第 3 章

LED 灯和按键控制案例

微控制器原理及应用是一门实践性很强的计算机硬件开发课程，教学中需要加强学生的实践动手能力，并在教学之外辅以大量的创新应用实践，才能让学生灵活掌握技术原理的应用。大部分高校在课程实践教学中都是使用微控制器实验箱进行实践教学，存在实验箱易误操作、易损坏、设备成本高的问题，而且实验课时不足，导致实验效果不佳。针对应用型本科院校微控制器原理及应用课程实验教学存在的问题，本书设计了丰富的微控制器原理及应用实验仿真案例，将传统物理实验箱的实验内容，如硬件设计、软件编程、系统调试和效果展现全部迁移到仿真软件系统中，能够完成课程大纲要求的课内实验和课外创新设计。

基于 Proteus 硬件仿真工具和 Keil 程序设计软件构成的仿真设计环境，本章设计了与 LED 灯和按键控制相关的基本案例。主要内容包括流水灯、单键识别、汽车灯光模拟控制、I/O 接口应用、汇编指令、键盘接口、74LS244 的应用、74LS138 译码器的应用、8255A 的应用和 RXT-51 应用，每个案例都提供了详尽的软硬件设计，包括仿真电路和完整的参考程序，有助于学生自主学习和掌握微控制器内部基本硬件模块的工作原理和应用编程。

3.1 流水灯

3.1.1 案例概述

通过 8051 系列微控制器的 P1 口接 8 个 LED 发光二极管，要求编写程序实现：

(1) 从下往上每次点亮一个 LED，当点亮所有 LED 时，全灭。再从上往下每次点亮一个 LED，当点亮所有 LED 时，全灭；

(2) 全灭、全亮 2 次；

(3) 隔一个交替灭、亮 2 次；

重复上述过程。

3.1.2 要求

(1) 学习微控制器 I/O 接口结构特点及相关寄存器的使用方法；

(2) 掌握一个简单具体的微控制器项目的开发流程；

(3) 熟悉 Proteus ISIS 软件及使用方法。

3.1.3　知识点

8051 微控制器内部并行 I/O 接口结构和寄存器的用法、C51 语言应用程序设计。

3.1.4　电路原理图

案例控制电路如图 3-1 所示。8 个 LED 灯通过灌电流连接方式连接在微控制器的 P1 口的 8 个引脚上。限流电阻阻值不宜太大，阻值设定为 200Ω，否则 LED 灯不亮。

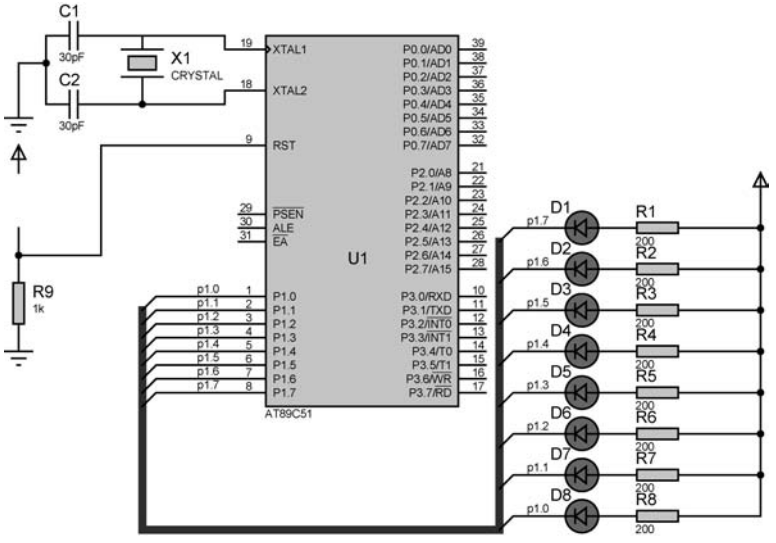

图 3-1　流水灯电路图

3.1.5　案例应用程序

```
#include <REG51.H>                          //8051 特殊功能寄存器定义
#define LED_PORT1 P1                        //用 P1 口驱动灯
void time(unsigned int ucMs);               //延时单位：ms
void main(void)
{
unsigned char ucTimes;
#define DELAY_TIME 400
while(1)
{
    LED_PORT1 = 0xff;                       //灭 8 个灯
    time(200);
    //从左往右依次点亮 LED
    for(ucTimes = 0;ucTimes < 8;ucTimes++){ //循环点亮 P1 口灯
        LED_PORT1 = LED_PORT1 - (0x80 >> ucTimes);  //低电平驱动灯亮
        time(DELAY_TIME);
    }

    LED_PORT1 = 0xff;                       //灭 P1 口灯
```

```
    time(200);
    //然后从右往左依次点亮 LED
    for(ucTimes = 0;ucTimes < 8;ucTimes++){              //循环点亮 P1 口灯
        LED_PORT1 = LED_PORT1 - (0x01 << ucTimes);       //低电平驱动灯亮
        time(DELAY_TIME);
    }

    LED_PORT1 = 0xff;time(DELAY_TIME);                   //全灭
    LED_PORT1 = 0;time(DELAY_TIME);                      //全亮
    LED_PORT1 = 0xff;time(DELAY_TIME);                   //全灭
    LED_PORT1 = 0;time(DELAY_TIME);                      //全亮

    LED_PORT1 = 0x55;time(DELAY_TIME);                   //隔一个点亮
    LED_PORT1 = 0xaa;time(DELAY_TIME);                   //交换

    LED_PORT1 = 0x55;time(DELAY_TIME);                   //隔一个点亮
    LED_PORT1 = 0xaa;time(DELAY_TIME);                   //交换
 }
}

void time(unsigned int ucMs)                             //延时单位: ms
{
#define DELAYTIMES 239
unsigned char  ucCounter;                                //延时设定的循环次数
 while (ucMs!= 0) {
    for (ucCounter = 0; ucCounter < DELAYTIMES; ucCounter++){}     //延时
    ucMs -- ;
    }
}
```

3.1.6 案例分析

因为 P1 口低电平驱动灯亮,高电平驱动灯灭。可以预先推算出控制流水灯亮灭状态的 P1 口驱动状态数据。

首先驱动 8 个灯全灭,然后要实现从下往上一个一个亮。可以用一个 for 循环实现状态控制。初始化时,先让右边第一个灯亮,然后再依次向下边顺序再点亮一个灯。本案例 P1 口的动态驱动数据设计为 0xff-(0x80 >> uctime),0x80 即是右边第一个灯灭的时候的驱动数据 1000 0000B,用 11111111B 减去它即为 0111 1111B,从而实现了第一个灯亮的控制。其他每个灯依次点亮的控制原理类似,通过 8 次循环执行 LED_PORT1=LED_PORT1-(0x80 >> ucTimes),实现点亮所有流水灯的效果。同理,如要实现从上往下一个一个亮,P1 口的动态驱动数据应设计为 LED_PORT1=LED_PORT1-(0x01 << ucTimes),通过 8 次循环执行实现需要的效果。

本案例还可以改变流水灯的变化方式,如从中间到两边依次亮。读者可以自己完成电路和相应的代码。

3.2　单键识别

3.2.1　案例概述

通过 8051 系列微控制器控制实现功能：每按一次按键，与 I/O 接口 P1 相连的 8 个发光二极管中点亮的一个往下移动一位。

注意：我们在手动按键时，一些机械抖动或是其他非人为的因素很有可能会造成误识别，一般手动按下一次键然后释放，按键两片金属膜接触的时间大约为 50ms，在按下瞬间到稳定的时间为 5～10ms，在松开的瞬间到稳定的时间也为 5～10ms。可以在首次检测到键被按下后延时 10ms 左右再去检测，这时如果是干扰信号将不会被检测到；如果确实有键被按下，则可确认。以上为按键识别去抖动的原理。

3.2.2　要求

(1) 学习微控制器 I/O 接口结构特点及相关寄存器的使用方法；

(2) 掌握一个简单具体的微控制器项目的开发流程；

(3) 熟悉 Proteus ISIS 软件及使用方法。

3.2.3　知识点

8051 微控制器的并行 I/O 接口知识、I/O 接口的结构和寄存器的用法。

3.2.4　电路原理图

如图 3-2 所示，控制按键接在 P3.4 引脚。8 个 LED 灯通过灌电流连接方式连接在微控制器的 I/O 接口 P1 的 8 个引脚上，限流作用的电阻阻值设定为 200Ω，否则 LED 灯不亮。

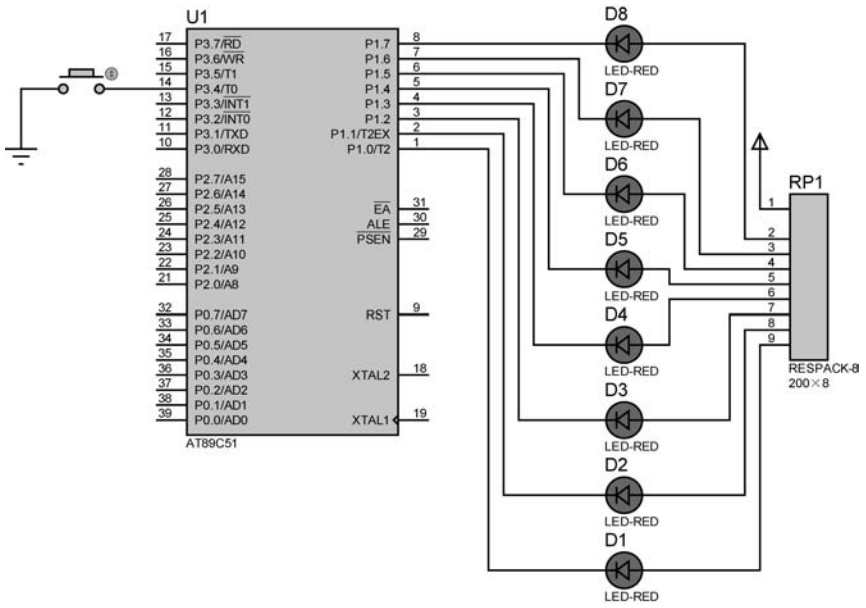

图 3-2　单键识别电路图

3.2.5 案例应用程序

```
#include <reg51.h>
Sbit BY1 = P3^4;                        //定义按键的输入端 S2 键
unsigned char count;                    //按键计数,每按一下,count 加 1
unsigned char temp;
unsigned char a,b;

void delay10ms(void)                    //延时程序
{
    unsigned char i,j;
    for(i = 20;i > 0;i -- )
    for(j = 248;j > 0;j -- );
}

key()                                   //按键判断程序
{
    if(BY1 == 0)                        //判断是否按下键盘
    {
      delay10ms();                      //延时,软件去干扰
      if(BY1 == 0)                      //确认按键按下
       {
       count++;                         //按键计数加 1
        if(count == 8)                  //计 8 次重新计数
        {
         count = 0;                     //将 count 清零
        }
       }
    while(BY1 == 0);                    //等待本次按键松开,确保每按一次 count 只加 1
    }
}

move()                                  //移动函数
{
    a = temp << count;
    b = temp >>(8 - count);
    P1 = a|b;
}

main()
{
count = 0;
temp = 0xfe;
P1 = 0xff;
P1 = temp;
while(1)                                //无限循环,判断按键是否按下
  {
   key();                               //调用按键识别函数
   move();                              //调用移动函数
  }
}
```

3.2.6　案例分析

在该案例中，按键判断函数 key()用于判断按键是否按下，每一次按下后就改变变量 count 的值，即加 1 运算。而 move()函数根据变量 count 的最新值改变变量 temp 的值，再由 temp 的新值控制 LED 灯的亮灭变化。

向左移动函数 move()的前 3 轮循环结果如下：

第一轮，

```
count = 0
temp = 0xfe;                    // temp = 11111110B
a = temp << 0;                  // a = 11111110B
b = temp >> 8;                  // b = 00000000B
P1 = a|b;                       // P1 = 11111110B
```

第二轮，

```
count = 1;
temp = 0xfe;                    // temp = 11111110B
a = temp << 1;                  // a = 11111100B
b = temp >> 7;                  // b = 00000001B
P1 = a|b;                       // P1 = 11111101B
```

第三轮，

```
count = 2
temp = 0xfe;                    // temp = 11111110B
a = temp << 2;                  // a = 11111000B
b = temp >> 6;                  // b = 00000011B
P1 = a|b;                       // P1 = 11111011B
```

以此类推，实现每按一次按键，与 P1 口相连的 8 个 LED 二极管依次点亮下一个。

3.3　汽车灯光模拟控制

3.3.1　案例概述

(1) 通过 8051 系列微控制器实现汽车左转向灯、右转向灯的模拟控制；

(2) 在左转向、右转向控制的基础上增加汽车故障灯，编程实现故障灯控制功能，要求故障灯控制不影响转向灯的控制；

(3) 在上述基础上增加汽车倒车灯，编程实现倒车灯控制功能，要求倒车灯打开后常亮，但是不影响故障灯和转向灯的控制功能的控制。

3.3.2　要求

(1) 学习使用微控制器各 I/O 接口的输入/输出功能及应用；

(2) 熟悉汽车灯光模拟控制应用程序的开发；

(3) 熟悉软件延时程序的编写方式。

3.3.3　知识点

微控制器 I/O 接口的输入/输出原理、C51 语言应用程序设计。

3.3.4 电路原理图

案例转向模拟控制电路如图 3-3 所示。

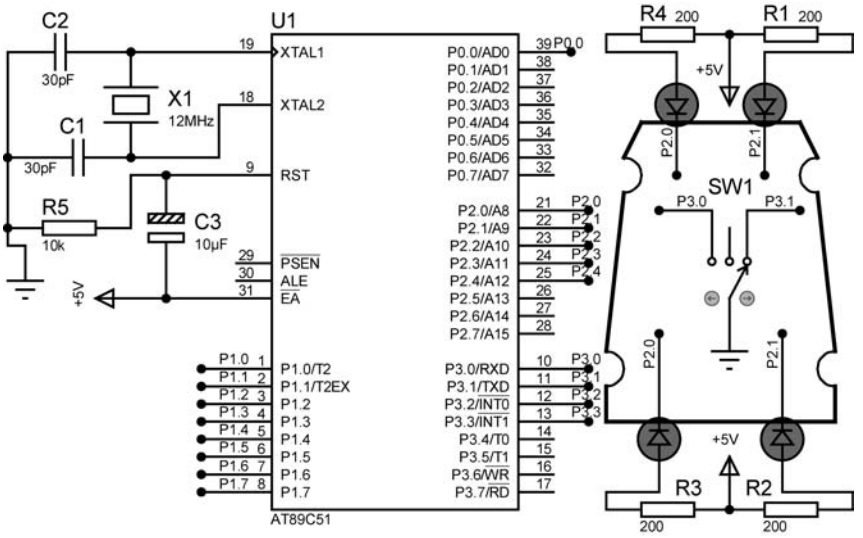

图 3-3 汽车左转向、右转向灯模拟控制

案例中，汽车左右转向由三向开关控制，案例中使用 P3 的 P3.0、P3.1 两个引脚分别作为左、右控制开关。汽车左右转向灯通过灌电流连接方式连接在微控制器的 I/O 接口 P2 的 P2.0、P2.1 两个引脚上，限流电阻阻值设定为 200Ω。

在汽车左转向灯、右转向灯模拟控制基础上，增加两个故障灯和两个倒车灯。使用微控制器的 P3.2 引脚连接倒车灯控制开关，使用 P3.3 引脚连接故障灯控制开关，如图 3-4 所示。

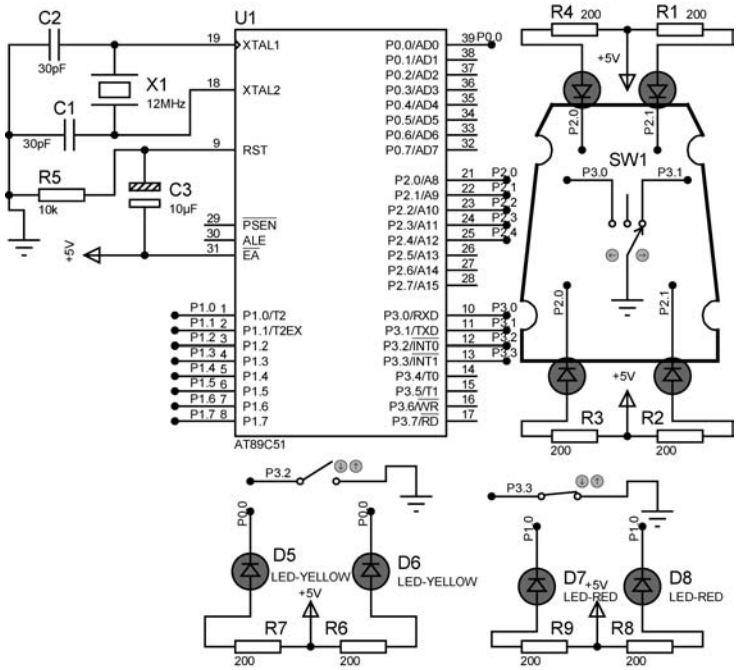

图 3-4 故障灯、倒车灯控制

3.3.5　案例应用程序

按照上述电路，相应的参考程序如下：

```
#include <REG51.H>                    //片内寄存器定义
#include <stdio.h>                    //输入/输出函数库
#include <intrins.h>                  //内部函数库
sbit L = P3^0;                        //左转向灯开关
sbit R = P3^1;                        //右转向灯开关
sbit B = P3^2;                        //倒车灯开关
sbit E = P3^3;                        //故障灯开关
sbit leftLed = P2^0;                  //左转向灯
sbit rightLed = P2^1;                 //右转向灯
sbit backLed = P0^0;                  //倒车灯
sbit errLed = P1^0;                   //故障灯
#define ON_leftLed    leftLed = 0
#define OFF_leftLed   leftLed = 1
#define ON_rightLed    rightLed = 0
#define OFF_rightLed   rightLed = 1
#define ON_backLed    backLed = 0
#define OFF_backLed   backLed = 1
#define ON_errLed     errLed = 0
#define OFF_errLed    errLed = 1
void time(unsigned int ucMs);         //延时单位：ms
void main (void)
{
    while(1){                         //3 种灯可以同时工作
        if (!L){                      //打开左转向灯,闪烁
        ON_leftLed;time(100);
        OFF_leftLed;time(100);
        }
        if (!R){                      //打开右转向灯,闪烁
        ON_rightLed;time(100);
        OFF_rightLed;time(100);
        }
        if (!B){ON_backLed;time(100);}    //打开倒车灯,常亮
        else  OFF_backLed;                //必须要关闭灯,否则倒车灯不能灭掉

        if (!E){                      //打开故障灯,闪烁
        ON_errLed;time(400);
        OFF_errLed;time(100);
        }
    }
}

void delay_5us(void)                  //延时 5μs,对于 22.1184MHz 晶振而言,需要 4 个_nop_();
                                      //对于 11.0592MHz 晶振而言,需要 2 个_nop_();
{
   _nop_();
   _nop_();
   _nop_();
   _nop_();
```

```
}

void delay_50us(void)                    //延时 50μs
{
    unsigned char i;
    for(i = 0;i < 4;i++)
    {
       delay_5us();
    }
}

void delay_100us(void)
{
    delay_50us();
    delay_50us();
}


void time(unsigned int ucMs)             //延时单位: ms
{
   unsigned char j;
   while(ucMs > 0){
      for(j = 0;j < 10;j++) delay_100us();
      ucMs -- ;
     }
}
```

另外,一个更简便的方法如图 3-5 所示。

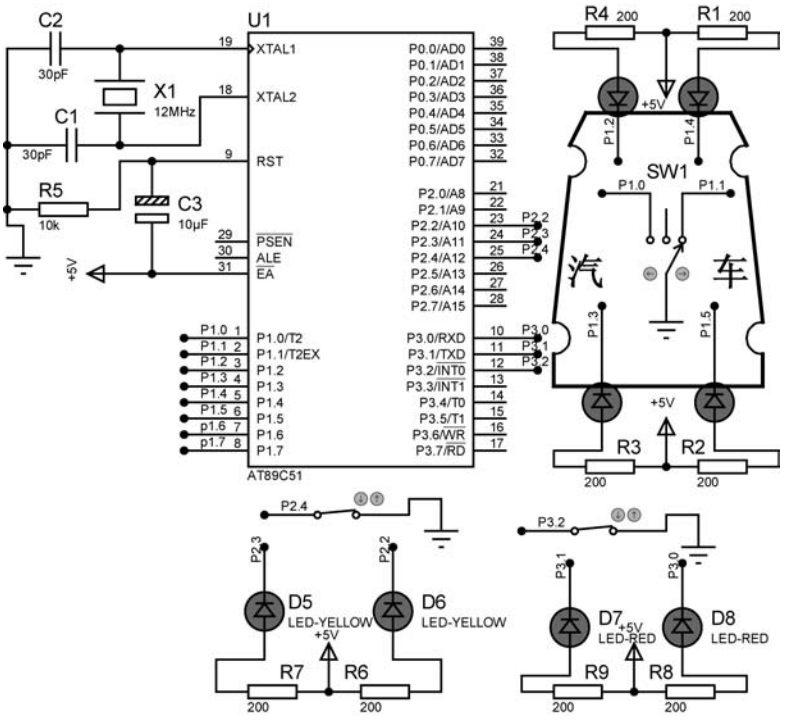

图 3-5 故障灯、倒车灯控制

在图 3-5 中，两个故障灯分别由 P2.2、P2.3 引脚驱动控制其亮灭，两个倒车灯分别由 P3.0、P3.1 引脚驱动控制其亮灭。两个左转向灯分别由 P1.0、P1.3 引脚驱动控制其亮灭，两个右转向灯分别由 P1.1、P1.5 引脚驱动控制其亮灭。使用微控制器的 P2.4 引脚连接倒车灯控制开关，使用 P3.2 引脚连接故障灯控制开关。汽车左右转向灯控制由三向开关控制。参考程序如下：

```
#include <reg51.h>
Sbit broken = P2^4;                    //故障灯控制开关,常亮
sbit back = P3^2;                      //倒车灯控制开关,闪烁
void delay(unsigned char com)
{
    unsigned char i,j,k;
    for(i = 5;i > 0;i -- )
      for(j = 132;j > 0;j -- )
        for(k = com;k > 0;k -- );
}

main(void){
    unsigned char dat;
    while(1){
        broken = 1;
        P1 = 0xff;                     //第二次循环开始控制左转向灯和右转向灯灭掉,形成闪烁
        P3 = 0xff;                     //第二次循环开始控制倒车灯灭掉,形成闪烁
        //P2 = 0xff;                   //故障灯常亮,不要闪烁,所以不能灭掉,除非开关断开灭掉
        delay(100);                    //灭一段时间
        dat = P1&0x03;                 //按位逻辑与,判断 P1.0、P1.1
        if(dat == 1)P1 = 0xcf;
        //若 P1.1 = 0,即 P1 = 0000 0001,P1.4 = P1.5 = 0,右转向灯亮即 P1 = 11001111
        if(dat == 2)P1 = 0xf3;
        //若 P1.0 = 0,即 P1 = 0000 0010,P1.2 = P1.3 = 0,左转向灯亮即 P1 = 11110011
        if(broken == 0){P2 = 0xf3;} else P2 = 0xff;
        //若 P2.4 为 0,则故障灯常亮,即设置 P2 为 11110011
        //此处必须控制,若开关断开后,灯要灭掉
        if(back == 0)P3 = 0xfc;        //倒车灯亮,闪烁  11111100
        delay(100);                    //控制灯亮一段时间
    }
}
```

3.3.6　案例分析

按照第二种案例程序实现方法，程序实现中，3 种汽车灯的控制开关都是直接接地，如果合上则开关连接的微控制器 I/O 接口引脚状态变为 0，表明开启相应的灯光控制。初始时汽车灯全灭，通过将汽车灯引脚置为高电平灭掉灯，并延时一段时间。

```
broken = 1;
P1 = 0xff;
P3 = 0xff;
delay(100);
```

然后判断左转向灯和右转向灯的控制开关是否合上，截取 P1 的最低 2 位的二进制数据(P1&0x03)值即可分析转向方向。P1 的最低 2 位为 1，即 P1.1 为 0，右转向灯开关合上，立即将 P1.1、P1.5 引脚置为 0，即可点亮右转向灯。同理，P1 的最低 2 位为 2，即 P1.0 为 0，左转向灯开关合上，立即将 P1.2、P1.3 引脚置为 0，即可点亮左转向灯。

倒车灯控制开关的状态由 if(back==0)来判断，如果合上，则设置 P3=0xfc，即将 P3.0、P3.1 引脚置为 0，即可点亮倒车灯。

故障灯控制开关的状态由 if(broken==0)来判断，如果合上，则设置 P2=0xf3，即将 P2.2、P2.3 引脚置为 0，即可点亮故障灯。如果故障灯控制开关断开，则设置 P2=0xff，必须控制故障灯灭掉。

3 种汽车灯的控制逻辑代码执行一遍之后，延时一段时间，从而实现转向灯和倒车灯的一次亮灭变化。由于控制逻辑代码是无限循环执行，故而可以实现转向灯和倒车灯的周期性亮灭变化，即闪烁效果。

另外，3 种汽车灯的控制互不干扰，而且故障灯开启后状态不应闪烁，而应处于常亮的状态，所以在故障灯控制中，通过“if(broken==0){P2=0xf3;} else P2=0xff;”实现故障灯开关断开后立即熄灯，从而不出现闪烁的效果。

3.4 I/O 接口应用

3.4.1 案例概述

通过 8051 系列微控制器编程实现如下的 32 个 LED 灯的显示效果：

(1) 从 L1～L32 按序轮流点亮一个 LED，然后熄灭。每个 LED 灯亮时间约 150ms。

(2) 在全部灯灭的情况下，按序依次从 L32～L1 点亮每个 LED，每个灯亮间隔 150ms。即点亮 L32，150ms 后再点亮 L31，依次按顺序点亮其他灯，最后点亮 L1，直到灯全部被点亮。

(3) 在全部灯亮的情况下，从 L1～L32 依次熄灭 LED，熄灭 L1，延时 150ms，然后熄灭 L2，延时 150ms，依次按顺序熄灭其他灯，最后熄灭 L32，最终全部灯被熄灭。

重复上述过程。

3.4.2 要求

(1) 学习微控制器 I/O 接口结构特点及相关寄存器的使用方法；

(2) 掌握一个简单具体的微控制器项目的开发流程；

(3) 熟悉 Proteus ISIS 软件及使用方法。

3.4.3 知识点

8051 系列微控制器的并行 I/O 接口、I/O 接口的结构和特殊功能寄存器的用法。

3.4.4 电路原理图

案例控制电路如图 3-6 所示，在微控制器的 4 个 I/O 接口 P1、P0、P3 和 P2 的 32 个引

脚上按顺序分别接一个 LED 灯，灯采用灌电流连接方式连接。限流电阻阻值不宜太大，阻值设定为 200Ω，否则 LED 灯不亮。

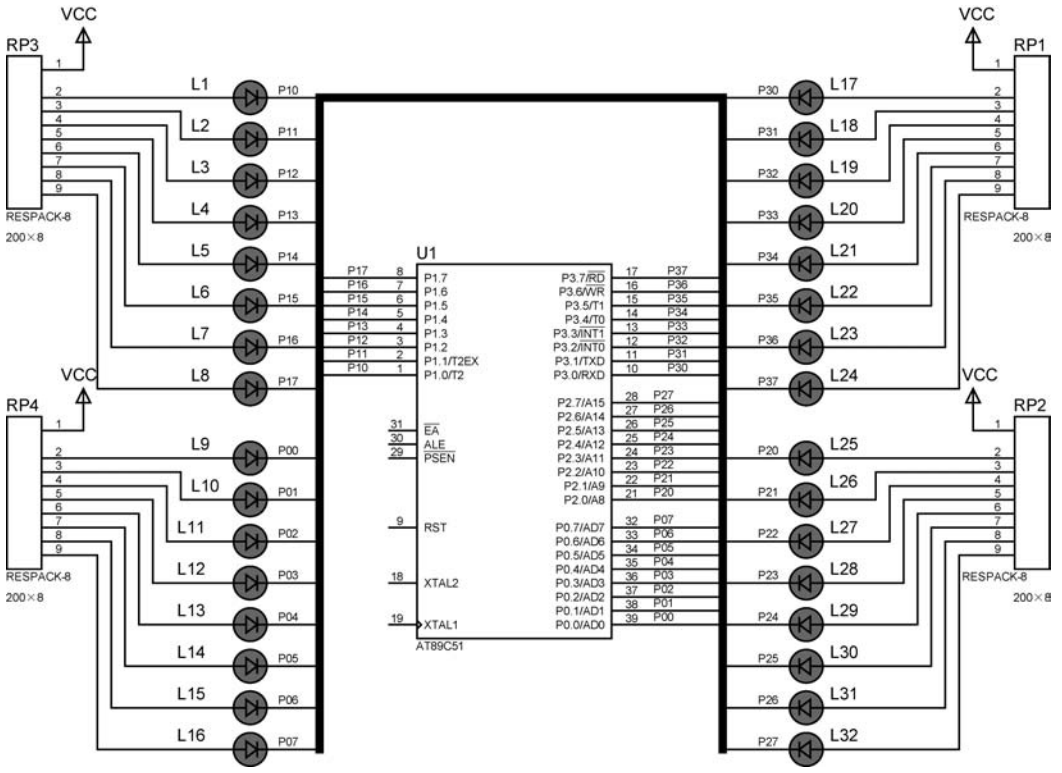

图 3-6　I/O 接口应用电路图

3.4.5　案例应用程序

1. 循环移位控制方法

通过循环移位控制 LED 灯的亮灭，代码如下：

```
#include <reg51.h>
void Delay()
{
    unsigned char i,j;
    for(i = 0;i < 255;i++)
    for(j = 0;j < 255;j++);
}

void main()
{
    unsigned char i;
    P1 = 0xff;
    P0 = 0xff;
    P3 = 0xff;
    P2 = 0xff;
```

```
while(1)
{
    //1
    for(i = 0; i < 8; i++)
    {
        P1 = P1 - (0x01 << i);
        Delay();
    }
    P1 = 0xff;
    for(i = 0; i < 8; i++)
    {
        P0 = P0 - (0x01 << i);
        Delay();
    }
    P0 = 0xff;
    for(i = 0; i < 8; i++)
    {
        P3 = P3 - (0x01 << i);
        Delay();
    }
    P3 = 0xff;
    for(i = 0; i < 8; i++)
    {
        P2 = P2 - (0x01 << i);
        Delay();
    }
    P2 = 0xff;

    //2
    for(i = 0; i < 8; i++)
    {
        P2 = 0x7f >> i;            //按位右移,i = 1,P2 = 00111111. i = 2,P2 = 00011111
        Delay();
    }
    for(i = 0; i < 8; i++)
    {
        P3 = 0x7f >> i;
        Delay();
    }
    for(i = 0; i < 8; i++)
    {
        P0 = 0x7f >> i;
        Delay();
    }
    for(i = 0; i < 8; i++)
    {
        P1 = 0x7f >> i;
        Delay();
    }

    //3
```

```
        for(i = 0; i < 8; i++)
        {
            P1 = P1|(0x01 << i);        //按位或,如 00000000|00000001、00000001|00000010 =
                                        //00000011  …
            Delay();
        }

        for(i = 0; i < 8; i++)
        {
            P0 = P0|(0x01 << i);
            Delay();
        }

        for(i = 0; i < 8; i++)
        {
            P3 = P3|(0x01 << i);
            Delay();
        }

        for(i = 0; i < 8; i++)
        {
            P2 = P2|(0x01 << i);
            Delay();
        }
    }
}
```

2. 定义数组数据控制方法

通过定义数组数据控制 LED 灯亮灭,代码如下:

```
#include <reg51.h>
unsigned char code tab1[] = {0xfe,0xfd,0xfb,0xf7,0xef,0xdf,0xbf,0x7f,0xff};  //左移单个
                                                                              //点亮
unsigned char code tab2[] = {0x7f,0x3f,0x1f,0x0f,0x07,0x03,0x01,0x00};  //右移逐个点亮
unsigned char code tab3[] = {0x01,0x03,0x07,0x0f,0x1f,0x3f,0x7f,0xff};  //左移逐个熄灭

void Delay()
{
    unsigned char i,j;
    for(i = 0;i < 255;i++)
    for(j = 0;j < 255;j++);
}

void main()
{
    unsigned char i;
    while(1)
    {
       //左移单个点亮
        for(i = 0;i < 9;i++)
        {
            P1 = tab1[i];          //单个点亮 L1～L8,0xfe = 11111110、0xfd = 11111101
            Delay();
        }
```

```
for(i = 0;i < 9;i++)
{
    P0 = tab1[i];          //单个点亮 L9～L16 ,0xfe = 11111110、0xfd = 11111101
    Delay();
}
for(i = 0;i < 9;i++)
{
    P3 = tab1[i];          //单个点亮 L17～L24
    Delay();
}
for(i = 0;i < 9;i++)
{
    P2 = tab1[i];          //单个点亮 L25～L32
    Delay();
}
//右移逐个点亮,最后全亮
for(i = 0;i < 8;i++)
{
    P2 = tab2[i];          //逐个点亮 L32～L25,0x7f = 01111111
                           //最后一个是 0x00,P2 = 0x00,保持不变.8 次循环后退出
    Delay();
}
for(i = 0;i < 8;i++)
{
    P3 = tab2[i];          //逐个点亮 L24～L17
    Delay();
}
for(i = 0;i < 8;i++)
{
    P0 = tab2[i];          //逐个点亮 L16～L9
    Delay();
}
for(i = 0;i < 8;i++)
{
    P1 = tab2[i];          //逐个点亮 L8～L1
    Delay();
}
//左移逐个熄灭
for(i = 0;i < 8;i++)
{
    P1 = tab3[i];          //逐个熄灭 L1～L8,0x01 = 00000001、0x03 = 00000011
    Delay();
}
for(i = 0;i < 8;i++)
{
    P0 = tab3[i];          //逐个熄灭 L9～L16
    Delay();
}
for(i = 0;i < 8;i++)
{
    P3 = tab3[i];          //逐个熄灭 L17～L24
    Delay();
}
for(i = 0;i < 8;i++)
```

```
        {
            P2 = tab3[i];          //逐个熄灭 L25～L32
            Delay();
        }
    }
}
```

3.4.6　案例分析

循环移位方法的程序分析如下：

(1) 实现从 L1～L32 按序轮流点亮一个 LED，然后熄灭。

初始时，32 个 LED 灯全灭，即将 I/O 接口 P1、P0、P3 和 P2 的 32 个引脚全部设置为高电平。要实现 32 个 LED 灯逐个轮流点亮，将每个 I/O 接口的 8 位二进制状态值按顺序逐个置为 0 即可。以 P1 为例，实现语句如下：

```
for(i = 0; i < 8; i++)
        {
            P1 = P1 - (0x01 << i);
            Delay();
        }
```

其中，语句 P1＝P1－(0x01≪i)在 8 次循环执行过程中，P1 的状态值分别为 11111110、11111100、…、00000000。经过 8 次循环后再将 P1 的值置为 0xff，为下次轮流点亮做准备。

(2) 在灯全部灭的情况下，实现按照顺序 L32～L1 依次点亮每个 LED，直到点亮全部灯结束。

初始时，32 个 LED 灯全灭，要实现按顺序依次从 L32～L1 点亮每个 LED，通过将每个 I/O 接口的 8 位二进制状态值按顺序依次置为 0 即可依次点亮每个 LED 灯。以 P1 为例，实现语句如下：

```
for(i = 0; i < 8; i++)
        {
            P2 = P2 & (0x7f >> i);
            Delay();
        }
```

其中，语句 P2＝P2&(0x7f≫i) 在 8 次循环执行过程中，P2 的状态值分别为 01111111、01111110、…、00000000。

(3) 在全部灯亮的情况下，要实现从 L1～L32 依次熄灭 LED，最后全部灯被熄灭。

由于前边的程序运行后，P1、P0、P3 和 P2 的值都为全 0，32 个灯处于全亮状态。通过将每个 I/O 接口的 8 位二进制状态值按顺序依次置为 1 即可依次熄灭每个 LED 灯。以 P1 为例，实现语句如下：

```
for(i = 0;i < 8;i++)
        {
            P1 = P1|(0x01 << i);
            Delay();
        }
```

其中，语句 P1＝P1|(0x01≪i) 在 8 次循环执行过程中，P1 的状态值分别为 00000001、00000011、…、11111111。

3.5 汇编指令

3.5.1 案例概述

通过 8051 系列微控制器控制实现功能：每按下一次按钮，计数值加 1，通过微控制器的 P2.0～P2.3 引脚连接的 LED 灯显示出其对应的二进制计数值，引脚低电平控制 LED 灯亮。

3.5.2 要求

(1) 熟悉 ISIS 模块的汇编程序编辑、编译与调试过程；
(2) 完成实验的汇编语言程序的设计与编译；
(3) 练习 ISIS 汇编程序调试方法，并最终实现实验的预期功能。

3.5.3 知识点

8051 微控制器的汇编指令语法、汇编应用程序设计、调试方法。

3.5.4 电路原理图

案例原理图如图 3-7 所示，图中输入电路由外接在 P1.3 引脚的 1 个按钮(but)组成；输出电路由外接在 P2 口的 8 只低电平驱动的发光二极管组成。此外，还包括时钟电路、复位电路和片选电路。

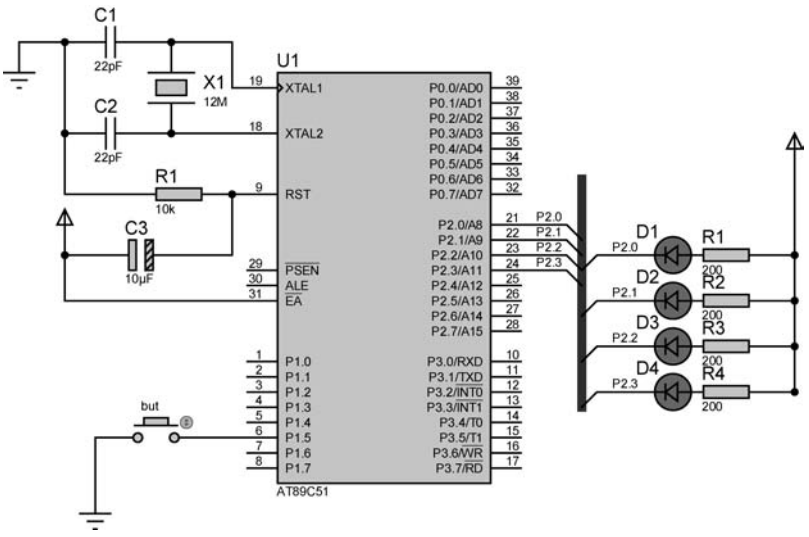

图 3-7 汇编指令实验原理图

3.5.5　案例应用程序

```
ORG 0100
Sbit   button = P1^5;
START: MOV R1, #00H          //R1 为 0,从 0 开始计数
MOV A, R1
CPL A                        //取反指令
MOV P2,A                     //送入 P1 口,由发光二极管显示
kkk: JB button,kkk           //判断 but 是否按下
LCALL DELAY50                //若 but 按下则延时 50ms
JB button,kkk                //消抖动,再判断 SP1 是否真的按下
INC R1                       //若确实按下则进行按键处理,即将计数内容加 1 后送入 P1 口
MOV A,R1 ;
CPL A                        //发光二极管显示
MOV P2,A ;
JNB button, $                //等待 but 释放
SJMP kkk                     //继续对 but 扫描
DELAY50: MOV R3, #50         //延时 12.5ms
L1: MOV R4, #125
DJNZ R4, $
DJNZ R3,L1
RET
END
```

3.5.6　案例分析

关于按钮的消除抖动问题,程序通过间隔 12.5ms 时间连续两次判断按钮是否按下来实现。

```
kkk: JB button,kkk           //判断 but 是否按下
LCALL   DELAY50              //若 but 按下则延时 12.5ms
JB button,kkk                //消抖动,再判断 SP1 是否真的按下
```

另外,关于按键恶意按住不放的问题,程序实现方法如下:

```
JB button,kkk                //若 button = 1 则转移到标号 kkk,否则顺序执行下一条指令
JNB button, $                //若 button = 0 则转移到标号 kkk,否则顺序执行下一条指令
```

此时,指令判断按钮是否按下,如果是按下则跳转到 $(本条指令的地址),即重复执行本条指令“JNB button, $”。如果判断按钮松开,则继续执行下一条指令。该指令作用相当于是等待按钮 but 释放才继续执行下一条指令。

程序编译调试方法:通过 ISIS 模块的汇编程序编辑、编译与调试工具,熟悉嵌入式微控制器的汇编语言的基本开发方法和仿真调试过程。首先单击 ISIS 主菜单“源代码”的“添加/移除源代码”命令,结果如图 3-8 所示。

如果已有程序文件,则单击“更改”按钮,找到程序文件所在的位置并选择,之后单击“确定”按钮。如果没有程序文件,则单击“新建”按钮,建立一个文件类型为.ASM 的文件,单击“打开”按钮,再单击“确定”按钮。接着单击“源代码”按钮,可以在框中看到程序文件,双击可打开编辑(注意:这里的程序文件只能是一个,否则编译的时候不知道是哪个)。

图 3-8 添加/移除源代码

建立源文件之后可以单击 ISIS 主菜单“源代码”的“全部编译”命令进行程序编译，在弹出的对话框中可以看到自己的代码是否有误。接着可以单击 ISIS 主菜单“调试”的“开始/重新启动调试”命令进行程序调试，同时可以打开“调试”菜单的最后 5 项，以便通过 Watch Window、Registers、SFR Memory、Internal(IDATA)Memory、Source Code 窗口观察分析程序执行的结果，如图 3-9 所示。

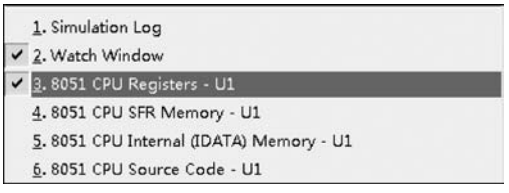

图 3-9 “调试”菜单的最后 5 项

在 Source Code 代码调试窗口中选择“单步越过命令行”按钮，即单步执行程序，可一步一步观察程序的运行结果，并进行逻辑功能分析，如图 3-10 所示。

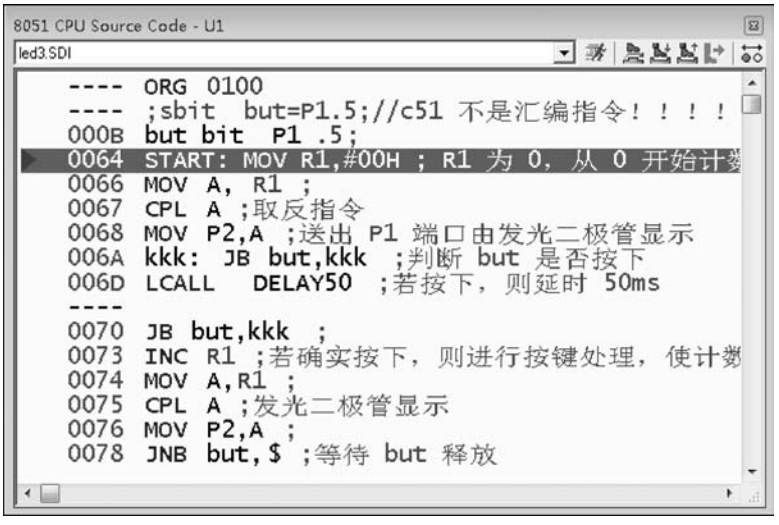

图 3-10 代码调试窗口

3.6　键盘接口应用

3.6.1　案例概述

通过 8051 系列微控制器编程编写 4×4 矩阵键盘按键扫描程序。按下任意键时，数码管会显示该键的序号。

3.6.2　要求

掌握微控制器控制行列式矩阵键盘的方法、接口电路设计和按键扫描应用程序设计。

3.6.3　知识点

行列式矩阵键盘原理、按键识别方法和应用程序开发。

3.6.4　电路原理图

1. 程序查询法

电路如图 3-11 所示。

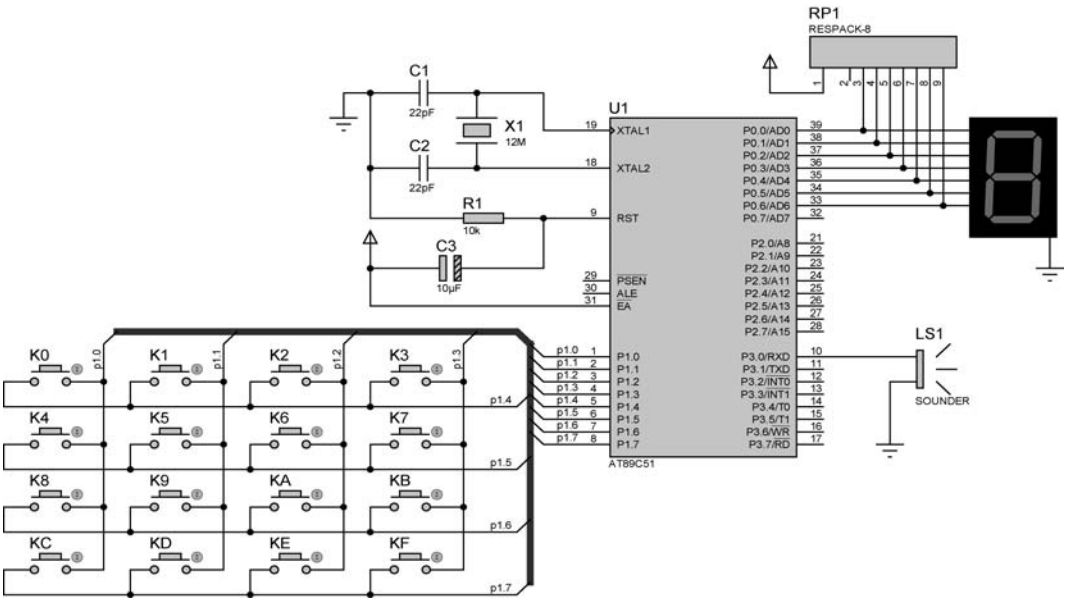

图 3-11　程序查询键盘接口电路图

案例电路图中，微控制器的 P1 口的高 4 位控制行列键盘的行信号，P1 口的低 4 位控制行列键盘的列信号。每一水平线（行线）与垂直线（列线）的交叉处接一个按键，由按键控制是否连通。利用这种行列矩阵结构，只需 4 个行线和 4 个列线即可组成 4×4 个按键的键盘。

另外，P0 口连接一个七段共阴极数码管，以便实时显示当前按下的按键序号。蜂鸣器直接用 P3.0 引脚的高电平驱动，以便提示当前有新的按键按下。

2. 程序中断扫描法

电路如图 3-12 所示，微控制器的 P1 口的高 4 位控制行列键盘的行信号，P1 口的低 4 位控制行列键盘的列信号。每一水平线(行线)与垂直线(列线)的交叉处接一个按键，由按键控制是否连通。另外，矩阵键盘的 4 个行信号引脚输入到一个与门，与门的输出连接到外部中断 1 的输入引脚 P3.3。

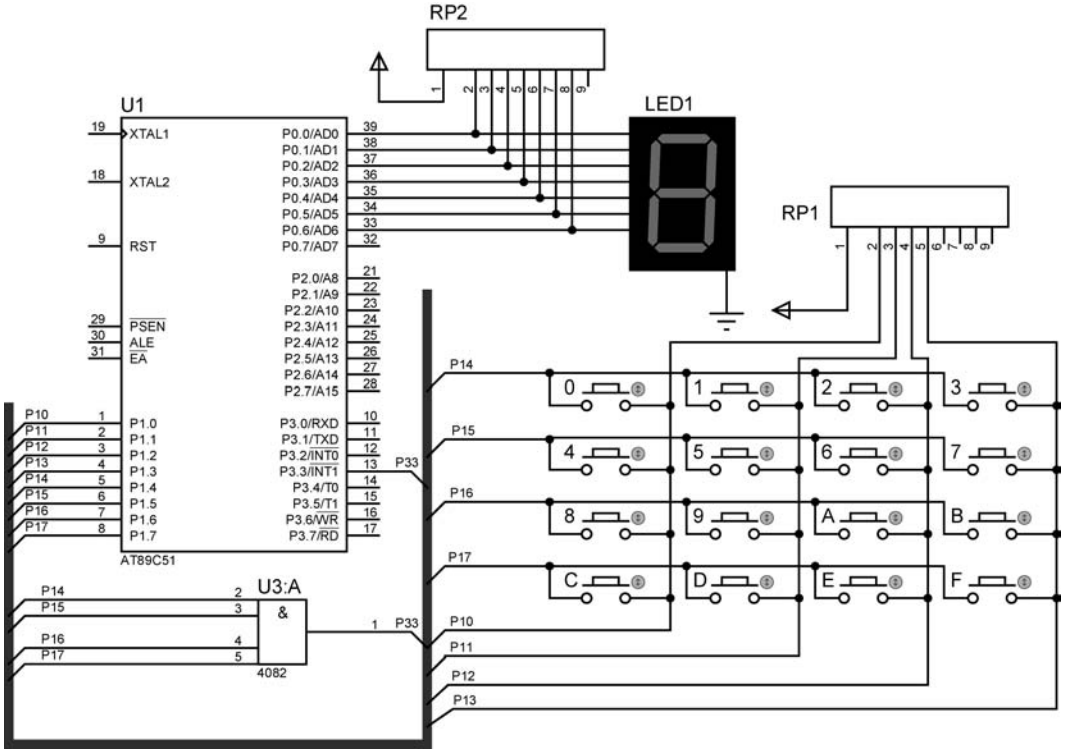

图 3-12　中断扫描键盘接口电路图

3.6.5　案例应用程序

为了减少键盘与微控制器接口时所占用 I/O 接口线的数目，在键数较多时，通常都将键盘排列成行列矩阵式。微控制器对按键的识别方法有以下两种方式：程序控制扫描方式(即利用程序连续地对键盘进行扫描)和中断扫描方式(即按键按下引起中断，然后微控制器对键盘进行扫描。该方法能够有效提高 CPU 的效率)。

1. 程序控制扫描方式

该方法通过 CPU 执行程序连续地对键盘进行扫描，以便查询按键是否按下并识别键号。

当按下任意一个按键时，键盘扫描程序首先判断按键发生在哪一列，然后根据所发生的行附加不同的值，从而得到按键的序号。实现代码如下所示：

```
#include <reg51.h>
#define uchar unsigned char
#define uint unsigned int
```

```
uchar code led[ ] = {0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90,0x88,0x83,0xc6,0xa1,
0x86,0x8e,0x00};
sbit BEEP = P3^0;
uchar oldKeyNo = 16,newKeyNo = 16;

void DelayMS(uint ms)
{
    uchar t;
    while(ms--)
    {
        for(t = 0;t < 120;t++);
    }
}

void Key()
{
    uchar T;
    P1 = 0x0f;
    DelayMS(1);
    T = P1 ^ 0x0f;
    switch(T)                    //求列号
    {
        case 1: newKeyNo = 0; break;
        case 2: newKeyNo = 1; break;
        case 4: newKeyNo = 2; break;
        case 8: newKeyNo = 3; break;
        default: newKeyNo = 16;
    }
    P1 = 0xf0;
    DelayMS(1);
    T = P1 >> 4 ^ 0x0f;
    switch(T)
    {
        case 1: newKeyNo += 0; break;
        case 2: newKeyNo += 4; break;
        case 4: newKeyNo += 8; break;
        case 8: newKeyNo += 12;
    }
}

void Beep()                      //蜂鸣器
{
    uchar i;
    for(i = 0;i < 100;i++)
    {
        DelayMS(1);
        BEEP = ～BEEP;
    }
    BEEP = 0;
}
```

```
void main()
{
    P0 = 0x00;
    while(1)
    {
        P1 = 0xf0;
        if(P1 != 0xf0)
            Key();
        if(oldKeyNo != newKeyNo)
        {
            P0 = ～led[newKeyNo];
            Beep();
            oldKeyNo = newKeyNo;
        }
        DelayMS(100);
    }
}
```

2. 中断扫描方法

```
#include <reg51.h>
#define uchar unsigned char
#define uint unsigned int
char led_mod[] = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f,0x77,0x7c,0x58,0x5e,
0x79,0x71};
//数码管段码

uchar oldKeyNo = 16,newKeyNo = 16;

void DelayMS(uint x)
{
    uchar i;
    while(x--) for(i = 0;i < 120;i++);
}

//外部中断 1 的中断函数
void Key() interrupt 2{
    uchar Tmp;
    P1 = 0x0f;              //P1 高 4 位的行置 0,  低 4 位的 4 列置 1
    DelayMS(1);
    Tmp = P1^0x0f;          //是异或运算符
    //求列号
    switch(Tmp)
    {
        case 1:newKeyNo = 0;break;
        case 2:newKeyNo = 1;break;
        case 4:newKeyNo = 2;break;
        case 8:newKeyNo = 3;break;
        default:newKeyNo = 16;
    }
```

```
P1 = 0xf0;                    //低 4 位的 4 列置 0,高 4 位的 4 行置 1
DelayMS(1);
Tmp = P1 >> 4^0x0f;

//求行号
switch(Tmp)                   //对 0～3 行分别附加起始值 0,4,8,12
{
    case 1:newKeyNo += 0;break;
    case 2:newKeyNo += 4;break;
    case 4:newKeyNo += 8;break;
    case 8:newKeyNo += 12;
}

//数码管显示键号
if(oldKeyNo!= newKeyNo)
    {
        P0 = led_mod[newKeyNo];
        oldKeyNo = newKeyNo;
    }
    DelayMS(100);
}

void main(void) {
   P0 = 0x00;
   IT1 = 1;
   EX1 = 1;
   EA = 1;
   P1 = 0xf0;
   while(1);
}
```

3.6.6　案例分析

1. 程序控制扫描方式

按下任意键时,数码管都会显示其键序号,扫描程序首先判断按键发生在哪一列,然后根据所发生的行附加不同的值,从而得到按键的序号。假设按键 5 按下,获取键号 5 的方法如下:

第一步,求列号。

按照该方法,先将键盘 4 个行引脚状态全设置为 0,4 个列引脚状态全设置为 1,通过设置 P1＝0x0f 实现。延时一段时间后,再次读取 P1 的低 4 位的状态值。然后根据低 4 位的值得到按键的列号。关键代码如下:

```
P1 = 0x0f;
DelayMS(1);
T = P1 ^ 0x0f;
```

例如,按下一个键后 P1 的状态值 0x0f 变成 0000xxxxB,低 4 位 xxxx 中一个为 0,3 个仍为 1,通过异或把 3 个 1 变为 0,唯一的 0 变为 1。比如序号 5 的键被按下后,P1.1＝0,即

P1＝00001101。故 T＝P1^0x0f＝2，再按照二进制序号，获得当前按键 5 的列号 KeyNo＝1。如果无按键按下，则当前按键的序号变量 KeyNo＝16，即无效键号。

```
case 1:KeyNo = 0;break;
case 2:KeyNo = 1;break;
case 4:KeyNo = 2;break;
case 8:KeyNo = 3;break;
default:KeyNo = 16;          //无键按下
```

第二步，求行号。

先将键盘 4 个行引脚状态全设置为 1，4 个列引脚状态全设置为 0，即通过设置 P1＝0x0f0 实现。延时一段时间后，再次读取 P1 的高 4 位的状态值，然后根据低 4 位的值得到按键的列号。关键代码如下：

```
P1 = 0x0f0;
DelayMS(1);
T = P1 >> 4 ^ 0x0f;
```

例如，按下一个键后 P1 的状态值 0x0f0 变成 xxxx0000，高 4 位 xxxx 中有一个为 0，3 个仍为 1；为了便于计算，再将高 4 位转移到低 4 位，并由异或操作得到改变的值。比如按键 5 被按下后，P1.5＝0，即 P1＝11010000，故 T＝P1 >> 4^0x0f＝2，即当前按键 5 的行号是 2。

第三步，求键值。

通过前面获得按键的行号之后，可以根据不同的行号对第一步得到的列号加一个附加码，从而计算出当前按键的键号。按照矩阵键盘的排列序号，第 0 ～ 3 行分别添加的附加码值分别为 0，4，8，12。关键代码如下：

```
switch(T)
    {
        case 1: newKeyNo += 0; break;
        case 2: newKeyNo += 4; break;
        case 4: newKeyNo += 8; break;
        case 8: newKeyNo += 12;
    }
```

例如，按键 5 被按下后，P1.5＝0，即 P1＝11010000，再将高 4 位转移到低 4 位变量 T＝P1 >> 4^0x0f＝2，即行号为 2，最终按键 5 的键值为 KeyNo＋4 即 5。

2. 中断扫描方式

本方式在程序控制扫描方式的基础上做了改进，即将中断技术引入按键扫描过程中，即只有当有按键被按下后，才引起外部中断，触发微控制器对键盘进行扫描并获取键号，否则微控制器执行其他程序，相当于键盘不存在。中断扫描键盘可以有效提升微控制器中 CPU 的利用率。具体如下：

主程序初始化外部中断 INT1 之后，再将键盘的 4 个行信号引脚全部设置为 1，键盘的 4 个列信号引脚全部设置为 0，即 P1＝0xf0。

如果有键按下，则该按键所在的行信号变为 0，4 个行信号输入与门，与门输出低电平送至外部中断 1 的输入引脚 P3.3，触发外部中断 1。此时暂停主程序，转而执行外部中断 1 的

服务程序。通过将程序控制扫描获取键值的代码放在中断服务程序中，每次触发一次外部中断1，就可以获取一个最新的按键的键值并送数码管显示。

没有按键按下的时候，4个行信号引脚与4个列信号引脚全部断开。4个行信号状态保持1111，与门输出高电平送至外部中断1的输入引脚P3.3，不会触发外部中断1。这样可大大减少无按键按下时按键扫描程序耗费的CPU的时间。

3.7　74LS244的应用

3.7.1　案例概述

8051系列微控制器通过扩展的三态缓冲器74LS244读取8个开关断开的状态，并统计断开的个数并送数码管显示。

3.7.2　要求

掌握利用缓冲器来扩展微控制器的输出引脚的方法。

3.7.3　知识点

三态缓冲器74LS244的硬件特点、接口设计和应用编程。

3.7.4　电路原理图

案例电路原理图如图3-13所示。

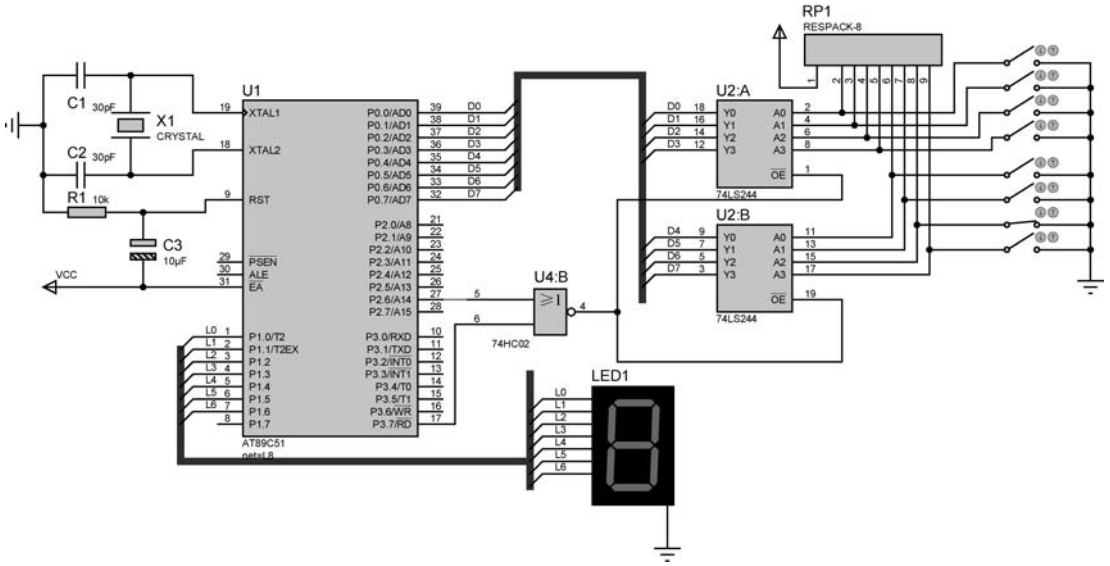

图3-13　74LS244应用电路图

74LS244是一种三态输出的8总线缓冲驱动器，无锁存功能，当$\overline{OE}$为低电平时，输入到引脚Ai的信号传送到输出引脚Yi；当$\overline{OE}$为高电平时，Yi处于禁止高阻状态。Proteus

中的 74LS244 是 4 位输出。

本案例利用 74LS244 作为输入口，读取 8 个开关状态，并将此状态通过 74LS244 输入到微控制器，由微控制器处理后送数码管显示开关的断开个数。

3.7.5 案例应用程序

```
#include <reg51.h>
#include <absacc.h>
#define a74ls244 XBYTE[0xBFFF]        //定义 74LS244 的外部 RAM 地址
unsigned char code LED[] = {0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90,0x88}; //共阳
                                      //极数码管的段码表
//计算变量中为 1 的个数
unsigned char jisuan(unsigned char k)
{
  unsigned char sss = 0;
  while(k)                            //P0 获取的开关状态的实时值,不为 0 就计算
  {
    if(k % 2 == 1)
    sss++;
    k/ = 2;
   }
   return sss;
}

int main(void)
{
  unsigned char i,ccc,t;
  P1 = ~LED[0];                       //初始时显示 0
  while(1)
  {
     for(i = 0;i < 250;i++)           //延时
     {
     }
     t = P0;                          //读取 74LS244 的端口的状态
     ccc  = jisuan(t);                //计算 1 的个数,即断开关的个数
     P1 = ~LED[ccc];                  //得到有几位拨码开关断开,并查表得到段码输出显示
  }
}
```

3.7.6 案例分析

将三态缓冲器 74LS244 作为微控制器的外部 RAM 设备，设置它的外部 RAM 地址，对该地址单元读入数据，以便获取 8 个开关的闭合断开状态的数据。

案例中，微控制器的读信号 $\overline{RD}$ 和 P2.6 配合，即二者输入到或非门 74HC02，或非门输出作为 74LS244 的 $\overline{OE}$ 信号，从而选中 74LS244 工作，再通过此 74HC273 输出数据的段码实现显示。两片 74LS244 的使能端 OE 都接或非门 74HC02 的输出端，即两片 74LS244 的端口地址相同，可被微控制器同时选中，74LS244 每个输入端 Ai 分别连接一个开关。拨动

开关，观察数码管的变化。

案例中将 74LS244 的外部 RAM 地址 0x7fff＝01111111 B，即 P2.6＝1。P2.6 与读信号 $\overline{RD}$(P3.7)送到或非门 74HC02 的输入端，或非门的输出为 0，送到 74LS244 的 $\overline{OE}$，使之工作。74LS244 读取 8 个开关的断开状态，通过 P0 口送入微控制器内部分析计算，从而将 8 个开关中处于断开状态的个数显示在数码管上。

3.8　74LS138 译码器的应用

3.8.1　案例概述

通过 8051 系列微控制器 I/O 接口控制 74LS138 译码器的输出信号变化，并控制 P1 连接的 8 个 LED 灯的流水亮灭。

3.8.2　要求

掌握利用 74LS138 译码器来扩展微控制器的输出引脚的方法。

3.8.3　知识点

74LS138 译码器的硬件特点、接口设计和应用编程。

3.8.4　电路原理图

案例控制电路如图 3-14 所示。

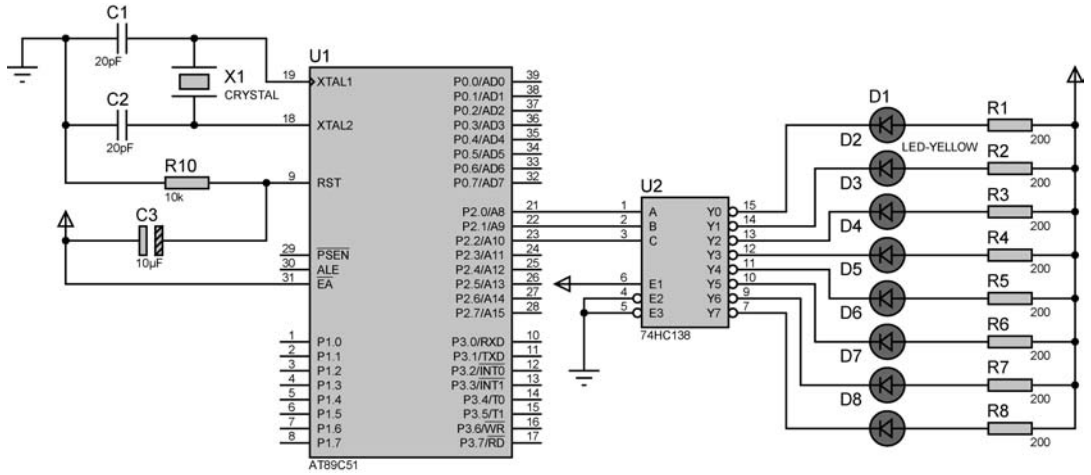

图 3-14　74LS138 译码器应用电路图

8 个 LED 灯通过灌电流连接方式连接在 74LS138 译码器的 8 个输出引脚 Y0～Y7 上，限流电阻阻值设定为 200Ω。原理图中 P2 口的 P2.0、P2.1、P2.2 引脚作为 74LS138 译码器的输入信号 ABC，微控制器通过 P2.0、P2.1 和 P2.2 输出的 8 种不同编码控制 74LS138 译码器的输出变化。3-8 译码器 74LS138 的真值表如表 3-1 所示。

表 3-1 74LS138 的真值表

E1	E2	E3	C	B	A	Y7 ～ Y0	有效输出
0	0	1	0	0	0	1 1 1 1 1 1 1 0	Y0
0	0	1	0	0	1	1 1 1 1 1 1 0 1	Y1
0	0	1	0	1	0	1 1 1 1 1 0 1 1	Y2
0	0	1	0	1	1	1 1 1 1 0 1 1 1	Y3
0	0	1	1	0	0	1 1 1 0 1 1 1 1	Y4
0	0	1	1	0	1	1 1 0 1 1 1 1 1	Y5
0	0	1	1	1	0	1 0 1 1 1 1 1 1	Y6
0	0	1	1	1	1	0 1 1 1 1 1 1 1	Y7

74LS138 译码器的输入地址信号 CBA 有 8 种编码，当一个选通端(E1)为高电平，另两个选通端(E2)和(E3)为低电平时，可根据输入端地址(CBA)的二进制编码控制输出端 Y0～Y7 对应的一个引脚以低电平译出。比如，当 CBA＝110 时，Y6 输出端输出低电平信号。

3.8.5 案例应用程序

```
#include <REG51.H>
bit   Flg = 0;                              //T1 标志位
unsigned char ccc = 0;

void Timer1(void) interrupt 3 using 2       //定时器 1 中断处理函数
{
  ET1 = 0;                                  //关闭 T1 中断
  TH1 = 0x4C;
  TL1 = 0x00;                               //初始化定时器初值
  ET1 = 1;                                  //打开 T1 中断
  ccc++;

  if(ccc  == 20)                            //定时 1s
  {
    Flg = 1;                                //定时器中断标志位置位
    ccc = 0;
  }
}

int main(void)
{
    TMOD = 0x10;                            //初始化 T1 工作方式 1
    TH1 = 0x4C;
    TL1 = 0x00;                             //初始化定时器值,50ms
    ET1 = 1;
    TR1 = 1;                                //启动 T1
    EA = 1;                                 //开中断
    P2 = 0;

    while(1)
    {
```

```
    while(Flg == 0);                    //1s 延时完成的标志位
    Flg = 0;
    P2 = (P2 + 1) % 8;                  //始终在 0～7 选择
    }
}
```

3.8.6　案例分析

案例代码中，主要有两个并行的函数：main()和 timer1()。主程序中，每隔 1s 就改变 I/O 接口 P2 的状态值，继而改变输入到 74LS138 译码器的输入信号 CBA 的值，最终改变 74LS138 译码器译码输出引脚的状态，因为只有一个输出引脚为低电平，故只有一盏灯亮。具体实现代码如下：

```
while(1)
     {
        while(Flg == 0);                //1s 延时完成的标志位
        Flg = 0;
        P2 = (P2 + 1) % 8;              //始终在 0～7 选择
     }
```

在定时器函数中，定时器硬件一次定时时间是 50ms，只有完成 20 次 50ms 的定时后，定时 1s 完成的标志变量 Flg 的值才设置为 1。主程序中就是通过变量 Flg 的值来判断 1s 定时是否完成。具体实现代码如下：

```
if(ccc == 20)                           //定时 1s
  {
    Flg = 1;                            //定时器中断标志位置位
    ccc = 0;
  }
```

3.9　8255A 的应用

3.9.1　案例概述

8051 系列微控制器通过 8255A 的输出口 A、B、C 控制连接的 3 个 LED 数码管显示 120s。

3.9.2　要求

掌握利用 8255A 来扩展微控制器的并行输出引脚的方法。

3.9.3　知识点

8255A 的硬件原理、特点、接口设计和应用编程。

3.9.4　电路原理图

案例电路原理图如图 3-15 所示。

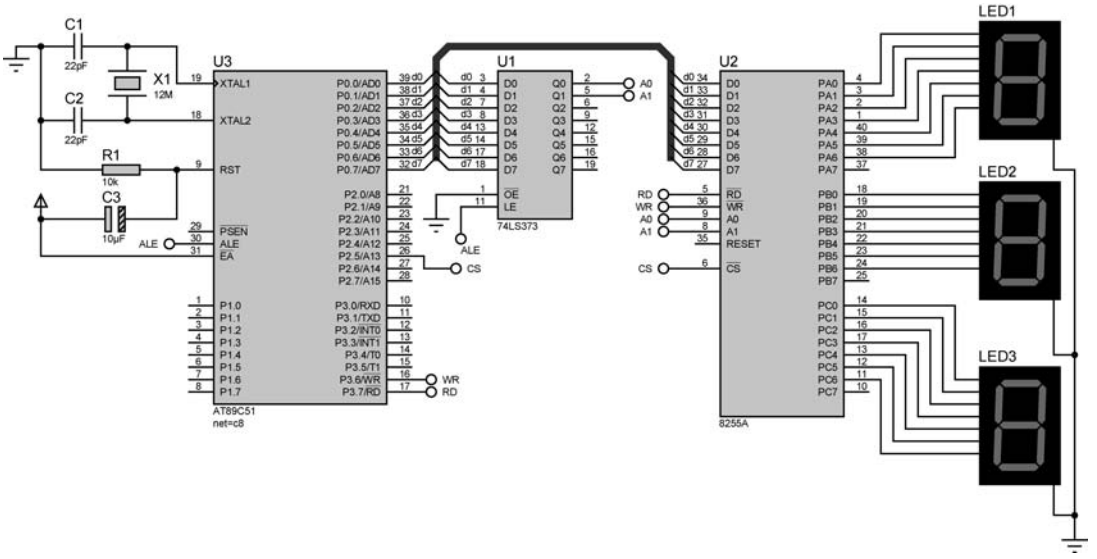

图 3-15　8255A 应用电路图

案例原理图中，采用 8255A 的 3 个输出端口 A、B、C 分别连接一个数码管，以便分别驱动数码管显示不同的内容。微控制器的 P0 口连接双向缓冲锁存器 74LS373 的 8 个输入引脚，微控制器就可通过 74LS373 来驱动 8255A（即微控制器通过 74LS373 向 8255A 发送地址信号，P0 的输出数据信号直接输出到 8255A 的数据输入端 D0～D7）。微控制器可以分别选中 8255A 的某一个输出口工作，再通过此输出口输出段码驱动一个数码管显示。微控制器的读、写控制信号 $\overline{RD}$、$\overline{WR}$ 分别连接 8255A 的 $\overline{RD}$、$\overline{WR}$。锁存器 74LS373 的输出 Q0、Q1 分别连接 8255A 的地址输入端 A0、A1 引脚。微控制器通过 P2.5 引脚控制 8255A 的片选信号引脚 $\overline{CS}$。

74LS373 是带有三态门的 8D 锁存器，当使能信号线 $\overline{OE}$ 为低电平时，三态门处于导通状态，允许 D0～D7 输出到 Q0～Q7，当 $\overline{OE}$ 端为高电平时，输出三态门断开，输出线处于浮空状态。LE 称为数据输入线，当 74LS373 用作地址锁存器时，首先应使三态门的使能信号 $\overline{OE}$ 为低电平，这时，当 LE 端输入端为高电平时，锁存器输出（Q0～Q7）状态和输入端（D0～D7）状态相同；当 LE 端从高电平返回到低电平（下降沿）时，输入端（D0～D7）的数据锁入 Q0～Q7 的 8 位锁存器中。

3.9.5　案例应用程序

```
#include <reg51.h>
#include <absacc.h>                          //外部内存空间宏定义
#define dA XBYTE[0xd000]                     //外部寄存器定义 A 端口
#define dB XBYTE[0xd001]                     //外部寄存器定义 B 端口
#define dC XBYTE[0xd002]                     //外部寄存器定义 C 端口
#define control XBYTE[0xd003]                //外部寄存器定义 8255A 的控制端口
unsigned char ccc = 0;                       //计数器
bit flg = 0;                                 //T0 标志位
unsigned char code LED[ ] = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f};    //共阴极
```

```
                                        //数码管的段码表

void t1(void) interrupt 3 using 1       //定时器 1 中断处理函数
{
    ET1 = 0;                            //关闭中断
    TH1 = 0x3C;                         //设置定时器的初值
    TL1 = 0xB0;
    ET1 = 1;                            //打开 T1 中断
    ccc++;
    if(ccc == 20)                       //定时 1s 的控制
    {
     flg = 1;                           //定时器中断标志位置位
     ccc = 0;
    }
}

int main(void)
{
  unsigned char ttt = 0;
  TMOD = 0x10;                          //初始化 T1 工作在方式 1
  TH1 = 0x3C;
  TL1 = 0xB0;                           //50ms 定时初值
  ET1 = 1;                              //开启定时器 1 中断
  TR1 = 1;                              //启动定时器
  EA = 1;                               //开中断
  control = 0x80;                       //初始化 8255A 的工作方式

  while(1)
    {                                   //循环输出 0～9

      while(flg == 0);                  //判断 1s 定时时间是否完成
      flg = 0;
      ttt++;
      if(ttt > 120)                     //秒计数器最大值为 120
      {
        ttt = 0;                        //到 120s 则恢复到 0
      }
      dA = LED[ttt/100];                //送百位显示
      dB = LED[ttt/10 % 10];            //送十位显示
      dC = LED[ttt % 10];               //送个位显示
    }
}
```

3.9.6　案例分析

本案例通过 8051 系列微控制器 I/O 接口 P0 控制 8255A 的输出端口 A、B、C 信号变化，进一步控制 A、B、C 连接的 3 个 LED 数码管显示状态。

8255A 可编程外围接口芯片是 Intel 公司生产的通用并行接口芯片，它具有 A、B、C 3 个并行接口，用＋5V 单电源供电，能在以下 3 种方式下工作：

方式 0，基本输入/输出方式。

方式 1，选通输入/输出方式。

方式 2，双向选通工作方式。

本实验中，8255A 端口 A、B、C 都工作在方式 0 并作为输出口，输出段码以便控制数码管的显示。

电路图中，将 8255A 作为微控制器的外部 RAM 设备，所以可以设置 8255A 的外部 RAM 地址。8255A 内部 4 个寄存器可定义 4 个 RAM 地址。本例中的 8255A 的外部 RAM 地址采用绝对宏 XBYTE 实现。因为 8255A 的 CS 信号接 P2.5，所以 A、B、C 端口的地址分别为 1101 0000 0000 0000、1101 0000 0000 0001、1101 0000 0000 0010，控制口地址为 1101 0000 0000 0011。另外，8255A 的 CS 接 P2.5，写、读控制信号 $\overline{WR}$ 和 $\overline{RD}$ 分别接微控制器的 P3.6 和 P3.7，A1、A0 分别接地址锁存器 74LS373 的输出引脚 Q1 和 Q0。

当地址为 0xd000，即 P2.5＝0，信号输入到 8255A 的 $\overline{CS}$，低电平有效，8255A 的输出口 A 口可以工作，微控制器再将秒计数变量 ttt 的百位值送到 A 口，并驱动数码管显示。

当地址为 0xd001，即 P2.5＝0，信号输入到 8255A 的 $\overline{CS}$，低电平有效，8255A 的输出口 B 口可以工作，微控制器再将秒计数变量 ttt 的十位值送到 B 口，并驱动数码管显示。

当地址为 0xd002，即 P2.5＝0，信号输入到 8255A 的 $\overline{CS}$，低电平有效，8255A 的输出口 C 口可以工作，微控制器再将秒计数变量 ttt 的个位值送到 C 口，并驱动数码管显示。

在定时器函数中，定时器硬件一次定时时间是 50ms，只有完成 20 次 50ms 的定时溢出中断后，定时标志变量 flg 的值才设置为 1。具体实现代码如下：

```
if(ccc == 20)                          //定时 1s
  {
    flg = 1;                           //定时器中断标志位置位
    ccc = 0;
  }
```

在主程序中，每隔 1s 时间就改变 3 个数码管的秒计数显示。1s 定时是否完成是通过循环检测定时标志变量 flg 的值来判断。1s 时间完成之后，就分别通过对 8255A 的 3 个外部 RAM 地址赋值，输出秒计数变量 ttt 的百、十、个位的值，驱动 3 个数码管显示。具体实现代码如下：

```
while(flg == 0);                       //判断 1s 定时时间是否完成
    flg = 0;
    ttt++;
    dA = LED[ttt/100];                 //送百位显示
    dB = LED[ttt/10 % 10];             //送十位显示
    dC = LED[ttt % 10];                //送个位显示
```

3.10 RTX-51 的应用

3.10.1 案例概述

假设在 8051 微控制器的 P2 口接有 8 个 LED，使用 RTX-51 Tiny，编写程序使 8 个

LED 以不同的频率闪烁。

3.10.2　要求

(1) 学习使用多任务实时操作系统 RTX-51 Tiny 软件设计方法；

(2) 熟悉 OS_WAIT()函数的使用。

3.10.3　知识点

(1) RTX-51 Tiny 的多任务并发的应用编程；

(2) OS_WAIT()函数的挂起、唤醒功能。

3.10.4　电路原理图

案例控制电路如图 3-16 所示，8 个 LED 灯连接在微控制器的 P2 口的 8 个引脚上，灯通过灌电流连接方式连接。限流电阻阻值设定为 200Ω。

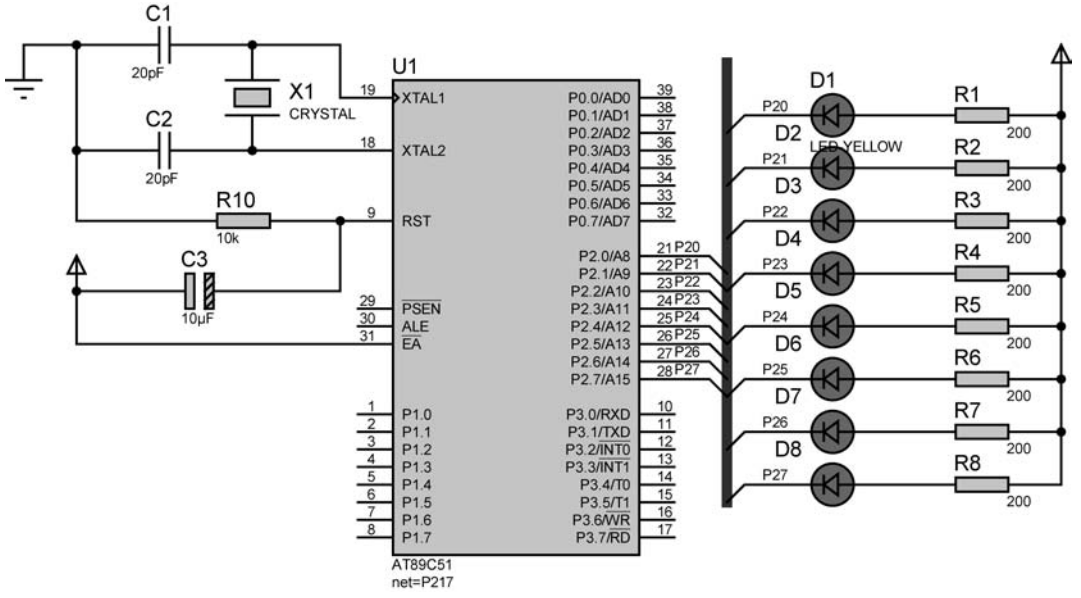

图 3-16　RTX-51 Tiny 的应用

3.10.5　案例应用程序

```
#include <reg51.h>
#include <rtx51tny.h>                    //包含 RTX-51 Tiny 的头文件
#define uint  unsigned  int
#define uchar unsigned char
sbit  P20 = P2^0;
sbit  P21 = P2^1;
sbit  P22 = P2^2;
sbit  P23 = P2^3;
sbit  P24 = P2^4;
```

```
sbit   P25 = P2^5;
sbit   P26 = P2^6;
sbit   P27 = P2^7;

void init(void)_task_ 0
{
os_create_task(1);
os_create_task(2);
os_create_task(3);
os_create_task(4);
os_create_task(5);
os_create_task(6);
os_create_task(7);
os_create_task(8);
os_delete_task(0);
}

void Pt0(void)_task_ 1
{
while(1)
  {
   P20 = !P20;
   os_wait(K_TMO,25,0);                    //如果将 25 修改为 100,则闪烁明显变慢
  }
}

void Pt1(void)_task_ 2
{
 while(1)
 {
  P21 = !P21;
  os_wait(K_TMO,35,0);
 }
}

void Pt2(void)_task_ 3
{
 while(1)
 {
  P22 = !P22;
  os_wait(K_TMO,50,0);
 }
}

void t3(void)_task_ 4
{
 while(1)
 {
  P23 = !P23;
  os_wait(K_TMO,95,0);
 }
```

```
}

void t4(void)_task_ 5
{
 while(1)
 {
  P24 = !P24;
  os_wait(K_TMO,95,0);
 }
}

void t5(void)_task_ 6
{
 while(1)
 {
  P25 = !P25;
  os_wait(K_TMO,50,0);
 }
}

void t6(void)_task_ 7
{
 while(1)
 {
  P26 = !P26;
  os_wait(K_TMO,35,0);
 }
}

void t7(void)_task_ 8
{
 while(1)
 {
  P27 = !P27;
  os_wait(K_TMO,25,0);
 }
}
```

3.10.6　案例分析

在8051系列微控制器中基于RTX-51 Tiny进行多任务并发程序设计之前，需要完成3个基本准备工作，即Keil中配置实用RTX-51 Tiny、工程代码中包含头文件RTX-51TNY.h以及工程中添加操作系统的配置CONF_TNY.A51。

首先，在工程目标名称上右击，选择Options for Target 'Target 1'命令，如图3-17所示。

在如图3-18所示的对话框中，在Target选项卡Operating下拉列表框中选择RTX-51 Tiny，即可使用嵌入式实时操作系统RTX-51精简版。

为了配置的有效和正确，必须将RTX-51的配置文件复制到工程目录下，并加入到工程

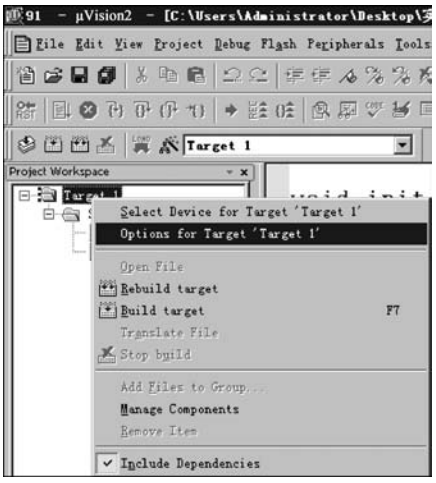

图 3-17 工程设置

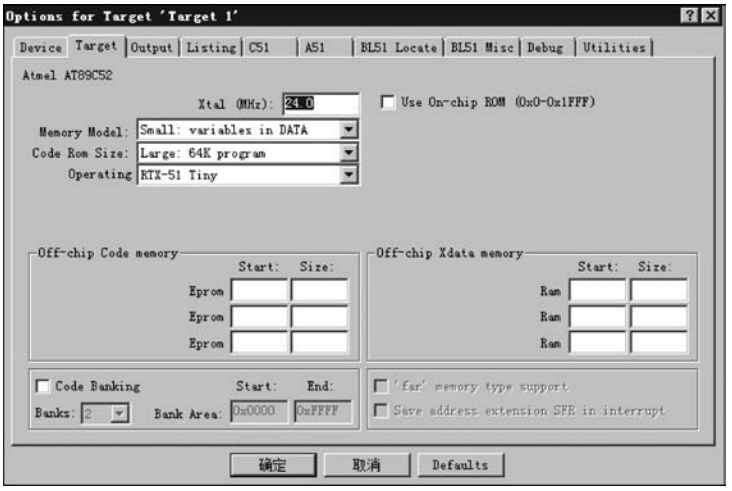

图 3-18 配置选项卡 Target

中。通过配置文件的设置来定制 RTX-51 的配置。CONF_TNY. A51 主要的配置参数有 INT_CLOCK 和 TIMESHARING。

INT_CLOCK 指定定时器产生中断前的指令周期数,默认值为 10 000。该参数用于计算定时器所设初值,即 65 536－INT_CLOCK。

TIMESHARING 是循环设置参数,默认的值是 5。该参数指定每个任务在循环任务切换前运行的滴答数,默认的值 5 表示多任务并发执行中每个任务切换的时间片时间长度是 5 个滴答。TIMESHARING 为 0 表示禁止循环任务切换,多任务并发执行采用挂起函数和唤醒函数切换。RTX-51 中一个滴答为 10 000 个机器周期,一个机器周期等于 12 个时钟周期。本案例禁止循环任务分时切换,即 TIMESHARING 和 INT_CLOCK 设置如下:

```
INT_CLOCK   EQU 10000;
TIMESHARING EQU  0;
```

本案例需要建立 9 个任务：初始化任务和 8 个 LED 闪烁任务，在初始化任务中建立 8 个 LED 闪烁任务，之后删除自身。使用 OS_WAIT()函数等待超时进行任务切换，修改 CONF_TNY.A51 中的 TIMESHARING 禁止循环人为切换。程序中使用的挂起任务函数 OS_WAIT()及其参数说明：

```
char os_wait(
unsigned char event_set,                    //要等待的事件
unsigned char ticks,                        //要等待的滴答数
unsigned int  dummy);                       //无用参数
```

该函数挂起当前任务，并等待一个或几个事件，如时间间隔、超时或从其他任务和中断发来的信号。参数 event_set 指定要等待的事件，事件类型如表 3-2 所示，可以是表中几种事件的组合。ticks 表示要等待的时间长度。本案例用到的等待事件是时间间隔 K_IVL。

表 3-2　3 种等待事件

事　　件	描　　述
K_IVL	等待滴答值为单位的时间间隔
K_SIG	等待一个信号
K_TMO	等待一个以滴答值为单位的超时

事件可以用竖线符(|)进行逻辑或。例如，K_TMO|K_SIG 指定任务等待一个超时或者一个信号。

ticks 参数指定要等待的时间间隔事件(K_IVL)或超时事件(K_TMO)的定时器滴答数。参数 dummy 是为了提供与 RTX-51 Full 的兼容性而设置的，在 RTX-51 Tiny 中并不使用。

函数返回值：当有一个指定的事件发生时，任务进入就绪态。任务恢复执行时，由返回的常数指出使任务重新启动的事件，返回值类型如表 3-3 所示。

表 3-3　返回值类型

返　回　值	描　　述
RDY_EVENT	任务的就绪标志位是被 os_set_ready 或 isr_set_ready 置位的
SIG_EVENT	收到一个信号
TMO_EVENT	超时完成，或时间间隔到
NOT_OK	event_set 参数的值无效

案例中关键程序原理分析：

(1) 案例中的每个任务实际花很短时间，即执行完 P2x=!P2x 后，该任务就挂起。因此 8 个任务瞬间都挂起了，等待间隔时间到了之后就被唤醒。

(2) 8 个任务瞬间都挂起了，但还是有先后次序的，有很短的时间差。

(3) 8 个任务挂起后等待间隔时间到了之后被唤醒，每个任务按照 OS_WAIT()函数中的定时时间长短依次被唤醒。

(4) 如果两个任务等待的定时时间相同，如两个定时都是等待 25 个滴答的任务，它们不会在同一时刻被唤醒，因为各个任务的挂起时间不同，有很小的差别。

(5) 定时 25、50 个滴答的任务也不会同时唤醒，因为挂起的时刻不一样。