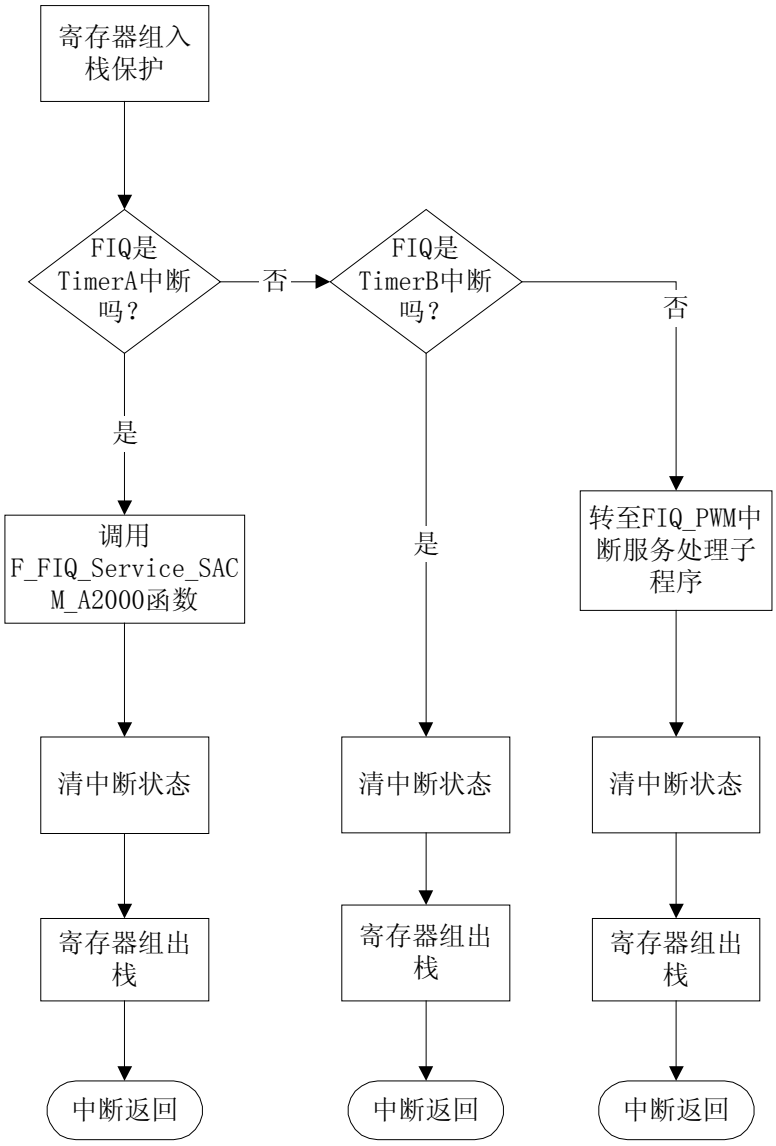
主程序流程图：



A2000 手动方式主程序流程图：



中断服务子程序：



【程序范例】

```
//*************************************************************
// Note: A2000 有两种播放方式，即自动方式和非自动方式，我们可以通
//过改变 Mode 的值来切换这两种方式：当 Mode=0 时以非自动方式播放，
//当 Mode=1 时，以自动方式播放。
//*************************************************************


#define     SPEECH_1        0
#define     DAC1            1
```

```
#define         DAC2                2
#define         Ramp_UpDn_Off       0
#define         Ramp_Up_On          1
#define         Ramp_Dn_On          2
#define         Ramp_UpDn_On        3
#define         Manual              0
#define         Auto                1
#define         Full                1
#define         Empty               2
#define         Mode                1

#include  "A2000.h"
main()
{
   extern long      RES_HH_24K_SA,RES_HH_24K_EA;
        //定义语音资源的首末地址标号
   long    int Addr;
        //定义地址变量
        int       Ret = 0;
        //定义获取语音数据变量并初始化
        if(Mode == 1)
        //采用自动方式播放
        {
                SACM_A2000_Initial(1);
                //自动方式播放初始化
                SACM_A2000_Play(SPEECH_1,DAC1,Ramp_UpDn_On);
                //定义语音索引号、播放通道、允许音量增/减调节
                while(1)
                SACM_A2000_ServiceLoop();
                //获取语音数据并将其填入解码队列
        }
        if(Mode == 0)                                   //采用非自动方式播放
        {
     Addr=RES_HH_24K_SA;                        //送入语音队列的首址

     SACM_A2000_Initial(0);                     //非自动方式播放的初始化

     SACM_A2000_InitDecoder(DAC1);
     //开始对 A2000 的语音数据以非自动方式解码
     while(1)
     {
          if(SACM_A2000_TestQueue()!=Full)
          {
                Ret =SP_GetResource(Addr);
```

```
                SACM_A2000_FillQueue(Ret);
                Addr++;
            }

          if(Addr< RES_HH_24K_EA )              //如果该段语音未播完，即未到达末地址时
            SACM_A2000_Decoder();
                  //获取资源并进行解码，再通过中断服务子程序送入 DAC 通道播放
          else
            SACM_A2000_Stop();                  //否则，停止播放
        }
      }
}
//中断程序（ISR.ASM）
.text
.include   hardware.inc
.include   A2000.inc
.include   Resource.inc


.public    _FIQ;
_FIQ:
      PUSH     r1,r4 to [sp];                   //压栈保护
      r1=0x2000;
      test r1,[P_INT_Ctrl];                    //判断是不是 FIQ_TMA 中断源
      jnz L_FIQ_TimerA;                        //是，则转
      r1=0x0800;
      test r1,[P_INT_Ctrl];                    //否则，判断是不是 FIQ_TMB 中断源
      jnz L_FIQ_TimerB;                        //是，则转
L_FIQ_PWM:
      r1=C_FIQ_PWM;
      [P_INT_Clear]=r1;                        //清除 P_INT_Clear 单元
      POP R1,R4 from[sp];
      reti;
L_FIQ_TimerA:
      [P_INT_Clear]=r1;
      call F_FIQ_Service_SACM_A2000;           //调用函数，完成播放
      pop r1,r4 from [sp];
      reti;
L_FIQ_TimerB:
      [P_INT_Clear]=r1;
      pop r1,r4 from [sp];
      reti;

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

实验二　SACM-480

【实验目的】

1)了解语音压缩和播放功能并学会编程

2)了解凌阳音频编码算法库(SACM_Lib)。

3)了解 SACM_s480 的语音文件。

【实验设备】

1)装有 u'nsp IDE 仿真环境的 PC 机一台

2)μ'nSP™十六位单片机实验箱一个

【实验原理】

将 PCM 音频格式的 8K16 位 WAV 文件经 SACM－S480 算法压缩后变为 CELP 的音频格式，压缩前为：*.wav ,8k/16bit；压缩后为：*.48k/*.72k fo 并生成*.out 文件用于对比试听(pc-play),即在 PC 机上试听*.out 文件并和实验板上的扬声器的声音进行对比。压缩比较大 80：3, 存储容量大,音质介于 A2000 和 S240 之间,适用于语音播放：如*"文曲星"*词库

【实验步骤】

1) 用 PC 机录制一个 wav 语音文件。

2) 用所给的 Windows 压缩工具将该 wav 文件压缩生成 4.8k（或 7.2k）的文件。

3）将 μ'nSP™ IDE 打开后,建立一个新工程。

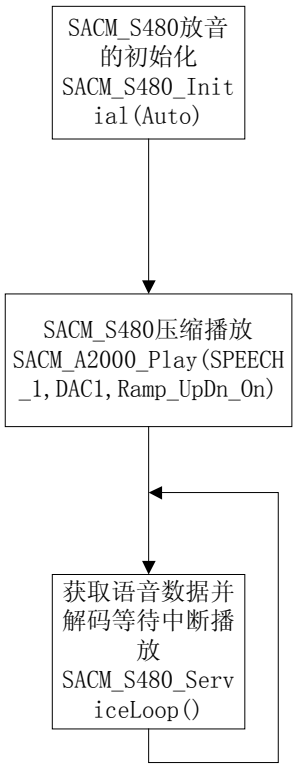
4) 添加 SACM_S480 的语音文件到程序中的资源文件夹（resource）下。

5）编写程序代码。

6）编译程序，观察结果。

【程序流程图】

主程序流程图：



【程序范例】

主程序（MAIN.ASM）

```
//*************************************************************
// Note: s480 只有自动播放方式,在中断 FIQ 的 FIQ_TMA 中断源中通过
//主程序的 SACM_S480_ServiceLoop()对语音数据进行解码，然后将其
//送入 DAC 通道播放
//*************************************************************
#define         SPEECH_1          2
#define         DAC1              1
#define         DAC2              2
#define         Ramp_UpDn_Off     0
#define         Ramp_Up_On        1
#define         Ramp_Dn_On        2
#define         Ramp_UpDn_On      3
#define         Auto              1
#define         Full              1
#define         Empty             2
#include  "s480.h"
main()
{
      SACM_S480_Initial(Auto);
      //自动方式播放初始化
      SACM_S480_Play(SPEECH_1,DAC1+DAC2,Ramp_UpDn_On);
      //定义语音索引号、播放通，允许音量增/减调节
      while(1)
      SACM_S480_ServiceLoop();
      //获取语音数据并将其填入解码队列
}
//中断程序（ISR.ASM）
.text
.include   hardware.inc
.include   S480.inc
.include   Resource.inc


.public     _FIQ;
_FIQ:
      PUSH     r1,r4 to [sp];                    //压栈保护
      r1=0x2000;
      test r1,[P_INT_Ctrl];                      //判断是不是 FIQ_TMA 中断源
      jnz L_FIQ_TimerA;                          //是，则转
      r1=0x0800;
      test r1,[P_INT_Ctrl];                      //否则，判断是不是 FIQ_TMB 中断源
      jnz L_FIQ_TimerB;                          //是，则转
```

```
L_FIQ_PWM:
        r1=C_FIQ_PWM;
        [P_INT_Clear]=r1;                       //清除 P_INT_Clear 单元
        POP R1,R4 from[sp];
        reti;
L_FIQ_TimerA:
        [P_INT_Clear]=r1;
        call F_FIQ_Service_SACM_S480;           //调用函数，完成播放
        pop r1,r4 from [sp];
        reti;
L_FIQ_TimerB:
        [P_INT_Clear]=r1;
        pop r1,r4 from [sp];
        reti;

}
```

实验三　SACM-240

【实验目的】

1)了解凌阳单片机以 SACM_S240 语音格式播放及程序的编写方法。

2)了解凌阳音频编码算法库(SACM_Lib)。

3)了解 SACM_S240 的语音文件。

【实验设备】

1)装有 u'nsp IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

将 PCM 音频格式的 8K16 位 WAV 文件经 SACM－S240 算法压缩后变为线性预测编码格式即 LPC，压缩前为：*.wav ,8k/16bit；压缩后为：*.24k，压缩比大 80：1.5,价格低,适用于对保真度要求不高的场合, 如：玩具类产品的批量生产，码率仅为 2.4 Kbps.

【实验步骤】

1)将 μ'nSP™ IDE 打开后,建立一个新工程。

2)在该项目的源文件夹(SOURCE FILES)下建立一个新的 C 语言文件。

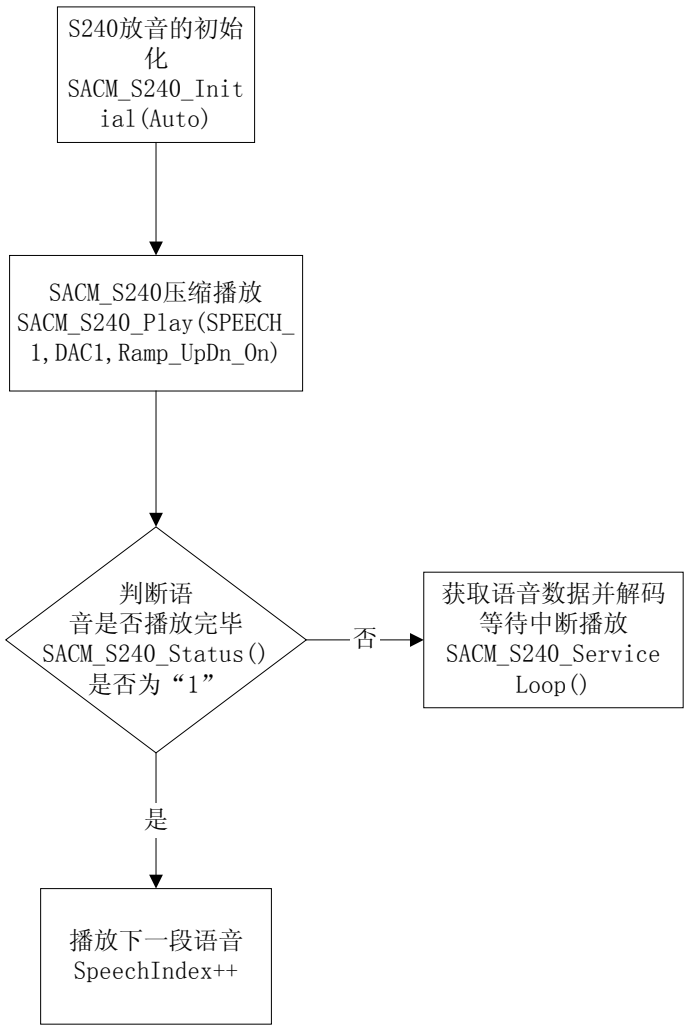
3)编写程序代码。

4)在资源文件夹下添加 SACM_S240 的语音文件。

5)编译程序，软件调试，观察结果。

【程序流程图】

主程序流程图：



【程序范例】

```
//主程序(MAIN.C):
//*********************************************************
// Note: s240 只有自动播放方式,在中断 FIQ 的 FIQ_TMA 中断源中通过
//主程序的 SACM_S240_ServiceLoop()对语音数据进行解码，然后将其
//送入 DAC 通道播放。
//*********************************************************
#include "hardware.h"
#include "s240.h"


#define     DAC1            1
#define     DAC2            2
#define     Ramp_UpDn_Off   0
#define     Ramp_Up_On      1
#define     Ramp_Dn_On      2
#define     Ramp_UpDn_On    3

#define MaxSpeechNum3                       //播放语音的最大个数

#define   Auto              1

main()
{

    int SpeechIndex = 0;                    // 选择第一首语音
    while(1)
    {
        SACM_S240_Initial(Auto);
            SACM_S240_Play(SpeechIndex,DAC1+DAC2, Ramp_UpDn_On);
            // 播放第一首
            while(SACM_S240_Status()&0x01)      //判断第一首是否播完
            SACM_S240_ServiceLoop();

            SpeechIndex++;
            SACM_S240_Play(SpeechIndex,DAC1+DAC2, Ramp_UpDn_On);
            //播放第二首
            while(SACM_S240_Status()&0x01)      //判断第二首是否播完
            SACM_S240_ServiceLoop();
            SpeechIndex++;
            SACM_S240_Play(SpeechIndex,DAC1+DAC2, Ramp_UpDn_On);
            //播放第三首
```

```
            while(1)
            SACM_S240_ServiceLoop();            //停
        }

}
//中断程序（ISR.ASM）
.TEXT
  .include hardware.inc
.include   S240.inc
.include   Resource.inc

.external  F_FIQ_Service_SACM_S240;

.public    _FIQ;
_FIQ:
    PUSH    r1,r4 to [sp];                  //压栈保护
    r1=0x2000;
    test r1,[P_INT_Ctrl];                   //判断是不是 FIQ_TMA 中断源
    jnz L_FIQ_TimerA;                       //是，则转
    r1=0x0800;
    test r1,[P_INT_Ctrl];                   //否则，判断是不是 FIQ_TMB 中断源
    jnz L_FIQ_TimerB;                       //是，则转
L_FIQ_PWM:
    r1=C_FIQ_PWM;
    [P_INT_Clear]=r1;                       //清除 P_INT_Clear 单元
    POP R1,R4 from[sp];
    reti;
L_FIQ_TimerA:
    [P_INT_Clear]=r1;
    call F_FIQ_Service_SACM_S480;           //调用函数，完成播放
    pop r1,r4 from [sp];
    reti;
L_FIQ_TimerB:
    [P_INT_Clear]=r1;
    pop r1,r4 from [sp];
    reti;
```

实验四　SACM_MS01 实验

【实验目的】

1)了解凌阳单片机以 SACM_A2000 格式播放语音的编程方法。

2)了解凌阳音频编码算法库(SACM_Lib)。

3)了解 SACM_A2000 的语音文件。

【实验设备】

1)装有 u'nsp IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

SACM-MS01 是 FM 语音及音乐合成方法。首先编曲获得乐谱即*.pop 文件,再通过 scfmV32.EXE 程序生成目标文件*.bin 、阅读文件*.asm、和中间文件*.ok1，其中*.bin 文件就是我们所需要的资源文件。以下是获得音乐编曲乐谱（"配器"总谱*.POP）文件的三种途径：

1）用 TXET EDIT 按 SPCE 编曲格式人工输入多信道器乐总谱，生成*.POP 文件（计算机音乐业内称为：KEY IN）。

特点：较繁琐，但只要具备音乐理论＆配器法＆和声学知识了解 SPCE 编曲格式者均可尝试。

2）遵照 SPCE 编曲格式 用 DTM＆MIDI（音源＋MIDI 键盘＋作曲软件）的方法，演奏自动生成*.mid 文件，再用凌阳 MIDI2POP.EXE 转成*.pop 文件。

特点：需要专业设备&软件，具备键盘乐演艺技能，了解 SPCE 编曲格式。

3）从网上下载 MIDI 文件或商业化 MIDI 作品，将其修改成凌阳格式并转成凌阳的*.pop 文件。

【实验步骤】

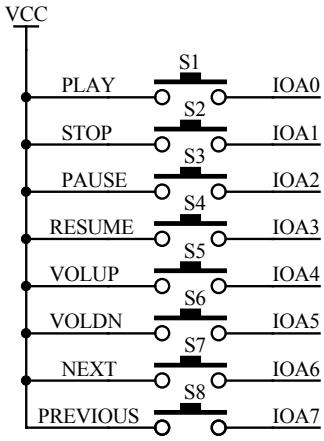
1)将 μ'nSP™ IDE 打开后,建立一个新工程。

2)编写程序代码。

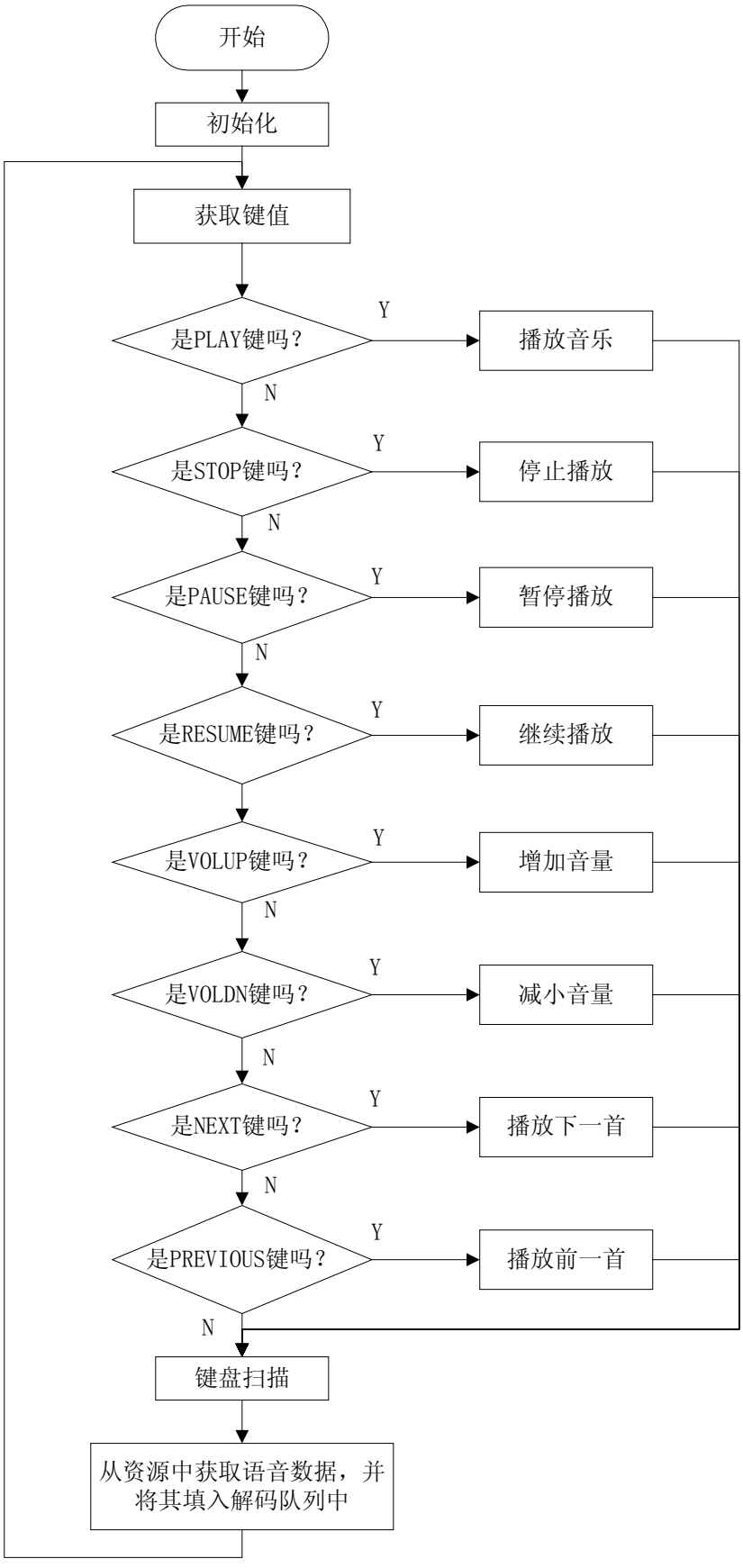
3)在资源文件夹下添加 SACM_MS01 的语音文件。

4)编译程序，软件调试，观察结果。

【硬件连接图】



【程序流程图】


开始
初始化
获取键值
是PLAY键吗？
Y
播放音乐
N
是STOP键吗？
Y
停止播放
N
是PAUSE键吗？
Y
暂停播放
N
是RESUME键吗？
Y
继续播放
是VOLUP键吗？
Y
增加音量
N
是VOLDN键吗？
Y
减小音量
N
是NEXT键吗？
Y
播放下一首
N
是PREVIOUS键吗？
Y
播放前一首
N
键盘扫描
从资源中获取语音数据，并
将其填入解码队列中

【程序范例】

```
/*****************************************************************************/
主程序(MAIN.C)
/*****************************************************************************/
#include "hardware.h"
#include "ms01.h"

#define Disable                 0
#define Enable                  1

#define MaxSongNum              11              // 最大语音资源文件个数
#define MaxVolume               15              //最大音量

#define Inst_Max_Num            35              // 乐器混合数
#define Drum_Max_Num            20              // 鼓点数

#define DAC_24K                 1
#define DAC_20K                 2
#define DAC_16K                 3

#define Manual                  0
#define    Auto                 1

int         Ret = 0;                            // 定义子程序返回值
int main()
{
            int Key = 0;                        // 定义存储器键值
            int SongIndex = 0;                  // 第一首曲子的初始化
            int VolumeIndex = 8;                //初始化中音
            int    OnIndex=0,OffIndex=0;
            int Channel=0, Instrument=0 ,Drum=0;
            // 2: 20K Current(TimerA) 3: 16K Current(TimerA)
            Ret = System_Initial();
            SACM_MS01_Initial(DAC_24K);
            SACM_MS01_Play(SongIndex,DAC1+DAC2, Ramp_UpDn_On);
            //播放音乐
            //SACM_MS01_ChannelOff(0);
            //SACM_MS01_ChannelOff(1);
            //SACM_MS01_ChannelOff(2);
            //SACM_MS01_ChannelOff(3);
            //SACM_MS01_ChannelOff(4);
            //SACM_MS01_ChannelOff(5);
                                    while(1)
            {    // 用 C 语言定义主函数范围
```

```
Key = SP_GetCh();
switch(Key)
{
case 0x00:
        break;
case 0x01:
        // 播放音乐
        SACM_MS01_Play(SongIndex,DAC1+DAC2, Ramp_UpDn_On);
        break;
case 0x02:
        // 停止播放
        SACM_MS01_Stop();
        break;
case 0x04:
        SACM_MS01_Pause();     // 暂停
        break;
case 0x08:
        SACM_MS01_Resume(); // 继续播放
            break;
    case 0x10:
                                    // 音量增加
            VolumeIndex++;
            if(VolumeIndex > MaxVolume)
                VolumeIndex = MaxVolume;
            SACM_MS01_Volume(VolumeIndex);
            break;
    case 0x20:
                                    // 音量减小
            if(VolumeIndex == 0)
                VolumeIndex = 0;
            else
                VolumeIndex--;
            SACM_MS01_Volume(VolumeIndex);
            break;
    case 0x40:
                                    //播放下一首
        if( ++SongIndex == MaxSongNum)
            SongIndex = 0;
        SACM_MS01_Play(SongIndex, DAC1+DAC2, Ramp_UpDn_On);
        break;

case 0x80:                          //播放前一首
        SACM_MS01_ChannelOff(0);
        SACM_MS01_ChannelOn(1);
```

```
                        SACM_MS01_ChannelOn(2);
                        SACM_MS01_ChannelOff(3);
                        SACM_MS01_ChannelOff(4);
                        SACM_MS01_ChannelOff(5);
                        SACM_MS01_SetInstrument(2,Instrument,0);
                        //f(channel, instrument,song command on/off)
                        if(++Instrument > Inst_Max_Num) Instrument=0;
                            SACM_MS01_SetInstrument(1,Drum,1);
                            //f(channel, instrument,song command on/off)
                        if(++Drum > Drum_Max_Num) Drum=0;
                        //if( --SongIndex < 0)
                        //SongIndex = MaxSongNum-1;
                        //SACM_MS01_Play(SongIndex,DAC1+DAC2);
                        break;
                default:
                        break;
                }
                Ret = System_ServiceLoop();             // 键扫描服务子程序
                Ret = SACM_MS01_ServiceLoop()    //     SACM-MS01  播放子程序
            }
            return 0;
}
```

系统文件(system.asm):

```
////////////////////////////////////////////////////////////////////
// Function: System commander
// Service for H/W, keyboard scan
// Input: None
// Output: None
// Functions:
// (In Assembly view)
//call F_System_Initial;
// (In C language view)
//System_Initial();
////////////////////////////////////////////////////////////////////
.include resource.inc;
.include hardware.inc;
.include key.inc;

.external F_SACM_MS01_SetInstrument;
.external F_SACM_MS01_ChannelOn;
.external F_SACM_MS01_ChannelOff;

.public R_SpeechType;
```

```
.RAM
.VAR R_SpeechType;

.CODE
.public _System_Initial;
.public    F_System_Initial;
_System_Initial: .PROC
F_System_Initial:
            //push      BP,BP to [SP];
            //BP = SP + 1;
            //r1 = [BP+3];
            //r2 = [BP+4];
            call  F_Key_Scan_Initial;                  //键盘扫描
            call  F_User_Init_IO;                      //  定义使用 IO
            // Add other general initialization here
            // test area
            jmp L_TestEnd;
            r1 = 0;
            r2 = 1;
            call  F_SACM_MS01_SetInstrument;
            r1 = 1
            r2 = 2;
            call  F_SACM_MS01_SetInstrument;
            r1 = 2;
            r2 = 3;
            call  F_SACM_MS01_SetInstrument;
            r1 = 3;
            r2 = 4;
            call  F_SACM_MS01_SetInstrument;
            r1 = 4;
            r2 = 5;
            call  F_SACM_MS01_SetInstrument;
            r1 = 5;
            r2 = 6;
            call  F_SACM_MS01_SetInstrument;

            r1 = 0;
            call  F_SACM_MS01_ChannelOn;
            r1 = 1;
            call  F_SACM_MS01_ChannelOn;

            r1 = 0;
            call  F_SACM_MS01_ChannelOff;
            r1 = 1;
```

```
        call  F_SACM_MS01_ChannelOff;
L_TestEnd:
        //r1 =0x0001;                           // 返回值
        //pop           BP,BP from [SP];
        retf;
        .ENDP;
//*************************************************************
// Function: Main Loop of system
// Input: None
// Output: None
// Using:
//     call F_System_ServiceLoop; (in assembly domain)
//   System_ServiceLoop(); (in C domain)
//*************************************************************
.public _System_ServiceLoop;
.public    F_System_ServiceLoop;
_System_ServiceLoop: .PROC
F_System_ServiceLoop:
        call  F_Key_DebounceCnt_Down;       // 调用键盘去抖子程序
        call      F_Key_Scan_ServiceLoop;       //调用键盘扫描子程序
        // Add other general service functions here
        R1=0x0001;                            // 清零看门狗
        [P_Watchdog_Clear]=R1;                //
        retf;
        .ENDP;

F_User_Init_IO:
        r1 = 0x0000;                          // 定义 IOA 口为输入
        [P_IOA_Dir] = r1;
        [P_IOA_Attrib] = r1;
        [P_IOA_Data] = r1;

        r1 = 0x0000;                          // 定义 IOB 口为输入
        [P_IOB_Dir] = r1;
        [P_IOB_Attrib] = r1;
        [P_IOB_Data] = r1;
        retf;
```

实验五　SACM_A2000 与 S480/S720 混合实验

【实验目的】

1)了解凌阳单片机以 SACM_A2000 与 S480/S720 混合语音格式播放及程序的编写方法。

2)了解凌阳音频编码算法库(SACM_Lib)。

3)了解 SACM_A2000 与 S480/S720 的语音文件。

【实验设备】

1)装有 u'nsp IDE 仿真环境的 PC 机一台

2)μ'nSP™十六位单片机实验箱一个

【实验原理】

SACM_A2000 格式和 S480 格式前面实验已经分别作了较详细的介绍，本实验是将这两种方式结合起来，即通过按键来选择播放的类型：SACM_A2000 或 SACM_480，这样可以针对不同的语音和音乐采用不同的方式来播放。

【实验步骤】

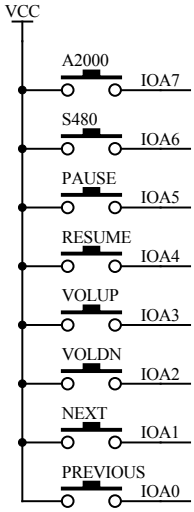
1)将 μ'nSP™ IDE 打开后,建立一个新工程。

2)编写程序代码。

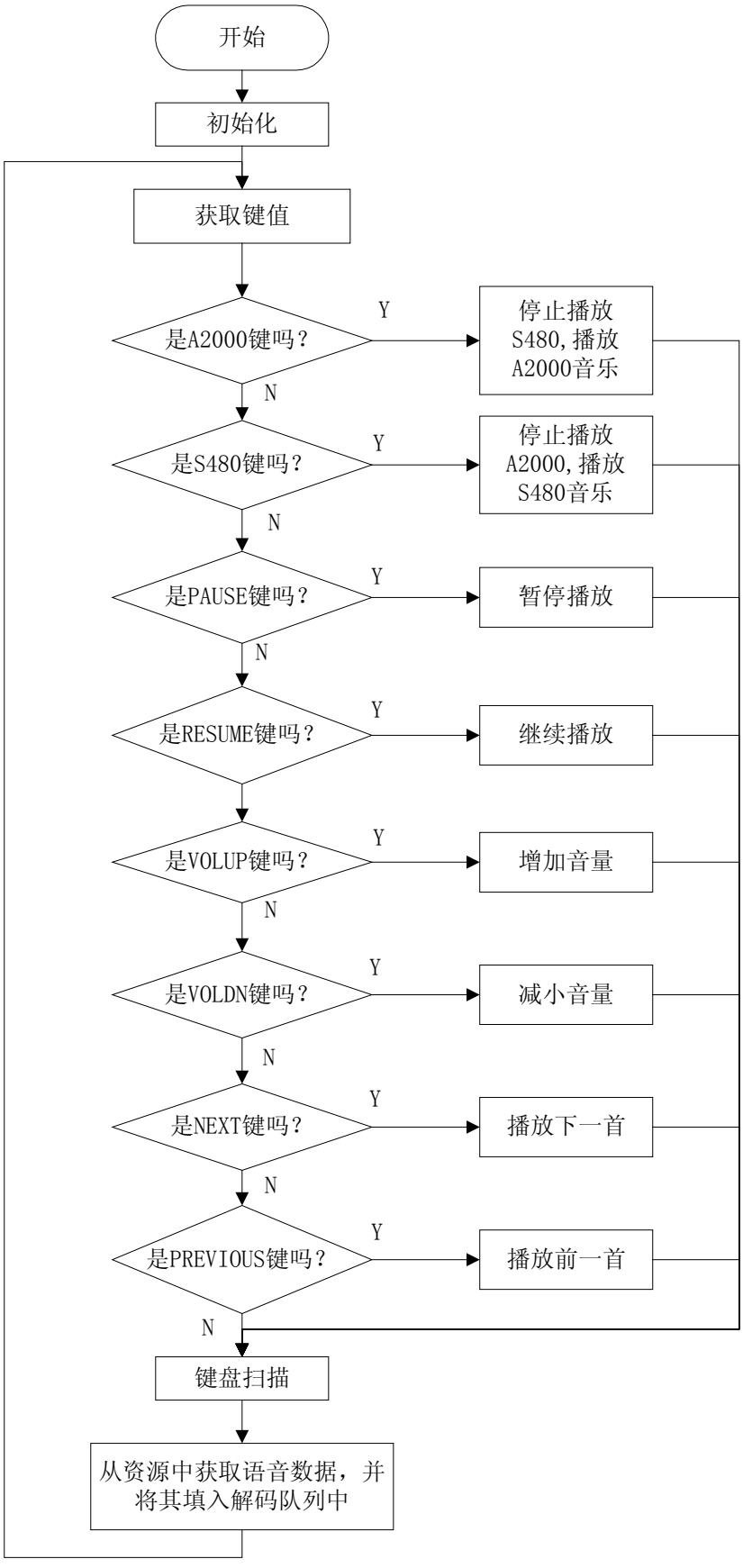
3)在资源文件夹下添加 SACM_A2000/S480/S720 的语音文件。

4)编译程序，软件调试，观察结果。

【硬件连接图】



【程序流程图】


开始
初始化
获取键值
是A2000键吗？
Y
停止播放
S480,播放
A2000音乐
N
是S480键吗？
Y
停止播放
A2000,播放
S480音乐
N
是PAUSE键吗？
Y
暂停播放
N
是RESUME键吗？
Y
继续播放
是VOLUP键吗？
Y
增加音量
N
是VOLDN键吗？
Y
减小音量
N
是NEXT键吗？
Y
播放下一首
N
是PREVIOUS键吗？
Y
播放前一首
N
键盘扫描
从资源中获取语音数据，并
将其填入解码队列中

【程序范例】

```
/*******************************************************************************
主程序(MAIN.C):
/*******************************************************************************
      Program: SACM-A2000, SACM-S480,S720 的播放实验

      使用函数的调用:
      System_Initial();用于 Hardware、Keyboard scan 的初始化，见  system.asm
      System_ServiceLoop();

      Sunplus Function call:
      int SP_GetCh();
      Return values of SP_GetCh():
{0x00,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80}
 *******************************************************************************/
#include "hardware.h"
#define         Disable             0
#define         Enable              1
      #define         A2000               1                   // SACM-A2000
#define         S480_720            2                 // SACM-S480/S720
#define         Max_Volume          15                //  最大音量
#define         Min_Volume          0                 //  最小音量
#define         Max_Num_A2000       3                 //  定义 A2000 播放的最大语音数
#define         Max_Num_S480        2                 //  定义 S480 播放的最大语音数
#define         Manual              0
#define         Auto                1
unsigned int    Ret = 0;                              //  定义子程序返回值
unsigned int    SpeechType = 1;                       //  设置 A2000
unsigned int    Key = 0;                              //  定义存储器键值
int main()
{
      unsigned int Index_A2000 = 0;
      unsigned int Index_S480 = 0;

      int Volume_A2000 = 8;
      int Volume_S480 = 8;
      int    State = 0;
      SpeechType = 0;
      System_Initial();
      while(1)
      {                                             //  主程序由 C 来实现
            Key = SP_GetCh();
            switch(Key)
           {
```

```
case 0x00:
        break;
case 0x01:
        SACM_S480_Stop();
        SpeechType = SACM_A2000_Initial(Auto);
        Set_SpeechType(SpeechType);
        //设置语音的类型
    SACM_A2000_Play(Index_A2000,DAC1+DAC2,Ramp_UpDn_On);〕
        //A2000 播放
        Index_A2000++;
        if(Index_A2000 == Max_Num_A2000)
            Index_A2000 = 0;
        break;
case 0x02:
        SACM_A2000_Stop();
        //A2000 播放停止
        SpeechType = SACM_S480_Initial(Auto);
        Set_SpeechType(SpeechType);
        //设置语音的类型
        SACM_S480_Play(Index_S480,DAC1+DAC2,Ramp_UpDn_On);
        Index_S480++;
        if(Index_S480 == Max_Num_S480)
        Index_S480 = 0;
        break;
case 0x04:
        switch(SpeechType)
        {
            case 1:
                SACM_A2000_Pause();
                break;
            case 2:
                SACM_S480_Pause();
                break;
        }
        break;

case 0x08:
        switch(SpeechType)
        {
            case 1:
                SACM_A2000_Resume();
                break;
            case 2:
                SACM_S480_Resume();
```

```
                    break;
                }

                break;

        case 0x10:
                switch(SpeechType)
                {   //判断播放类型，是 A2000 的还是 S480 的
                    case A2000:         //A2000 放音
                        Volume_A2000++;
                        if(Volume_A2000 > Max_Volume)
                            Volume_A2000 = Max_Volume;
                        SACM_A2000_Volume(Volume_A2000);
                        break;
                    case S480_720:      //S480 放音
                        Volume_S480++;
                        if(Volume_S480 > Max_Volume)
                            Volume_S480 = Max_Volume;
                        SACM_S480_Volume(Volume_S480);
                        break;
                }
                break;
        case 0x20:
                switch(SpeechType)
                {
                    case A2000:         //A2000 类型的播放
                        if(Volume_A2000 > Min_Volume)
                            Volume_A2000--;
                        SACM_A2000_Volume(Volume_A2000);
                        break;
                    case S480_720:      //S480 类型的播放
                        if(Volume_S480 > Min_Volume)
                            Volume_S480--;
                        SACM_S480_Volume(Volume_S480);
                        break;
                }
                break;
        case 0x40:                      // 播放下一首
        case 0x80:                      // 播放前一首
        default:
                break;
}
Ret = System_ServiceLoop();             //调用键扫描服务循环函数
switch(SpeechType)
```

```
        {
            case A2000:
                SACM_A2000_ServiceLoop(); // A2000 的自动播放与停止
                break;
            case S480_720:
                SACM_S480_ServiceLoop();  // S480 的自动播放与停止
                break;
        }

    }
    return 0;
}
```

系统文件(system.asm):

```
/*****************************************************************************
// Function: System commander
// Service for H/W, keyboard scan
// Input: None
// Output: None
// Functions:
//     (In Assembly view)
//     call F_System_Initial;
//     (In C language view)
//     System_Initial();
/*****************************************************************************
.include resource.inc;
.include hardware.inc;
.include key.inc;

.public    R_SpeechType;
.RAM
.var       R_SpeechType;
.external  R_Flag_A2000;
.external  R_Flag_S480;
.CODE
.public _Set_SpeechType;
.public F_Set_SpeechType;
F_Set_SpeechType:
_Set_SpeechType:  .PROC
            push BP,BP to [SP];
            BP = SP + 1;
            r1 = [BP+3];
            [R_SpeechType] = r1;
            pop        BP,BP from [SP];
```

```
        retf;
        .ENDP

.public _System_Initial;
.public    F_System_Initial;
_System_Initial: .PROC
F_System_Initial:
        push BP,BP to [SP];                    // 寄存器入栈保护
        call  F_Key_Scan_Initial;              // 键盘扫描
        // Add other general initialization here
        //BP = SP + 1;
        //r1 = [BP+3];

        call  F_User_Init_IO;

        //r1 =0x0001;                          //返回值

        pop        BP,BP from [SP];
        retf;
        .ENDP;
//*************************************************************
// Function: Main Loop of system
// Input: None
// Output: None
// Using:
//     call F_System_ServiceLoop; (in assembly domain)
//   System_ServiceLoop(); (in C domain)
//*************************************************************
.public _System_ServiceLoop;
.public    F_System_ServiceLoop;
_System_ServiceLoop: .PROC
F_System_ServiceLoop:
        call  F_Key_DebounceCnt_Down;        // 去抖
        call       F_Key_Scan_ServiceLoop;      //调用键盘扫描子程序
        //call       F_Key_Scan_ServiceLoop_2; // 调用键扫描子程序 2
        R1=0x0001;                           // 清除看门狗
        [P_Watchdog_Clear]=R1;
        retf;
        .ENDP;
```

实验六　SACM-DVR

【实验目的】

1)通过实验了解凌阳单片机对语音的录制,压缩和播放的功能及过程。

2)学会编程并会扩展 SRAM。

【实验设备】

1)装有 u'nsp IDE 仿真环境的 PC 机一台。

2)μ'nSP™十六位单片机实验箱一个。

【实验原理】

SACM-DVR 具有录音和放音功能，并采用 A2000 的算法，录音时采用 16K 数据率及 8K 采样率获取语音资源，经过 A2000 压缩后存储在扩展的 SRAM 628128A 里，录音 32 秒后完毕，并自动开始放音。

【实验步骤】

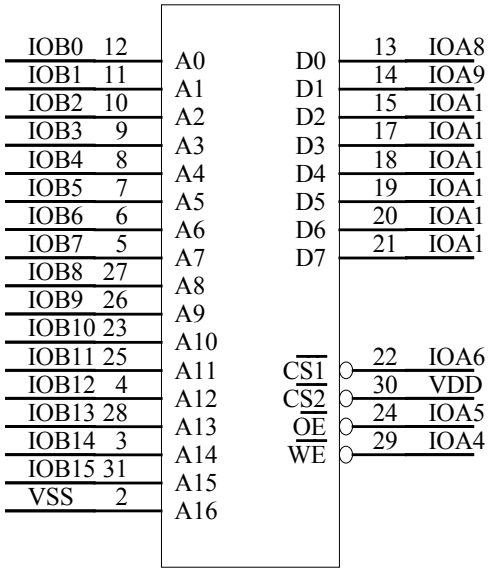
1)连接硬件线路。

2)编写程序。

3)编译程序、调试程序。

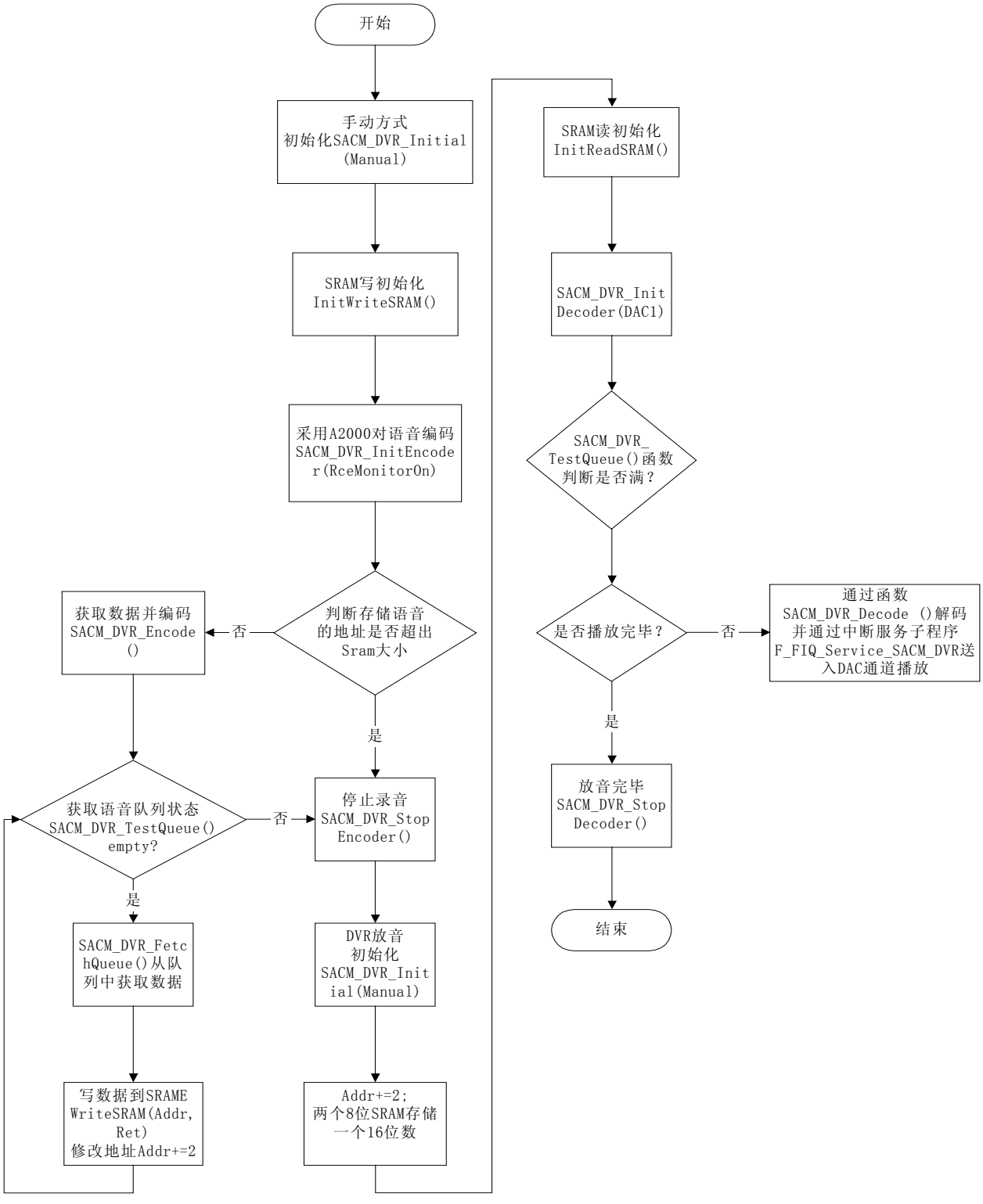
4)执行程序后开始录音，并在 32 秒后开始放音。

【硬件连接图】



628128A

手动方式主程序流程图：



【程序范例】

主程序（MAIN.C）

```
//*****************************************************************
// Note: DVR 有两种录放音方式，即自动方式和非自动方式，我们可以通
//过改变 Mode 的值来切换这两种方式：当 Mode=0 时以非自动方式录放音，
```

```
//当 Mode=1 时，以自动方式录放音，而且中断中分别用 FIQ 中断实现语音的
//播放，用 IRQ1 中断实现语音的录制。这里我们用非自动方式：
//*********************************************************************

#define Manual      0
#define   Auto      1
#define   SRAM_Size   0xffff-2
#define   Stop      0
#define   Record    1
#define   Play      2
#define   RceMonitorOff     0
#define   RceMonitorOn      1
#define   DAC1    1
#define   DAC2    2
#define   Full      1
#define   Empty     2

int main()
{
    int Ret=0,
    int Addr, Addr_Save;
    //Mode = Auto;                              //选择自动方式录放音
    Mode = Manual;                              //选择手动方式录放音
    if(Mode == Manual)                          //采用手动方式
    {
**************************录音*************************************
    SACM_DVR_Initial(Manual);                   //手动方式初始化
    Addr = 0;                                   //定义语音存放的首址变量
    InitWriteSRAM();//
    SACM_DVR_InitEncoder(RceMonitorOn);   //开始采用 A2000 对语音以非自动方式编码
    while(Addr<SRAM_Size)                 //判断存储语音的地址是否超出存储单元的大小
    {
        SACM_DVR_Encode ();               //获取数据并编码
        if(SACM_DVR_TestQueue()!= Empty)
        {
            Ret=SACM_DVR_FetchQueue();   // 从队列中获取数据
            WriteSRAM(Addr,Ret);          // 写入用户定义的存储单元区
            Addr+=2;                      //两个 8 位 SRAM 存储一个 16 位数据
         }

    }
    SACM_DVR_StopEncoder();

**********************放音*************************************
```

```
    SACM_DVR_Initial(Manual);                  //非自动方式播放的初始化
    InitReadSRAM();
        Addr=0;
    SACM_DVR_InitDecoder(DAC1);                //开始对 A2000 的语音数据以非自动方式解码
    while(1)
    {
        if(SACM_DVR_TestQueue()!=Full)         //测试并获取语音队列的状态
        {
            Ret =ReadSRAM(Addr);               //从存储区里获取一个字型语音数据
            SACM_DVR_FillQueue(Ret);           //获取语音编码数据并填入语音队列等候解码
            Addr+=2;
        }
        if(Addr<SRAM_Size)                     //如果该段语音播完，即到达末地址时
                SACM_DVR_Decode ();                //获取资源并进行解码，再通过中断服务子程
                                                   //送入 DAC 通道播放
        else
            SACM_DVR_StopDecoder();            //否则，停止播放
    }
    }
}
//中断程序 (ISR.ASM)
.public _FIQ;
 .public _IRQ1;
 .include hardware.inc;
.TEXT

.include dvr.inc;
_FIQ:
        push r1,r5 to [sp];
        call    F_FIQ_Service_SACM_DVR;   //语音播放中断
        r1=0xa800
        [P_INT_Clear]=r1
        pop r1,r4 from [sp];
        reti;

_IRQ1:
        push r1,r5 to [sp];
        call    F_IRQ1_Service_SACM_DVR; //语音录制中断
         r1=0x1000
        [P_INT_Clear]=r1
        pop r1,r4 from [sp];
        reti;
```