的计算公式如下：

$$TCNT1 = 65536 - F\_CPU/分频 \times 定时长度(s)$$

上述公式中减号后面的部分计算出来的值必须在 0～65535 以内。

本例其他设计类似于 3.4 节 LED 模拟交通灯的案例，大家可以自行比对阅读。

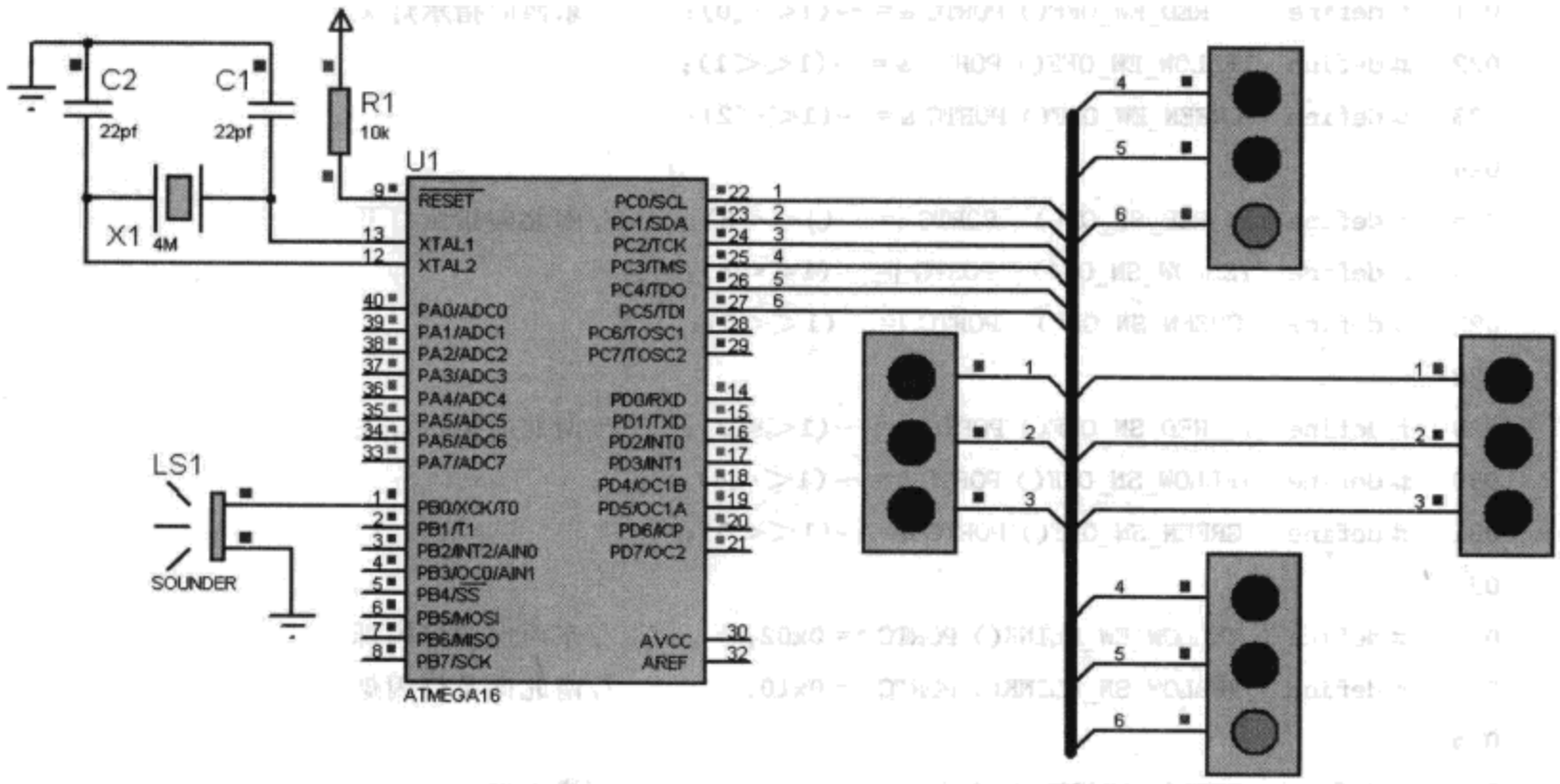


图 3-19　TIMER1 控制交通指示灯

2. 实训要求

① 用 T0 定时器重新设计本例，实现相同的显示效果。

② 重新调整切换与闪烁时间，实现完整的交通指示灯仿真效果。

3. 源程序代码

```
001  //-------------------------------------------------------------------
002  //  名称：定时器 T1 控制交通指示灯
003  //-------------------------------------------------------------------
004  //  说明：东西向绿灯亮 5 s 后，黄灯闪烁，闪烁 5 次后亮红灯
005  //        红灯亮后，南北向由红灯变为绿灯，5 s 后南北向黄灯闪烁
006  //        闪烁 5 次后亮红灯，东西向绿灯亮，如此往复。
007  //        本例将时间设得较短是为了调试的时候能较快地观察到运行效果
008  //
009  //-------------------------------------------------------------------
010  #define F_CPU   4000000UL
011  #include <avr/io.h>
012  #include <avr/interrupt.h>
013  #include <util/delay.h>
014  #define INT8U    unsigned char
015  #define INT16U   unsigned int
016
```

```
017  #define     RED_EW_ON()   PORTC |=   (1<<0);     //东西向指示灯开
018  #define  YELLOW_EW_ON()   PORTC |=   (1<<1);
019  #define   GREEN_EW_ON()   PORTC |=   (1<<2);
020
021  #define     RED_EW_OFF() PORTC &= ~(1<<0);     //东西向指示灯关
022  #define  YELLOW_EW_OFF() PORTC &= ~(1<<1);
023  #define   GREEN_EW_OFF() PORTC &= ~(1<<2);
024
025  #define     RED_SN_ON()   PORTC |=   (1<<3);     //南北向指示灯开
026  #define  YELLOW_SN_ON()   PORTC |=   (1<<4);
027  #define   GREEN_SN_ON()   PORTC |=   (1<<5);
028
029  #define     RED_SN_OFF() PORTC &= ~(1<<3);     //南北向指示灯关
030  #define  YELLOW_SN_OFF() PORTC &= ~(1<<4);
031  #define   GREEN_SN_OFF() PORTC &= ~(1<<5);
032
033  #define   YELLOW_EW_BLINK() PORTC ^= 0x02;       //东西向黄灯闪烁
034  #define   YELLOW_SN_BLINK() PORTC ^= 0x10;       //南北向黄灯闪烁
035
036  #define   BEEP() (PORTB ^= 0x01)                 //蜂鸣器
037
038  //延时倍数,闪烁次数,操作类型变量
039  INT8U Time_Count = 0, Flash_Count = 0, Operation_Type = 1;
040  //-------------------------------------------------------------------
041  // 主程序
042  //-------------------------------------------------------------------
043  int main()
044  {
045      DDRB   = 0xFF;  PORTC = 0xFF;                  //配置输出端口
046      DDRC   = 0xFF;  PORTC = 0x00;
047      TCCR1B = 0x03;                                 //T1 预设分频:64
048      TCNT1   = 65536 - F_CPU/64.0 * 0.5;            //晶振 4 MHz,0.5 s 定时初值
049      TIMSK   = _BV(TOIE1);                          //允许 T1 定时器溢出中断
050      sei();                                         //开中断
051      while (1);
052  }
053
054  //-------------------------------------------------------------------
055  // 黄灯警报声音输出
056  //-------------------------------------------------------------------
057  void Yellow_Light_Alarm()
058  {
```

```
059     INT8U i;
060     for (i = 0; i<100; i ++ )
061     {
062         BEEP(); _delay_us(380);
063     }
064 }
065
066 //-------------------------------------------------------------------
067 // T1 定时器溢出中断服务程序(控制交通指示灯切换显示)
068 //-------------------------------------------------------------------
069 ISR (TIMER1_OVF_vect)
070 {
071     TCNT1 = 65536 - F_CPU/64.0 * 0.5;                    //重装 0.5 s 定时初值
072     switch (Operation_Type)
073     {
074         case 1:  //东西向绿灯与南北向红灯亮,5 s 后绿灯灭
075                  RED_EW_OFF(); YELLOW_EW_OFF(); GREEN_EW_ON();
076                  RED_SN_ON();  YELLOW_SN_OFF(); GREEN_SN_OFF();
077                  //5 s 后切换操作 (0.5 s * 10 = 5 s)
078                  if ( ++ Time_Count != 10) return;
079                  Time_Count = 0;
080                  Operation_Type = 2;                     //下一操作
081                  break;
082
083         case 2:  //东西向绿灯灭,黄灯开始闪烁
084                  Yellow_Light_Alarm();
085                  GREEN_EW_OFF();
086                  YELLOW_EW_BLINK();
087                  //闪烁 5 次
088                  if ( ++ Flash_Count != 10) return;
089                  Flash_Count = 0;
090                  Operation_Type = 3;                     //下一操作
091                  break;
092
093         case 3:  //东西向红灯与南北向绿灯亮
094                  RED_EW_ON();  YELLOW_EW_OFF(); GREEN_EW_OFF();
095                  RED_SN_OFF(); YELLOW_SN_OFF(); GREEN_SN_ON();
096                  //南北向绿灯亮 5 s 后切换(0.5 s * 10 = 5 s)
097                  if ( ++ Time_Count != 10) return;
098                  Time_Count = 0;
099                  Operation_Type = 4;                     //下一种操作类型
100                  break;
```

```
101
102             case 4:  //南北向绿灯灭,黄灯开始闪烁
103                      Yellow_Light_Alarm();
104                      GREEN_SN_OFF();
105                      YELLOW_SN_BLINK();
106                      //闪烁 5 次
107                      if ( ++ Flash_Count != 10) return;
108                      Flash_Count = 0;
109                      Operation_Type = 1;                    //回到第一种操作
110         }
111 }
```

3.20 TIMER1 与 TIMER2 控制十字路口秒计时显示屏

本例运行时,东西向蓝色数码管与南北向红色数码管同步倒计时,若干秒后交换,如此往复。在倒计时过程中,如果仅剩下 5 s 时,系统会发现报警提示声音。本例同时启用了 2 个定时器 T/C1 和 T/C2,其中 16 位的 T/C1 定时器负责递减秒数及切换方向,8 位的 T/C2 定时器负责刷新数码管显示。本例电路及部分运行效果如图 3－20 所示。

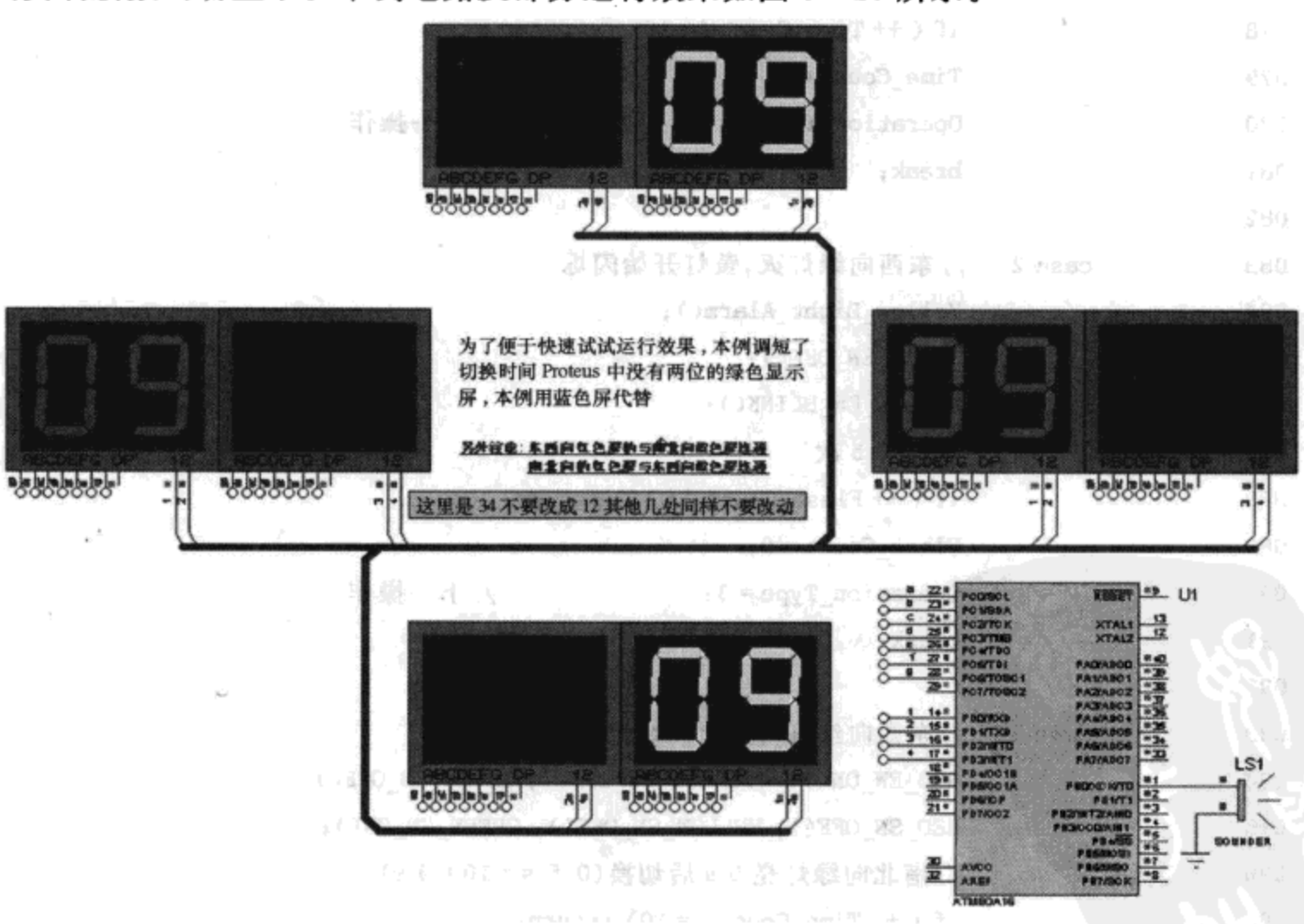


图 3－20　TIMER1 与 TIMER2 控制十字路口秒计时显示屏

1. 程序设计与调试

本例同时启用 16 位的 T/C1 和 8 位的 T/C2 定时器。以下代码分别设置了 T/C1 与 T/C2的分频比及定时初值,定时中断屏蔽寄存器 TIMSK 设置为同时允许 T/C1 和 T/C2 定时溢出中断。

```
TCCR1B = 0x03;                          //T1 预设分频:64
TCNT1 = 65536 - F_CPU/64.0 * 1.0;       //晶振 4 MHz,1 s 定时初值
TCCR2 = 0x04;                           //T2 预设分频:64
TCNT2 = 256 - F_CPU/64.0 * 0.004;       // 4 MHz 系统时钟,0.004 s 定时初值
TIMSK = _BV(TOIE1) | _BV(TOIE2);        //同时允许 T1、T2 定时器溢出中断
```

T/C1 定时溢出中断每秒触发一次,中断程序控制秒数递减,并在仅剩下 5 s 时开始发出警报声音。T/C2 定时溢出中断程序以 4 ms 周期刷新数码管显示。该中断程序中根据方向的切换,控制(0、1)位或(2、3)位数码管的刷新显示。

2. 实训要求

① 进一步完善本例设计,仿真十字路口计时屏的切换效果。

② 重新用 T/C0 和 T/C2 定时器改编本例。

3. 源程序代码

```
001 //-------------------------------------------------------------------
002 //  名称:TIMER1 与 TIMER2 控制十字路口秒计时显示屏
003 //-------------------------------------------------------------------
004 //  说明:本例运行时,东西向蓝色数码管与南北向红色数码管同步倒计时,
005 //        若干秒后交换,如此反复。
006 //        本例使用的两个定时器中,T/C1 定时器负责递减秒数及切换方向,
007 //        T/C2 定时器负责刷新显示数码管
008 //
009 //-------------------------------------------------------------------
010 #define F_CPU   4000000UL
011 #include <avr/io.h>
012 #include <avr/interrupt.h>
013 #include <util/delay.h>
014 #define INT8U   unsigned char
015 #define INT16U  unsigned int
016
017 #define BEEP() (PORTB ^= 0x01)
018
019 //设置最大秒数为 12 s,每 12 s 将切换通行方向
020 //这里为了能尽快观察到运行效果而将该值设得较小
021 #define MAX_SECOND 12
022
023 //通行方向类型(东西/南北)
024 enum TRAFFIC_DIRECTION {EW,SN} Current_Direct;
```

```
025
026  //0～9 的数码管段码
027  const INT8U SEG_CODE[] = {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90};
028
029  //当前剩余秒数(两位)及秒显示缓冲
030  int   Remain_Second;
031  INT8U Second_Display_Buffer[] = {0,0};
032  //-------------------------------------------------------------------
033  // 根据剩余秒数 Remain_Second 刷新秒显示缓冲
034  //-------------------------------------------------------------------
035  void Refresh_Second_Display_Buffer()
036  {
037      Second_Display_Buffer[0] = Remain_Second/10;
038      Second_Display_Buffer[1] = Remain_Second % 10;
039  }
040
041  //-------------------------------------------------------------------
042  // 警报声输出函数
043  //-------------------------------------------------------------------
044  void Alarm()
045  {
046      INT8U i;
047      for (i = 0 ; i< 80; i ++)
048      {
049         BEEP(); _delay_us(300);
050      }
051  }
052
053  //-------------------------------------------------------------------
054  // 主程序
055  //-------------------------------------------------------------------
056  int main()
057  {
058      Current_Direct = EW;                    //初始通行方向设为东西方向
059      Remain_Second =   MAX_SECOND;           //初始剩余秒数为最大秒数
060      Refresh_Second_Display_Buffer();        //刷新秒显示缓冲
061
062      TCCR1B = 0x03;                          //T1 预设分频:64
063      TCNT1   = 65536 - F_CPU/64.0 * 1.0;     //晶振 4 MHz, 1 s 定时初值
064      TCCR2   = 0x04;                         //T2 预设分频:64
065      TCNT2   = 256 - F_CPU/64.0 * 0.004;     //晶振 4 MHz,0.004 s 定时初值
066      TIMSK   = _BV(TOIE1) | _BV(TOIE2);      //允许 T1、T2 定时器溢出中断
067      DDRB = 0xFF;                            //配置输出端口
```

```
068       DDRC = 0xFF;
069       DDRD = 0xFF;
070       sei();                                          //开中断
071       while (1);
072   }
073
074   //-------------------------------------------------------------------
075   // T1 定时器溢出中断程序,控制倒计时
076   //-------------------------------------------------------------------
077   ISR (TIMER1_OVF_vect)
078   {
079       //重装 1 s 定时初值
080       TCNT1    = 65536 - F_CPU/64.0 * 1.0;
081       //计时值递减,递减到终点后重新从最大秒数 MAX_SECOND 开始,
082       //同时切换通行方向
083       if (-- Remain_Second == -1 )
084       {
085          Remain_Second = MAX_SECOND;
086          Current_Direct = Current_Direct == EW ? SN : EW;
087       }
088       //刷新秒显示缓冲
089       Refresh_Second_Display_Buffer();
090       //剩余时间在 5 s 以内时输出声音
091       if (Remain_Second <= 5) Alarm();
092   }
093
094   //-------------------------------------------------------------------
095   // T2 定时器溢出中断程序,控制数码管扫描显示
096   //-------------------------------------------------------------------
097   ISR (TIMER2_OVF_vect)
098   {
099       //当前待显示位索引(注意设为静态变量)
100       static INT8U i = 0;
101       //位间延时 4 ms
102       TCNT2    = 256 - F_CPU/64.0 * 0.004;
103       //先关闭段码
104       PORTC = 0xFF;
105       //输出数码管段码
106       PORTC = SEG_CODE[ Second_Display_Buffer[i] ];
107       //根据当前方向输出(0,1)位或(2,3)位的扫描码
108       if (Current_Direct == EW)
109          PORTD = _BV(i);
110       else
```

```
111            PORTD = _BV(i + 2);
112        //i 在 0,1 之间切换(扫描输出的数位将是 0,1 或 2,3)
113        i = (i == 0) ? 1 : 0;
114    }
```

3.21 用工作于计数方式的T/C0实现100以内的脉冲或按键计数

本例T/C0时钟由T0(PB0)引脚输入。利用这一特点,本例实现了按键和脉冲计数及显示功能,学习调试要点在于掌握T/C0工作于计数方式的程序设计方法。案例电路及部分运行效果如图3-21所示。

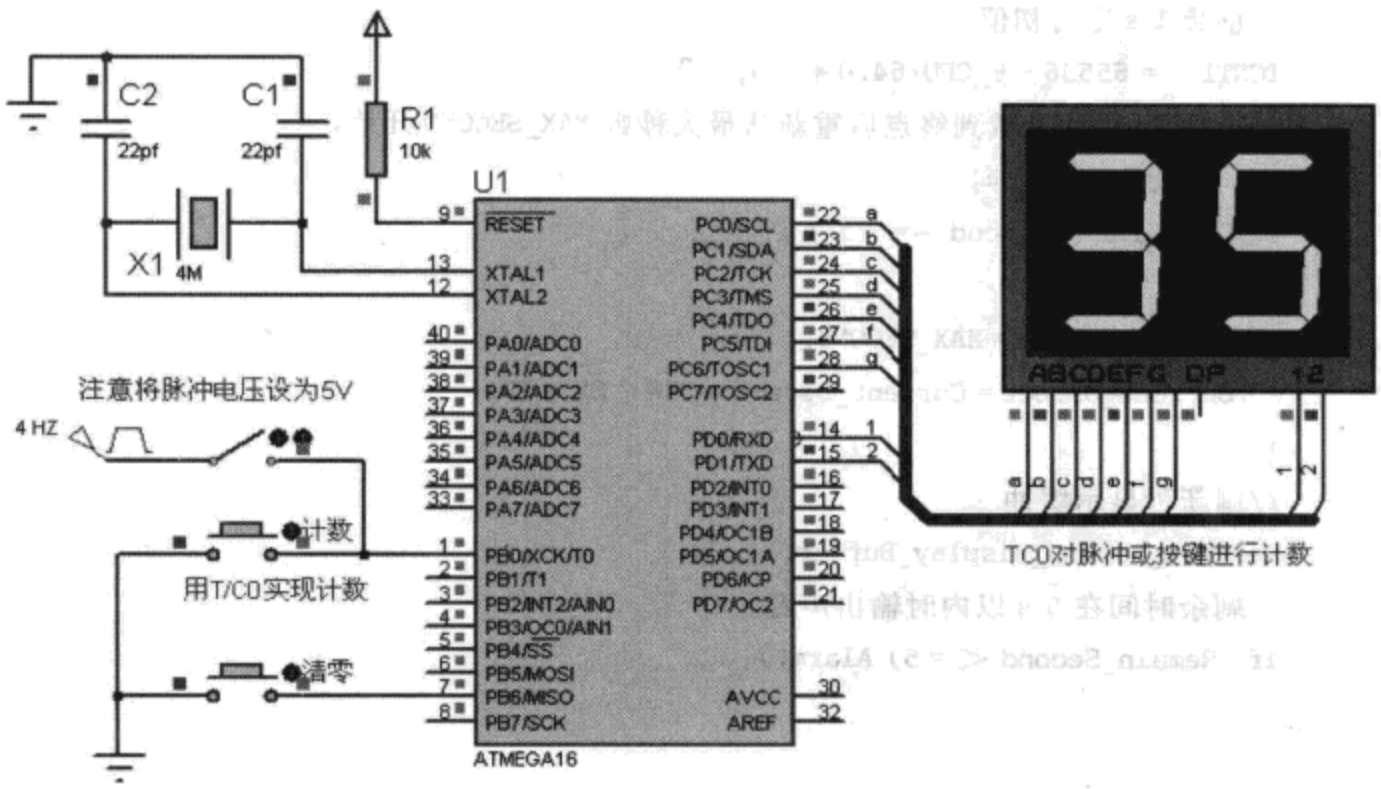


图3-21 用工作于计数方式的T/C0实现100以内的脉冲或按键计数

1. 程序设计与调试

此前案例中使用的T/C0、T/C1、T/C2均工作于定时方式,本例中T/C0工作于计数方式,其区别在于此前TCNTx的计数时钟由系统时钟分频后提供。所提供的时钟具有固定频率(周期),因而TCNTx的计数可换算成计时。

本例中的T/C0工作于计数方式,TCNT0的计数时钟不再由系统时钟分频提供,而是由来自T0(PB0)引脚的外部信号提供,这些信号可能是无固定周期的(例如本例的按键计数操作),也可能是有固定周期的(例如本例中的外部输入计数脉冲)。

主程序中TCCR0=0x06,该行代码将TCCR0的低3位(CS02、CS01、CS00)设为110,它使得TCNT0的计数时钟来自于T0(PB0)引脚,且为下降沿触发。主程序中设置TCCR0后,T0(PB0)引脚每次由高电平到低电平的跳变都将使TCNT0递增1。

主程序中的while循环完成了清零控制,显示上限控制,数码管刷新显示控制等操作。

2. 实训要求

① 利用T/C0计数溢出中断实现计数,每次按键或脉冲输入信号的下降沿使计数变量累

加，程序中注意要将 TCNT0 初值设为 255，这样才能使得每次下降沿累加 TCNT0 都会导致溢出，触发 T/C0 计数溢出中断，在中断程序中完成对计数变量的累加。显然，如果将 TCNT0 设为 254 的话，每 2 次脉冲或按键输入才会实现 1 次计数变量累加。

② 改用 4 位集成式数码管，实现更大范围的计数操作。

3. 源程序代码

```
01 //-------------------------------------------------------------------
02 //  名称：用工作于计数方式的 TC0 实现 100 以内的脉冲或按键计数
03 //-------------------------------------------------------------------
04 //  说明：TC0 工作于计数器方式且设为下降沿触发，外部的每次按键或脉冲
05 //        出现的下降沿都将导致 TCNT0 累加计数一次，TCNT0 的计数值将被实时
06 //        刷新并显示在两位数码管上
07 //
08 //-------------------------------------------------------------------
09 #define  F_CPU   4000000UL              //4 MHz 晶振
10 #include <avr/io.h>
11 #include <util/delay.h>
12 #define INT8U   unsigned char
13 #define INT16U  unsigned int
14
15 //清除键定义
16 #define Clear_Key_DOWN() ((PINB & 0x40) == 0x00)
17
18 //0～9 的数字编码
19 const INT8U SEG_CODE[] = {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
20 //-------------------------------------------------------------------
21 // 在两位数码管上显示计数值
22 //-------------------------------------------------------------------
23 void Show_Count_ON_DSY()
24 {
25     PORTD = 0xFF;
26     PORTC = SEG_CODE[TCNT0/10];
27     PORTD = 0xFE;
28     _delay_ms(2);
29     PORTD = 0xFF;
30     PORTC = SEG_CODE[TCNT0 % 10];
31     PORTD = 0xFD;
32     _delay_ms(2);
33 }
34
35 //-------------------------------------------------------------------
36 // 主程序
```

```
37  //-------------------------------------------------------------------
38  int main()
39  {
40      DDRC = 0xFF; PORTD = 0xFF;              //配置输出端口
41      DDRD = 0xFF; PORTD = 0xFF;
42      DDRB = 0x00; PORTB = 0xFF;              //配置输入端口
43
44      TCCR0 = 0x06;                           //T0 工作于计数方式,下降沿触发
45      TCNT0 = 0x00;                           //设置计数初值
46      while(1)
47      {
48          if(Clear_Key_DOWN()) TCNT0 = 0;     //如果按下清零键则将 TCNT0 重新置 0
49          if (TCNT0 >= 100) TCNT0 = 0;        //计数值限制在 100 以内
50          Show_Count_ON_DSY();                //持续刷新显示
51      }
52  }
```

3.22 用定时器设计的门铃

本例用定时器控制蜂鸣器模拟发出“叮咚”的门铃声,其中“叮”的声音用较短定时形成较高频率,“咚”的声音用较长定时形成较低频率。仿真电路中加入了虚拟示波器,按下按键时除听到门铃声外,还会从示波器中观察到2种不同频率的波形。本例电路及输出的部分波形如图3-22所示。

1. 程序设计与调试

主程序控制变量soundDelay分别取值-700与-1000,它影响定时器溢出中断程序中TCNT1的取值,从而控制输出2种不同频率的声音。

定时器初始定时为700 μs,按键使能定时器溢出中断后,在前300 ms时间内的每次中断触发时间间隔都是0.7 ms(初值为-700),这使得DoorBell()可以输出1000/(0.7×2)≈714 Hz的声音频率,在后500 ms内的每次中断触发时间间隔为1 ms(初值为-1000),DoorBell()输出1000/(1.0×2)=500 Hz声音频率。按下按键后的“叮咚”声正是这样产生的。

在一次“叮咚”声音输出完成后,TMISK=0x00使定时器溢出中断被禁止,声音输出也随即停止。

设置T/C1定时器初值时所使用的公式是:

$$TCNT1 = 65536 - F\_CPU/分频 \times 定时长度(s)$$

本例最初定时长度设为700 μs,即0.0007 s,代入公式可得:TCNT1=65536-1000000/1×0.0007=65536-700。

对于语句TCNT1=65536-700,在编写程序时还可直接写成TCNT1=-700,因为65536即17位二进制数10000000000000000,其最高位为1,其余16位为0。对于16位的寄存器,65536与0是相等的,因此有TCNT1=0-700=-700,实际存入TCNT1是-700的补

码，将 700 进行二进制数分解可得 0000001010111100，将其取反加 1 得到 1111110101000100，即 64836，而 65536－700 也等于 64836。

图 3－22　用定时器设计的门铃

可见，对于 16 位的 T/C1 定时器，在 1 MHz 时钟频率、分频为 1 时，要实现 x 微秒(μs)的延时，可直接使用语句 TCNT1＝－ x，当然，x 的取值应在 0～65535 范围之内。同样，对于 8 位的 T/C0 与 T/C2 定时器，在同样时钟与分频比下，设置延时初值时也可以有 TCNT0＝－ x 和 TCNT2＝－ x。当然，这里 x 取值应在 0～255 范围之内。

2. 实训要求

① 修改定时器初值－700 和－1000，并改变 2 种频率声音的输出时长，再观察输出效果。

② 重新编写程序，用定时器控制输出另一种包含 3 个不同频率的门铃声音效果。

3. 源程序代码

```
01  //-------------------------------------------------------------------
02  //  名称：用定时器设计的门铃
03  //-------------------------------------------------------------------
04  //  说明：按下按键时蜂鸣器发出叮咚的门铃声
05  //
06  //-------------------------------------------------------------------
07  #define F_CPU   1000000UL                    //1 MHz 晶振
```

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#define INT8U   unsigned char
#define INT16U  unsigned int

#define DoorBell() (PORTD ^= 0x01)              //门铃定义
#define Key_DOWN() ((PINB & 0x80) == 0x00)      //按键定义
volatile INT16U soundDelay;                     //2个不同取值分别对应于"叮"、"咚"
//-------------------------------------------------------------
// 主程序
//-------------------------------------------------------------
int main()
{
    DDRB = 0x00; PORTB = 0xFF;                  //配置输入/输出端口
    DDRD = 0xFF;
    TCCR1B = 0x01;                              //T1 预设分频:1
    TCNT1 = -700;                               //1 MHz 时钟,定时初值设为 700 μs
    sei();                                      //开总中断
    while(1)
    {
        if(Key_DOWN())                          //按下 K1 时启动定时溢出中断
        {
            TIMSK = _BV(TOIE1);                 //允许 T1 定时器溢出中断
            soundDelay = -700;                  //700 μs 延时对应输出"叮"的声音
            _delay_ms(400);                     //声音保持 400 ms
            soundDelay = -1000;                 //1000 μs 延时对应输出"咚"的声音
            _delay_ms(600);                     //声音保持 600 ms
            TIMSK = 0x00;                       //禁止 T1 定时器溢出中断
        }
    }
}

//-------------------------------------------------------------
// T/C1 定时器中断程序控制门铃声音输出
//-------------------------------------------------------------
ISR(TIMER1_OVF_vect )
{
    DoorBell();                                 //门铃声音输出
    TCNT1 = soundDelay;                         //按主程序设定的-700或-1000设置延时
}
```

3.23 报警器与旋转灯

本例运行时，按下报警开启按键后系统将发出逼真的警报声音，连接 PC 端口的 LED 中相邻的 3 只将随之不断循环滚动点亮，按下关闭键时警报输出停止，所有 LED 熄灭。本例同时启用了 3 个中断程序。案例电路及部分运行效果如图 3－23 所示。

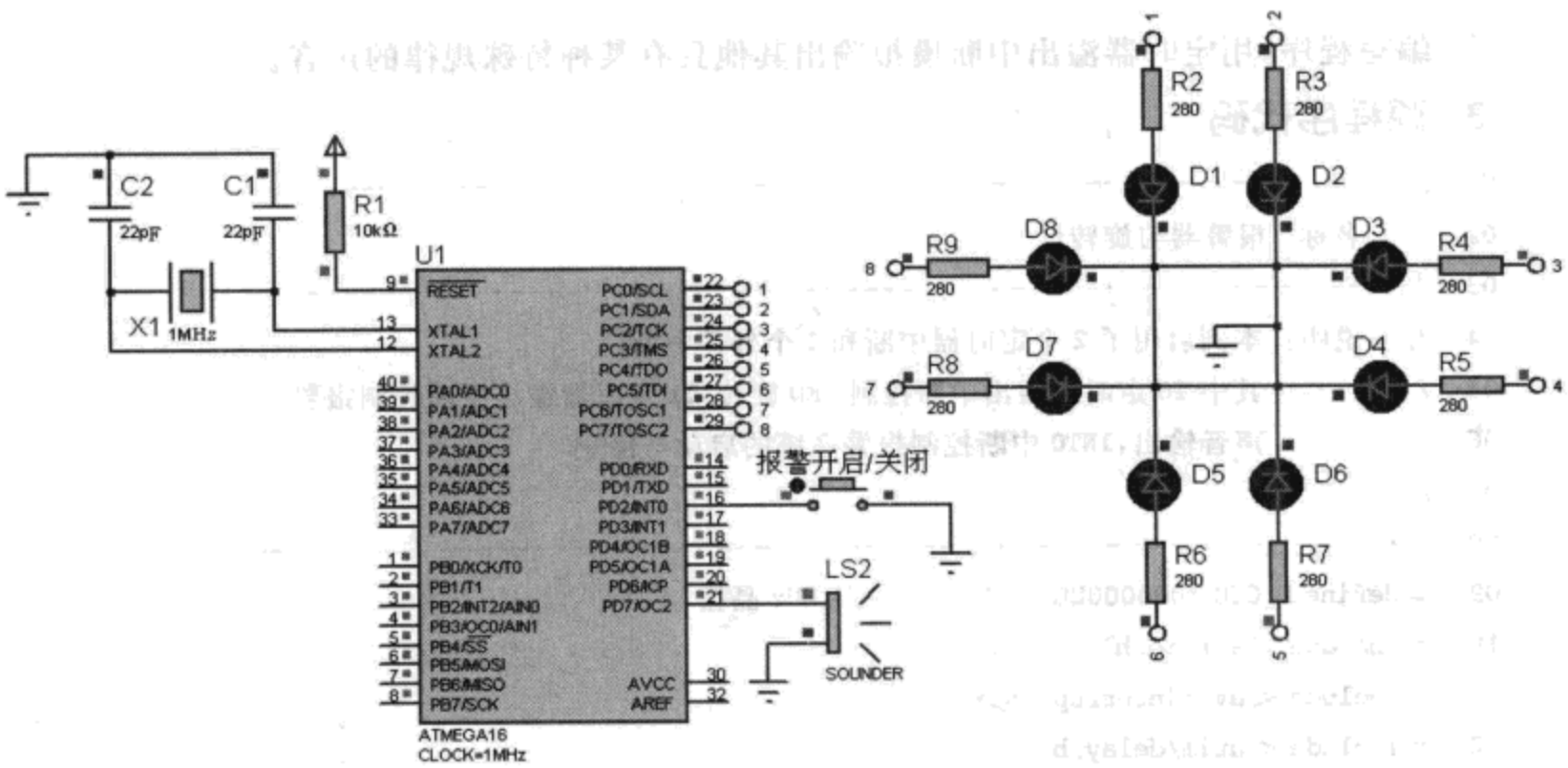


图 3－23 报警器与旋转灯

1. 程序设计与调试

本例同时启用了 INT0 中断、T/C0 溢出中断、T/C1 溢出中断，它们分别负责系统启停控制、LED 旋转滚动显示与警报声音输出。本例的报警声音非常逼真，它模拟了声音频率均匀拉高、还原、再拉高的过程。

本例中 16 位的 T/C1 定时/计数寄存器 TCNT1＝0xFE00＋FRQ，其中 FRQ 由主程序控制在 0x00～0xFF(0～255)之间反复递增循环取值，由于 FRQ 的变化由 main 函数控制，T/C1 溢出中断程序中需要使用该变量，因此要注意在定义 FRQ 时添加 volatile 关键字。

本例程序中 T/C1 溢出中断程序输出的频率范围计算如下：

① TCNT1 计数寄存器取值范围为 0xFE00～0xFEFF，即 65 024～65 279；

② 延时值范围为(65 536－65 024)～(65 536－65 279)，即 512～257；

③ 对于 1 MHz 的时钟，其输出频率范围为 1 000 000/(512×2)～1 000 000/(257×2)Hz，即 976～1 945 Hz。

由于 FRQ 类型为 INT8U，主程序中的 while 循环控制 FRQ 变量由 0x00 持续递增，FRQ 将在 0x00～0xFF 范围内反复循环取值，而 T/C1 中断程序内的 TCNT1 寄存器在每次中断触发时都将获取这个不断变化的 FRQ 值，使得 TCNT1 不断由 0xFE00 向 0xFEFF 循环递增，每次中断触发都使下一次的中断触发时间变得更短，T/C1 中断的触发频率越来越高，中断程序输出的…01010101…序列的频率也越来越高，从而形成了 976～1 945 Hz 频率的平滑递增

输出，案例输出的报警器声音效果非常逼真。

2. 实训要求

① 修改主程序中 while 循环内_delay_ms 函数参数为 1、2、3 或 4 等，重新编译并运行程序，试听输出警报声音的急促程度是否会发生变化。

② 第 5 章有与射击游戏相关的程序，参考该游戏程序修改本例代码，模拟出枪支射击的声音。

③ 编写程序，用定时器溢出中断模拟输出其他具有某种特殊规律的声音。

3. 源程序代码

```
01  //-------------------------------------------------------------------
02  //  名称：报警器与旋转灯
03  //-------------------------------------------------------------------
04  //  说明：本例启用了 2 个定时器中断和 1 个外部中断
05  //        其中 T0 定时器溢出中断控制 LED 旋转，T1 定时器溢出中断控制报警
06  //        声音输出，INT0 中断控制报警系统的启动与停止
07  //
08  //-------------------------------------------------------------------
09  #define F_CPU 1000000UL                  //1 MHz 晶振
10  #include <avr/io.h>
11  #include <avr/interrupt.h>
12  #include <util/delay.h>
13  #define INT8U   unsigned char
14  #define INT16U  unsigned int
15
16  //蜂鸣器输出定义
17  #define SPK() (PORTD ^= _BV(PD7))
18
19  volatile INT8U FRQ = 0x00;          //定时初值循环递增控制频率循环递增(volatile 不可省略)
20  INT8U ON_OFF = 0;                   //开关变量
21  INT8U Pattern = 0xE0;               //旋转灯端口花样初值 11100000
22  //-------------------------------------------------------------------
23  // 主程序
24  //-------------------------------------------------------------------
25  int main()
26  {
27      DDRC = 0xFF;                                //配置 LED 输出端口
28      DDRD = ~_BV(PD2); PORTD = _BV(PD2);         //配置按键输入与蜂鸣器输出端口
29      TCCR0  = 0x05;                              //T0 预设分频：1024
30      TCNT0  = 256 - F_CPU/1024.0 * 0.1;          //1 MHz 时钟，0.1 s 定时初值
31      TCCR1B = 0x01;                              //T1 预设分频：1
32      MCUCR = 0x02;                               //INT0 为下降沿触发
33      GICR  = 0x40;                               //INT0 中断使能
```

```
34      sei();                                  //开中断
35      while (1)
36      {
37          //定时初值循环递增控制频率循环递增
38          // FRQ 在超过 255 溢出后从 0 开始再继续递增
39          FRQ ++ ;
40          //改变延时参数可调整报警声音输出的急促程度(例如 1、2、3、4)
41          _delay_ms(1);
42      }
43  }
44
45  //-------------------------------------------------------------------
46  // 外部中断 0,启停报警器声音和 LED 旋转
47  //-------------------------------------------------------------------
48  ISR (INT0_vect )
49  {
50      ON_OFF = !ON_OFF;          //启停切换
51      if(ON_OFF )
52      {
53         TIMSK |= 0x05;          //开启 2 个定时器中断,分别控制报警器和 LED
54         Pattern = 0xE0;         //11100000,开 3 个灯旋转
55      }
56      else
57      {
58         TIMSK = 0x00;           //关闭所有定时器中断
59         PORTC = 0x00;           //关闭所有 LED
60         PORTD &= ~_BV(PD7);     //在蜂鸣器连接的 PD7 引脚输出低电平
61      }
62  }
63
64  //-------------------------------------------------------------------
65  // T0 定时器中断程序控制 LED 旋转
66  //-------------------------------------------------------------------
67  ISR (TIMER0_OVF_vect )
68  {
69      //重装 0.1 s 计时初值
70      TCNT0   = 256 - F_CPU/1024.0 * 0.1;
71      //以下两行实现 111 的循环左移(高位为 1 时左移后右端补 1,否则直接左移)
72      if (Pattern & 0x80)  Pattern = (Pattern << 1) | 0x01;
73      else                 Pattern << = 1;
74      PORTC = Pattern;            //LED 显示
75  }
76
```

```
77  //-------------------------------------------------------------------
78  // T1 定时器中断控制报警器声音输出
79  //-------------------------------------------------------------------
80  ISR (TIMER1_OVF_vect)
81  {
82      TCNT1 = 0xFE00 + FRQ;       //主程序中 FRQ 的递增导致输出频率递增
83      SPK();
84  }
```

3.24 100000 s以内的计时程序

本例程序运行时，首次按下K1即开始启动精度为0.1 s的计时，计时刷新显示在6位数码管上，再次按下K1时暂停计时，当前计时值保持显示在数码管上，第三次按下K1时计时值归0。本例最大计时为99999.9 s。案例电路及部分运行效果如图3-24所示。

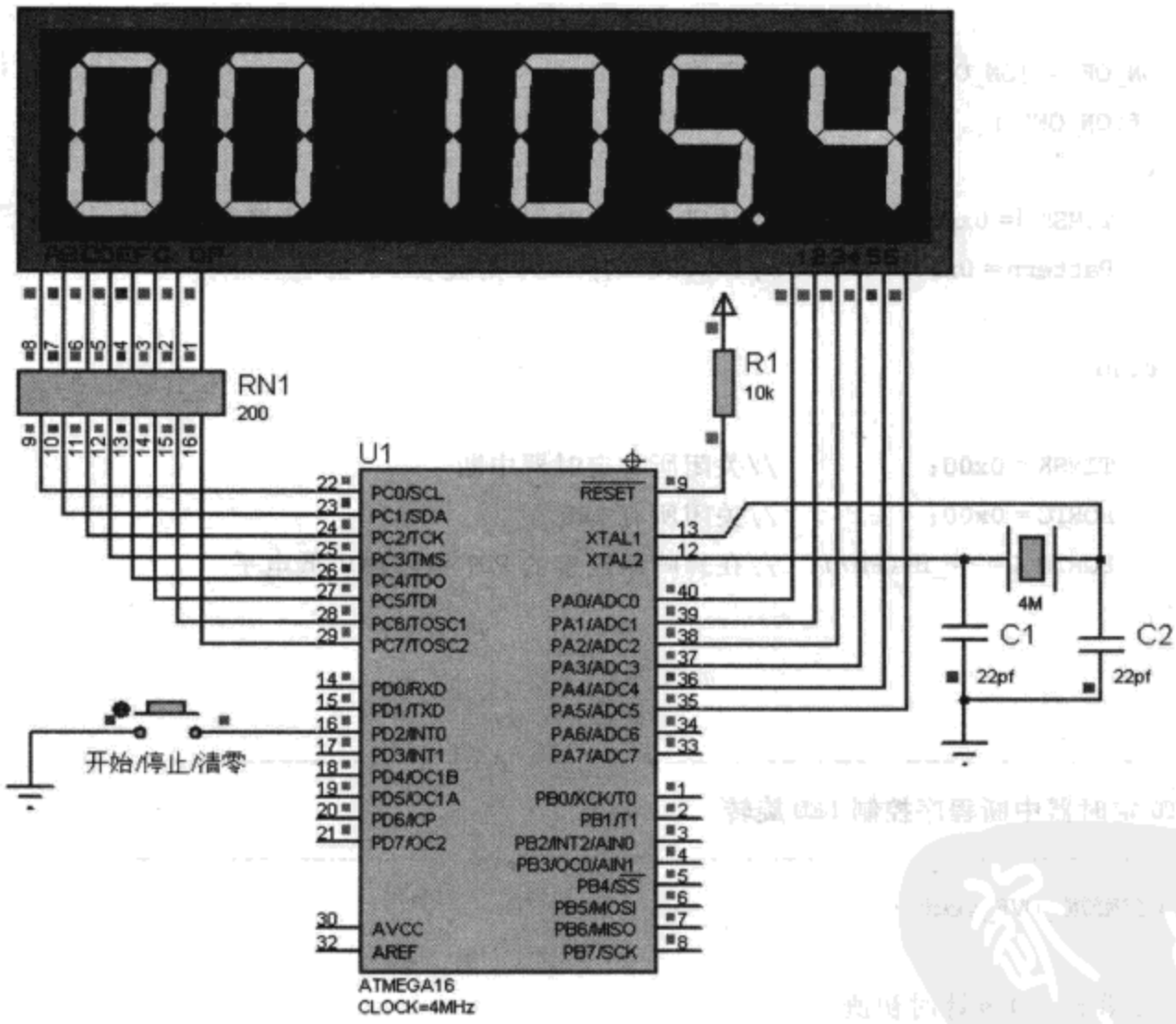


图3-24 100000 s以内的计时程序

1. 程序设计与调试

本例设计与调试要点如下：

① 本例计数(计时)方法与此前案例不同，此前案例的计数值一般是保存在全局变量中，或者是保存在计数寄存器内，需要显示在数码管上时，再将待显示数据分解为多个数位。本例

直接定义了含 6 个元素的数组 Digits_Buffer，定时中断子程序每 0.1 s 对 Digits_Buffer [0]累加，累加到 10 时即进位到下一元素，以此类推。这样设计后的计数值在显示时不需要再用整除(/)和取余(%)进行分解。

② 对于同一按键上实现的 3 种操作，程序中使用变量 KeyOperation 进行标识，每次出现按键中断时递增 KeyOperation，根据不同的 KeyOperation 值完成不同的操作。变量 KeyOperation 可以定义为全局变量，也可以在函数内部定义为静态变量，本例使用的是后一种定义方式，这样可读性更好一些。注意：编写调试程序不可忽略了 static 关键字。

③ 由于主程序中第 47 行已设置 TIMSK 允许定时器溢出中断，按键中断函数中第 89 行只需要设置 TCCR1B 为非 0 的分频比(本例设为 8 分频)即可开始启动计时并每隔 0.1 s 触发中断。停止或清零计时时，第 91 行也只需要将 TCCR1B 设置为 0 分频(无时钟，T/C1 不工作)即可，虽然 TIMSK 仍然允许定时器溢出中断，但由于 T/C1 已无时钟，溢出中断也亦不会发生，0.1 s 的计时亦停止。

④ 案例使用了 6 位集成式七段数码管，在完成 0.1 s 精度计时的同时完成数码管的刷新显示，并实现对小数点的显示控制。由于 Digits_Buffer [0]～Digits_Buffer [5]分别存放的依次是小数位、个位、十位一直到最高位，在 6 位数码管上显示该数组时，循环控制变量 i=0～5，如果用 PORTC=～_BV(i)来输出位码，这会将小数位显示在数码管最左边，而最高位却显示在最右边，因此输出位码的语句应为 PORTC=～_BV(5－i)，凡是共阴数码管均需要添加“～”来输出位码，而_BV 的参数 5－i 显然是起到了将顺序读取的 0～5 号数组元素在数码管上逆向显示的作用。

2. 实训要求

① 修改程序，将计时精度设为 0.01 s，并重新进行调试与仿真运行。

② 在本例中实现两段计时，第一次暂停后再次按下 K1 时，可在前一段时间的基础上再继续计时，最后按下 K1 时才将计时清零。

3. 源程序代码

```
01  //-------------------------------------------------------------------
02  //  名称：100000 s 以内的计时程序
03  //-------------------------------------------------------------------
04  //  说明：在 6 只数码管上完成 00000.0～99999.9 s 计时
05  //
06  //-------------------------------------------------------------------
07  #define  F_CPU   4000000UL                    //4 MHz 晶振
08  #include <avr/io.h>
09  #include <avr/interrupt.h>
10  #include <util/delay.h>
11  #define INT8U   unsigned char
12  #define INT16U  unsigned int
13
14  //共阴数码管 0～9 的数字段码
15  const INT8U SEG_CODE[] =
```

```
16  { 0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F };
17
18  //6 只数码管上显示的数字缓冲
19  INT8U Digits_Buffer[] = {0,0,0,0,0,0};
20  //-------------------------------------------------------------------
21  // 主程序
22  //-------------------------------------------------------------------
23  void Show_Count_ON_DSY()
24  {
25      INT8U i;
26      for (i = 0; i <= 5; i ++ )
27      {
28          PORTC = 0x00;                          //暂时关闭段码
29          PORTA = ~_BV(5 - i);                   //输出位码
30          PORTC = SEG_CODE[ Digits_Buffer[i] ];  //输出段码
31          if (i == 1) PORTC |= 0x80;             //在个位数上加小数点
32          _delay_ms(3);                          //位间延时
33      }
34  }
35
36  //-------------------------------------------------------------------
37  // 主程序
38  //-------------------------------------------------------------------
39  int main()
40  {
41      DDRA = 0xFF; PORTA = 0xFF;             //配置端口
42      DDRC = 0xFF; PORTC = 0xFF;
43      DDRD = 0x00; PORTD = 0xFF;
44      MCUCR = 0x02;                          //INT0 为下降沿触发
45      GICR  = 0x40;                          //INT0 中断使能
46      TCNT1 = 65536 - F_CPU/8 * 0.1;         //0.1 s 定时(T1 预设 8 分频在 INT0 中断中完成)
47      TIMSK = _BV(TOIE1);                    //允许 T1 定时器溢出中断
48      sei();                                 //开中断
49      while(1) Show_Count_ON_DSY();          //持续刷新显示
50  }
51
52  //-------------------------------------------------------------------
53  // T1 定时器溢出中断实现计时
54  //-------------------------------------------------------------------
55  ISR (TIMER1_OVF_vect )
56  {
57      INT8U i;
```

```
58      TCNT1 = 65536 - F_CPU/8 * 0.1;          //重设 0.1 s 定时初值
59      Digits_Buffer[0] ++ ;                   //0.1 s 位累加
60      for (i = 0;i <= 5; i ++ )               //进位处理
61      {
62          if(Digits_Buffer[i] == 10)
63          {
64              Digits_Buffer[i] = 0;
65              //如果是 0～4 位则分别向高一位进位
66              if(i != 5 ) Digits_Buffer[i + 1] ++ ;
67          }
68          //循环过程中如果某个低位没有进位,则循环可提前结束
69          else break;
70      }
71  }
72
73  //-------------------------------------------------------------
74  // INT0 中断函数完成 K1 按键的 3 种操作
75  //-------------------------------------------------------------
76  ISR (INT0_vect)
77  {
78      INT8U i;
79      //按键操作标识:0 停止, 1 开始, 2 暂停
80      static INT8U KeyOperation = 0;
81      //每次按键时,操作标识在 0,1,2 中循环选择
82      if ( ++ KeyOperation == 3) KeyOperation = 0;
83
84      switch (KeyOperation)
85      {
86        case 0: TCCR1B = 0x00;                //停止(清零)
87                for (i = 0; i<6; i ++ ) Digits_Buffer[i] = 0;
88                break;
89        case 1: TCCR1B = 0x02;                //开始计时(提供 8 分频时钟)
90                break;
91        case 2: TCCR1B = 0x00;                //暂停计时
92                break;
93      }
94  }
```

3.25 用 TIMER1 输入捕获功能设计的频率计

本例 T/C1 工作于一般模式下的定时器方式,外部被测信号从 ICP(PD6)输入,程序利用 T/C1 的输入捕获(Input Capture)功能,在按下 K1 按键时,通过检测 2 次捕获的计数差值计

算出被测信号频率，并显示在4位数码管上。本例电路及部分运行效果如图3-25所示。

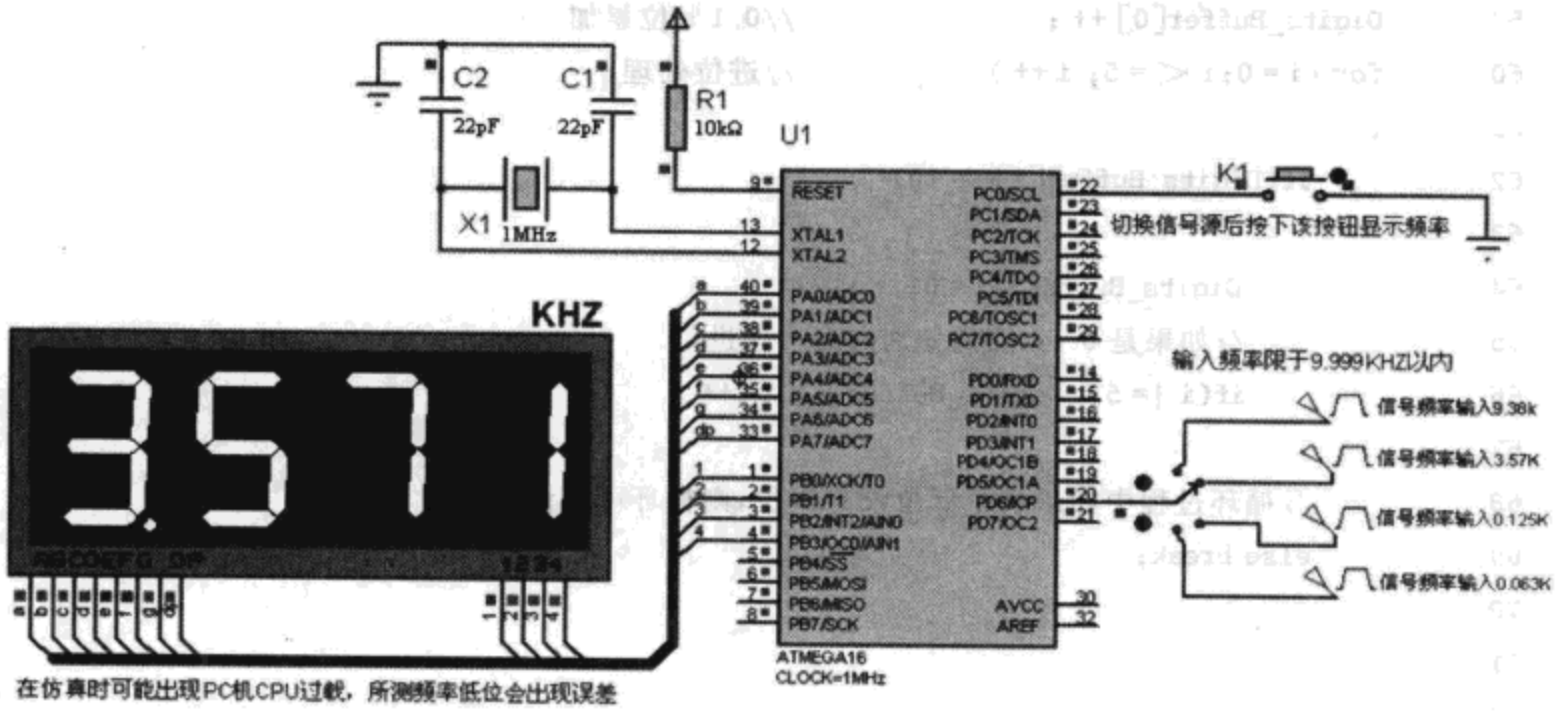


图3-25　用TIMER1输入捕获功能设计的频率计

1. 程序设计与调试

本例T/C1工作于定时器方式，TCNT1计数脉冲由系统时钟分频后提供，本例系统时钟为1 MHz，TCCR1B设置分频比为1，计数时钟仍为1 MHz，在该计数时钟下，每1 μs时间TCNT1计数1次。

主程序通过设置TCCR1B＝_BV(ICNC1)｜_BV(ICES1)，设置了输入捕获噪音消除(ICNC1＝1)位和ICP上升沿触发输入捕获(ICES1＝1)位。由于PD6(ICP)输入捕获引脚输入信号的每次上升沿都将触发捕获，在捕获发生时，当前计数寄存器TCNT1的计数值被复制到输入捕获寄存器ICR1中；在连续2次ICP引脚上升沿触发捕获中断时，中断服务程序通过计算2次所读取的ICR1的差值，即可得出相邻2次TCNT1的计数差值。本例的输入捕获中断向量名称为TIMER1_CAPT_vect。

在本例配置下，TCNT1的每次计数为1 μs，如果差值为8，则该输入信号周期为8 μs，倒数处理后即可得到信号频率。

因本例仿真电路中放入了多路外部信号源，Proteus仿真时可能会提示PC机CPU过载，仿真未能在实时模式下运行，这会影响所检测频率的精度。在运行本例检测外部信号频率时，所显示出来的频率低位数可能会出现误差。

2. 实训要求

① 进一步改进本例，使每次按下K1按键后能重复进行6次捕获，求得3个捕获差值后计算平均频率，以提高系统检测结果的精度。

② 设置T/C1工作于计数方式，重新编写本例程序实现频率检测。

3. 源程序代码

```
01  //-------------------------------------------------------------
02  //  名称：用 TIMER1 输入捕获功能设计的频率计
03  //-------------------------------------------------------------
```

```
04  //  说明：本例运行时，切换不同的频率输入，然后按下 K1 按键，数码管上将
05  //        显示当前频率值。2 次捕获的时间差值即为当前输入频率的周期，
06  //        周期倒数即可得到当前频率
07  //
08  //-------------------------------------------------------------------
09  #define  F_CPU  1000000UL                          //1 MHz 晶振
10  #include <avr/io.h>
11  #include <avr/interrupt.h>
12  #include <util/delay.h>
13  #define INT8U   unsigned char
14  #define INT16U  unsigned int
15
16  //共阴数码管 0～9 的数字编码，最后一位为黑屏
17  const INT8U SEG_CODE[] =
18  {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x00};
19
20  //分解后的待显示数位
21  INT8U Display_Buffer[] = {0,0,0,0};
22
23  //连续 2 次捕获计数变量
24  INT16U CAPi = 0,CAPj = 0;
25  //-------------------------------------------------------------------
26  // 数码管显示频率
27  //-------------------------------------------------------------------
28  void Show_FRQ_ON_DSY()
29  {
30      INT8U i = 0;
31      for (i = 0; i<4; i ++ )
32      {
33          PORTA = 0x00;                              //先暂时关闭段码
34          PORTB = ~_BV(i);                           //发送扫描码
35          PORTA = SEG_CODE[ Display_Buffer[i] ];     //发送数字段码
36          if (i == 0) PORTA |= 0x80;                 //最高位加小数点
37          _delay_ms(2);
38      }
39  }
40
41  //-------------------------------------------------------------------
42  // 主程序
43  //-------------------------------------------------------------------
44  int main()
45  {
46      INT8U LastKey = 0xFF;                          //最近按键状态
```

```
47      DDRA = 0xFF;                                    //配置输出端口
48      DDRB = 0xFF;
49      DDRC = 0x00; PORTC = 0xFF;
50      DDRD = 0x00; PORTD = 0xFF;
51      //输入捕获噪音消除,ICP上升沿触发输入捕获,
52      //分频系数:1(1 MHz,每1 μs计数1次)
53      TCCR1B = _BV(ICNC1) | _BV(ICES1);               //初始时无分频,按下K1后提供1分频
54      sei();                                          //开中断
55      while(1)
56      {
57          if(LastKey != PINC )                        //PC端口有键按下(K1按下)
58          {
59              TIMSK = _BV(TICIE1);                    //使能TC1输入捕获中断
60              TCCR1B |= 0x01;                         //提供1分频计数时钟
61              LastKey = PINC;                         //保存最近按键状态
62          }
63          Show_FRQ_ON_DSY();                          //数码管显示频率
64      }
65  }
66
67  //-------------------------------------------------------------
68  // T1输入捕获中断子程序
69  //-------------------------------------------------------------
70  ISR (TIMER1_CAPT_vect)
71  {
72      INT8U i;
73      if (CAPi == 0) CAPi = ICR1;                     //第1次捕获
74      else                                            //第2次捕获
75      {
76          CAPj = ICR1 - CAPi;                         //2次相减得到周期(μs)
77          CAPj = 1000000UL/CAPj;                      //周期倒数后乘以1000000得到频率
78          TIMSK = 0x00;                               //第2次捕获后禁止输入捕获中断
79          TCCR1B &= 0xFC;                             //关闭计数时钟
80          for (i = 3; i != 0xFF; i -- )               //分解频率数位并放入显示缓冲
81          {
82              Display_Buffer[i] = CAPj % 10;
83              CAPj /= 10;
84          }
85          TCNT1 = CAPi = CAPj = 0;                    //相关变量和寄存器清零
86      }
87  }
```

3.26 用工作于异步模式的 T/C2 控制的可调式数码管电子钟

本例所使用的 T/C2 定时器工作于异步模式，由 PB6（TOSC1）与 PB7（TOSC1）外接 32768 Hz 晶振提供时钟，利用晶振提供的钟表时钟，本例设计了可调式数码管电子钟。案例电路及部分运行效果如图 3－26 所示。

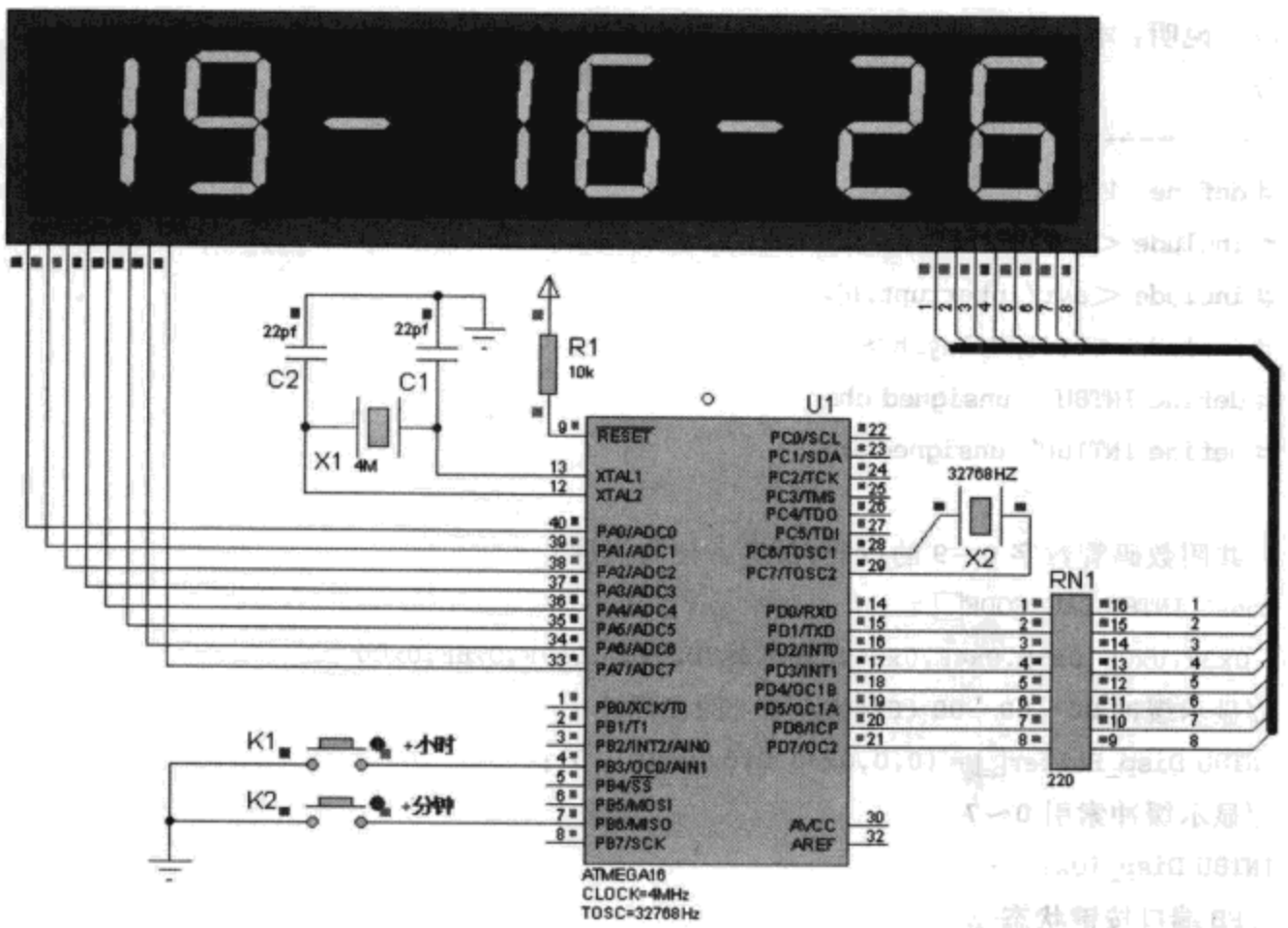


图 3－26 用工作于异步模式的 T2 控制的可调式数码管电子钟

1. 程序设计与调试

对于 T/C2 定时/计数器，通过设置异步状态寄存器 ASSR 可选择 T/C2 的时钟源，程序中 72～74 行对 T/C2 进行了相应设置，ASSR 寄存器中的 AS2 位是 T/C2 的时钟选择位，其中 72 与 73 行：

```
ASSR |= _BV(AS2);        用于选择外部时钟
TCCR2 = 0x04;            将分频系数设为 64
```

由这两行设置可得 T/C2 计数时钟频率为 32768 Hz/64＝512 Hz。

第 74 行将 8 位定时/计数器 T/C2 的计数寄存器 TCNT2 初值设为 0，计数 256 次后溢出，在 512 Hz 时钟频率下耗时 0.5 s，T/C2 溢出中断程序将每隔 0.5 s 被调用，中断程序利用 0.5 s 实现时分秒分隔标志“－”的闪烁显示，在每遇到第 2 个 0.5 s 时递增秒数，并进行秒分时的进位处理。

本例数码管的刷新显示则由 T/C0 定时器每隔 4 ms 刷新完成。

2. 实训要求

① 修改电路，使用 3 组 2 位集成数码管，分别显示时分秒，每组数码管之间用 2 位 LED

实现闪烁。

② 将T/C2分频系数改为32,重新修改代码实现本例功能。

3. 源程序代码

```
001  //-------------------------------------------------------------------
002  //  名称：用工作于异步模式的T2控制的可调式数码管电子钟
003  //-------------------------------------------------------------------
004  //  说明：本例T2使用外部32768 Hz时钟,K1、K2分别用来调整小时和分钟
005  //
006  //-------------------------------------------------------------------
007  #define  F_CPU   4000000UL //1 MHz晶振
008  #include <avr/io.h>
009  #include <avr/interrupt.h>
010  #include <util/delay.h>
011  #define INT8U   unsigned char
012  #define INT16U  unsigned int
013
014  //共阴数码管数字0～9的段码(最后一位为黑屏)
015  const INT8U SEG_CODE[] =
016  { 0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x00 };
017  //显示缓冲00-00-00(0x40为"-"的段码)
018  INT8U Disp_Buffer[] = {0,0,0x40,0,0,0x40,0,0};
019  //显示缓冲索引0～7
020  INT8U Disp_Idx;
021  //PB端口按键状态
022  INT8U Key_State = 0xFF;
023  //时分秒
024  INT8U h,m,s;
025  //-------------------------------------------------------------------
026  // 小时处理函数
027  //-------------------------------------------------------------------
028  void Increase_Hour()
029  {
030      if( ++h > 23) h = 0;
031      Disp_Buffer[0] = SEG_CODE[h/10];
032      Disp_Buffer[1] = SEG_CODE[h % 10];
033  }
034
035  //-------------------------------------------------------------------
036  // 分钟处理函数
037  //-------------------------------------------------------------------
038  void Increase_Minute()
039  {
```

```
040      if( ++ m > 59)
041      {
042          m = 0; Increase_Hour();
043      }
044      Disp_Buffer[3] = SEG_CODE[m/10];
045      Disp_Buffer[4] = SEG_CODE[m % 10];
046  }
047
048  //------------------------------------------------------------------
049  // 秒处理函数
050  //------------------------------------------------------------------
051  void Increase_Second()
052  {
053      if( ++ s > 59)
054      {
055          s = 0; Increase_Minute();
056      }
057      Disp_Buffer[6] = SEG_CODE[s/10];
058      Disp_Buffer[7] = SEG_CODE[s % 10];
059  }
060
061  //------------------------------------------------------------------
062  // 主程序
063  //------------------------------------------------------------------
064  int main()
065  {
066      DDRA = 0xFF; PORTA = 0xFF;                    //配置端口
067      DDRD = 0xFF; PORTD = 0xFF;
068      DDRB = 0x00; PORTB = 0xFF;
069
070      TCCR0 = 0x03;                                 //预设分频:64
071      TCNT0 = 256 - F_CPU/64.0 * 0.004;             //晶振 4 MHz,4 ms 定时初值
072      ASSR = 0x08;                                  //异步时钟使能
073      TCCR2 = 0x04;                                 //预设分频:64,32768 Hz/64 = 512 Hz
074      TCNT2 = 0;                                    //T2 计时初值
075      TIMSK = _BV(TOIE2) | _BV(TOIE0);              //允许 T0、T2 定时器中断
076
077      h = 12;     m = s = 0;
078      //将初始时分秒段码放入显示缓冲
079      Disp_Buffer[0] = SEG_CODE[h/10];
080      Disp_Buffer[1] = SEG_CODE[h % 10];
081      Disp_Buffer[3] = SEG_CODE[m/10];
082      Disp_Buffer[4] = SEG_CODE[m % 10];
```

```
083        Disp_Buffer[6] = SEG_CODE[s/10];
084        Disp_Buffer[7] = SEG_CODE[s % 10];
085
086        sei();                                          //开中断
087        while (1)
088        {
089            if(PINB ^ Key_State)                        //如果按键状态变化
090            {
091                _delay_ms(10);                          //延时消抖
092                if(PINB ^ Key_State) .                  //再次判断按键状态是否变化
093                {
094                    Key_State = PINB;                   //获取当前按键状态
095                    if(!(Key_State & _BV(PB3)))         //K1
096                       Increase_Hour();                 // + 小时
097                    else
098                    if(!(Key_State & _BV(PB6)))         //K2
099                       Increase_Minute();               // + 分钟
100                }
101            }
102        }
103    }
104
105    //-------------------------------------------------------------------
106    // T0 定时器溢出中断程序(控制数码管扫描显示)
107    //-------------------------------------------------------------------
108    ISR (TIMER0_OVF_vect)
109    {
110        static INT8U Disp_Idx = 0;                      //显示数位索引
111        TCNT0 = 256 - F_CPU/64.0 * 0.004;               //位间延时 4 ms
112        PORTA = 0x00;                                   //先关闭段码(共阴)
113        PORTA = Disp_Buffer[Disp_Idx];                  //输出数码管段码
114        //输出位码(本例使用的是共阴数码管,注意将_BV取反)
115        PORTD = ~_BV(Disp_Idx);
116        Disp_Idx = (Disp_Idx + 1) & 0x07;               //数码管位索引在0~7内循环
117    }
118
119    //-------------------------------------------------------------------
120    // T2 中断控制时钟运行
121    //-------------------------------------------------------------------
122    ISR (TIMER2_OVF_vect)
123    {
124        //由于TCNT2溢出时自动归0,因此不需要在中断函数中重装初值
125        //TCNT2 = 0;
```

```
126      //T2 时钟为 512 Hz,TCNT2 由 0 计数到 256 时溢出,故每 0.5 s 中断一次
127      if (Disp_Buffer[2] == 0x40)
128      {
129          Disp_Buffer[2] = Disp_Buffer[5] = 0x00; //前 0.5 s 关闭"-"显示
130      }
131      else
132      {
133          Disp_Buffer[2] = Disp_Buffer[5] = 0x40; //后 0.5 s(即 1 s)打开"-"显示
134          Increase_Second();                      //秒递增
135      }
136  }
```

3.27 TIMER1 定时器比较匹配中断控制音阶播放

本例运行时,按下 K1 按键将输出一段有 14 个音符的音阶,音符输出由定时器控制完成。在输出声音时,通过连接的虚拟示波器可观察到脉宽逐步缩小,频率不断升高。如果 PC 机 CPU 因连接虚拟示波器而过载,导致声音播放失真,这时可断开示波器再播放。本例电路及部分波形如图 3-27 所示。

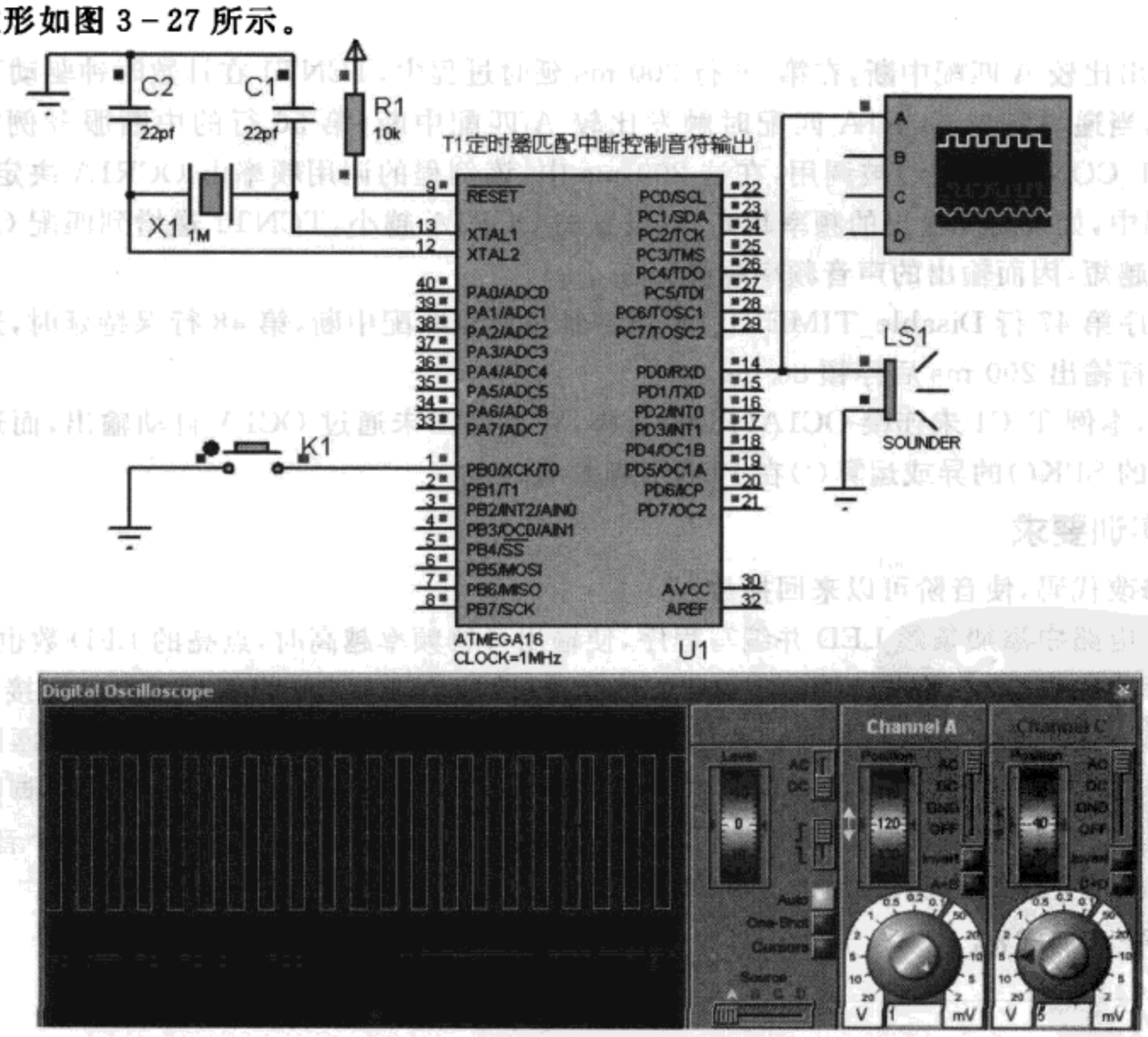


图 3-27 TIMER1 定时器比较匹配中断控制音阶播放

1. 程序设计与调试

本例程序运行时将输出DO、RE、ME、……的声音，下面是其中前7个音符的频率：

简　谱	1	2	3	4	5	6	7
音　符	C5	D5	E5	F5	G5	A5	B5
频　率	523	587	659	698	784	880	987

以上频率的计时初值可根据下面的关系式推出：

① 根据频率可得方波周期：t = 1 /频率×1000000（单位为 μs）；

② 由于所输出的 t（μs）周期方波中，高/低电平各占50%，因此定时器定时（计数）长度为 Count= $t/2$，即 Count=1000000/2/频率。

主程序中第32、33两行将TCCR1A与TCCR1B分别设为0x00与0x09，它们共同将WGM1[3:0]设为0100，使T/C1工作于CTC模式（即OCR1A/B与TCNT1比较匹配时清零T/C1），TCCR1B=0x09还将TCNT1计数时钟设为使用1分频的系统时钟，即F_CPU。

根据公式Count=1000000/2/频率，可设置输出比较寄存器OCR1A：

OCR1A=F_CPU/2/TONE_FRQ[i]

本例F_CPU为1 MHz，由此设定的OCR1A决定了相应的输出频率。

设置OCR1A寄存器的下一行将TCNT1设为0，随后的第45行Enable_TIMER1_OCIE()允许输出比较A匹配中断，在第46行200 ms延时过程中，TCNT1在计数时钟驱动下每微秒递增1，当递增到与OCR1A匹配时触发比较A匹配中断，第56行的中断服务例程ISR(TIMER1_COMPA_vect)被调用，在这200 ms中，该例程的调用频率由OCR1A决定，在第43行代码中，如果需要输出的频率越高，则设置的OCR1A越小，TCNT1递增到匹配OCR1A的周期也越短，因而输出的声音频率越高。

主程序第47行Disable_TIMER1_OCIE()禁止比较匹配中断，第48行保持延时，这样可使每个音符输出200 ms后停顿80 ms。

另外，本例T/C1未连接OC1A(PD5)引脚，频率输出未通过OC1A自动输出，而通过中断程序中的SPK()的异或运算(^)在PD0引脚输出。

2. 实训要求

① 修改代码，使音阶可以来回播放。

② 在电路中添加条形LED并编写程序，使输出音符频率越高时，点亮的LED数也越多。

③ 修改代码32行的TCCR1A=0x00，将0x00改为0x40，这样设置后T/C1连接OC1A(PD5)引脚，在每次比较匹配时OC1A引脚会自动取反。通过该设置，程序中的比较匹配中断允许与禁止语句可删除，中断服务程序也可以删除，将蜂鸣器改接至OC1A(PD5)引脚同样可听到音符输出。根据以上说明修改代码进行调试时需要注意的是，在输出完最后一个音符后，由于比较匹配会继续出现，最后的高频率音符将持续输出，这个问题要注意解决。

3. 源程序代码

```
01  //-------------------------------------------------------------------
02  //  名称：TIMER1 定时器比较匹配中断控制音阶播放
03  //-------------------------------------------------------------------
```

```
04 //  说明：本例运行时，按下 K1 将在定时器控制下演奏一段音阶 1,2,3,4,5,6,7...
05 //        本例使用了 T1 的定时器比较匹配中断实现不同频率音符输出
06 //
07 //-------------------------------------------------------------------
08 #define  F_CPU   1000000UL
09 #include <avr/io.h>
10 #include <avr/interrupt.h>
11 #include <util/delay.h>
12 #define INT8U   unsigned char
13 #define INT16U  unsigned int
14
15 #define K1_DOWN() ((PINB & _BV(PB0)) == 0x00)        //按键定义
16 #define SPK()       (PORTD ^= _BV(PD0))              //蜂鸣器定义
17 //TC1 输出比较 A 匹配中断使能开关
18 #define Enable_TIMER1_OCIE()  (TIMSK |=  _BV(OCIE1A))
19 #define Disable_TIMER1_OCIE() (TIMSK &= ~_BV(OCIE1A))
20
21 //C 调 15 个音符频率表
22 const INT16U TONE_FRQ[] =
23  { 0,262,294,330,349,392,440,494,523,587,659,698,784,880,988,1046 };
24 //-------------------------------------------------------------------
25 // 主程序
26 //-------------------------------------------------------------------
27 int main()
28 {
29     INT8U i;
30     DDRB = 0x00; PORTB = 0xFF;                    //配置端口
31     DDRD = 0xFF; PORTD = 0xFF;
32     TCCR1A = 0x00;                                //TC1 与 OC1A 不连接，禁止 PWM 功能
33     TCCR1B = 0x09;                                //TC1 预设分频：1
34                                                   //CTC 模式(比较匹配时 TC1 自动清零)
35     sei();                                        //开中断
36     while(1)
37     {
38        while (!K1_DOWN());                        //未按键等待
39        while(K1_DOWN());                          //等待释放
40
41        for( i=1; i<16; i++ )
42        {
43            OCR1A = F_CPU/2/TONE_FRQ[i];           //根据频率计算延时，设置 OCR1A
44            TCNT1 = 0;                             //在播放新的音符之前将 TCNT0 清零
45            Enable_TIMER1_OCIE();                  //允许 TC1 比较匹配中断，播放当前音符
46            _delay_ms(200);                        //播放延时
```

```
47              Disable_TIMER1_OCIE();                  //禁止 TC1 比较匹配中断,停止播放
48              _delay_ms(80);                          //停顿延时
49          }
50      }
51  }
52
53  //-------------------------------------------------------------
54  // T1 定时器比较匹配中断程序,控制音符频率输出
55  //-------------------------------------------------------------
56  ISR (TIMER1_COMPA_vect)
57  {
58      SPK();
59  }
```

3.28 用 TIMER1 输出比较功能调节频率输出

本例运行时,通过按下 K1～K4 按键可分别调整输出频率的千位、百位、十位与个位数,当前频率将显示在 4 只数码管上。案例电路图及部分运行效果如图 3-28 所示。

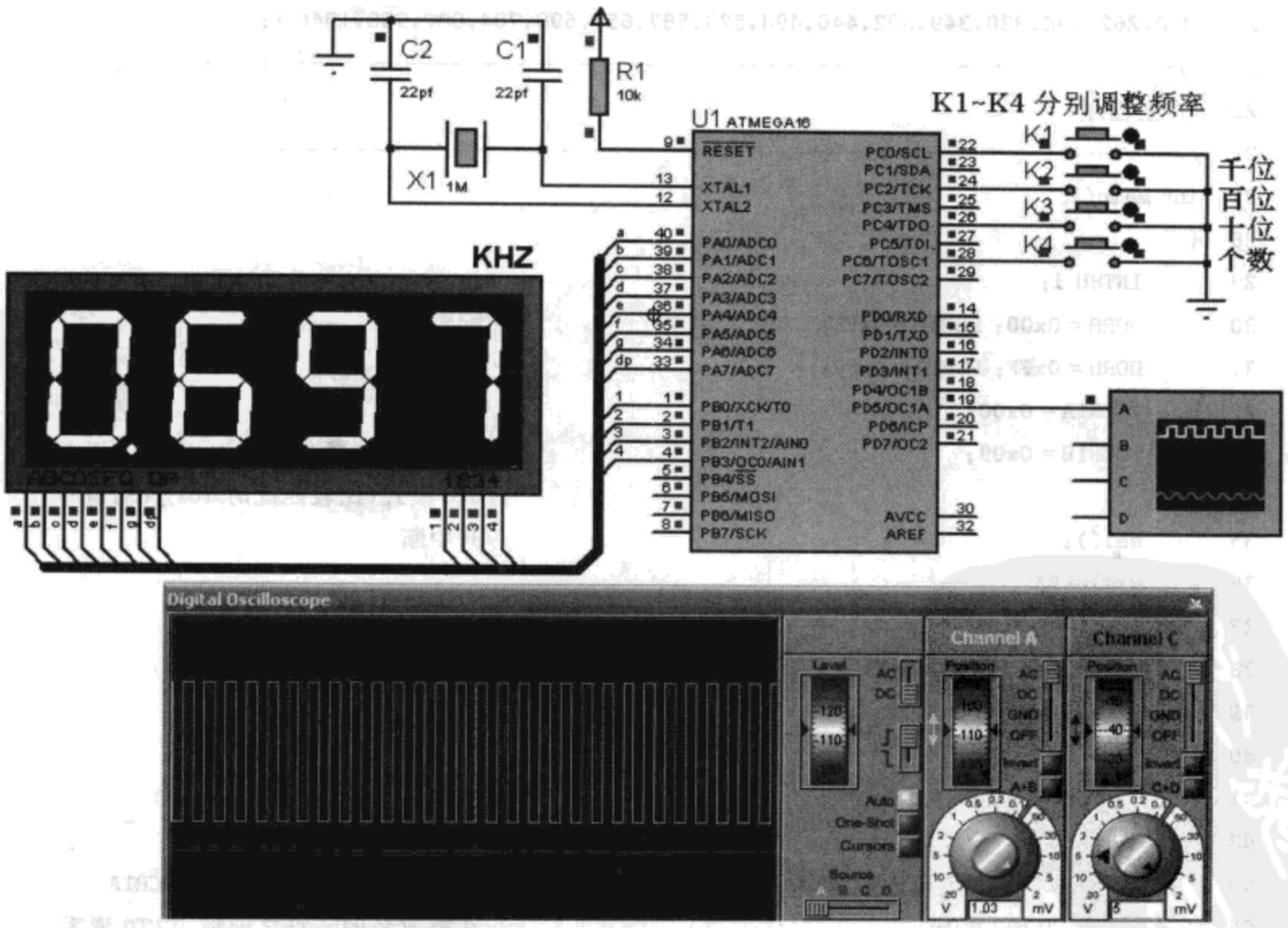


图 3-28 用 TIMER1 输出比较功能调节频率输出

1. 程序设计与调试

本例仍然使用 T/C1 的 CTC(比较匹配清零计数器)模式,相关设置与上一案例很相似。在阅读调试本例时,有 2 个要点:

① 上一案例中 T/C1 与 OC1A 不连接,本例 T/C1 连接了 OC1A 引脚(PD5),所生成的频率由该引脚输出。

② 本例程序中对按键的扫描未使用"未释放则等待的语句",这是因为数码管刷新显示与按键扫描都由主程序中的 while(1)循环控制完成,在此循环中,如果对按键使用 while (按键未释放);这样的语句来等待释放,在按下某按键调整频率时,如果未及时释放则会出现数码管不能被连贯快速扫描而出现缺位的现象。

2. 实训要求

① 本例电路中各按键只能对相应频率数位进行递增循环调节,完成本例调试仿真后修改仿真电路及程序,使按键可以任意增减输出频率。

② 将频率输出引脚改为 OC1B(PD6),重新修改程序,使用比较匹配中断或非中断方式在该引脚输出所设定的频率。

3. 源程序代码

```
01  //-------------------------------------------------------------
02  //  名称:用 TIMER1 比较输出功能调节频率输出
03  //-------------------------------------------------------------
04  //  说明:本例运行过程中,通过 K1~K4 这 4 个不同按键分别调节频率值的
05  //        千位、百位、十位、个位,通过虚拟示波器可以观察到不同的频率输出
06  //
07  //-------------------------------------------------------------
08  #define  F_CPU   1000000UL                    //1 MHz 晶振
09  #include <avr/io.h>
10  #include <util/delay.h>
11  #define INT8U   unsigned char
12  #define INT16U  unsigned int
13
14  //定义按键
15  #define K1 (INT8U)(~_BV(PC0))
16  #define K2 (INT8U)(~_BV(PC2))
17  #define K3 (INT8U)(~_BV(PC4))
18  #define K4 (INT8U)(~_BV(PC6))
19
20  //共阴数码管 0~9 的数字编码
21  const INT8U SEG_CODE[] =
22  {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
23  //分解后的待显示频率数位(初值为 100 MHz)
24  INT8U FRQ_DATA[] = {0,1,0,0};
25  //-------------------------------------------------------------
```

```
26  // 数码管显示频率
27  //-------------------------------------------------------------------
28  void Show_FRQ_ON_DSY()
29  {
30      INT8U i = 0;
31      for (i = 0; i<4; i ++ )
32      {
33          PORTB = ~_BV(i);                    //发送扫描码
34          PORTA = SEG_CODE[ FRQ_DATA[i] ];    //发送数字段码
35          if (i == 0) PORTA |= 0x80;
36          _delay_ms(2);
37      }
38  }
39
40  //-------------------------------------------------------------------
41  // 频率设置
42  //-------------------------------------------------------------------
43  void Set_Frequency()
44  {
45      INT16U f;
46      f = FRQ_DATA[0] * 1000 +                //根据FRQ_DATA数组中的各数位
47          FRQ_DATA[1] * 100 +                 //计算出频率
48          FRQ_DATA[2] * 10 +
49          FRQ_DATA[3];
50      OCR1A = F_CPU/2.0/f;                    //由频率计算出输出比较寄存器OCR1A初值
51  }
52
53  //-------------------------------------------------------------------
54  // 主程序
55  //-------------------------------------------------------------------
56  int main()
57  {
58      INT8U i = 0, Key_State = 0xFF;
59      DDRA = 0xFF; PORTA = 0xFF;              //配置端口
60      DDRB = 0xFF; PORTB = 0xFF;
61      DDRD = 0xFF; PORTD = 0xFF;
62      DDRC = 0x00; PORTC = 0xFF;
63      TCCR1A = 0x40;                          //TC1连接OC1A引脚,每次比较匹配时OC1A取反
64      TCCR1B = 0x09;                          //CTC模式(比较匹配时清零计数器),分频:1
65      TCNT1 = 0;                              //清除定时器值
66      Set_Frequency();                        //设置频率
67      while(1)
68      {
```

```
69          if(PINC ^ Key_State )                    //如果按键状态变化
70          {
71                Key_State = PINC;                  //获取当前按键状态
72                if(Key_State != 0xFF)              //如果有键按下
73                {
74                    switch (Key_State)             //根据不同按键分别调整千、百、十、个位
75                    {
76                       case K1: i = 0; break;
77                       case K2: i = 1; break;
78                       case K3: i = 2; break;
79                       case K4: i = 3; break;
80                    }
81                    //修改频率数组的第 i 位(千、百、十、个位)
82                    FRQ_DATA[i] = (FRQ_DATA[i] + 1) % 10;
83                    Set_Frequency();               //设置频率
84                }
85          }
86          Show_FRQ_ON_DSY();                       //数码管显示频率
87      }
88  }
```

3.29 TIMER1 控制的 PWM 脉宽调制器

本例运行时，调节可变电阻，系统程序进行模/数转换后，根据不同的转换结果调节输出不同占空比波形，驱动电机以不同速度转动。在仿真运行过程中，连接虚拟示波器以后如果提示 PC 机 CPU 过载，这会使电机转速不正常，这时可删除示波器再运行仿真。案例电路及部分运行效果如图 3－29 所示。

1. 程序设计与调试

T/C1 可工作于一般模式(定时/计数)、CTC 模式(比较匹配时清零计数器)，以及快速 PWM、相位可调 PWM 及相位频率可调的 PWM 模式，通过外部运放可构成 8 位、9 位、10 位或 16 位的 D/A 转换器。

本例主程序中 TCCR1A＝0x83(10000011)、TCCR1B＝0x02(00000010)完成如下设置：

① 0x83 的低 2 位将 TCCR1A 寄存器最低 2 位的波形发生器模式(Waveform Generation mode)选择位 WGM1[1:0]设为 11，0x02 的中间 2 位 00 将 WGM1[3:2]设为 00，因此 WGM1[3:0]为 0011，根据这 4 位 WGM 的设置可知 T/C1 工作模式为 10 位 PWM，相位可调。

② 0x83 的高 4 位 1000 对应于 TCCR1A 的高 4 位 COM1A[1:0]和 COM1B[1:0]，它们用于设置 T/C1 的比较输出模式(Compare Output Modulation:COM)。

由 TCCR1A 与 TCCR1B 共同配置的 4 位波形发生模式位 WGM1[3:0]及由 TCCR1A 高 4 位 COM1A[1:0]与 COM1B[1:0]的取值可知比较输出模式为：加 1 计数与 OCR1A 比较匹配时将 OC1A 引脚清零，减 1 计数与 OCR1A 比较匹配时将 OC1A 引脚置 1，程序中将该行注

释为:10 位正向 PWM。

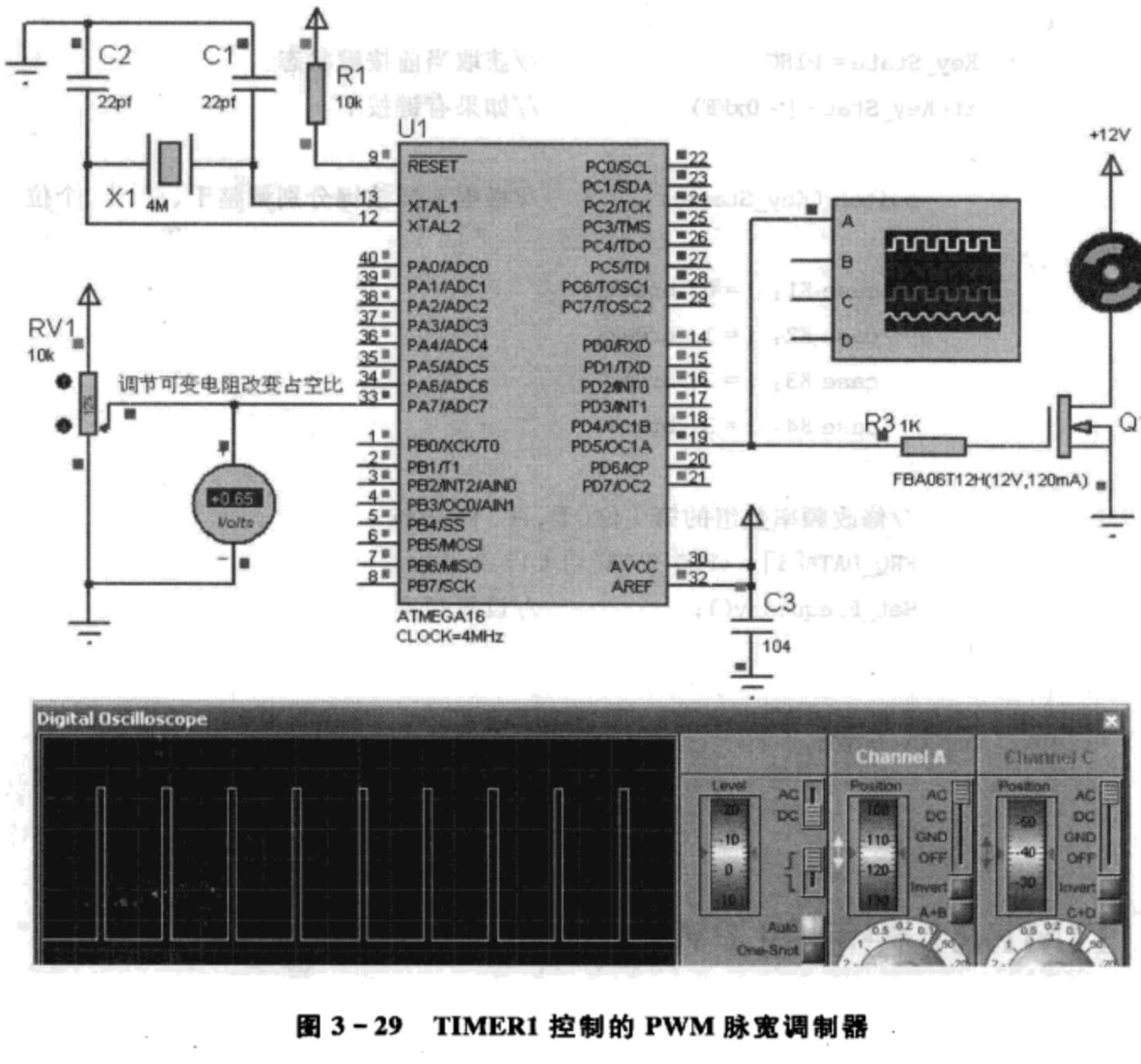


图 3－29　TIMER1 控制的 PWM 脉宽调制器

对于本例 10 位的 PWM,其上限 TOP 值为 1023,主程序中 10 位的模/数转换结果为 0～1023,通过设置 OCR1A 为模/数转换的值,可使 OCR1A 取值范围在 0～1023 之间,OCR1A 的取值决定了 OC1A(PD5)引用脚输出脉冲的起始相位和脉宽。在正向 PWM 模式下,OCR1A 取值越小时,占空比越低;OCR1A 取值越大时,占空比越高。但对于 2 个极值 0 与 1023例外。

在本例相位可调的 PWM 模式下,PWM 波形频率由以下公式确定:

F_OC1A＝F_CPU/N/2/TOP　(其中 F_CPU 为系统时钟,N 为分频设置)

根据本例配置数据可得 OC1A 引脚输出 PWM 波形频率为:

F_OC1A＝4000000/8/2/1023≈244 Hz

2. 实训要求

① 修改程序,改用 OC1B(PD4)引脚控制电机转速。

② 本例 T/C1 配置为相位可调的 PWM 模式,在调试本例后重新将 T/C1 配置为快速 PWM 模式,在 OC1A 或 OC1B 引脚输出占空比可调的 PWM 波形。

3. 源程序代码

```
01 //-------------------------------------------------------------------
02 //  名称:TIMER1 控制的 PWM 脉宽调制器
```

```
//-------------------------------------------------------------------
//  说明：本例运行过程中，调节 RV1 可改变输出波形的占空比
//
//-------------------------------------------------------------------
#define  F_CPU   4000000UL
#include <avr/io.h>
#include <util/delay.h>
#define INT8U   unsigned char
#define INT16U  unsigned int

//-------------------------------------------------------------------
// 对通道 CH 进行模/数转换
//-------------------------------------------------------------------
INT16U ADC_Convert(INT8U CH)
{
    int Result;
    ADMUX = CH;                              //ADC 通道选择
    Result = (INT16U)(ADCL + (ADCH << 8));   //读取转换结果
    return Result;
}

//-------------------------------------------------------------------
// 主程序
//-------------------------------------------------------------------
int main()
{
    INT16U x = 0, PRE_ADC_Result = 0;
    //float Duty = 1.0;                      //占空比
    DDRA = 0x00; PORTA = 0xFF;               //配置端口
    DDRD = 0xFF; PORTD = 0xFF;
    DDRC = 0xFF;
    ADCSRA = 0xE6;                           //10 位 ADC 转换置位，启动转换，64 分频
    _delay_ms(3000);                         //延时等待系统稳定
    TCCR1A = 0x83;                           //10 位 PWM(1023)，正向 PWM
    TCCR1B = 0x02;                           //时钟 8 分频，PWM 频率：F_CPU/8/2046
    while(1)
    {
       x = ADC_Convert(7);
       if (x != PRE_ADC_Result)              //如果模/数转换值变化则修改 OCR1A
       {
           PRE_ADC_Result = x;               //保存最后一次模/数转换结果
           //Duty = x/1023.0;                //由模/数转换结果计算占空比 Duty
           //x = 1023 * Duty;                //根据占空比求 OCR1A
```

```
46                                                    //上述两行未改变x的值,因此可注销
47              if (x == 1023) x = 0;                 //如果调节到极值时将极值交换
48              else if (x == 0) x = 1023;
49              OCR1A = x;                            //设置输出比较寄存器OCR1A(0～1023)
50          }
51      }
52  }
```

3.30 数码管显示两路A/D转换结果

本例单片机对PA端口输入的多路模拟量进行A/D转换,最大转换精度为10位。在本例中,ADC0(PA0)、ADC1(PA1)引脚外接两组可变电阻,分别调节RV0与RV1时,对应两组的模拟电压在进行数字转换后将显示在8位数码管上。本例电路及部分运行效果如图3-30所示。

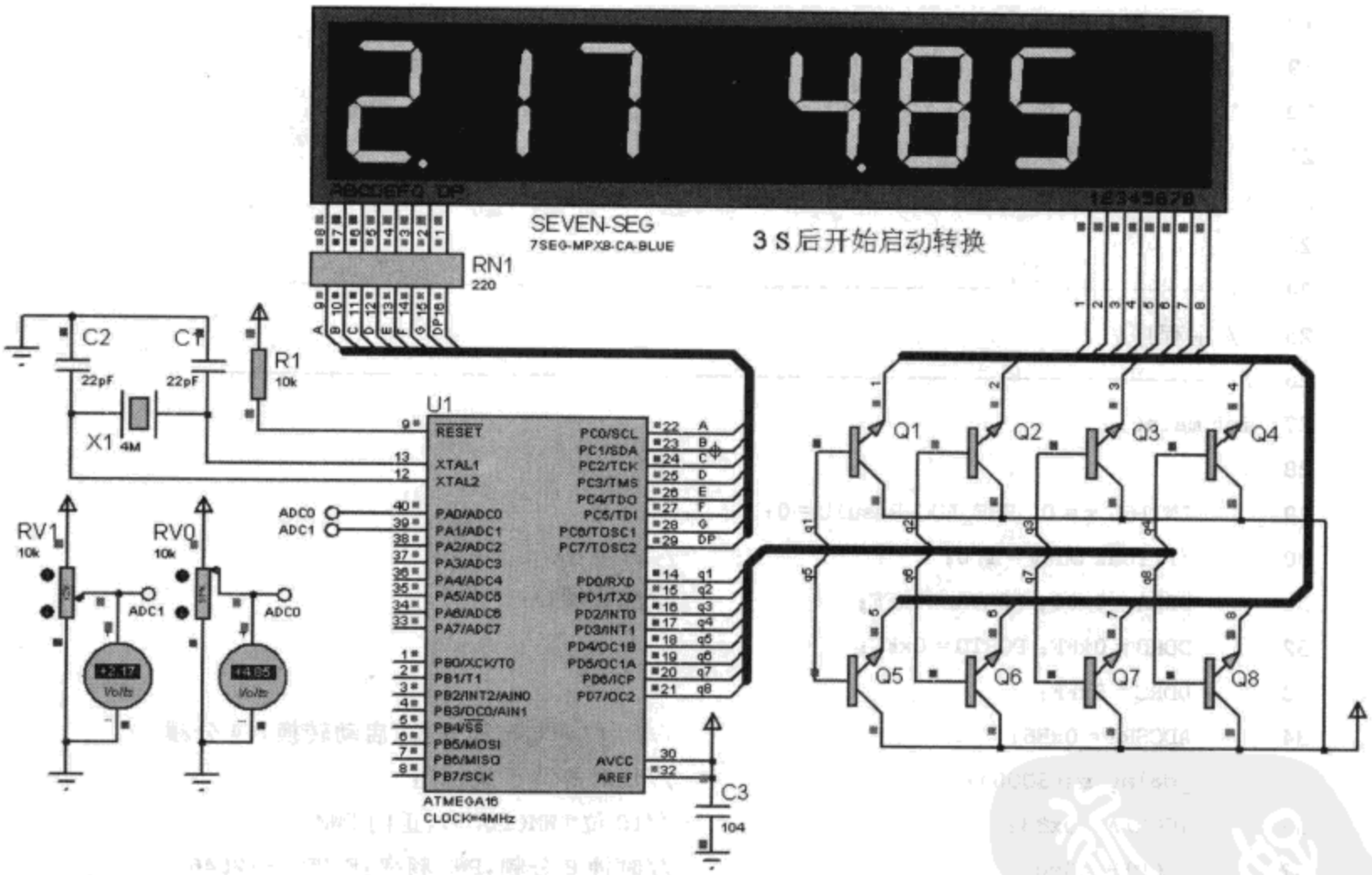


图3-30 数码管显示两路A/D转换结果

1. 程序设计与调试

本例程序设计要点在于:

ADC控制与状态寄存器A——ADCSRA,程序中其取值为0xE6(11100110),其最高位ADEN置位启动ADC,ADSC置位开始转换,ADATE置位启动ADC自动触发功能。ADCSRA的低3位ADPS[2:0]设置为110,分频比设为64。

多路复用选择寄存器——ADMUX，其低 4 位 MUX3～MUX0 为模拟通道选择位，用于选择模拟通道，取值 0000～0111 对应于 ADC0～ADC7。

ADC 数据寄存器——ADC(ADCH/ADCL)，转换结束后数据存放于该寄存中。在读取 ADC 中的数据时，本例 26 行和 27 行提供了分 ADCL 与 ADCH 读取的语句和直接通过 ADC 读取的语句，经编译测试，两种写法都能获取正确的转换结果。

本例程序中单独编写了 ADC 转换函数，参数为待转换通道 CH，函数中 ADMUX 直接等于通道参数 CH。CH 分别取 0 和 1 时，它相当于将低 4 位 MUX3～MUX0 分别设为 0000 和 0001，选择 ADC0 通道和 ADC1 通道进行转换。第 26 行将转换结果除以 1023 再乘以 500，可将 10 位的模/数转换结果 0x0000～0x03FF(即 0～1023)转换为 000～500 之间的待显示数据值(它们对应于电压 0.00～5.00 V)，在显示时两组数值的高位后面单独附加小数点。

2. 实训要求

① 选择其他 ADC 通道进行模/数转换并显示结果。

② 重新编写本例程序，利用 ADC 中断程序完成转换结果显示。

3. 源程序代码

```
01  //-------------------------------------------------------------------
02  //  名称：数码管显示两路 A/D 转换结果
03  //-------------------------------------------------------------------
04  //  说明：调节 RV1 和 RV2 时，两路模拟电压将显示在 8 只集成式数码管上
05  //
06  //-------------------------------------------------------------------
07  #define F_CPU 4000000UL                    //4 MHz
08  #include <avr/io.h>
09  #include <util/delay.h>
10  #define INT8U   unsigned char
11  #define INT16U  unsigned int
12
13  //各数字的数码管段码，最后一位为空白
14  const INT8U SEG_CODE[] =
15  {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xFF};
16  //两路模拟转换结果显示缓冲，显示格式为：X.XX X.XX ，第 4 位和第 8 位不显示
17  INT8U Display_Buffer[] = {0,0,0,10,0,0,0,10};
18  //-------------------------------------------------------------------
19  // 对通道 CH 进行模/数转换
20  //-------------------------------------------------------------------
21  void ADC_Convert(INT8U CH)
22  {
23      int Result;
24      ADMUX = CH;                            //ADC 通道选择
25      //读取转换结果，并转换为电压值
26      Result = (int)((ADCL + (ADCH << 8)) * 500.0/1023.0);
27      //或使用语句：Result = (int)(ADC * 500.0/1023.0);
```

```
28
29      //ADC0 的结果放入数组 0、1、2 单元,ADC1 的结果放入数组 4、5、6 单元
30      Display_Buffer[CH * 4]     = Result/100;
31      Display_Buffer[CH * 4 + 1] = Result/10 % 10;
32      Display_Buffer[CH * 4 + 2] = Result % 10;
33  }
34
35  //-------------------------------------------------------------
36  // 主程序
37  //-------------------------------------------------------------
38  int main()
39  {
40      INT8U i;
41      DDRA = 0xFC;                                  //配置 A/D 转换端口 ADC0、ADC1 为输入
42      DDRC = 0xFF; PORTC = 0x00;                    //配置数码管显示端口
43      DDRD = 0xFF; PORTD = 0x00;
44      ADCSRA = 0xE6;                                //ADC 转换置位,启动转换,64 分频
45      _delay_ms(3000);                              //延时等待系统稳定
46      while(1)
47      {
48          ADC_Convert(0);  ADC_Convert(1);          //对 2 个通道进行 A/D 转换
49          for(i = 0; i<8; i ++ )
50          {
51              PORTC = 0xFF;                         //先关闭段码
52              PORTD = _BV(i);                       //发送数码管位码
53              PORTC = SEG_CODE[ Display_Buffer[i] ]; //发送数字段码
54              if(i == 0 || i == 4) PORTC &= 0x7F;   //对整数位加小数点
55              _delay_ms(4);
56          }
57      }
58  }
```

3.31 模拟比较器测试

本例程序运行时,模拟比较器的正极 AN0 与负极 AN1 所输入的模拟电压将进行比较,如果 AN0 上的电压高于 AN1 上的电压时,LED0 点亮,否则 LED1 点亮。本例电路及部分运行效果如图 3-31 所示。

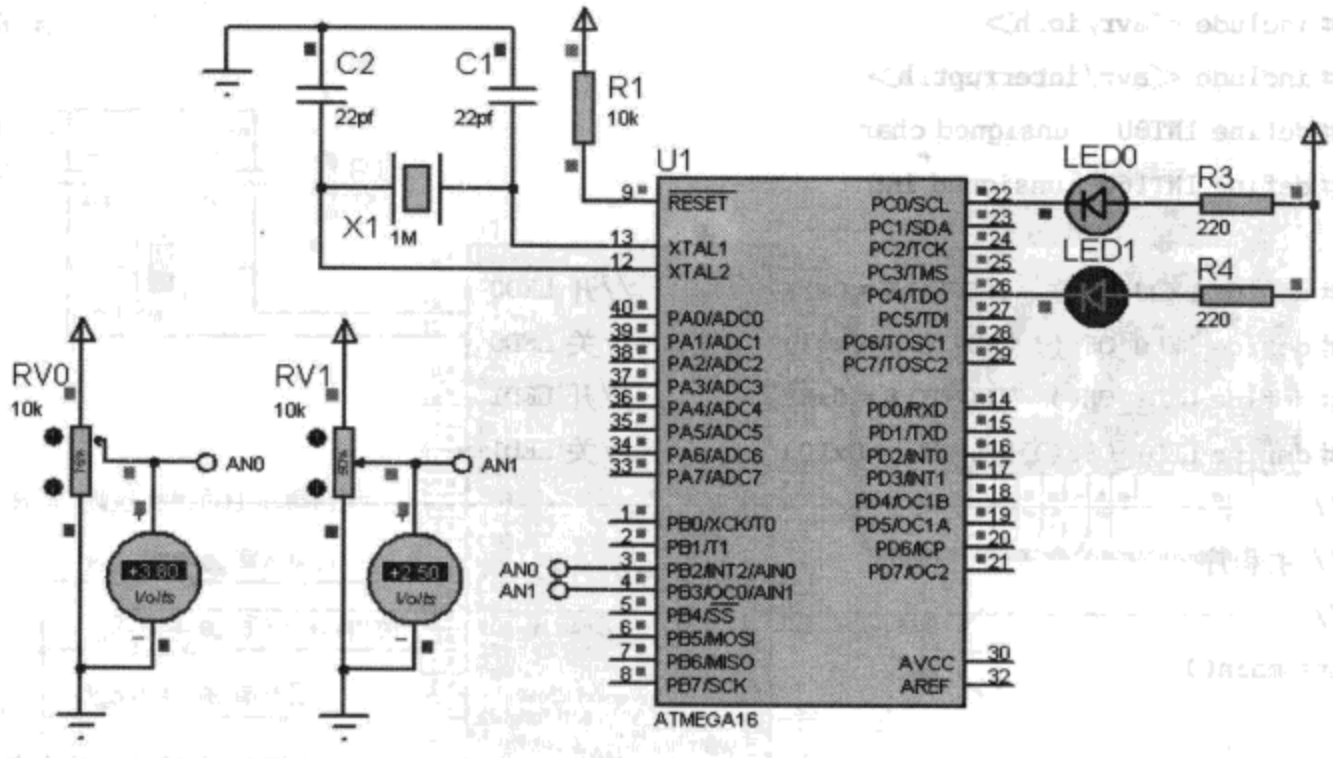


图 3-31　模拟比较器测试

1. 程序设计与调试

本例程序要点在于 SFIOR(Special Function IO Register)与 ACSR(Analog Comparator Control and Status Register)寄存器的设置：

① 主程序将特殊功能 I/O 寄存器 SFIOR 中的 ACME 位清零，使 AIN1 连接比较器的负极输入端，在模/数转换状态寄存器 ADCSRA 中的 ADEN 为 0 时，如果将 ACME 置位，模拟比较器将使用 ADC 的多路输入作为负极输入端。

② 将 SFIOR 中的 PUD 置位，禁用内部上拉电阻。

③ 将模拟比较器控制及状态寄存器 ACSR 中的 ACIE 位置位，以允许模拟比较器中断。

在完成上述设置并开中断后，如果 AN0 上的电压高于 AN1 上的电压时，LED0 点亮，否则 LED1 点亮。

2. 实训要求

① 重新配置 25 行的特殊功能 I/O 寄存器 SFIOR 的 ACME 位，并将模/数转换控制与状态寄存器 ADCSRA 的 ADEN 位置 0，利用 ADC 多路输入选择器 ADMUX 将 ADC0～ADC7 中的某一路模拟电压作为模拟比较器的反相(即“－”端)输入源，完成上述模拟比较器测试。

② 在 AN1 引脚提供 1.5 V 电压，编程检测 AN0 引脚模拟电压向上穿越 1.5 V 电压的次数。

3. 源程序代码

```
01  //-------------------------------------------------------------------
02  //  名称：模拟比较器测试
03  //-------------------------------------------------------------------
04  //  说明：当 AN0 上的电压高于 AN1 时，模拟比较器置位，LED1 点亮，反之 LED2 点亮
05  //
06  //-------------------------------------------------------------------
07  #define F_CPU   1000000UL                    //1 MHz 晶振
```

```
#include <avr/io.h>
#include <avr/interrupt.h>
#define INT8U   unsigned char
#define INT16U  unsigned int

#define LED0_ON()  (PORTC &= 0xFE)          //开 LED0
#define LED0_OFF() (PORTC |= 0x01)          //关 LED0
#define LED1_ON()  (PORTC &= 0xEF)          //开 LED1
#define LED1_OFF() (PORTC |= 0x10)          //关 LED1
//-------------------------------------------------------------
// 主程序
//-------------------------------------------------------------
int main()
{
    DDRB = 0x00;                            //PB2,PD3(AIN0/AIN1)设置为输入(无内部上拉)
    DDRC = 0xFF;                            //PC 端口设置为输出(外接 LED)
    SFIOR &= ~_BV(ACME);                    //AIN1 连接比较器的负极输入端
    SFIOR |=  _BV(PUD);                     //禁用内部上拉电阻
    ACSR  =  _BV(ACIE);                     //允许模拟比较器中断
    sei();                                  //开总中断
    while(1);
}

//-------------------------------------------------------------
// 模拟比较器中断服务程序
//-------------------------------------------------------------
ISR (ANA_COMP_vect)
{
    if (ACSR & _BV(ACO))                    //检查 ACO 位,判断 AN0 电压是否大于 AN1 电压
    {
        LED0_ON();  LED1_OFF();
    }
    else
    {
        LED0_OFF(); LED1_ON();
    }
}
```

3.32 EEPROM 读/写与数码管显示

AVR 单片机内部存储器有 Flash、SRAM、EEPROM 这 3 种。本例程序演示了 EEPROM 数据存储空间透明地址数据和不透明地址数据的读/写与显示。案例电路及部分运行效果如

图 3－32所示。

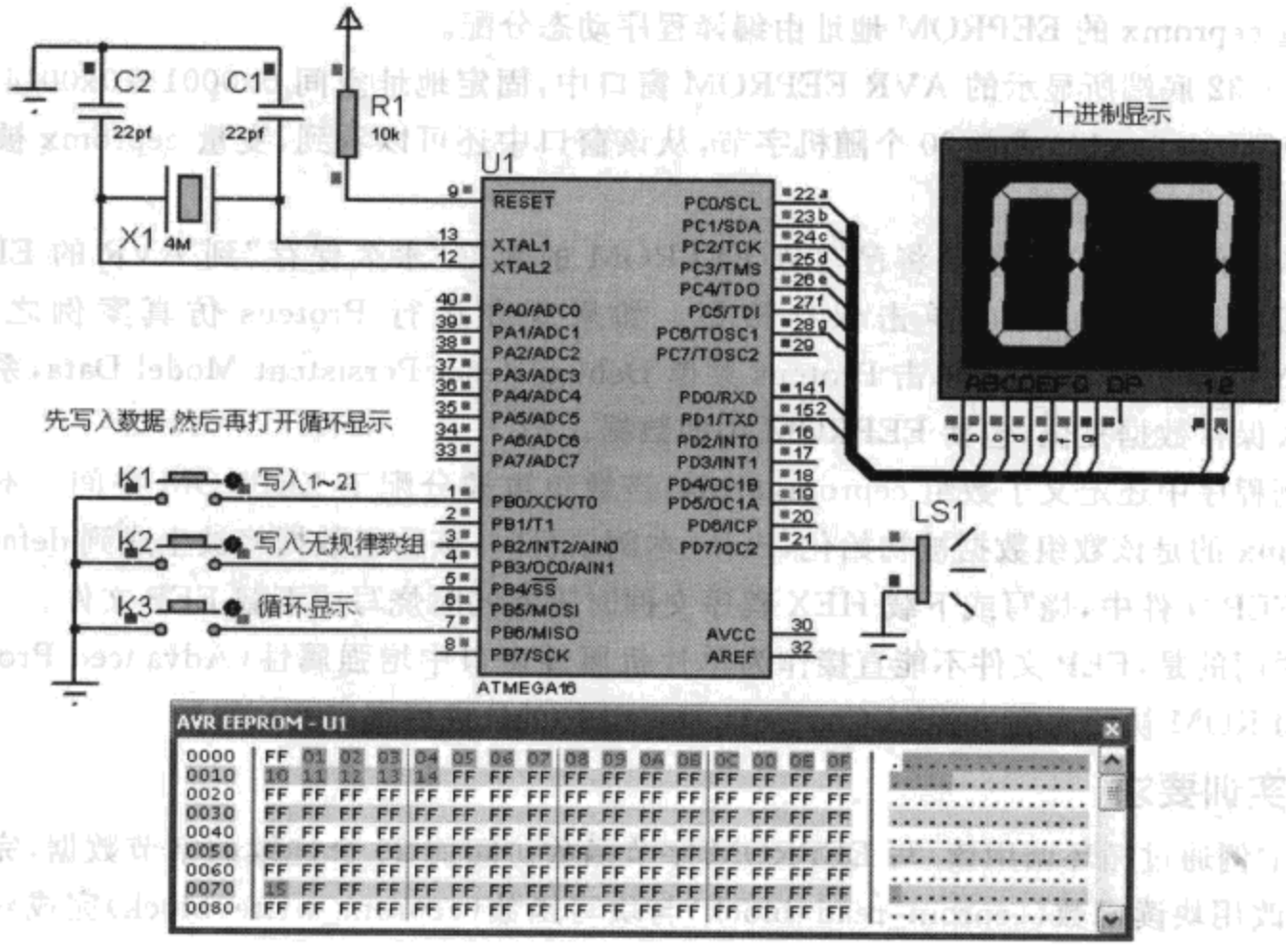


图 3－32　EEPROM 读/写与数码管显示

1. 程序设计与调试

AVR－GCC 提供了专门用于访问 EEPROM 的函数，这使得 EEPROM 的读/写变得非常简单。在编写本例程序时，需要添加头文件＜avr/eeprom. h＞。打开 AVRStudio 帮助菜单中的 avr－libc参考手册，然后打开 Library Manual，可找到 avr/eeprom. h。该头文件给出了 EEPROM 操作的重要函数：

```
eprom_is_ready()                          //EEPROM 就绪
eeprom_busy_wait()                        //EEPROM 忙等待
eeprom_read_byte(地址)                    //从指定地址读取并返回一字节数据
eeprom_write_byte(地址,一字节数据)        //向指定地址写入一字节数据
```

在参考手册中，还可以看到对字数据进行读/写的函数。

本例读/写的 21 个字节数据中，前 20 个数据地址是固定的，其地址空间为 0x0001～0x0014，第 21 个字节数据(对应于 eepromx)的地址是编译程序分配的，字节变量 eepromx 通过以下语句申明：

```
INT8U eepromx __attribute__((section("eeprom")));
```

该语句还可以写成：

```
INT8U eepromx EEMEM;
```

其中 EEMEM 即__attribute__((section("eeprom"))),它定义在头文件＜avr/eeprom. h＞

里面。

变量 eepromx 的 EEPROM 地址由编译程序动态分配。

图 3-32 底端所显示的 AVR EEPROM 窗口中，固定地址空间 0x0001～0x0014 将被写入 1～20(0x01～0x14)或者 20 个随机字节，从该窗口中还可以看到，变量 eepromx 被编译器动态分配到 0x0070 地址。

调试本例程序时，如果要将已写入 EEPROM 的数据“永久保存”到 AVR 的 EEPROM 空间，在退出 Proteus 时应单击保存按钮。如果要在运行 Proteus 仿真案例之前清除 EEPROM中原有的数据，可单击 Proteus 菜单 Debug/Reset Persistent Model Data，系统会将所有持久保存数据复位，包括 EEPROM 中的数据。

本例程序中还定义了数组 eeprom_array，该数组也被分配于 EEPROM 空间。不同于变量 eepromx 的是该数组数据被初始化，Build 本例项目时，该数组数据将被生成到 default 文件夹下的 EEP 文件中，烧写或下载 HEX 程序文件时需要同时烧写或下载 EEP 文件。

要说明的是，EEP 文件不能直接作为单片机属性窗口中增强属性(Advanced Properties)下的 EEPROM 初始内容(Initial Contents of EEPROM)进行绑定。

2. 实训要求

① 本例通过循环调用读/写 EEPROM 字节函数访问前 20 个连续的字节数据，完成本例调试后，改用块读函数(eeprom_read_block)与块写函数(eeprom_write_block)完成对它们的读/写操作。

② 编程向 EEPROM 空间中先写入 20 个字数据(word data)，然后读取显示。

③ 编程向 eeprom_array 数组中写入新的数据，然后读取并显示。

3. 源程序代码

```
001  //-------------------------------------------------------------------
002  //  名称：EEPROM 读/写与数码管显示
003  //-------------------------------------------------------------------
004  //  说明：本例运行时，按下 K1 将向 EEPROM 中写入 1～21，按下 K2 时写入无规律
005  //        的 21 个数，按下 K3 时读取 EEPROM 中的 21 个数并循环显示。
006  //        所读/写的 21 个数中，前 20 个在 EEPROM 中的地址是透明的(0x0001～0x0014)，
007  //        最后的第 21 个数其地址是不透明的
008  //
009  //-------------------------------------------------------------------
010  #define  F_CPU   4000000UL                    //4 MHz 晶振
011  #include <avr/io.h>
012  #include <avr/eeprom.h>
013  #include <util/delay.h>
014  #include <stdlib.h>
015  #define INT8U   unsigned char
016  #define INT16U  unsigned int
017
018  //按键定义
```

```
#define Write_1_21_Key_DOWN()   ((PINB & 0x01) == 0x00) //写 1～21(0x01～0x15)
#define Write_Random_Key_DOWN() ((PINB & 0x08) == 0x00) //写随机数
#define Loop_Show_Key_DOWN()    ((PINB & 0x40) == 0x00) //循环显示
//蜂鸣器定义
#define BEEP() (PORTD ^= 0x80)

//将字节变量 eepromx 分配于 EEPROM 存储器(地址不透明)
INT8U eepromx __attribute__((section("eeprom")));

//下面的数组也被分配于 EEPROM,编译后生成 EEP 文件
//(其中 EEMEM 即__attribute__((section("eeprom")))
//后续代码未使用该数组
INT8U eeprom_array[] EEMEM =
{
  0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x1A,0x1B,0x1C,0x1D,0x1E,0x1F,
  0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,
};

//0～9 的数字编码,最后一位为黑屏
const INT8U SEG_CODE[] =
{0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x00};
//分解后的待显示数位
INT8U Display_Buffer[] = {0,0};
//-------------------------------------------------------------------
// 数码管显示字节
//-------------------------------------------------------------------
void Show_Count_ON_DSY()
{
    PORTD = 0xFF;
    PORTC = SEG_CODE[Display_Buffer[1]];
    PORTD = 0xFE;
    _delay_ms(2);
    PORTD = 0xFF;
    PORTC = SEG_CODE[Display_Buffer[0]];
    PORTD = 0xFD;
    _delay_ms(2);
}

//-------------------------------------------------------------------
// 响铃子程序
//-------------------------------------------------------------------
void Play_BEEP()
```

```
{
    INT16U i;
    for (i = 0 ; i< 300; i ++ )
    {
        BEEP(); _delay_us(200);
    }
}

//-------------------------------------------------------------
// 主程序
//-------------------------------------------------------------
int main()
{
    INT8U Current_Data,LOOP_SHOW_FLAG = 0;
    INT16U i,Current_Read_Addr = 0x0001;
    DDRC = 0xFF; PORTD = 0xFF;                    //配置输出端口
    DDRD = 0xFF; PORTD = 0xFF;
    DDRB = 0x00; PORTB = 0xFF;                    //配置输入端口
    srand(200);                                   //设置随机种子
    while(1)
    {
       start:
    //K1:循环显示-----------------------------------------------
    if(Loop_Show_Key_DOWN())
    {
        Current_Read_Addr = 0x0001;       //设为从地址 0x0001 开始显示
        eeprom_busy_wait();
        Current_Data = eeprom_read_byte((INT8U * )Current_Read_Addr);

        //因为写入的数据都不超过 100,如果读 0x0001 时出现 0xFF(255),
        //则说明还未写入过新的数据,这时循环显示标志 LOOP_SHOW_FLAG 仍为关闭
        //否则才打开循环显示标志
        if (Current_Data != 0xFF) LOOP_SHOW_FLAG = 1;
        Play_BEEP();
        while (Loop_Show_Key_DOWN());    //等待释放
    }
    //K2:写入 1~21(0x01~0x15)-----------------------------------
    if (Write_1_21_Key_DOWN())
    {
       LOOP_SHOW_FLAG = 0;
       for (i = 1; i <= 20; i ++ )        //根据 Atmel 公司建议,不使用 0 地址
       {
           eeprom_busy_wait();
```

```
104             eeprom_write_byte((INT8U*)i,(INT8U)i);
105         }
106         //前 20 个数的 EEPROM 地址透明,第 21 个数的 EEPROM 地址不透明
107         eeprom_busy_wait();
108         eeprom_write_byte(&eepromx,0x15);
109         Play_BEEP();
110         while (Write_1_21_Key_DOWN()); //等待释放
111     }
112     //K3:写入 21 个随机数----------------------------------
113     if (Write_Random_Key_DOWN())
114     {
115         LOOP_SHOW_FLAG = 0;
116         for (i = 1; i <= 20; i++)
117         {
118             eeprom_busy_wait();
119             eeprom_write_byte((INT8U*)i,rand() % 100);
120         }
121         //前 20 个数地址透明,第 21 个数地址不透明
122         eeprom_busy_wait();
123         eeprom_write_byte(&eepromx,rand() % 100);
124
125         Play_BEEP();
126         while (Write_Random_Key_DOWN()); //等待释放
127     }
128     if (LOOP_SHOW_FLAG)//--------------------------------
129     {
130         eeprom_busy_wait();
131         if (Current_Read_Addr != 21)     //前 20 个数从透明地址读取
132           Current_Data = eeprom_read_byte((INT8U*)Current_Read_Addr);
133         else                             //第 21 个数从不透明地址读取
134           Current_Data = eeprom_read_byte(&eepromx);
135
136         Display_Buffer[1] = Current_Data/10;
137         Display_Buffer[0] = Current_Data % 10;
138
139         //每个数显示保持一段时间
140         for (i = 0 ; i<160; i++)
141         {
142             Show_Count_ON_DSY();
143             //显示过程中如果某个写入键按下则停止显示
144             if (Write_1_21_Key_DOWN() || Write_Random_Key_DOWN())
145             {
146                 LOOP_SHOW_FLAG = 0;
```

```
147                     PORTD = 0xFF;
148                     goto start;
149                 }
150             }
151             //地址循环递增(1～21)
152             Current_Read_Addr = Current_Read_Addr % 21 + 1;
153         }
154     }
155 }
```

3.33 Flash程序空间中的数据访问

对于在程序运行过程不会发生变化，而且占用空间较大的数据块，本例将其保存到Flash程序空间内，并演示了对这些数据的读取与显示。由于本例通过串口发送数据到虚拟终端显示，程序还应用了异步串行接口程序设计技术。本例电路及部分运行效果如图3-33所示。

1. 程序设计调试

AVR-GCC提供了访问Flash程序内存空间的相关函数，在编写本例程序时，需要添加头文件<avr/pgmspace.h>，相关细节说明可参考avr-libc参考手册。

程序中第19行和22行分别用prog_int8_t类型和prog_int16_t类型将含有320个字节的数组Flash_Byte_Array及含有60个字的Flash_Word_Array数组保存到Flash程序内存中，这大大节省了对AVR单片机RAM空间的占用。

本例分别使用了pgm_read_byte和pgm_read_byte函数从Flash程序内存中读取字节数据与字数据。

为显示所读取的数据，仿真电路中虚拟终端的RXD引脚连接单片机的TXD引脚，从单片机串口发送的字节数据和字数据将显示到虚拟终端上。

在应用串口发送数据时，需要先初始化串口，初始化步骤如下：

① 设置异步串行通信波特率，收发双方的设置要完全一致，否则会出现收发失败或出现乱码。

② 确定USART字符帧结构，包括数据位位数、奇偶校验类型及停止位个数等。

③ 使能发送或接收。

本例的Init_USART函数中编写了如下语句：

```
UCSRB = _BV(TXEN);                              //允许发送
UCSRC = _BV(URSEL) | _BV(UCSZ1)| _BV(UCSZ0);    //8位数据位,1位停止位
UBRRL = (F_CPU/9600/16 - 1) % 256;              //波特率:9600
UBRRH = (F_CPU/9600/16 - 1)/256;
```

前两行设置了USART的控制与状态寄存UCSRB与UCSRC：

第1行将UCSRB中的TXEN置位，允许串口数据发送，如果要允许接收，可再将RXEN置位。

第2行UCSRC寄存中的UCSZ1，UCSZ0位与第一行UCSRB寄存器中的UCSZ2位，即

UCSZ[2：0]共 3 位，共同设置发送或接收字符帧中的数据位位数大小。UCSZ[2：0]3 位的取值为 000、001、010、011、111 时，字符帧数据位位数大小分别为 5、6、7、8、9。本例的设置为 011，即数据位位数为 8 位。另外，由于 UCSRC 中的停止位 USBS 位未置位，取值为 0，表示停止位为 1 位，如果将 USBS 置位则停止位为 2 位。由于该行的设置与默认值相同，故此行可以省略。

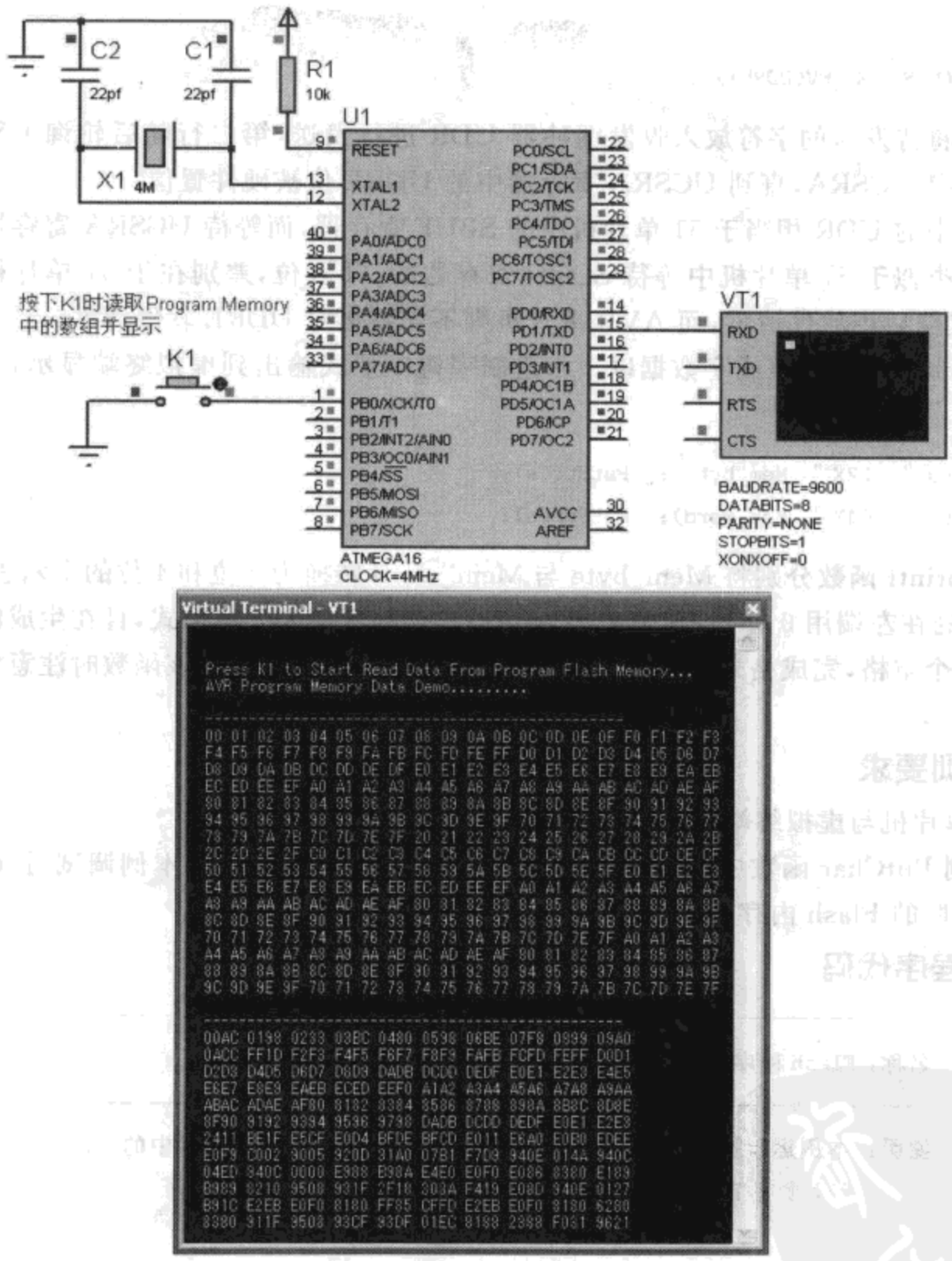


图 3-33　Flash 程序空间中的数据访问

初始化程序最后还需要设置波特率，不同于 51 单片机是，AVR 单片机含有独立的高精度波特率发生器，不需要像 51 单片机那样占用一个定时/计数器。

波特率寄存器 UBRR 由 UBRRH 与 UBRRL 构成，其中 UBRRH 的低 4 位与 UBRRL 的 8 位共 12 位用于保存波特率设置值，UBRRH 的低 4 位是 12 位波特率的高 4 位，UBRRL 中

存放的则是12位波特率的低8位。

初始化程序中给出了根据波特率设置UBRRL与UBRRH的公式，本例将波特率设置为9600。公式计算结果与精确值之间会存在一定误差，一般误差在5%以内是允许的。

完成上述步骤后，就可以进入第③步，利用串口收发数据。本例仅应用串口进行数据发送操作。在第66行的PutChar函数中，发送字符c的关键语句如下：

```
UDR = c;
while(!(UCSRA & _BV(UDRE)));
```

第1行将待发送的字符放入收发缓冲器UDR进行发送，第二行随后轮询USART的控制与状态寄存UCSRA，直到UCSRA寄存器中的UDRE位被硬件置位。

这两行中的UDR相当于51单片机中的SBUF寄存器，而等待UCSRA寄存器的UDRE硬件置位则类似于51单片机中等待SCON寄存器的TI置位，差别在于51单片机中需要在TI被硬件置位后用软件清零，而AVR单片机则不需要再对UDRE软件清零。

为了将十六进制字节或字数据以十六进制字符串形式输出到虚拟终端显示，本例使用了语句：

```
sprintf(s,"%02X ", Mem_byte);  PutStr(s);
sprintf(s,"%04X ", Mem_word);  PutStr(s);
```

其中sprintf函数分别将Mem_byte与Mem_word转换为2位和4位的十六进制数，不足2位或4位的在左端用0补齐，其中a～f与A～F全部转换为大写形式，且在生成的字符串后面补充了一个空格，完成格式转换后，直接输出字符串s即可。引用该函数时注意添加头文件<stdio.h>。

2. 实训要求

① 为单片机与虚拟终端重新选择另一波特率，完成数据发送。

② 本例PutChar函数中使用轮询标志的方式发送字符，在完成本例调试后，改用中断方式发送所读取的Flash内存数据。

3. 源程序代码

```
001  //-------------------------------------------------------------------
002  //  名称：Flash程序空间的数据访问
003  //-------------------------------------------------------------------
004  //  说明：本例运行时，按下K1将读取并显示存放于Flash程序内存中的
005  //        320个字节数据及60个字数据
006  //
007  //-------------------------------------------------------------------
008  #define  F_CPU   4000000UL                    //4 MHz 晶振
009  #include <avr/pgmspace.h>
010  #include <stdio.h>
011
012  #define INT8U   unsigned char
013  #define INT16U  unsigned int
```

```
014
015  //按键定义
016  #define K1_DOWN() (PINB & _BV(PB0)) == 0x00
017
018  //存放于 Flash 程序内存中的字节数据(16 * 20 = 320 个字节)
019  prog_int8_t Flash_Byte_Array[] =
020  {
021    0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,
022    0xF0,0xF1,0xF2,0xF3,0xF4,0xF5,0xF6,0xF7,0xF8,0xF9,0xFA,0xFB,0xFC,0xFD,0xFE,0xFF,
023    0xD0,0xD1,0xD2,0xD3,0xD4,0xD5,0xD6,0xD7,0xD8,0xD9,0xDA,0xDB,0xDC,0xDD,0xDE,0xDF,
024    0xE0,0xE1,0xE2,0xE3,0xE4,0xE5,0xE6,0xE7,0xE8,0xE9,0xEA,0xEB,0xEC,0xED,0xEE,0xEF,
025    0xA0,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,0xA8,0xA9,0xAA,0xAB,0xAC,0xAD,0xAE,0xAF,
026    0x80,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x8A,0x8B,0x8C,0x8D,0x8E,0x8F,
027    0x90,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9A,0x9B,0x9C,0x9D,0x9E,0x9F,
028    0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7A,0x7B,0x7C,0x7D,0x7E,0x7F,
029    0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,0x28,0x29,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,
030    0xC0,0xC1,0xC2,0xC3,0xC4,0xC5,0xC6,0xC7,0xC8,0xC9,0xCA,0xCB,0xCC,0xCD,0xCE,0xCF,
031    0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5A,0x5B,0x5C,0x5D,0x5E,0x5F,
032    0xE0,0xE1,0xE2,0xE3,0xE4,0xE5,0xE6,0xE7,0xE8,0xE9,0xEA,0xEB,0xEC,0xED,0xEE,0xEF,
033    0xA0,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,0xA8,0xA9,0xAA,0xAB,0xAC,0xAD,0xAE,0xAF,
034    0x80,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x8A,0x8B,0x8C,0x8D,0x8E,0x8F,
035    0x90,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9A,0x9B,0x9C,0x9D,0x9E,0x9F,
036    0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7A,0x7B,0x7C,0x7D,0x7E,0x7F,
037    0xA0,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,0xA8,0xA9,0xAA,0xAB,0xAC,0xAD,0xAE,0xAF,
038    0x80,0x81,0x82,0x83,0x84,0x85,0x86,0x87,0x88,0x89,0x8A,0x8B,0x8C,0x8D,0x8E,0x8F,
039    0x90,0x91,0x92,0x93,0x94,0x95,0x96,0x97,0x98,0x99,0x9A,0x9B,0x9C,0x9D,0x9E,0x9F,
040    0x70,0x71,0x72,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7A,0x7B,0x7C,0x7D,0x7E,0x7F
041  };
042
043  //存放于 Flash 程序内存中的字数据(10 * 6 = 60 个字)
044  prog_int16_t Flash_Word_Array[] =
045  {
046    0x00AC,0x0198,0x0233,0x03BC,0x0480,0x0598,0x06BE,0x07F8,0x0899,0x09A0,
047    0x0ACC,0xFF1D,0xF2F3,0xF4F5,0xF6F7,0xF8F9,0xFAFB,0xFCFD,0xFEFF,0xD0D1,
048    0xD2D3,0xD4D5,0xD6D7,0xD8D9,0xDADB,0xDCDD,0xDEDF,0xE0E1,0xE2E3,0xE4E5,
049    0xE6E7,0xE8E9,0xEAEB,0xECED,0xEEF0,0xA1A2,0xA3A4,0xA5A6,0xA7A8,0xA9AA,
050    0xABAC,0xADAE,0xAF80,0x8182,0x8384,0x8586,0x8788,0x898A,0x8B8C,0x8D8E,
051    0x8F90,0x9192,0x9394,0x9596,0x9798,0xDADB,0xDCDD,0xDEDF,0xE0E1,0xE2E3
052  };
053  //-------------------------------------------------------------
054  // USART 初始化
055  //-------------------------------------------------------------
056  void Init_USART()
```

```
057 {
058     UCSRB = _BV(TXEN);                                  //允许发送
059     UCSRC = _BV(URSEL) | _BV(UCSZ1)| _BV(UCSZ0);        //8位数据位,1位停止位
060     UBRRL = (F_CPU/9600/16 - 1) % 256;                  //波特率:9600
061     UBRRH = (F_CPU/9600/16 - 1)/256;
062 }
063 //---------------------------------------------------------------------
064 // 发送一个字符
065 //---------------------------------------------------------------------
066 void PutChar(char c)
067 {
068     if(c == '\n') PutChar('\r');
069     UDR = c;                                            //将待发送字符放入收发缓冲器
070     while(!(UCSRA & _BV(UDRE)));                        //等待UDRE被硬件置位(发送完毕)
071 }
072 //---------------------------------------------------------------------
073 // 发送字符串
074 //---------------------------------------------------------------------
075 void PutStr(char * s)
076 {
077     while ( * s) PutChar( * s ++ );
078 }
079
080 //---------------------------------------------------------------------
081 // 主程序
082 //---------------------------------------------------------------------
083 int main()
084 {
085     INT8U  Mem_byte;
086     INT16U Mem_word, i,j = 0;;
087     char s[6];
088
089     Init_USART();                                       //串口初始化
090     PutStr("\n\n  Press K1 to Start Read Data From Program Flash Memory...");
091
092     DDRB = 0x00; PORTB = 0xFF;                          //配置端口
093     DDRD = 0xFF;
094
095     while(1)
096     {
097        if (K1_DOWN())
098        {
099            PutStr("\n  AVR Program Memory Data Demo.....\n  ");
```

```
100             PutStr("\n  ------------------------------------\n  ");
101             //读取所有字节并显示
102             for (i = 0,j = 0; i< sizeof(Flash_Byte_Array); i ++ )
103             {
104                 //从 Flash 中读取 1 字节
105                 Mem_byte = pgm_read_byte(&Flash_Byte_Array[i]);
106                 //将 1 字节转换为字符串(后带 1 个空格)并送虚拟终端显示
107                 sprintf(s," % 02X ", Mem_byte);  PutStr(s);
108                 if ( ++ j == 20 ) //每行显示 20 个字节
109                 {
110                     j = 0; PutStr("\n  ");
111                 }
112             }
113             PutStr("\n  ------------------------------------\n  ");
114             //读取所有字并显示
115             for (i = 0,j = 0; i< sizeof(Flash_Word_Array); i ++ )
116             {
117                 //从 Flash 中读取 1 个字
118                 Mem_word = pgm_read_word(&Flash_Word_Array[i]);
119                 //将读取的 1 个字整数转换为字符串(后带 1 个空格)并送虚拟终端显示
120                 sprintf(s," % 04X ", Mem_word);  PutStr(s);
121                 if ( ++ j == 10 ) //每行显示 10 个字(整数)
122                 {
123                     j = 0;  PutStr("\n  ");
124                 }
125             }
126         }
127     }
128 }
```

3.34 单片机与 PC 机双向串口通信仿真

通常情况下，虚拟仿真系统是不能与物理环境交互通信的，但 Proteus 虚拟系统模拟了这种能力，它使 Proteus 仿真环境下的系统能与实际的物理环境直接交互，这种模型被称为物理接口模型(PIM)。Proteus 的 COMPIM 组件是一种串行接口组件，当由 CPU 或 UART 软件生成的数字信号出现在 PC 机物理 COM 端口时，它能缓存所接收的数据，并将它们以数字信号的形式发送给 Proteus 仿真电路。如果不希望使用物理串口而使用虚拟串口，使串口调试助手软件能与 Proteus 仿真系统中的单片机串口直接交互，这时还需要安装虚拟串口驱动软件 Virtual Serial Port Driver，简称 VSPD。

本例设计的系统中，单片机可接收 PC 机的串口调试助手软件所发送的数字串，并逐个显示在数码管上，当按下单片机系统的 K1 按键时，会有一串中文字串由单片机串口发送给串口

调试助手软件并显示在软件接收窗口中。

本例电路如图3-34所示，串口调试助手的运行效果如图3-35所示。

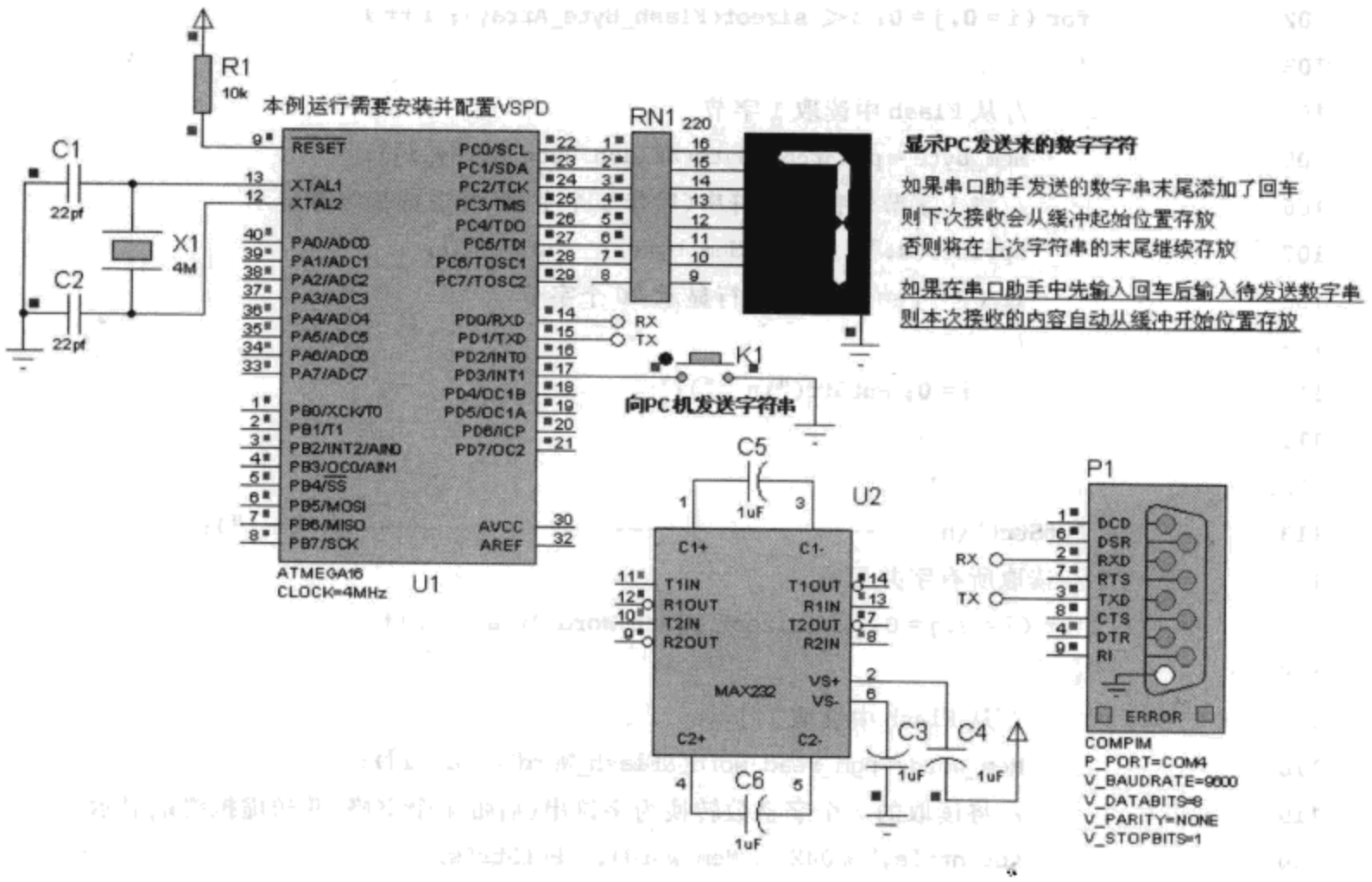


图3-34　单片机与PC机双向串口通信仿真

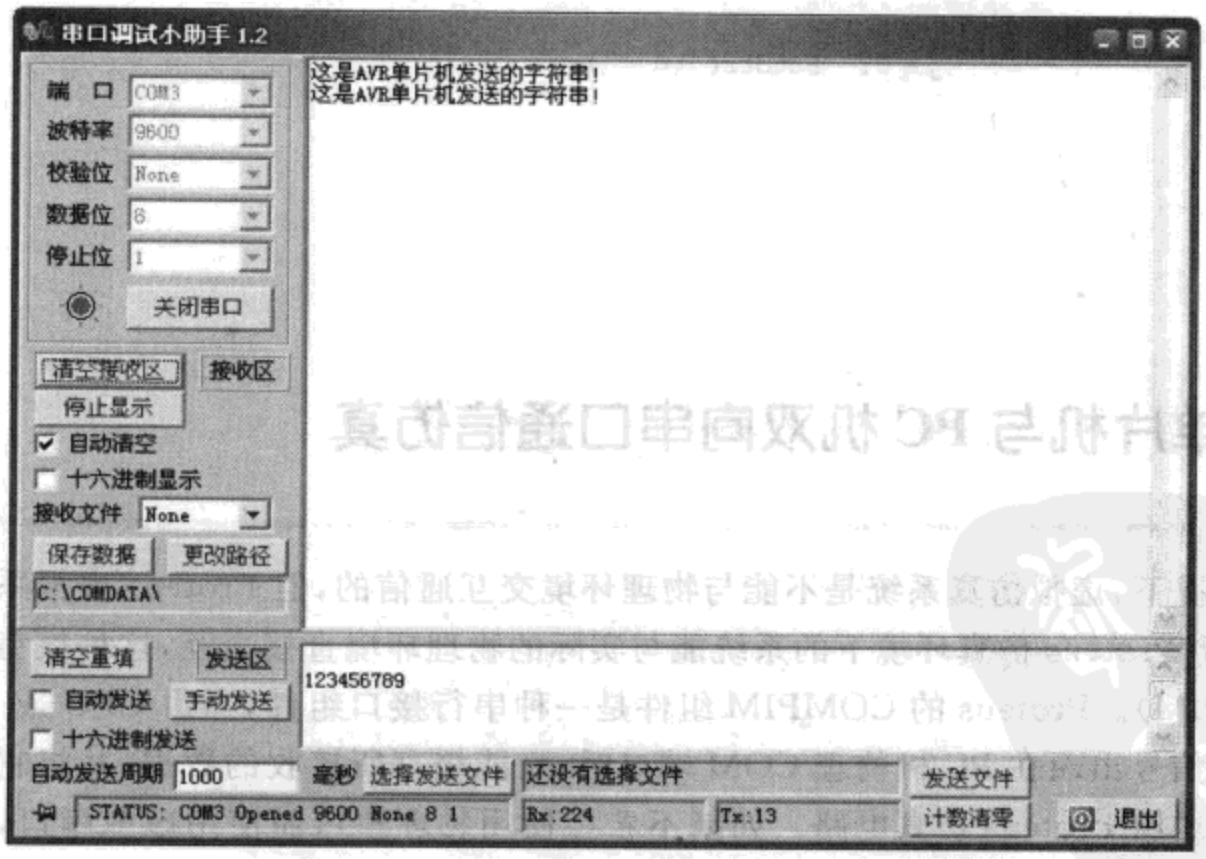


图3-35　串口调试助手

1. 程序设计与调试

与上一案例中有关串口程序设计代码相比，本例有如下差别：

① 初始化程序中第 45 行添加对 RXEN 与 RXCIE 的置位，分别允许接收及允许接收中断。

② 对字符的接收，本例编写了串口接收中断函数 ISR（USART_RXC_vect），读取所接收的字符时使用语句：c=UDR。

③ 在接收与缓存数字串时，使用了数据结构中线性表结构形式。

以上这些部分要重点阅读与调试。

因为本例实现的是 PC 机与单片机之间的双向通信，而且是在纯虚拟仿真环境完成的，下面重点说明本案例的调试方法。

本例实现的 PC 与单片机通信，实际上是 PC 机与 Proteus 中单片机仿真系统的通信，两者的通信通过串口进行，而串口又有虚拟串口和物理串口两种，对于本例也就有了以下几种调试方式，现假设 Proteus 安装在 PC1 中，如果都使用物理串口，调试方法有：

方法一：将串口调试助手软件安装在 PC2，然后用交叉串口线连接 PC1 与 PC2，如果两机都是使用的 COM1，那么在连接好串口线后，应设置 PC1 中的 COMPIM 属性，将串口设为 COM1，波特率等按程序要求设置，对 PC2 中的串口调试软件也要选择 COM1，波特率等要设成与 PC1 中的 COMPIM 相同。完成这些设置后，打开 PC2 中的串口调试软件，并运行 PC1 中的 Proteus 仿真系统，这时如果在串口助手软件输入一串数字并单击发送，PC1 中的数码管即会依次显示这些数字，如果按下 PC1 中单片机系统的 K1 按键，PC2 中的串口调试助手软件会显示：“这是由 AVR 单片机发送的字符串！”并换行，这样即实现了 PC 机与仿真单片机之间的物理串口通信。当然，如果两 PC 都使用 COM2 或一个连接 COM1、另一个连接 COM2 也可以，只是要注意在 COMPIM 组件和串口调试助手上也要做相应改动。

方法二：如果希望串口调试软件与单片机仿真系统同在一台 PC 中运行，假定使用的是 PC1，如果 PC1 有物理串口 COM1 和 COM2，这时可以将这两个串口用交叉线连接，然后仍按上述方法进行调试。不同的是 COMPIM 组件与串口调试软件要分别占用 COM1 和 COM2，不能占用同一个端口。

上述两种方式均使用的是物理串口，如果没有找到合适的串口线，或者使用的 PC 机没有物理串口，这就需要以虚拟串口软件为桥梁，实现串口调试助手与 Proteus 仿真单片机系统的串口通信。调试过程如下：

① 安装虚拟串口驱动程序 VSPD（Virtual Serial Port Driver），安装完成后运行该程序，在图 3－36 所示窗口的 First Port 中选择 COM3，在 Second Port 中选择 COM4（当然，也可以选择 COM5 和 COM6，除非它们已被占用），然后单击 Add Paris 按钮，这两个端口会立即出现在左边的 Virtual Ports 分支下，且会有蓝色虚线将它们连接起来。如果打开 PC 机的设备管理器，会发现在其中的端口下多出了两个串口。显示窗口如图 3－37 所示。

② 将这两个串口中的 COM4 分配给 COMPIM 组件使用，COM3 分配给串口助手使用，由于 COM3 与 COM4 这两个虚拟串口已经由虚拟串口驱动程序 VSPD 虚拟连接，运行同一台 PC 中的串口调试助手软件和 Proteus 中的单片机仿真系统时，两者之间就可以进行正常通信了，这如同使用物理串口连接一样。

2. 实训要求

① 在仿真电路中改用一组 8 位的数码管，重新编程程序，将从 PC 机串口助手软件发送的数据滚动显示在数码管上。

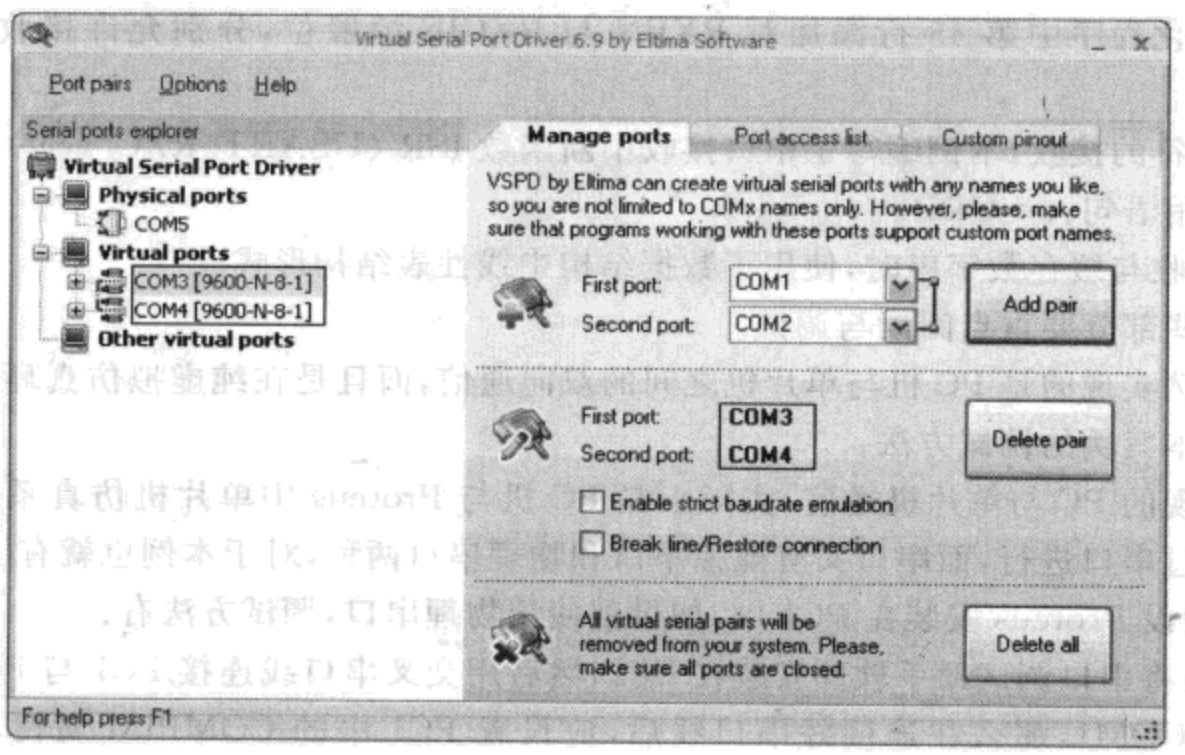


图 3-36　虚拟串口驱动软件

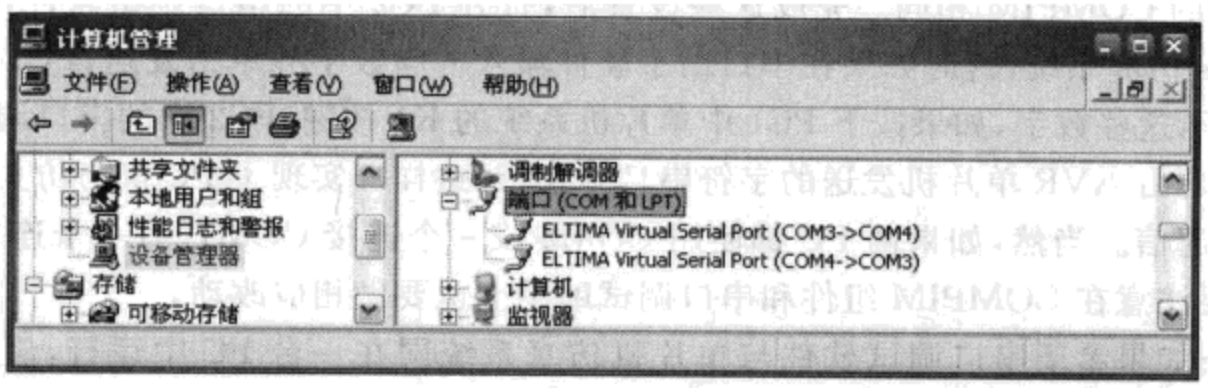


图 3-37　计算机端口管理

② 自编一个上位机 Windows 软件(使用 VB6、VC6、VS. NET 等开发工具),实现对下位单片机系统的控制。在软件中单击“开”按钮时,单片机能控制电机启动;单击“关”按钮时电机停止。调节单片机电路中的可变电阻 RV1 时,模/数转换结果能发送给上位机软件显示。

3. 源程序代码

```
001  //-------------------------------------------------------------------
002  //  名称:单片机与 PC 机双向串口通信仿真
003  //-------------------------------------------------------------------
004  //  说明:单片机可接收 PC 机发送的数字字符,按下单片机 K1 按键时,单片机
005  //        可向 PC 机发送字符串。在 Proteus 环境下完成本实验时,需要先安
006  //        装 Virtual Serial Port Driver 和串口调试助手软件。
007  //        建议在 VSPD 中将 COM3 和 COM4 设为对联端口。Proteus 中设 COMPIM
008  //        为 COM4,在串口助手中选择 COM3,然后实现单片机程序与 XP 下串口
009  //        助手的通信
010  //
011  //        本例缓冲为 100 个数字字符,如果发送的字符串末尾没有回车符,
012  //        则下次接收的字符串将在上次接收字符串的后面接着存放,否则
013  //        将重新从开始位置存放
```

```
014  //
015  //          如果本次 PC 发送的数字串是先输入回车符,再输入任意数字串,
016  //          则本次新接收的数字串也将从缓冲开始位置存放
017  //
018  //-------------------------------------------------------------
019  #define  F_CPU   4000000UL              //4 MHz 晶振
020  #include <avr/io.h>
021  #include <avr/interrupt.h>
022  #include <util/delay.h>
023  #define INT8U   unsigned char
024  #define INT16U  unsigned int
025
026  //数字串接收缓冲
027  struct
028  {
029      INT8U Buf_Array[100];              //缓冲空间
030      INT8U Buf_Len;                     //当前缓冲长度
031  } Receive_Buffer ;
032
033  //清空缓冲标志
034  INT8U Clear_Buffer_Flag = 0;
035  //0~9 的数字编码,最后一位为黑屏
036  const INT8U SEG_CODE[] =
037  {0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x00};
038
039  char * s = "这是 AVR 单片机发送的字符串! \n", * p;
040  //-------------------------------------------------------------
041  // USART 初始化
042  //-------------------------------------------------------------
043  void Init_USART()
044  {
045      UCSRB = _BV(RXEN) | _BV(TXEN) | _BV(RXCIE);      //允许接收和发送,接收中断使能
046      UCSRC = _BV(URSEL)| _BV(UCSZ1)| _BV(UCSZ0);      //8 位数据位,1 位停止位
047      UBRRL = (F_CPU/9600/16 - 1) % 256;               //波特率:9600
048      UBRRH = (F_CPU/9600/16 - 1)/256;
049  }
050  //-------------------------------------------------------------
051  // 发送一个字符
052  //-------------------------------------------------------------
053  void PutChar(char c)
054  {
055      if(c == '\n') PutChar('\r');
056      UDR = c;
```

```
057         while(!(UCSRA & _BV(UDRE)));
058     }
059     //-------------------------------------------------------------------
060     // 显示所接收的数字字符(数字字符由PC串口发送,AVR串口接收)
061     //-------------------------------------------------------------------
062     void Show_Received_Digits()
063     {
064         INT8U i;
065         for (i = 0; i< Receive_Buffer.Buf_Len; i ++ )
066         {
067             PORTC = SEG_CODE[ Receive_Buffer.Buf_Array[i] ];
068             _delay_ms(400);
069         }
070     }
071     //-------------------------------------------------------------------
072     // 主程序
073     //-------------------------------------------------------------------
074     int main()
075     {
076         Receive_Buffer.Buf_Len = 0;
077         DDRB = 0x00; PORTB = 0xFF;              //配置端口
078         DDRC = 0xFF; PORTC = 0x00;
079         DDRD = 0x02; PORTD = 0xFF;
080         MCUCR = 0x08;                           //INT1中断下降沿触发
081         GICR   = _BV(INT1);                     //INT1中断许可
082         Init_USART();                           //串口初始化
083         sei();
084         while(1) Show_Received_Digits();    //显示所接收到数字
085     }
086
087     //-------------------------------------------------------------------
088     // 串口接收中断函数
089     //-------------------------------------------------------------------
090     ISR (USART_RXC_vect)
091     {
092         INT8U c = UDR;
093         //如果接收到回车换行符则设置清空缓冲标志
094         if (c == '\r' || c == '\n') Clear_Buffer_Flag = 1;
095         if (c >= '0' && c <= '9')
096         {
097             //如果上次曾收到清空缓冲标志,则本次从缓冲开始位置存放
098             if (Clear_Buffer_Flag == 1)
099             {
100                 Receive_Buffer.Buf_Len = 0;
```

```
101                Clear_Buffer_Flag = 0;
102            }
103            //缓存新接收的数字
104            Receive_Buffer.Buf_Array[Receive_Buffer.Buf_Len] = c - '0';
105            //刷新缓冲长度(不超过最大长度)
106            if (Receive_Buffer.Buf_Len<100 ) Receive_Buffer.Buf_Len ++ ;
107        }
108    }
109
110    //-------------------------------------------------------------------
111    // INT1 中断函数(向 PC 发送字符串)
112    //-------------------------------------------------------------------
113    ISR (INT1_vect)
114    {
115        INT8U i = 0;
116        while (s[i]!= '\0') PutChar(s[i ++ ]); //向 PC 发送字符串
117    }
```

3.35 看门狗应用

单片机的工作常会受到来自外界电磁场的干扰,造成程序跑飞,单片机系统无法继续正常工作。本例演示了启动看门狗,用定时器喂狗以及停止喂狗导致单片机重启的过程。在启动完成后,LED1 熄灭,LED2 开始持续闪烁,一旦停止喂狗则系统自动重启,LED1 在启动时被再次点亮一次,然后熄灭,LED2 再次重新开始闪烁,系统重新进入正常运行状态。本例电路及部分运行效果如图 3-38 所示。

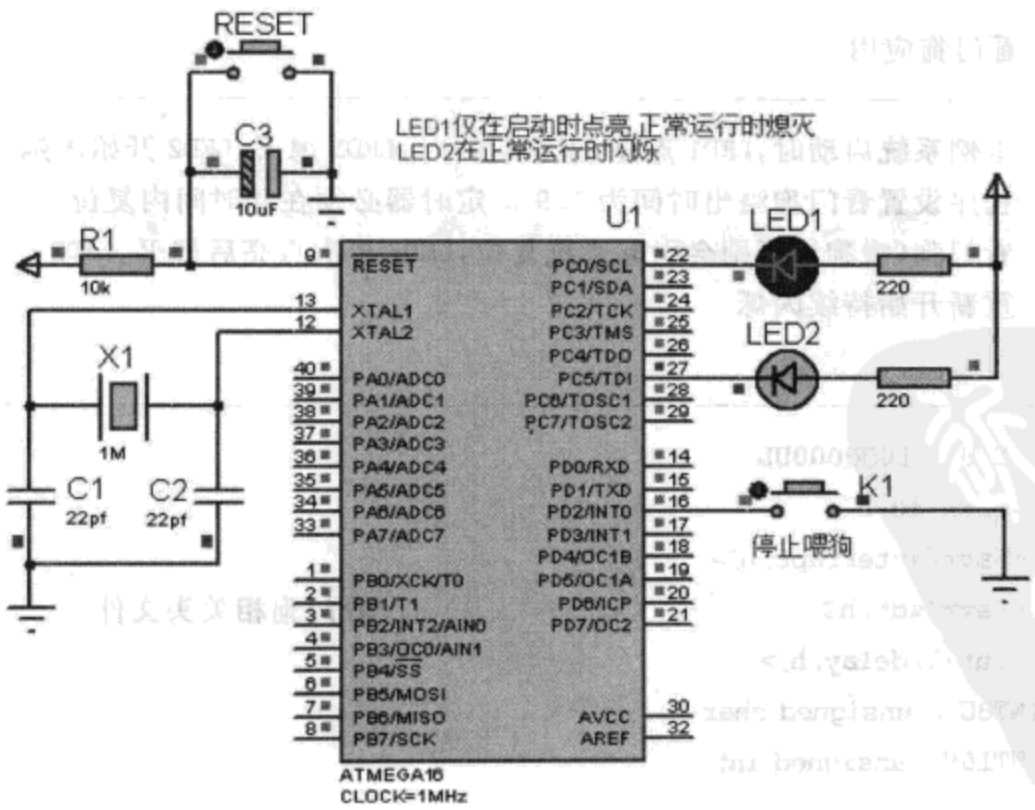


图 3-38 看门狗应用

1. 程序设计与调试

AVR-GCC提供了看门狗(watchdog)的相关控制函数,在应用看门狗时需要添加头文件:<avr/wdt.h>。通过wdt.h的宏定义wdt_enable和wdt_reset可以非常方便地启用和复位看门狗。

电路中LED1仅在开始时点亮,完成T/C1定时器溢出中断与外部INT0中断配置后,通过调用wdt_enable启动看门狗,喂狗时间设为2 s(1.9 s),LED2熄灭,随后即进入主程序中的while(1)循环,应用系统要正常处理的事务将在该循环中完成,本例所放置的代码仅控制LED2的闪烁,它表示单片机处于正常运行状态下。

当LED2开始闪烁时,表示程序开始运行正常,T/C1定时器溢出中断函数每隔1.5 s(<1.9 s)调用wdt_reset复位看门狗(喂狗),这样可使系统持续正常运行。当按下K1时,它模拟了异常事件导致定时器停止工作。系统出现故障、喂狗停止、程序跑飞的状态,由于喂狗时间超过2 s,这导致单片机应用系统自动重启。LED1被再次点亮,然后熄灭,随后LED2再次开始持续闪烁,系统重新恢复正常。

在调试运行本例时,按下K1停止喂狗可使单片机自动重启。这与按下电路中的热启动键RESET所观察到的效果是一样的,区别在于按下RESET键是"手动重启"系统,而按下K1则模拟了系统遇到故障后"自动重启"的过程。

2. 实训要求

① 重新配置喂狗时间为8 s,并修改相关定时器中断程序,实现上述仿真效果。

② 本例将LED2闪烁作为正常运行的任务,在完成本例调试后,将数码管显示当前时钟信息作为主程序的正常运行任务,在系统出现故障时能自动重启,然后再重新进入正常运行状态。

3. 源程序代码

```
01 //-------------------------------------------------------------------
02 //  名称:看门狗应用
03 //-------------------------------------------------------------------
04 //  说明:本例系统启动时,LED1点亮,正常运行时,LED1熄灭,LED2开始闪烁
05 //        程序设置看门狗溢出时间为1.9 s,定时器必须在此时间内复位
06 //        看门狗(喂狗),否则会引起系统复位,LED1再次点亮后熄灭,LED2
07 //        重新开始持续闪烁
08 //
09 //-------------------------------------------------------------------
10 #define F_CPU   1000000UL
11 #include <avr/io.h>
12 #include <avr/interrupt.h>
13 #include <avr/wdt.h>                          //看门狗相关头文件
14 #include <util/delay.h>
15 #define INT8U   unsigned char
16 #define INT16U  unsigned int
17
18 //分别定义LED1开/关,LED2闪烁
19 #define LED1_ON()     (PORTC &= ~_BV(PC0))
```

```
20  #define LED1_OFF()    (PORTC |=   _BV(PC0))
21  #define LED2_BLINK() (PORTC ^=   _BV(PC5))
22  //-------------------------------------------------------------------
23  // 主程序
24  //-------------------------------------------------------------------
25  int main()
26  {
27      DDRC = 0xFF; PORTC = 0xFF;                  //配置端口
28      DDRD = 0x00; PORTD = 0xFF;
29      LED1_ON();                                  //LED1 点亮
30      _delay_ms(1600);
31
32      MCUCR = 0x02;                               //INT0 中断下降沿触发
33      GICR   = _BV(INT0);                         //INT0 中断许可
34      TCCR1B = 0x03;                              //T1 预设分频:64
35      TCNT1   = 65536 - F_CPU/64.0 * 1.5;         //晶振 4 MHz,1.5 s 定时初值
36      TIMSK   = 0x04;                             //允许 T1 定时器溢出中断
37      wdt_enable(WDTO_2S);                        //启动看门狗(溢出时间 1.9 s,约等于 2.0 s)
38      //WDTCR = 0x0F;                             //用这一行也可以
39      LED1_OFF();                                 //LED1 熄灭
40      sei();                                      //开中断
41      while(1)
42      {
43          LED2_BLINK();                           //LED2 闪烁
44          _delay_ms(200);
45      }
46  }
47
48  //-------------------------------------------------------------------
49  // 定时器 1 中断程序负责喂狗(1.9 s 以内)
50  //-------------------------------------------------------------------
51  ISR (TIMER1_OVF_vect)
52  {
53      TCNT1 = 65536 - F_CPU/64.0 * 1.5;           //1.5 s 定时初值
54      wdt_reset() ;                               //看门狗复位
55  }
56
57  //-------------------------------------------------------------------
58  // INT0 中断函数(按下 K1 时关闭定时器,停止喂狗)
59  //-------------------------------------------------------------------
60  ISR (INT0_vect)
61  {
62      TIMSK = 0x00;
63  }
```

第4章

硬件应用

通过对第3章基础案例的学习与调试，大家已经熟悉了AVRStudio＋WinAVR开发环境下单片机内部资源的基本程序设计方法，知道如何利用AVR单片机C语言程序设计实现基本的系统功能。本章将在此基础上就单片机的外围硬件扩展提出数十个案例，这些硬件包括大量数字逻辑芯片、驱动芯片、机电器件、显示器件、传感器件等。通过认真的学习研究与跟踪调试，以及对实训要求的认真完成，大家一定会进一步熟悉和掌握单片机外围扩展硬件的应用方法与技巧，积累更多应用经验，进一步提高AVR单片机应用系统的C语言程序开发能力，为单片机系统的综合设计打下基础。

4.1 74HC138与74HC154译码器应用

本例单片机PB与PC端口分别外接3-8译码器与4-16译码器，程序在PB端口低3位输出000、001、010、011、…、111，通过3-8译码器控制8只LED滚动点亮。在PC端口低4位则循环输出0000、0001、0010、0011、…、1111，通过4-16译码器控制16只LED循环滚动显示。本例电路及部分运行效果如图4－1所示。

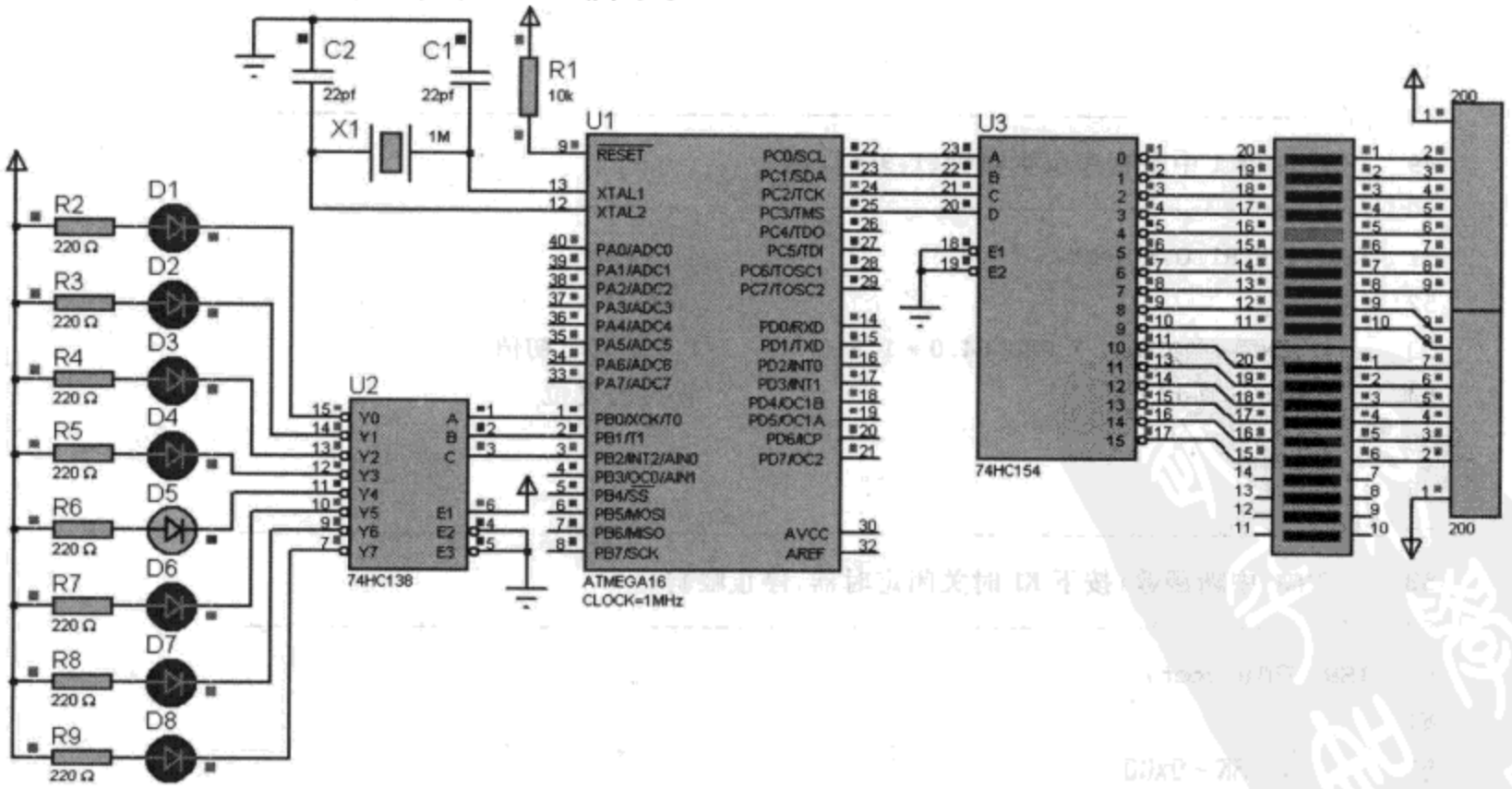


图4－1　74HC138与74HC154译码器应用

1. 程序设计与调试

表 4－1 是 3-8 译码器 74HC138 的真值表。PB 端口低 3 位连接 3-8 译码器的 CBA 输入端，依次输入 000、001、010、011、…、111。根据 3-8 译码器的真值表可知，在向 3-8 译码器输入 000 时，输出端 Y0 引脚为 0；输入 001 时，输出端 Y1 引脚为 0；输入 111 时，输出端 Y7 引脚为 0，这样即形成了 8 只 LED 逐个滚动点亮的效果。

表 4－1　3-8 译码器 74HC138 的真值表

输　入					输　出							
使能位		选择位										
G1	G2＊	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	H	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

注：　G2＝G2A＋G2B(本例中 GA＝E2＋E3)。

控制 3-8 译码器的语句是 PORTB＝(PORTB＋1) & 0x07。该语句使 PB 端的输出范围为 0～7，即 00000000～00000111，其高 5 位保持为 00000，而低 3 位由 000、001、010、……，一直递增到 111，经译码器译码后即形成 LED 滚动显示的效果。

单片机 PC 端口低 4 位连接 4-16 译码器的 DCBA 输入端，依次输入 0000、0001、0010、0011、……，直到 1111。根据 4-16 译码器的真值表可知，输入 0000 时，输出端引脚 0(Y0)为 0；当输入 0001 时，输出端引脚 1(Y1)为 0；当输入 1111 时，输出端引脚 15(Y15)为 0；16 只 LED 逐个滚动点亮的效果由此形成。表 4－2 给出了 4-16 译码器 74HC154 的真值表。

表 4－2　4-16 译码器 74HC154 的真值表

输　入			输　出														
G1G2		DCBA	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	L	LLLL	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	LLLH	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	LLHL	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H	H
L	L	LLHH	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H	H
L	L	LHLL	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H	H
L	L	LHLH	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H	H

续表 4-2

输入			输出														
G1G2		DCBA	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	L	LHHL	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H	H
L	L	LHHH	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H	H
L	L	HLLL	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H	H
L	L	HLLH	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H	H
L	L	HLHL	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H	H
L	L	HLHH	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H	H
L	L	HHLL	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H	H
L	L	HHLH	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H	H
L	L	HHHL	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L	H
L	L	HHHH	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	L
L	H	XXXX	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	L	XXXX	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	H	XXXX	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H

控制 4-16 译码器的语句是 PORTC=(PORTC+1) & 0x0F,它使 PC 端口的输出范围为 0～15(即 00000000～00001111),其高 4 位保持为 0000,而低 4 位由 0000、0001、0010、……,一直递增到 1111,经译码器译码后使相应的 LED 点亮,形成 LED 滚动显示的效果。

2. 实训要求

① 删除所有连接 3-8 译码器的 LED,重新加入 8 位七段数码管,用 3-8 译码器控制数码管位码,用 PA 端口控制段码,实现数码管数据显示。

② 在成功调试 4.9 节有关 LED 点阵屏的案例后,删除本例中连接 4-16 译码器的条形 LED,重新加入两片水平并排 8×8 LED 点阵显示屏,用 4-16 译码器控制列码(两片共 16 列),行码由 PA 端口控制,实现 2 片点阵屏的静态或滚动显示效果。这样设计可大大减少对单片机 I/O 端口的占用,如果直接控制 16 列,单片机将有 2 个端口被完全占用,使用译码器后只需要一个端口的 4 只引脚即可。

3. 源程序代码

```
01  //-------------------------------------------------------------------
02  //  名称：74HC138 与 74HC154 译码器应用
03  //-------------------------------------------------------------------
04  //  说明：本例运行时,PB 与 PC 端口分别循环输出 0x00～0x07,0x00～0x0F
05  //        两译码器的输出端 Y0～Y7 与 Y0～Y15 分别逐个呈现低电平
06  //        两组 LED 分别循环滚动显示
07  //
08  //-------------------------------------------------------------------
09  #define  F_CPU   1000000UL
10  #include <avr/io.h>
```

```
#include <util/delay.h>
#define INT8U   unsigned char
#define INT16U  unsigned int

//-------------------------------------------------------------------
// 主程序
//-------------------------------------------------------------------
int main()
{
    DDRB = 0xFF; PORTB = 0x00;              //配置 PB,PC 端
    DDRC = 0xFF; PORTC = 0x00;              //初始输出均为 0x00
    while(1)
    {
        PORTB = (PORTB + 1) & 0x07;         //3-8 译码器输出
        PORTC = (PORTC + 1) & 0x0F;         //4-16 译码器输出
        //以上两行还可以改写成:
        //PORTB = (PORTB + 1) % 8;
        //PORTC = (PORTC + 1) % 16;
        _delay_ms(80);                      //延时
    }
}
```

4.2 74HC595 串入并出芯片应用

本例单片机外接一片串入并出芯片 74HC595,该芯片仅占用单片机 PC 端口 3 只引脚,驱动单只数码管实现数字滚动显示。74HC595 芯片在后续有关 LED 点阵显示屏的案例中还会再次用到,通过本例调试要熟练掌握该芯片的程序设计方法。本例电路及部分运行效果如图 4-2 所示。

1. 程序设计与调试

74HC595 的输出端为 Q0～Q7,这 8 位并行输出端可以直接控制数码管的 8 个管段(本例数码管没有小数点,仅连接了数码管的 7 个引脚)。Q7′为级联输出端,它用来连接下一片 595 的串行数据输入端 DS。

74HC595 的控制端说明如下:

① SH_CP(11 脚)用于输入移位时钟脉冲,在上升沿时移位寄存器(Shift Register)数据移位,Q0→Q1→Q2→Q3→Q4→Q5→Q6→Q7→Q7′,其中 Q7′用于 595 的级联。本例中的 595 串行输入函数 Serial_Input_595 使用了 SH_CP 引脚及下面的 DS 引脚。

② DS(14 脚)为串行数据输入引脚,Serial_Input_595 函数通过移位运算符由高位到低位将位数据通过 DS 引脚串行送入 595 芯片,串行发送时由 SH_CP 引脚提供移位时钟。for 循环控制完成 8 次移位即可完成一个字节的串行传送。

③ ST_CP(12 脚)提供锁存脉冲,在上升沿时移位寄存器的数据被传入存储寄存器,由于

$\overline{OE}$引脚接地，传入存储寄存器的数据会直接出现在输出端 Q0～Q7。在串行输入函数完成一个字节的传送后，并行输出函数 Parallel_Output_595 在 ST_CP 的上升沿将数据送出。

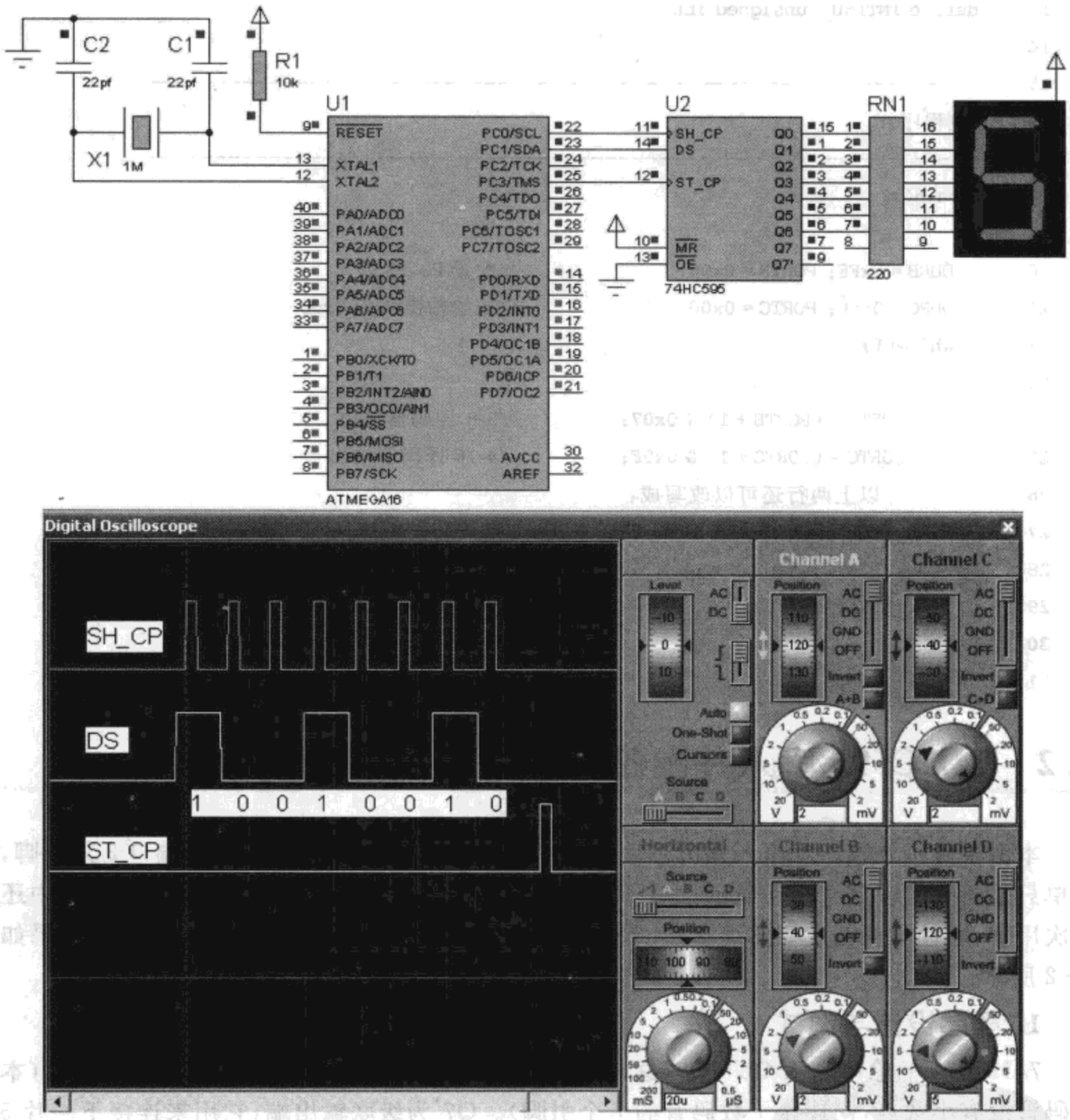


图 4－2　74HC595 串入并出芯片应用

④ $\overline{MR}$(10 脚)在低电平时将移位寄存器数据清零，本例中该引脚直接连接 Vcc。

⑤ $\overline{OE}$(13 脚)在高电平时禁止输出(高阻态)，本例中该引脚接地。

75HC595 其主要优点是能锁存数据，在移位过程中输出端的数据保持不变，这有利于使数码管在串行速度较慢的场合不会出现闪烁感。

图 4－2 中所示虚拟示波器的 A、B、C 通道与 SH_CP、DS、ST_CP 引脚对应，当前显示的波形与显示数字“5”(段码为 0x92，即 10010010)的操作时序对应，其中 A、B 通道波形与函数 Serial_Input_595 对应，该函数向 DS 引脚发送数据与并向 SH_CP 引脚输出移位时钟。通道 C 的波形与函数 Parallel_Output_595 对应，在完成一个字节发送后，向 ST_CP 引脚输入锁存

脉冲，在脉冲上升沿将所输入的字节送到输出锁存器。

本例的重要函数 Serial_Input_595 通过 for 循环在 SH_CP 引脚模拟输出 8 个时钟周期，将一个字节由高到低逐位通过 DS 线串行移入 595。该函数的编写模式对其他串行器件的数据写入代码编写都有参考作用，大家要熟练掌握。

2. 实训要求

① 思考源程序中第 42 行为什么可以省略，移到 for 循环后面又有什么作用？

② 再添加 1 片 74HC595 和 1 只数码管，将 2 片 74HC595 级联，仍使用 PC 端口 3 只引脚，实现对两只独立数码管的显示控制。

③ 重新修改本例电路与程序，用两片 595 芯片分别控制 8 位集成式七段数码管的段码与位码，以静态或滚动方式显示指定数据信息。

3. 源程序代码

```
01  //-------------------------------------------------------------------
02  //  名称：74HC595 串入并出芯片应用
03  //-------------------------------------------------------------------
04  //  说明：74HC595 具有一个 8 位串入并出的移位寄存器和一个 8 位输出锁存器
05  //        本例使用 74HC595，通过串行输入数据来控制数码管显示
06  //
07  //-------------------------------------------------------------------
08  #define  F_CPU   1000000UL
09  #include <avr/io.h>
10  #include <util/delay.h>
11  #define INT8U   unsigned char
12  #define INT16U  unsigned int
13
14  //595 引脚定义
15  #define SH_CP PC0                    //移位时钟脉冲
16  #define DS    PC1                    //串行数据输入
17  #define ST_CP PC3                    //输出锁存器控制脉冲
18
19  //595 引脚操作定义
20  #define SH_CP_0()  PORTC &= ~_BV(SH_CP)
21  #define SH_CP_1()  PORTC |=  _BV(SH_CP)
22  #define DS_0()     PORTC &= ~_BV(DS)
23  #define DS_1()     PORTC |=  _BV(DS)
24  #define ST_CP_0()  PORTC &= ~_BV(ST_CP)
25  #define ST_CP_1()  PORTC |=  _BV(ST_CP)
26
27  //数码管段码表
28  const INT8U SEG_CODE[] =
29  {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90};
30  //-------------------------------------------------------------------
```

```
31  // 串行输入子程序
32  //-------------------------------------------------------------------
33  void Serial_Input_595(INT8U dat)
34  {
35     INT8U i;
36     for(i = 0; i<8; i ++ )
37     {
38        if (dat & 0x80) DS_1(); else DS_0();  //发送高位
39        dat <<= 1;                            //次高位左移到高位
40        SH_CP_0(); _delay_us(2); //移位时钟线拉低
41        SH_CP_1(); _delay_us(2); //放在DS线的0或1在移位时钟脉冲上升沿被移入595
42        SH_CP_0(); _delay_us(2); //本行可以省略,也可移到for循环后面
43     }
44  }
45
46  //-------------------------------------------------------------------
47  // 并行输出子程序
48  //-------------------------------------------------------------------
49  void Parallel_Output_595()
50  {
51      ST_CP_0(); _delay_us(1);
52      ST_CP_1(); _delay_us(1);  //上升沿将数据送到输出锁存器
53      ST_CP_0(); _delay_us(1);
54  }
55
56  //-------------------------------------------------------------------
57  // 主程序
58  //-------------------------------------------------------------------
59  int main()
60  {
61      INT8U i = 0;
62      DDRC = 0xFF;                  //PC端口设为输出
63      while (1)
64      {
65          for(i = 0; i<10; i ++ )
66          {
67             //将数字i的段码字节串行输入595
68             Serial_Input_595(SEG_CODE[i]);
69             //595移位寄存器数据传输到存储寄存器并出现在输出端
70             Parallel_Output_595();
71             _delay_ms(300);
72          }
73      }
74  }
```

4.3 用 74LS148 与 74LS21 扩展中断

本例所选用单片机的 PD2(INT0)、PD3(INT1)、PB2(INT2)用于输入外部中断信号，当需要对更多的外部中断信号作出响应时就需要进行中断扩展了。实现中断扩展的方法较多，本例使用 8-3 编码芯片 74LS148 实现中断扩展，8 路外部中断信号可按优先级进行处理。另外，本例还利用具有双四路输入的与门芯片 74LS21 扩展中断，两者的差别将在程序设计与调试部分中阐述。本例电路及部分运行效果如图 4-3 所示。

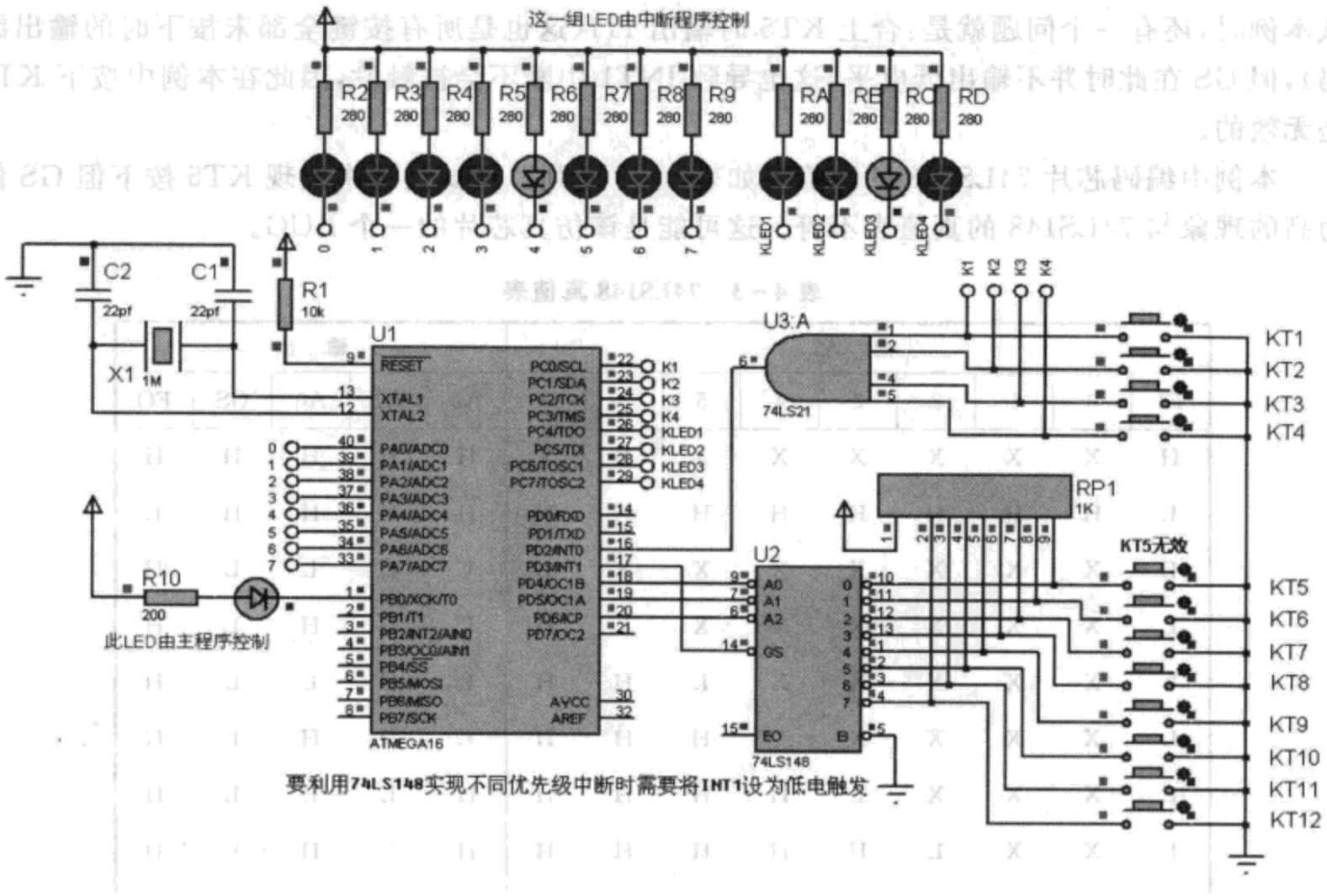


图 4-3 用 74LS148 与 74LS21 扩展中断

1. 程序设计与调试

74LS148 是带优先级的 8-3 编码芯片，对于外部的 8 路数据输入线，只要有 1 路或几路被置为 0，编码芯片即会按由高到低的优先级进行编码，并由 A2～A0 引脚输出 3 位二进制数，而且 GS 引脚会自动变为 0。在没有任何输入、8 路数据线均为高电平时，GS 自动变为 1。

本例将 GS 连接单片机的 PD3(INT1)引脚，当 GS 为 0 时即会触发 INT1 中断，中断程序根据 A2～A0 引脚输入的 3 位二进制编码执行相应操作。

由于 74LS148 是带优先级的，按键 KT12～KT5 模拟的中断级别由高到低，如果单击 KT9 右边的红色双向箭头将 KT9 按下锁住，这时再按下 KT5～KT8 中的任何按键，8-3 编码器的输出都不会发生变化，只有按下 KT10～KT12 时输出的 3 位编码才会变化。

在调试本例时会发现，如果将 KT9 锁住(保持按下状态)，这时按下更高级别的按键，虽然输出的 3 位编码会发生变化，但对应的指示 LED 却没有移动，这是因为本例设 INT1 为下降

沿触发，如果希望有低级别按键按下且锁定时，按下高级别按键还能立即触发INT1中断，这要通过设MCUCR=0x02将INT1配置为低电平触发。

这样设置后，即使低级别按键未释放，高级别按键事件也会马上被处理，指示LED会立即变化。如果释放高级别按键，LED会立即回到原位，除非低级别按键也释放了。

在设为低电平触发后，大家又会发现另一个问题，那就是左边由主程序控制的LED不能正常闪烁了。这是因为设为低电平触发后，只要有按键没有释放，INT1的中断程序就会处于无限次调用之中，主程序中控制LED闪烁的语句也就没有足够的时间执行了。

对于8-3编码器，0～7号引脚按键按下时，输出编码为111～000(不是000～111)。在调试本例时，还有一个问题就是：合上KT5时输出111(这也是所有按键全部未按下时的输出编码)，但GS在此时并不输出低电平，这也导致INT1中断不会被触发，因此在本例中按下KT5是无效的。

本例中编码芯片74LS148的真值表如表4-3所列。调试过程中发现KT5按下但GS仍为高的现象与74LS148的真值表不符。这可能是该仿真芯片的一个BUG。

表4-3　74LS148真值表

输　入									输　出				
EI	0	1	2	3	4	5	6	7	A2	A1	A0	GS	EO
H	X	X	X	X	X	X	X	X	H	H	H	H	H
L	H	H	H	H	H	H	H	H	H	H	H	H	L
L	X	X	X	X	X	X	X	L	L	L	L	L	H
L	X	X	X	X	X	X	L	H	L	L	H	L	H
L	X	X	X	X	X	L	H	H	L	H	L	L	H
L	X	X	X	X	L	H	H	H	L	H	H	L	H
L	X	X	X	L	H	H	H	H	H	L	L	L	H
L	X	X	L	H	H	H	H	H	H	L	H	L	H
L	X	L	H	H	H	H	H	H	H	H	L	L	H
L	L	H	H	H	H	H	H	H	H	H	H	L	H

对于另一组中断扩展，电路中使用了双四路输入的74LS21与门芯片(本例电路中只用了“半片”74LS21)，按键KT1～KT4右端接地，左端接与门输入端，且全部由PC端口高4位内部上拉(设为输入)，显然，KT1～KT4中任何一个铵键按下，与门输出端都会向PD2(INT0)引脚输入0，触发INT0中断，INT0中断程序通过读取PC端口(PINC)高4位即可知道是哪一按键触发中断，这种设计不具有8-3编码器所具有的优先级，占用的引脚数也更多。

2. 实训要求

① 将INT1配置为低电平触发，使多路按键按下时能按不同优先级作出响应。模拟多路按键按下时，可先单击一个或多个按键右上角的红色双向箭头将其按下并锁住。

② 使用整片74LS21实现对外部8路中断的处理(输出端占用INT0与INT1)。

③ 搜索Proteus芯片库，选用其他数字芯片实现中断扩展。

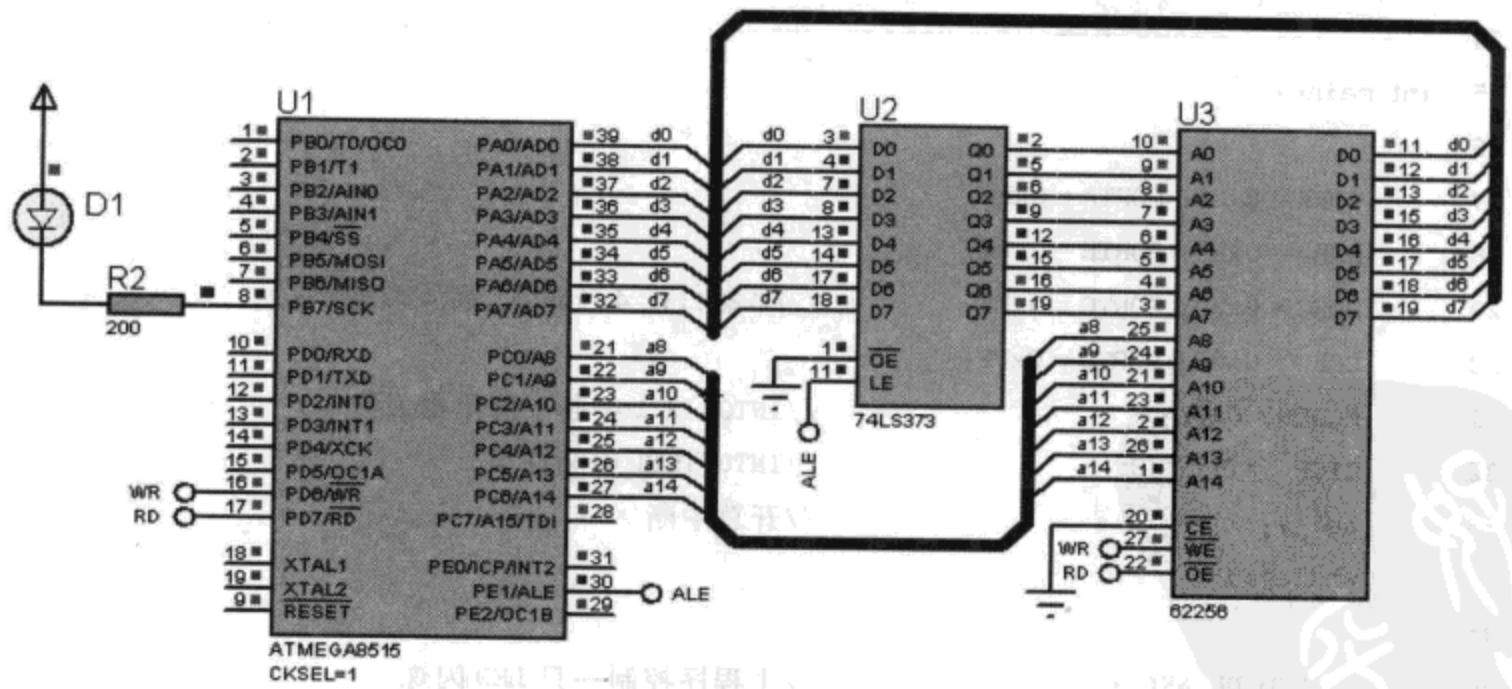
3. 源程序代码

```
01  //-------------------------------------------------------------------
02  //  名称：用 74LS148/74LS21 扩展中断
03  //-------------------------------------------------------------------
04  //  说明：本例利用 74LS148 扩展外部中断，对于外部的 8 个控制开关，任意
05  //        一个开关合上都将在 GS 引脚输出低电平，触发外部中断，优先级最
06  //        高的是输入引脚 7，最低的是输入引脚 0。中断触发后，中断例程通过
07  //        读取 A2、A1、A0 的输出，判断是哪一路按键触发中断
08  //
09  //        对于 74LS21，任何一个按键都会触发中断，它并没能真正实现中断
10  //        扩展，而是仅利用了 INT0，省去了对多个按键的轮询判断
11  //
12  //-------------------------------------------------------------------
13  #include <avr/io.h>
14  #include <avr/interrupt.h>
15  #include <util/delay.h>
16  #define INT8U   unsigned char
17  #define INT16U  unsigned int
18
19  //此 LED 由主程序控制
20  #define LED_BLINK()  PORTB ^= _BV(PB0)
21  //-------------------------------------------------------------------
22  // 主程序
23  // 说明：由于 Proteus 中 74LS148 存在问题，与输入引脚 0 对应的开关控制无效
24  //-------------------------------------------------------------------
25  int main()
26  {
27      DDRA = 0xFF; PORTA = 0xFF;
28      DDRB = 0xFF; PORTB = 0xFF;
29      DDRC = 0xF0; PORTC = 0xFF;          //PC 端口低 4 位输入，高 4 位输出
30      DDRD = 0x00; PORTD = 0xFF;
31      MCUCR = 0x0A;                       //INT0、INT1 中断下降沿触发
32      GICR  = 0xC0;                       //INT0、INT1 中断许可
33      sei();                              //开总中断
34      while(1)
35      {
36         LED_BLINK();                     //主程序控制一只 LED 闪烁
37         _delay_ms(100);                  //延时
38      }
39  }
40
41  //-------------------------------------------------------------------
```

```
42  // INT0 中断服务程序(4 个按键中任何一个按下时都会触发 INT0 中断)
43  //-------------------------------------------------------------------
44  ISR (INT0_vect)
45  {
46      PORTC = PINC << 4 | 0x0F;          //"|0x0F"用于保持 PC 低 4 位内部上拉
47  }
48
49  //-------------------------------------------------------------------
50  // INT1 中断服务程序(当有按钮按下时,GS 为零,触发 INT1 中断)
51  //-------------------------------------------------------------------
52  ISR (INT1_vect)
53  {
54      INT8U bidx = (PIND >> 4) & 0x07;   //得到按键编号
55      PORTA = ~_BV(bidx);                //点亮对应的 LED
56  }
```

4.4 62256扩展内存实验

本例给出了ATMEGA8515单片机外部内存扩展电路。所使用的是62256SRAM存储器,该芯片共有地址线15根,可提供2^{15}=32K字节空间,提供地址锁存的是74LS373,它是常用的地址锁存器芯片,其实质是一个带三态缓冲输出的8D触发器。本例演示了内存扩展芯片62256的读/写实验,这种扩展对学习后续案例中有关接口扩展的案例也有很好的参考作用。案例电路如图4-4所示。



注:LED点亮时数据传输开始,闪烁时表示读写完成,
此时可暂停运行,打开DEBUG中的Memory Contents,
查看62256内存数据。

图4-4 62256扩展内存实验

1. 程序设计与调试

设计本例仿真电路时，要掌握三总线(CB、AB、DB)的连接方法，本例中控制总线涉及ALE、$\overline{RD}$、$\overline{WR}$；对于由PA、PC端口提供的16位地址总线A0～A15，本例使用了A0～A14，数据总线则复用了PA端口的D0～D7。在程序设计方面，应熟练掌握MCUCR中SRE位的设置及外部内存地址定义等：

① 仿真电路的74LS373、62256与AVR单片机的连接。其中，单片机ALE引脚(Address Latch Enable，地址锁存使能)与74LS373的LE(锁存使能，Latch Enable)引脚的连接，74LS373地址锁存由单片机ALE引脚控制。单片机读/写控制引脚$\overline{RD}$、$\overline{WR}$与62256的$\overline{OE}$(Output Enable，输出使能)、$\overline{WE}$(Write Enable，写使能)连接。这3条控制总线引脚负责地址锁存及读/写控制。

② 为了访问外部扩展内存，一定要在主程序内将ATMEGA8515单片机MCUCR寄存的最高位SRE(External SRAM/XMEM Enable)置位，这样才能访问外部内存(或访问外部扩展接口地址)。在ATMEGA系列中，8515/64/128等单片机提供了三总线以扩展外部内存或接口，但8/16/32等单片机则没有提供扩展总线。

③ 外部内存地址(或接口地址)访问定义。本例中定义为：

```
#define EXTMEM_ADDR (INT8U *)0x8000
```

62256地址线共有15根，所定义的0x8000超出内部SRAM地址空间，指向某个外部内存地址，0x8000即1000000000000000地址，后面共15个0，它们与62256的15条地址线对应，高位的1与62256无关，当从0x8000地址开始读/写时，实际上是从62256的0地址开始读/写。

④ 主程序中第32行和37行对外部内存进行读/写，语句如下：

```
*(EXTMEM_ADDR + i) = i + 1; //写操作
*(EXTMEM_ADDR + i + 0x0100) = *(EXTMEM_ADDR + 199 - i); //读/写操作
```

在搞清楚上述内容后，对程序所完成的其他任务就容易理解了。程序运行时首先向62256开始处写入1～200，接着读取这些数，并将其逆向写到62256内存中0x0100开始的位置。

前面已经提到了单片机的$\overline{WR}$与$\overline{RD}$引脚，单片机通过这两只引脚对读/写时序进行自动管理，删除$\overline{WR}$连线时会出现写入失败，删除$\overline{RD}$连线时会导致读取失败。

本例程序完成对外部SRAM的读/写操作后，LED开始闪烁。如果要观察62256芯片内的数据，可按下Pause按钮暂停程序，然后单击Debug菜单，打开Memory Contents即可观察到图4-5所示窗口中显示的内存数据。

2. 实训要求

① 重新编写程序，向62256写入1001～1200共200个整数。这些整数不能用200个单字节来保存，因为它们已经超过了INT8U类型的最大值255，这时所占空间应为400个字节。编程时注意定义指针类型。

② 在Proteus中搜索ROM存储芯片对单片机外部ROM进行扩展，将固定数据绑定到该芯片后，在程序中读取外部ROM中的数据并通过虚拟终端显示。

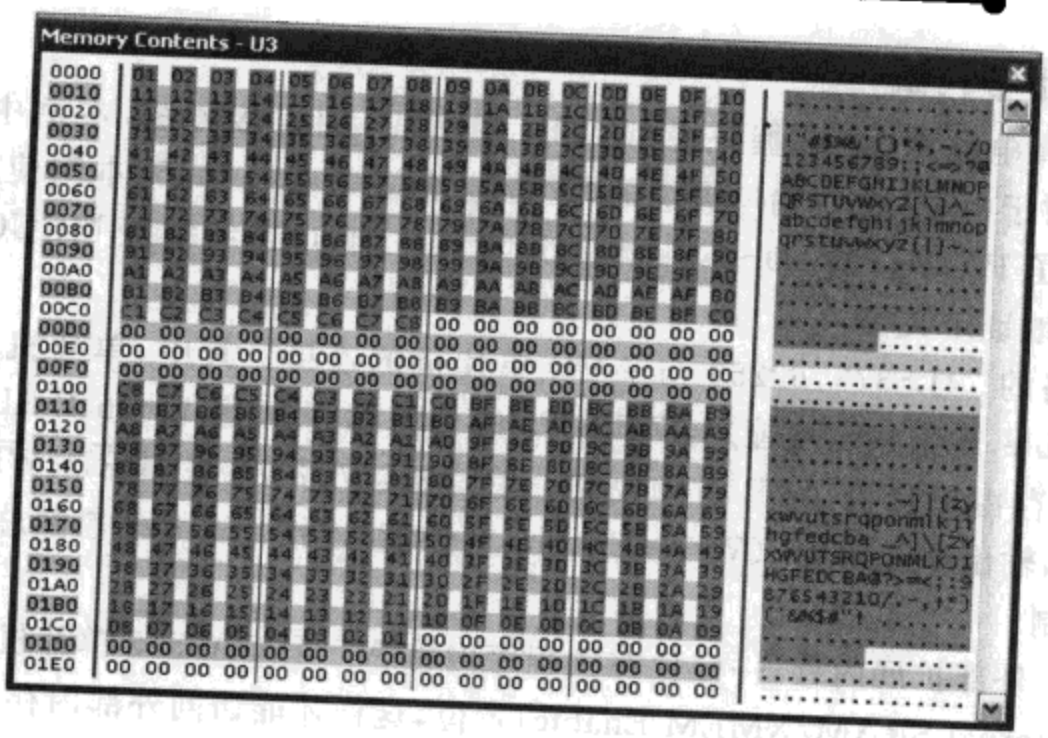


图4-5　62256内存内容

3. 源程序代码

```
01  //-------------------------------------------------------------------------
02  //  名称：用62256扩展内存(32 KB)
03  //-------------------------------------------------------------------------
04  //  说明：程序运行时首先向62256开始处写入1～200,然后读取这些数据,并将
05  //        其逆向写到62256内存中0x0100开始位置
06  //
07  //-------------------------------------------------------------------------
08  #define F_CPU 1000000UL
09  #include <avr/io.h>
10  #include <util/delay.h>
11  #define INT8U   unsigned char
12  #define INT16U  unsigned int
13
14  //外部内存地址定义
15  #define EXTMEM_ADDR   (INT8U*)0x8000
16  //LED控制
17  #define LED_ON()      (PORTB &= ～_BV(PB7))   //LED点亮
18  #define LED_BLINK()   (PORTB ^=  _BV(PB7))    //LED闪烁
19  //-------------------------------------------------------------------------
20  // 主程序
21  //-------------------------------------------------------------------------
22  int main()
23  {
24      INT8U i;
25      DDRB = 0xFF; PORTB = 0xFF;
```

```
26      LED_ON(); _delay_ms(1000);
27      //允许访问外部存储器
28      MCUCR |= 0x80;
29      //向 62256 的 0x0000 地址开始写入 1～200
30      for (i = 0; i<200; i ++ )
31      {
32          * (EXTMEM_ADDR + i) = i + 1;
33      }
34      //将 62256 中的 1～200 逆向拷贝到 0x0100 开始处
35      for (i = 0; i<200; i ++ )
36      {
37          * (EXTMEM_ADDR + i + 0x0100) = * (EXTMEM_ADDR + 199 - i);
38      }
39      //扩展内存数据读/写操作完成后 LED 闪烁
40      //这时可暂停 Proteus,打开菜单 Debug/Memory Contents 查看数据
41      while (1)
42      {
43          LED_BLINK();
44          _delay_ms(200);
45      }
46  }
```

4.5 用 8255 实现接口扩展

本例利用 ATMEGA8515 的三总线，通过 8255 接口扩展芯片控制 8 只集成式七段数码管显示，在 8255 的 PC 端口还添加有 3 个按键，用于调节所显示时间数据。仿真本例时要注意给 8255 单独添加 VDD 引脚。本例电路及部分运行效果如图 4-6 所示。

1. 程序设计与调试

本例的接口扩展电路与上一案例中的数据内存扩展电路非常相似，都使用了地址锁存芯片 74LS373。单片机的控制引脚 ALE、$\overline{RD}$、$\overline{WR}$连接方法也与上一案例类似。

正是因为本例的接口扩展电路与上一案例非常类似，上一案例中扩展内存的地址访问方法同样可以应用到本例中的扩展接口的地址访问上。

表 4-4 列出了 8255 的基本操作，通过仔细对比表格与本例电路即可得出 8255 的 3 个 I/O端口和 1 个命令端口的定义。由于 8255 的接口地址仅需要单片机地址端口的高 8 位控制，这 8 位地址中实际仅使用了低 3 位，它们分别对应 CS、A1、A0，其中 A1 与 A0 地址线可选择 8255 的 4 个端口地址之一。

以 PB 端口为例，由于 A1/A0 为 01，且 CS 为 0，则地址可定义为 11111111 00000001(定义中将未使用的高 8 位地址全部设为 1)，由此可得 8255PB 端口地址为 0xFF01。在向 8255PB 端口写入数据时，单片机会自动将$\overline{WR}$置为低电平，读 8255PB 端口数据时单片机则自动将$\overline{RD}$置为低电平。

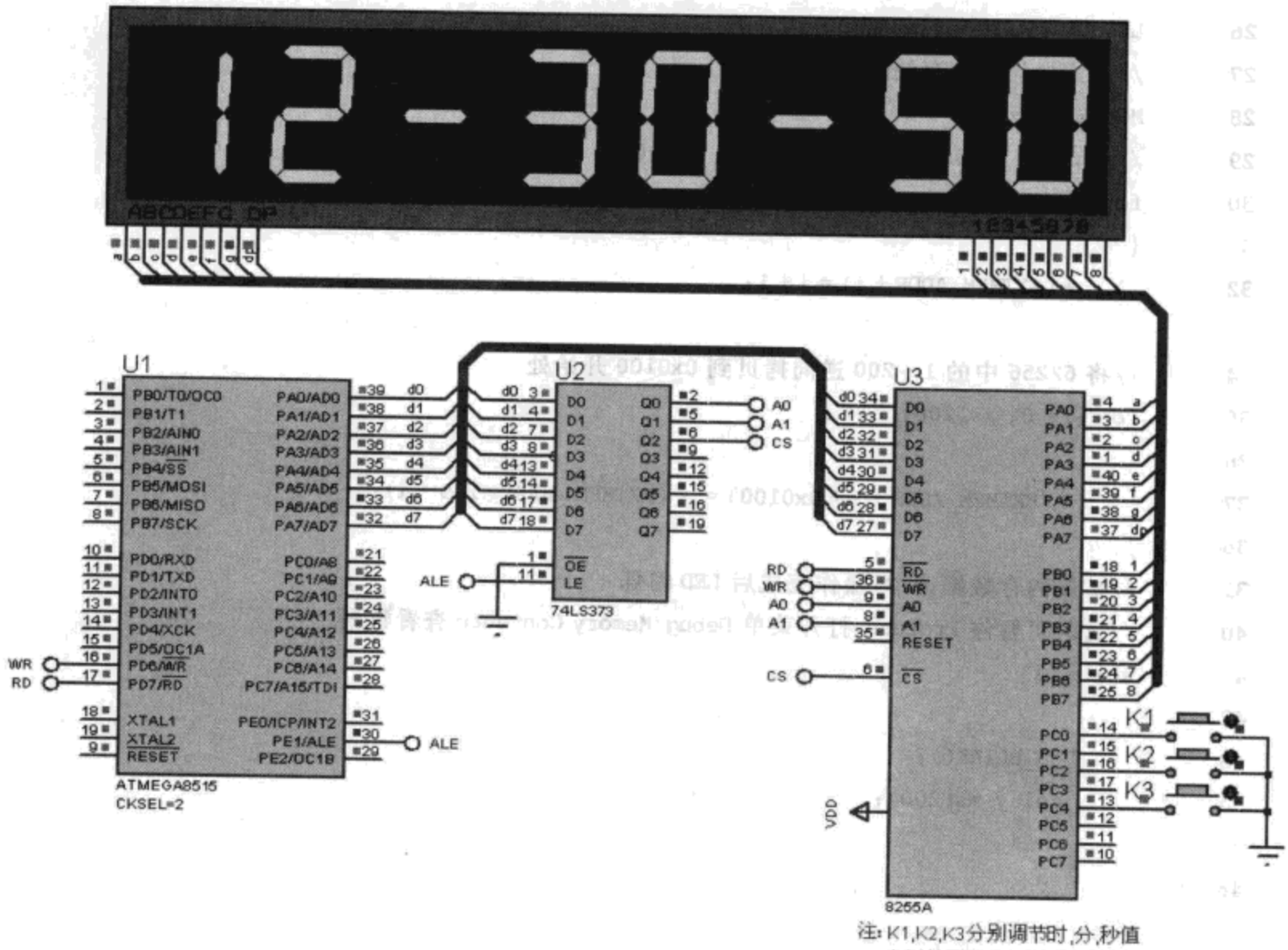


图4-6　用8255实现接口扩展

以下是8255的PA、PB、PC及命令端口的地址定义：

```
#define PA    (INT8U*)0xFF00
#define PB    (INT8U*)0xFF01
#define PC    (INT8U*)0xFF02
#define COM   (INT8U*)0xFF03
```

表4-4　8255的基本操作

操　作	A1	A0	$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	说　明
输入(读)	0	0	0	0	1	PA→数据总线
	0	1	0	0	1	PB→数据总线
	1	0	0	0	1	PC→数据总线
	1	1	0	0	1	控制字→数据总线
输出(写)	0	0	0	1	0	数据总线→PA
	0	1	0	1	0	数据总线→PB
	1	0	0	1	0	数据总线→PC
	1	1	0	1	0	数据总线→控制字

8255 命令口对工作方式的设置可参阅 8255 芯片的技术手册文件。图 4-7 给出了 8255 工作模式字节格式及本例所选择的设置。本例选择模式 0,8255 工作于基本 I/O 模式,使用 PA 和 PB 端口输出控制数码管显示,PC 端口则用于读取按键状态并进行相应处理,各位配置如图 4-7 右半部分所示。

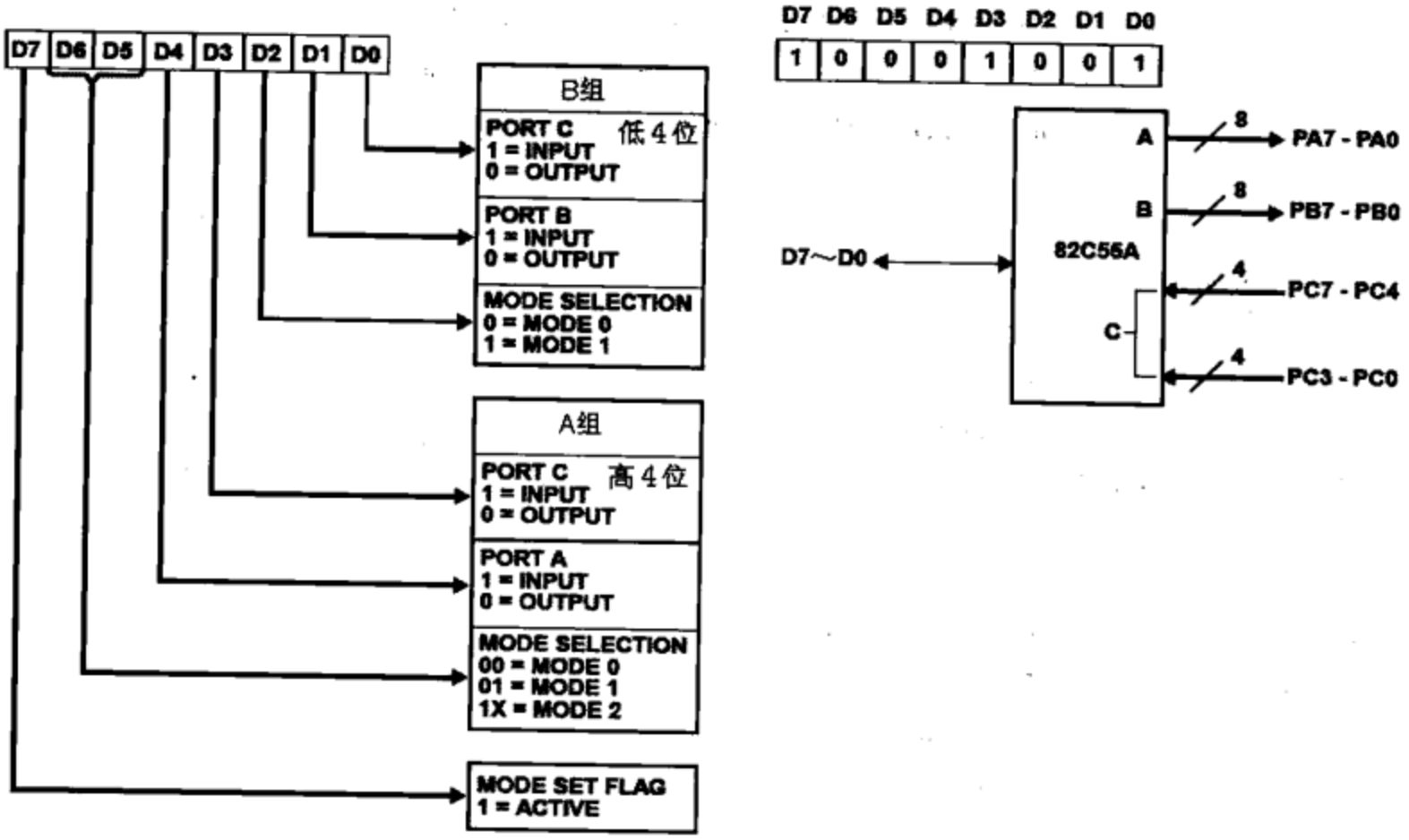


图 4-7 8255 工作模式字节格式(左)及本例设置(右)

在完成相关定义与配置后,其他操作与扩展内存的操作就很相似了:

① 源程序第 67 行向命令口 COM 写控制字节,实现对 8255 工作方式的配置:

```
* COM = 0B10001001;
```

② 第 73、74 行通过 PB、PA 端口输出位码与段码,控制数码管扫描显示;

```
* PB = _BV(i);
* PA = (INT8U)SEG_CODE[ Disp_Buffer[i] ];
```

③ 第 34 行读入 8255PC 端口的按键状态,以便分别进行时分秒的调节:

```
Key_State = * PC;
```

2. 实训要求

① 重新设计本例电路,再加一组相同的 8 位数码管,用 PA 控制两组数码管段码,PB 与 PC 用于控制 16 位的扫描码。在两组数码管上同时显示出年月日和时分秒信息。

② 保持本例的电路设计,仅将 PC 端口按键改成 4×4 键盘矩阵,利用键盘矩阵控制数码管显示、关闭及时分秒调节等自定义功能。

3. 源程序代码

```
01  //-------------------------------------------------------------------
02  //  名称：用8255实现接口扩展
03  //-------------------------------------------------------------------
04  //  说明：8255的PA、PB端口分别连接8位数码管的段码和位码
05  //        PC端口连接3只按键，正常运行时数码管显示一组时间值
06  //        PC端口的3只按键可对时间值的各部分分别进行调整
07  //
08  //-------------------------------------------------------------------
09  #define F_CPU 2000000UL
10  #include <avr/io.h>
11  #include <util/delay.h>
12  #define INT8U   unsigned char
13  #define INT16U  unsigned int
14
15  //PA,PB,PC端口及命令端口地址定义
16  #define PA   (INT8U*)0xFF00
17  #define PB   (INT8U*)0xFF01
18  #define PC   (INT8U*)0xFF02
19  #define COM  (INT8U*)0xFF03
20
21  //0～9的共阳数码管段码表，最后的0xBF表示"-"
22  const INT8U SEG_CODE[] =
23  { 0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xBF };
24  //待显示信息缓冲 12-30-50
25  INT8U Disp_Buffer[] = {1,2,10,3,0,10,5,0};
26  //上次按键状态
27  INT8U Pre_Key_State = 0x00;
28  //-------------------------------------------------------------------
29  // 8255PC端口按键处理
30  //-------------------------------------------------------------------
31  void Key_Process()
32  {
33      INT8U Key_State, t;
34      Key_State = *PC;                                  //读8255PC端口按键状态
35      if (Key_State == Pre_Key_State) return;
36      Pre_Key_State = Key_State;
37      switch (Key_State)
38      {
39          case (INT8U)~_BV(0):                          //K1:小时递增
```

```
40              t = Disp_Buffer[0] * 10 + Disp_Buffer[1];
41              if ( ++t == 24) t = 0;
42              Disp_Buffer[0] = t / 10;
43              Disp_Buffer[1] = t % 10;
44               break;
45          case (INT8U)~_BV(2):                          //K2:分钟递增
46              t = Disp_Buffer[3] * 10 + Disp_Buffer[4];
47              if ( ++t == 60) t = 1;
48              Disp_Buffer[3] = t / 10;
49              Disp_Buffer[4] = t % 10;
50               break;
51          case (INT8U)~_BV(4):                          //K3:秒数递增
52              t = Disp_Buffer[6] * 10 + Disp_Buffer[7];
53              if ( ++t == 60) t = 1;
54              Disp_Buffer[6] = t / 10;
55              Disp_Buffer[7] = t % 10;
56               break;
57          default: break;
58      }
59  }
60
61  //-------------------------------------------------------------------
62  // 主程序
63  //-------------------------------------------------------------------
64  int main()
65  {
66      INT8U i;
67      MCUCR |= 0x80;              //允许访问外部存储器/接口等
68       * COM = 0B10001001;       //8255 工作方式选择:工作于方式 0,PA、PB 输出,PC 输入
69      while(1)
70      {
71          for(i = 0; i<8; i ++ )                        //数码管显示
72          {
73              * PB = _BV(i);                            //向 PB 端口发送位码
74              * PA = (INT8U)SEG_CODE[ Disp_Buffer[i] ]; //向 PA 端口发送段码
75              _delay_ms(2);
76              Key_Process();                            //PC 端口按键处理
77          }
78      }
79  }
```

4.6 可编程接口芯片8155应用

可编程接口芯片8155内含256字节RAM存储器、2个可编程的8位并行端口、1个6位并行端口及1个14位的定时/计数器。本例用8155的PA与PB端口控制数码管显示，PC端口连接按键，案例演示了8155控制数码管显示，通过按键调整定时初值、启/停8155定时器，用定时器中断触发蜂鸣器，以及写8155内存等。本例电路及部分运行效果如图4-8所示。

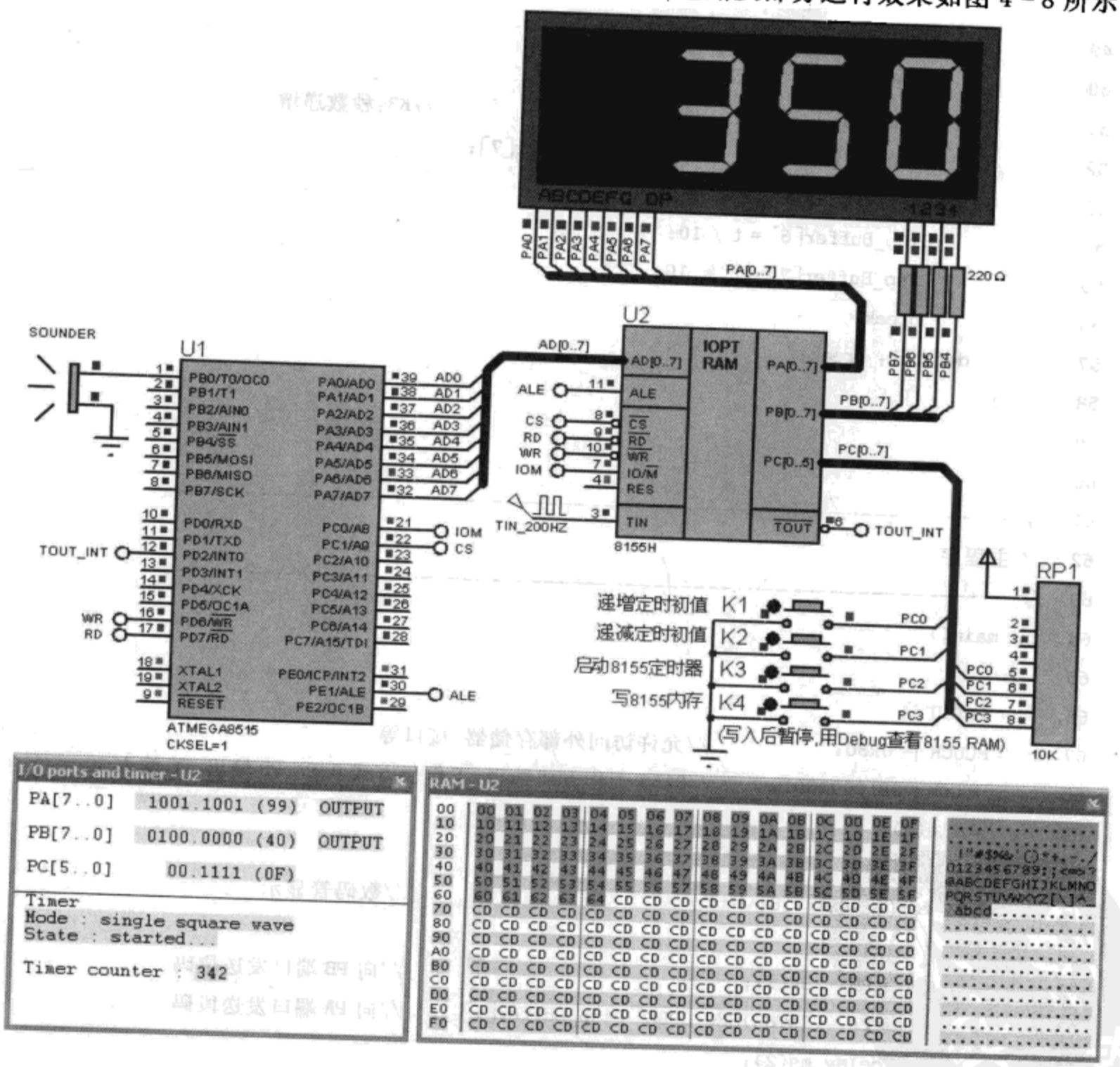


图4-8 可编程接口芯片8155应用

1. 程序设计与调试

图4-8所示电路中，8155的AD[0：7]为三态数据/地址线，TIN是定时/计数器输入引脚，$\overline{TOUT}$是定时器输出引脚，可以是方波或脉冲波形。IO/$\overline{M}$是I/O与RAM选择线，设为1

时选择 I/O，设为 0 时选择 RAM。其他引脚与 8255 类似。

本例程序重点在于以下地址定义：

```
#define COMM_8155        (INT8U*)0xFD00        //命令字端口
#define PA_8155          (INT8U*)0xFD01        //PA 端口地址
#define PB_8155          (INT8U*)0xFD02        //PB 端口地址
#define PC_8155          (INT8U*)0xFD03        //PC 端口地址
#define CONT_8155_L8     (INT8U*)0xFD04        //计数器低 8 位地址
#define CONT_8155_H8     (INT8U*)0xFD05        //计数器高 6 位 + 2 位方式地址
#define PMEM_8155        (INT8U*)0xFC00        //8155RAM 地址
```

单片机 PC 端口提供地址的高 8 位，其中 PC7～PC2 未用，定义中将它们全部设为 1，PC1 连接的 CS 位设为 0，PC0 对应的 IO/$\overline{M}$分别取 0/1，因此上述地址高 4 位定义中，除最后的 PMEM 定义为 0xFC 以外，其他全部为 0xFD。其他地址定义可根据表 4－5 所列的 8155 内部 I/O 地址表得到。

表 4－5　8155 内部 I/O 地址表

A2	A1	A0	I/O 端口
0	0	0	命令端口
0	0	1	PA 端口
0	1	0	PB 端口
0	1	1	PC 端口
1	0	0	定时器低 8 位
1	0	1	定时器高 6 位及方式

完成地址定义后，还需要根据 8155 命令字对端口及定时器进行配置管理，程序中第 56 行与 123 行对定时/计数器进行设置，并对端口进行管理。8155 命令字格式如表 4－6 所列。

表 4－6　8155 命令字格式

TM2	TM1	IEB	IEA	PC2	PC1	PB	PA

完成地址定义后，还需要根据 8155 命令字对端口及定时器进行配置管理，程序中第 56 行与 123 行的设置都向 8155 写入了命令字，其中：

第 123 行设 * COMM_8155＝0B00000011，其中 TM2/TM1 为 00，定时器空操作。同时低 4 位 0011 设置 PA 与 PB 端口为输出，PC 端口为输入。

第 56 行设 * COMM_8155＝0B11000011，它将 TM2/TM1 设为 11，在装入定时器方式和初值后立即启动计数。PA、PB、PC 端口配置不变。

本例运行时：

按下 K4 可向 8155RAM 中写入 0～100(0x00～0x64)，在暂停程序后可通过 Proteus 的 Debug 菜单下的 RAM 菜单查看 8155RAM 数据，如图 4－8 右下角所示。

按下 K3 时可启动定时器，程序已经给 14 位的定时器设置了固定初值，定时溢出时，8155 的$\overline{TOUT}$引脚触发单片机 INT0 中断，输出报警声音，同时还原定时初值，使中断能在同样时间后继续触发。

K1 与 K2 按键则用于改变 8155 定时器初值，在不同定时初值定义下，中断的触发间隔不同，这通过报警声音输出的间隔就可以分辨出来。

在暂停程序运行时，按下 Debug 菜单下的 I/O ports and timer 菜单可查看 I/O 端口与定时器配置及工作状态，如图 4－8 左下部分窗口所示。

2. 实训要求

① 修改本例程序，对TIN引脚输入的脉冲进行计数(改用按键或低频率脉冲)，并将计数值显示在4位数码管上。

② 在实现计数的基础上进一步修改程序，当计数值每次累加达到5000时，将当前计数值累加到8155RAM中的指定地址，然后再从0开始累加计数。

3. 源程序代码

```
001  //-------------------------------------------------------------------
002  //  名称：可编程序接口芯片8155应用
003  //-------------------------------------------------------------------
004  //  说明：本例利用8155的PA、PB连接数码管，显示8155当前定时初值
005  //        PC端口连接按键，分别用于调整定时初值，启动定时器，写8155RAM等
006  //        启动定时器后，在定时溢出时8155 TOUT将触发INT0中断，输出提示音
007  //        在调节的定时初值不同时，声音输出的间隔也不同
008  //
009  //-------------------------------------------------------------------
010  #define F_CPU 2000000UL
011  #include <avr/io.h>
012  #include <avr/interrupt.h>
013  #include <util/delay.h>
014  #define INT8U   unsigned char
015  #define INT16U  unsigned int
016
017  //8155地址定义
018  #define COMM_8155     (INT8U*)0xFD00   //命令字端口
019  #define PA_8155       (INT8U*)0xFD01   //PA端口地址
020  #define PB_8155       (INT8U*)0xFD02   //PB端口地址
021  #define PC_8155       (INT8U*)0xFD03   //PC端口地址
022  #define CONT_8155_L8  (INT8U*)0xFD04   //计数器低8位地址
023  #define CONT_8155_H8  (INT8U*)0xFD05   //计数器高6位+2位方式地址
024  #define PMEM_8155     (INT8U*)0xFC00   //8155RAM地址
025
026  //蜂鸣器定义
027  #define BEEP() PORTB^= _BV(PB0)
028  //0～9的共阳数码管段码表，最后一位为黑屏幕
029  const INT8U SEG_CODE[] =
030  { 0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xFF };
031  //待显示信息缓冲
032  INT8U Disp_Buffer[4] = {10,3,5,0};
033  //8155定时计数初值
```

```
034  volatile INT16U cnt_8155 = 350;
035  //定时初值递增或递减
036  enum OP_Type {ADD,SUB};
037  //-------------------------------------------------------------------
038  // 输出提示音
039  //-------------------------------------------------------------------
040  void Sounder()
041  {
042      INT8U i;
043      for (i = 0; i<50; i ++ )
044      {
045          BEEP(); _delay_us(160);
046      }
047  }
048
049  //-------------------------------------------------------------------
050  // 设置 8155 定时初值
051  //-------------------------------------------------------------------
052  void Set_8155_TC()
053  {
054      * CONT_8155_L8 = (INT8U)cnt_8155;          //装入定时初值低字节
055      * CONT_8155_H8 = (INT8U)(cnt_8155>>8);     //装入定时初值高字节
056      * COMM_8155 = 0B11000011;                  //设置 PA、PB、PC 端口方式及定时器命令
057  }
058
059  //-------------------------------------------------------------------
060  // 8155 定时初值调整
061  //-------------------------------------------------------------------
062  void adjust_tCount(enum OP_Type op)
063  {
064      INT8U i;
065      INT16U cnt;
066      cnt_8155 = (op == ADD) ? cnt_8155 + 50: cnt_8155 - 50;
067      if      (cnt_8155 > 500) cnt_8155 = 500;
068      else if (cnt_8155<100) cnt_8155 = 100;
069      cnt = cnt_8155;
070      for (i = 3; i >= 1; i -- )
071      {
072          Disp_Buffer[i] = cnt % 10;  //从低位开始逐位分解
073          cnt / = 10;
074      }
```

```
075 }
076
077 //-------------------------------------------------------------------
078 // 8155PC端口按键处理
079 //-------------------------------------------------------------------
080 void Key_Process()
081 {
082     INT8U i;
083     //上次按键状态
084     static INT8U Pre_Key_State = 0xFF;
085     //读8255PC端口按键状态
086     INT8U curr_Key_State = *PC_8155 | 0xF0;
087     //按键状态未改变则返回
088     if (Pre_Key_State == curr_Key_State) return;
089     //保存当前按键状态(用于下一次判断状态是否改变)
090     Pre_Key_State = curr_Key_State;
091     //处理按键操作
092     switch (curr_Key_State)
093     {
094       case (INT8U)~_BV(0): //K1:递增8155定时初值,每次递增50
095                            adjust_tCount(ADD);
096                            break;
097       case (INT8U)~_BV(1): //K2:递减8155定时初值,每次递减50
098                            adjust_tCount(SUB);
099                            break;
100       case (INT8U)~_BV(2): //K3:设置并启动8155定时器
101                            Set_8155_TC();
102                            break;
103       case (INT8U)~_BV(3): //K4:写8155RAM: 0~100
104                            for (i=0 ; i<=100; i++)
105                            {
106                              *(PMEM_8155+i) = i;
107                            }
108                            break;
109     }
110 }
111
112 //-------------------------------------------------------------------
113 // 主程序
114 //-------------------------------------------------------------------
115 int main()
```

```
{
    INT8U  i;
    DDRA = 0xFF;                    //配置端口
    DDRB = 0xFF;
    DDRC = 0xFF;
    DDRD = 0x00; PORTD = 0xFF;
    MCUCR = 0x82;                   //允许访问外部存储器/接口等,INT0 中断下降沿触发
    * COMM_8155 = 0B00000011;       //设置 8155 命令字:PA、PB 输出,PC 输入,不影响计数器工作
    GICR  = _BV(INT0);              //INT0 中断使能
    sei();                          //开中断
    while(1)
    {
        for(i = 0; i<4; i ++ )                          //4 位数码管显示
        {
            * PB_8155 = 0x00;                           //暂时关闭
            * PA_8155 = SEG_CODE[Disp_Buffer[i]];       //向 8155 PA 端口发送段码
            * PB_8155 = _BV(7 - i);                     //向 8155 PB 端口发送位码
            _delay_ms(4);
            Key_Process();                              //8155 PC 端口按键处理
        }
    }
}

//-------------------------------------------------------------
// INT0 中断子程序
//-------------------------------------------------------------
ISR (INT0_vect)
{
    Sounder();                  //蜂鸣器输出
    Set_8155_TC();              //重置 8155 TC 初值并启动
}
```

4.7 可编程外围定时/计数器 8253 应用

8253 可编程定时/计数器片内含 3 个独立的 16 位计数器,计数器均为递减计数,各计数器的工作方式与初值由软件设置。本例运行时,如果按下 8255 上的“启动 8253TC0”按键,单片机将向 8253 写入随机定时初值,由于定时初值不同,定时溢出中断将以不同时间间隔触发,每次触发时单片机输出报警声音。单片机随机写入 8253 的定时初值由 8255 控制数码管显示。本例电路及部分运行效果如图 4-9 所示。

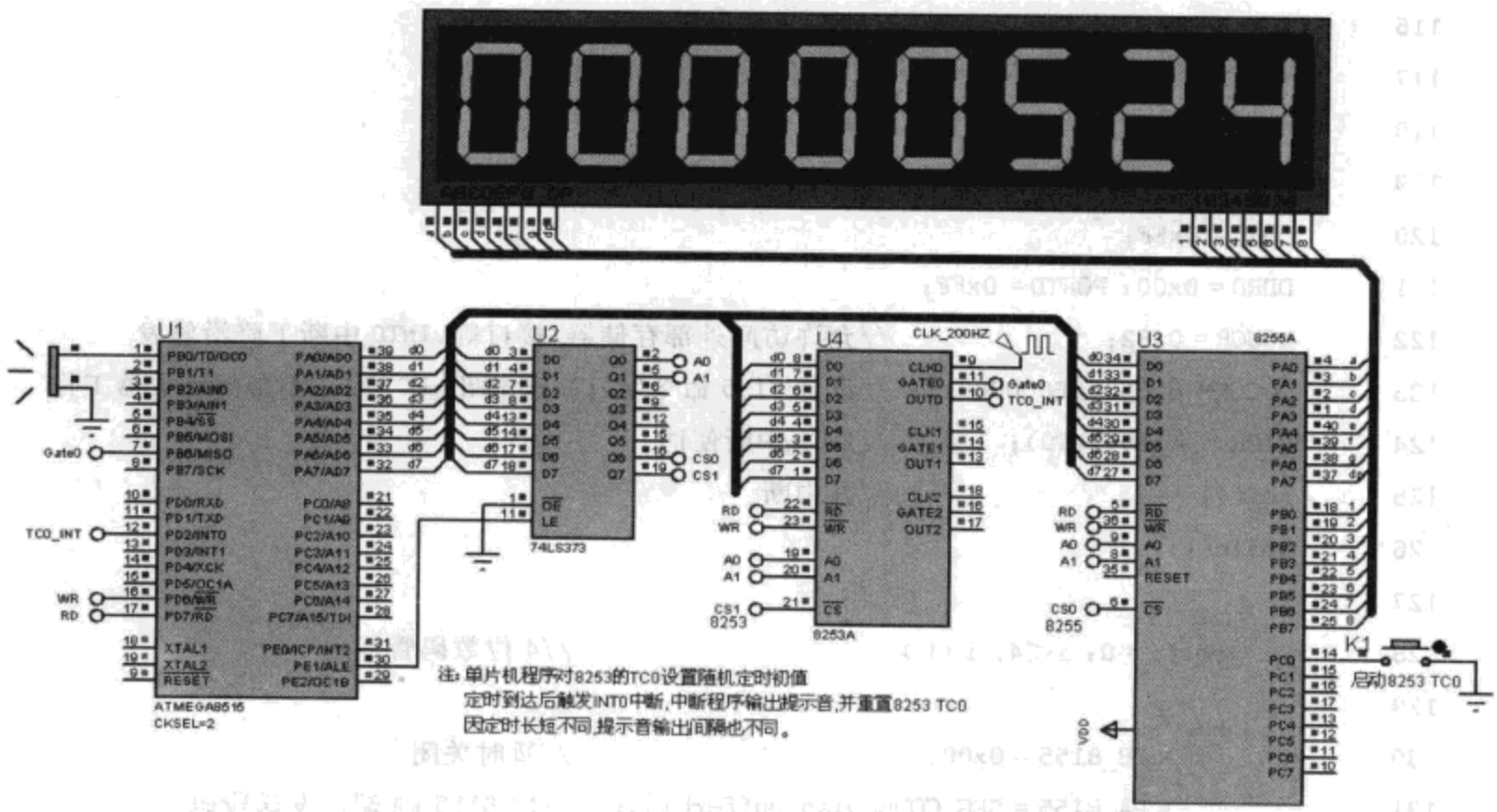


图 4-9　可编程外围定时/计数器 8253 应用

1. 程序设计与调试

本例要点之一在于 8255 与 8253 的接口扩展地址定义：

不同于上一案例中 8255 地址定义的是，本例 8255 地址定义的最前面添加有“*”号，这样定义后，再读/写所定义的地址空间时就不需要在 PA_8255 等符号前面添加“*”。对于 8253 的地址定义则未添加“*”，其用法与上一案例中的 8255 端口操作类似。

由于 8255 与 8253 都没有占用 16 位接口扩展地址的高 8 位(即 PC 端口 A8～A15 不连接 8255 与 8253)，因此地址定义中将它们全部设为全 1(FF)。地址低 8 位中的 PA7 与 PA6 分别通过地址锁存器 74LS373 连接 8253 的 CS1 和 8255 的 CS0 引脚，在片选 8253 与 8255 时，它们分别互斥为 0，地址低 8 位中的高 4 位中后 2 位未用，因此 8253 与 8255 低 8 位地址中高 4 位分别为 B(1011)和 7(0111)，8255 地址的最低 4 位定义可参考前面的 8255 案例。

```
#define PA_8255      *(INT8U*)0xFFB0
#define PB_8255      *(INT8U*)0xFFB1
#define PC_8255      *(INT8U*)0xFFB2
#define COM_8255     *(INT8U*)0xFFB3
```

在 16 位的扩展接口地址中，8253 和 8255 一样都有 A1 与 A0 引脚，其定义也很相似。查阅 8253 的技术手册可知，A1 与 A0 组合为 00、01、10、11 时，分别选择计数器 0、1、2、及控制寄存器。由于本例仅使用了 8253 的 TC0 并需要对其进行命令控制，于是有如下地址定义：

```
#define TC0_8253    (INT8U*)0xFF70
#define COM_8253    (INT8U*)0xFF73
```

下面再来看一下函数 Set_8253_TC0 中的代码：

```
*COM_8253 = 0x3A;                          //工作方式为5,先读/写低字节,后读/写高字节
TC0_Count = rand() % 600;                  //初值限制于600以内
*(INT16U*)TC0_8253 = (INT8U)TC0_Count;     //送低字节
*TC0_8253 = (INT8U)(TC0_Count>>8);         //送高字节
```

在阅读这些代码之前要参阅8253的控制字格式,表4-7给出了8253的命令字格式。

表4-7 8253命令字格式

SC1	SC0	RL1	RL0	M2	M1	M0	BCD

其中高2位SC1/0用于选择计数器,取值00、01、10、11分别对应计数器0、1、2、非法。

RL1/0用于设定对计数器的读/写顺序,00、01、10、11分别表示闩锁、只读/写高字节、只读/写低字节、先读/写低字节后读/写高字节。

M2/1/0取值000～101,分别用于选择计数器的工作方式0～5,本例选择的工作方式为5,即硬件触发选通方式。写入方式控制字和计数初值后,输出保持高电平,只有在门控信号GATE的上升沿之后才开始计数,完成最后一个计数后输出一个时钟周期的负脉冲。

最后一位BCD取0～1表示按二进制计数或按BCD码计数。

上述代码中*COM_8253取值0x3A(00111010),设定计数器0工作方式为5,先读/写低字节后读/写高字节。

2. 实训要求

① 修改本例程序,对CLK0与CLK1两路输入脉冲进行计数,计数值分成两组显示在数码管上。

② 修改本例电路,利用8255控制条形LED,利用8253控制扬声器,实现自定义音乐片段输出,根据不同的输出频率,8255控制的条形LED可实现同步闪烁。

3. 源程序代码

```
001  //-------------------------------------------------------------------
002  //  名称:可编程外围定时计数器8253应用
003  //-------------------------------------------------------------------
004  //  说明:本例运行时,按下8255 PC端口按键K1可启动8253定时器0,定时器0
005  //        工作于方式5,程序给8253提供随机的定时初值,定时到达后GATE0
006  //        触发INT0中断,中断程序输出提示音,并重置定时器
007  //
008  //-------------------------------------------------------------------
009  #define F_CPU 2000000UL
010  #include <avr/io.h>
011  #include <avr/interrupt.h>
012  #include <util/delay.h>
013  #include <stdlib.h>
014  #define INT8U   unsigned char
015  #define INT16U  unsigned int
```

```
016
017  //8255 PA、PB、PC 端口及命令端口地址定义
018  //(定义前面加 * 可使得后面使用时不用再加 * )
019  #define PA_8255    *(INT8U*)0xFFB0
020  #define PB_8255    *(INT8U*)0xFFB1
021  #define PC_8255    *(INT8U*)0xFFB2
022  #define COM_8255   *(INT8U*)0xFFB3
023
024  //8253 定时/计数器 0 及命令端口地址定义
025  #define TC0_8253   (INT8U*)0xFF70
026  #define COM_8253   (INT8U*)0xFF73
027
028  //8253 定时/计数器 0/1 的门控制位操作定义
029  #define TC0_G1()   PORTB |=  _BV(PB6)
030  #define TC0_G0()   PORTB &= ~_BV(PB6)
031
032  //蜂鸣器定义
033  #define BEEP() PORTB ^= _BV(PB0)
034  //0~9 的共阳数码管段码表,最后一位为黑屏
035  const INT8U SEG_CODE[] =
036  { 0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0xFF };
037
038  //待显示信息缓冲
039  INT8U Disp_Buffer[8] = {0,0,0,0,0,0,0,0};
040  //上次按键状态
041  INT8U Pre_Key_State = 0x00;
042  //对 TC0 设置的定时/计数初值
043  volatile INT16U TC0_Count = 510;
044  //-------------------------------------------------------------
045  // 输出提示音
046  //-------------------------------------------------------------
047  void Sounder()
048  {
049      INT8U i;
050      for (i = 0; i<100; i ++ )
051      {
052          BEEP(); _delay_us(180);
053      }
054  }
055
056  //-------------------------------------------------------------
```

```
057  // 设置 8253 TC0 定时初值
058  //-------------------------------------------------------------------
059  void Set_8253_TC0()
060  {
061      TC0_G0();              //先关闭 TC0 门控制位
062      * COM_8253 = 0x3A;     //工作方式为 5,先读/写低字节,后读/写高字节
063      TC0_Count = rand() % 600;                    //初值限制于 600 以内
064      * (INT16U * )TC0_8253 = (INT8U)TC0_Count;   //送低字节
065      * TC0_8253 = (INT8U)(TC0_Count>>8);         //送高字节
066      TC0_G1();                                    //开 TC0 门控制位
067  }
068
069  //-------------------------------------------------------------------
070  // 8255 PC 端口按键处理
071  //-------------------------------------------------------------------
072  void Key_Process()
073  {
074      INT8U Key_State;
075      Key_State = PC_8255;                         //读 8255 PC 端口按键状态
076      if (Key_State == Pre_Key_State) return;
077      Pre_Key_State = Key_State;
078      switch (Key_State)
079      {
080          case (INT8U)~_BV(0): Set_8253_TC0();  //K1:重置 8253 TC0
081                               sei();           //使能总中断
082                               break;
083          //这里可添加 case,用于 8255 PC 端口的其他按键处理
084          default: break;
085      }
086  }
087
088  //-------------------------------------------------------------------
089  // 主程序
090  //-------------------------------------------------------------------
091  int main()
092  {
093      INT8U i; INT16U cnt;
094      DDRA = 0xFF; DDRB = 0xFF;
095      DDRD = ~(_BV(PD2) | _BV(PD3));
096      srand(87);                                   //设置随机种子
097
```

```
098     MCUCR |= 0x82;              //允许访问外部存储器/接口等,INT0 中断下降沿触发
099     COM_8255 = 0B10001001;      //8255 工作方式选择:工作于方式 0,PA、PB 输出,PC 输入
100     GICR   = 0x40;              //INT0 中断使能
101     while(1)
102     {
103         cnt = TC0_Count;
104         for(i = 0; i<8; i ++ )                                  //数码管显示
105         {
106            Disp_Buffer[i] = cnt % 10;
107            cnt / = 10;
108            PB_8255 = _BV(7 - i);                                //向 8255PB 端口发送位码
109            PA_8255 = (INT8U)SEG_CODE[ Disp_Buffer[i] ];         //向 8255PA 端口发送段码
110            _delay_ms(2);
111            Key_Process();                                       //8255PC 端口按键处理
112         }
113     }
114 }
115
116 //-------------------------------------------------------------
117 // INT0 中断子程序
118 //-------------------------------------------------------------
119 ISR (INT0_vect)
120 {
121     Sounder();                                                  //蜂鸣器输出
122     Set_8253_TC0();                                             //重置 8253 TC0
123 }
```

4.8 数码管BCD解码驱动器7447与4511应用

此前有关数码管显示的案例中,单片机必须向数码管发送段码。本例使用的七段数码管显示译码器7447与4511各自仅占用PB端口高/低各4位引脚,单片机向7447与4511分别写入4位8421BCD码,经2块芯片译码后再向数码管输出数字段码,实现数码管显示。本例电路及部分运行效果如图4-10所示。

1. 程序设计与调试

本例使用了七段数码管显示驱动器7447与4511,它们接收数字0～9的4位BCD编码,译码后输出0～9的段码,因此本例代码中没有出现数码管段码表,待显示的数字可以直接输出。

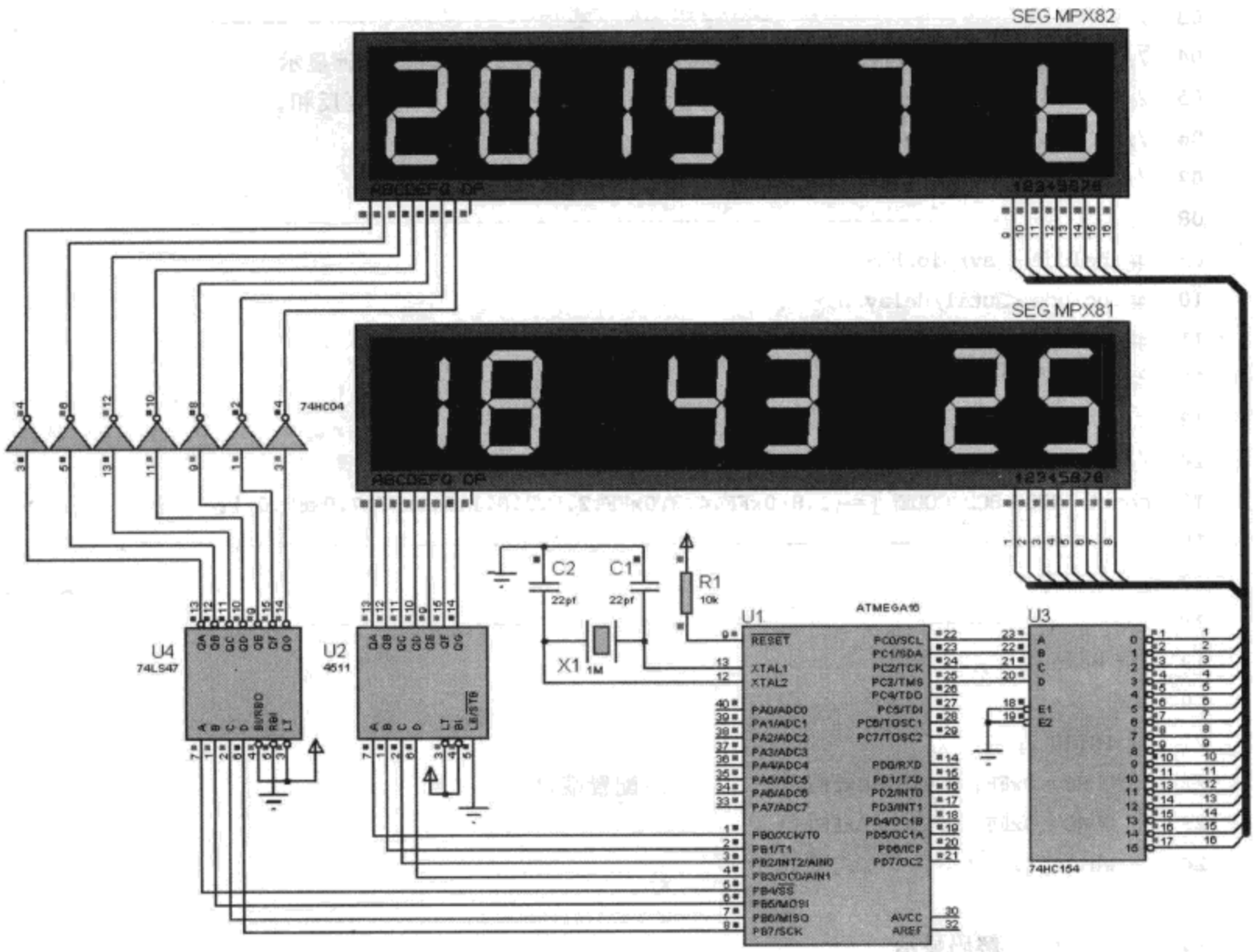


图 4-10　数码管 BCD 解码驱动器 7447 与 4511 应用

传输给 4511 的 4 位 BCD 码只能是 0000～1001，即 0～9 的 BCD 码；超过 1001 的编码会使输出为 00000000，共阴数码管各段均不显示，数码管黑屏。本例中发送给 4511 的 BCD 码为 1、8、0xFF、4、3、0xFF、2、5，数码管显示是 18 43 25。

传输给 7447 的 4 位 BCD 码与 4511 类似，由于 7447 是驱动共阳数码管的，同样的 BCD 码输入后，它的段码输出与 4511 完全相反。

本例两组数码管全部是共阴的，4511 可以直接驱动，但对 7447 则需要在输出端添加非门，如果改用共阳数码管，则输出端不需要添加非门，但本例中由 4-16 译码器控制的扫描码输出端(标号为 9～16)就需要添加非门了。

2. 实训要求

① 将两组数码管全部改用 4511 或 7447 驱动显示。

② 在 Proteus 中输入“bcd to 7 - segment”可以找到多种其他七段码数码管译码/驱动器，尝试改用搜索到的其他译码/驱动芯片控制数码管显示。

3. 源程序代码

```
01  //-------------------------------------------------------------------
02  //  名称：数码管 BCD 解码驱动器 7447 与 4511 应用
```

```
//-------------------------------------------------------------------
//  说明：BCD码经7447或4511译码后输出数码管段码，实现数码管显示
//        (7447驱动共阳数码管，本例用的是共阴数码管，因此需要反相，
//        4511驱动共阴极数码管)
//
//-------------------------------------------------------------------
#include <avr/io.h>
#include <util/delay.h>
#define INT8U   unsigned char
#define INT16U  unsigned int

//待显示的数字串"18 43 25"和"2015 7 6"，其中0xFF是不显示的
const INT8U BCD_CODE[] = {1,8,0xFF,4,3,0xFF,2,5,2,0,1,5,0xFF,7,0xFF,6,};
//-------------------------------------------------------------------
// 主程序
//-------------------------------------------------------------------
int main()
{
    INT8U i;
    DDRB = 0xFF; PORTB = 0xFF;              //配置端口
    DDRC = 0xFF; PORTC = 0xFF;
    while(1)
    {
      //4511解码显示
      for(i = 0;i<8;i ++ )
      {
          //译码器输出0～7对应的扫描码，控制数码管7SEG_MPX81
          PORTC = i;
          //向4511输出待显示数字的BCD码(非段码)
          PORTB = BCD_CODE[i];
          _delay_us(500);
      }
      //7447解码显示
      for(i = 8;i<16;i ++ ) //或改成for (; i<16; i ++ )
      {
          //译码器输出8～15对应的扫描码，控制数码管7SEG_MPX82
          PORTC = i;
          //向7447输出待显示数字的BCD码(非段码)
          //因为7447的输入端DCBA连接PC端口高4位，故这里需要左移
          PORTB = BCD_CODE[i] << 4;
          _delay_us(500);
      }
    }
}
```

4.9 8×8 LED 点阵屏显示数字

本例 8×8 LED 点阵屏的行驱动由 PC 端口控制，列选通由 PD 端口控制，程序运行时，8×8 LED 点阵屏依次循环显示数字 0～9，刷新过程由 T/C0 定时器溢出中断程序控制完成。本例电路及部分运行效果如图 4－11 所示。

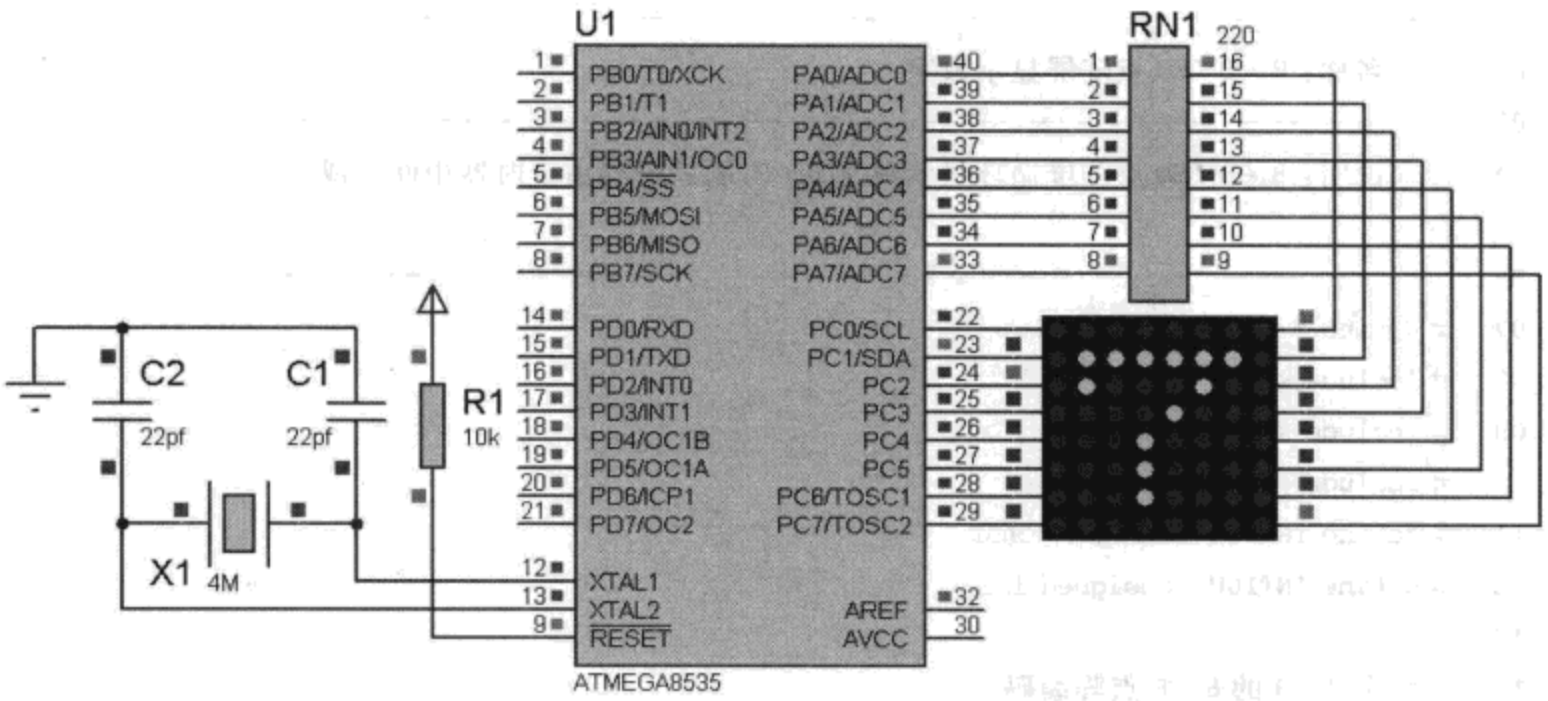


图 4－11 8×8 LED 点阵屏显示数字

1. 程序设计调试

点阵显示屏的动态刷新显示与集成式 8 位数码管的动态刷新显示非常相似，8 位数码管中的一只相当于点阵屏中的一列，数码管的段码相当于点阵屏的行码，位码则相当于点阵屏的列码，两者逻辑结构完全相同，只是外观不一样。由于逻辑结构相似，因此本例点阵屏的中断刷新显示代码与上一案例中的代码也非常相似。

如果点阵屏中每一行 LED 是共阳连接，那么每一列必定是共阴连接；如果将其旋转 90°，则行是共阴连接，列是共阳连接；如果将其旋转 180°旋转，则逻辑结构没有改变，但在编程控制显示时，点阵的取法或行码与列码字节的发送顺序需要作相应调整。

程序中数组 Table_OF_Digits 共有 64 个字节，每 8 个字节为一个数字的点阵代码，其中每个字节的 8 位对应于一列中的 8 个点，例如数组中第 0 行的 8 个字节 0x00、0x3C、0x66、0x42、0x42、0x66、0x3C、0x00 就是数字 0 第 0～7 列的点阵编码，这类似于数码管一个数字的段码，在点阵屏中就是行码，它们将被分别发送到显示屏的第 0～7 列。各字节的高位对应于列中上面的点还是下面的点，这由 PA 端口与显示屏 8 只行引脚的连接顺序决定。

变量 Num_Index 标明了将要显示的数字，取值范围为 0～9，变量 i 的取值范围为 0～7，表达式 Table_OF_Digits[Num_Index * 8＋i]使程序取得第 Num_Index 个数字的第 i 个字节，因为每个数字的点阵编码由 8 个字节构成，每次取得 0～7 个字节中的一个，通过 PC 端口发送到点阵屏的行引脚上。在发送行码之前，PC 端口先发送相应的列码选通对应列。由语句 PORTC＝_BV(i)可以看出，在本例点阵屏连接方式下，各列中的 LED 是共阳的，_BV(i)总是使第 i 列变为高电平，其他列为低电平，这时发送的行码将仅仅显示在第 i 列，这类似于当前发送给 8 位集成式数码管中的段码将仅仅显示在第 i 位上。

2. 实训要求

① 应用单片机的4个端口，控制16×16点阵LED显示屏显示。

② 利用2片595串入并出芯片分别控制8×8点阵屏的行码与列码，实现数字或字符显示。

3. 源程序代码

```
01  //-------------------------------------------------------------------
02  //   名称：8 * 8 LED点阵屏显示数字
03  //-------------------------------------------------------------------
04  //   说明：8 * 8 LED点阵屏循环显示数字0～9，刷新过程由定时器中断完成
05  //
06  //-------------------------------------------------------------------
07  #define F_CPU 4000000UL
08  #include <avr/io.h>
09  #include <avr/interrupt.h>
10  #include <util/delay.h>
11  #define INT8U   unsigned char
12  #define INT16U  unsigned int
13
14  //数字0～9的8 * 8点阵编码
15  const INT8U Table_OF_Digits[] =
16  {
17      0x00,0x3C,0x66,0x42,0x42,0x66,0x3C,0x00,//0
18      0x00,0x08,0x38,0x08,0x08,0x08,0x3E,0x00,//1
19      0x00,0x3C,0x42,0x04,0x08,0x32,0x7E,0x00,//2
20      0x00,0x3C,0x42,0x1C,0x02,0x42,0x3C,0x00,//3
21      0x00,0x0C,0x14,0x24,0x44,0x3C,0x0C,0x00,//4
22      0x00,0x7E,0x40,0x7C,0x02,0x42,0x3C,0x00,//5
23      0x00,0x3C,0x40,0x7C,0x42,0x42,0x3C,0x00,//6
24      0x00,0x7E,0x44,0x08,0x10,0x10,0x10,0x00,//7
25      0x00,0x3C,0x42,0x24,0x5C,0x42,0x3C,0x00,//8
26      0x00,0x38,0x46,0x42,0x3E,0x06,0x3C,0x00 //9
27  };
28
29  //-------------------------------------------------------------------
30  // 主程序
31  //-------------------------------------------------------------------
32  int main()
33  {
34      DDRA = 0xFF; PORTA = 0xFF;                    //配置端口
35      DDRC = 0xFF; PORTC = 0xFF;
36      TCCR0 = 0x03;                                 //预设分频:64
37      TCNT0 = 256 - F_CPU / 64.0 * 0.004;           //晶振4 MHz,4 ms定时初值
38      TIMSK = 0x01;                                 //允许T0定时器溢出中断
```

```
39          sei();                                          //开总中断
40          while (1);
41  }
42
43  //-------------------------------------------------------------------
44  // T0 定时器中断控制 LED 点阵屏刷新显示
45  //-------------------------------------------------------------------
46  ISR (TIMER0_OVF_vect)
47  {
48      static INT8U i = 0,t = 0,Num_Index = 0;
49      TCNT0 = 256 - F_CPU / 64.0 * 0.004;             //列间延时 4 ms
50      PORTC = _BV(i);                                 //列码
51      PORTA = ~Table_OF_Digits[Num_Index * 8 + i]; //行码(用~反相显示)
52      if( ++ i == 8) i = 0;                           //每屏一个数字由 8 个字节构成
53      if( ++ t == 250)                                //每个数字刷新显示一段时间
54      {
55          t = 0;
56          if( ++ Num_Index == 10) Num_Index = 0;      //显示下一个数字
57      }
58  }
```

4.10 8 位数码管段位复用串行驱动芯片 MAX6951 应用

Maxim 公司推出的 MAX6950/51 都是串行接口的共阴数码管显示驱动器，工作电压可低至 2.7 V，它们可分别驱动 5 位或 8 位的七段数码管。驱动芯片内置十六进制字符译码器(0～9，A～F)、复用扫描电路、段码和位码驱动器以及用于存储每一位数字的静态 RAM。

使用 MAX6950/51 时可以为每一位数字选择十六进制译码或非译码模式驱动任何七段数码管，每位数字不需要重写整个显示器即可单独寻址和刷新。该器件具有低功耗关断模式、数字亮度控制电路、扫描范围寄存器(允许用户选择 1～8 位显示数字)，各驱动器可相互保持同步的段闪烁控制以及强制所有 LED 打开的测试模式。本例电路及部分运行效果如图4－12所示。

1. 程序设计与调试

在设计 MAX6951 应用电路与应用程序之前，先简要介绍一下 6951 的引脚：

$\overline{\text{DIN}}$：串行数据输入，在 CLK 的上升沿将数据移入内部 16 位移位寄存器。

CLK：串行时钟输入，片选 CS 有效时，在 CLK 的上升沿，数据移入内部移位寄存器。

DIGX/SEGX：位码与段码复用驱动位，DIGX 吸入来自数码管共阴极的电流，SEGX 输出电流，位码与段码在关断时处于高阻状态。

ISET：电流设定，此引脚与 GND 之间串接电阻 R_{SET}，设置峰值电流，该电阻器还与电容器 C_{SET} 一起设置多路复用的显示时钟频率。

OSC：多路复用显示时钟输出。

CS：片选引脚，低电平时串行数据移入移位寄存器，在 CS 的上升沿锁存最后的 16 位数据。

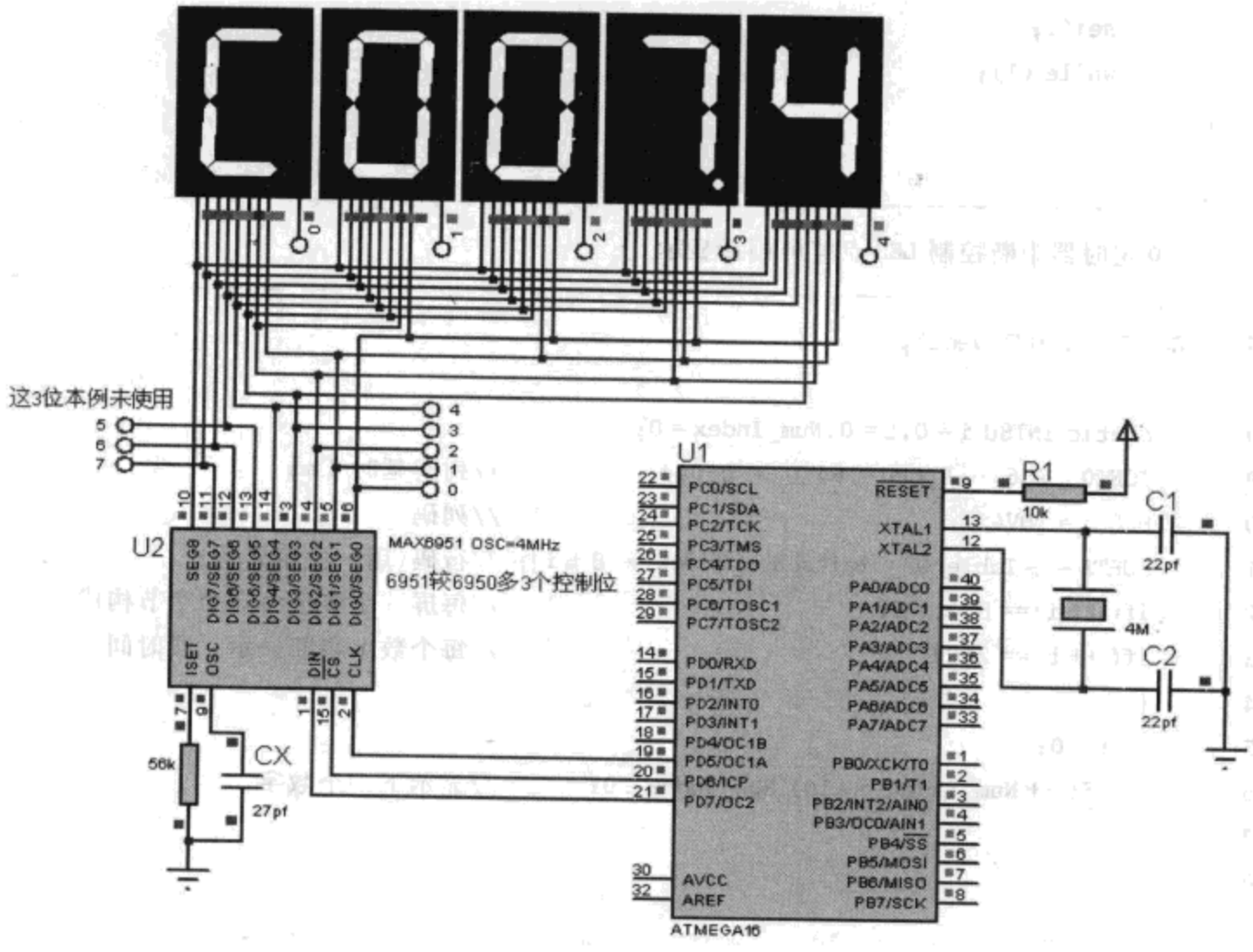


图 4－12　8 位数码管段位复用串行驱动芯片 MAX6951 应用

表 4－8 给出了 6951 与分立式数码管的连接方法，由于使用了段/位复用设计，本例中不能选用集成式数码管。图 4－12 中的 5 只独立数码管正是参照表 4－8 连接的，本例电路中还可以再添加 3 只数码管。

表 4－8　MAX6951 与 8 位分立式数码管的连接方法

	DIG/SEG0	DIG/SEG 1	DIG/SEG 2	DIG/SEG 3	DIG/SEG 4	DIG/SEG 5	DIG/SEG 6	DIG/SEG 7	SEG8
位 0	CC0	DP	G	F	E	D	C	B	A
位 1	DP	CC1	G	F	E	D	C	B	A
位 2	DP	G	CC2	F	E	D	C	B	A
位 3	DP	G	F	CC3	E	D	C	B	A
位 4	DP	G	F	E	CC4	D	C	B	A
位 5	DP	G	F	E	D	CC5	C	B	A
位 6	DP	G	F	E	D	C	CC6	B	A
位 7	DP	G	F	E	D	C	B	CC7	A

MAX6950/51 的 16 位串行数据格式为：D15～D0，其中高 8 位为地址，低 8 位为数据。本例函数 Write 完成对 16 位地址与数据字节的串行写入操作。本例中数码管 0～7 的地址分别为 0x60～0x67(本例实际使用了 5 位，即 0x60～0x64)，主程序中的 3 种演示将分别对这些地址进行写入。

为尽可能充分演示 MAX6950/51 的功能，主程序完成了全解码演示、部分解码演示、全部

不解码演示。主程序中的 86、90、94 行代码如下：

```
Write(0x01,0B00011111);      //解码模式:对 0～4 位全部解码
Write(0x01,0B00010110);      //解码模式:对 1,2,4 解码,第 0,3 位不解码
Write(0x01,0B00000000);      //解码模式:全部不解码
```

它们分别向 MAX6950/51 的 0x01 地址(即解码模式地址)写入了不同字节,其中置为 1 的对应位为解码,置为 0 的则不解码。以下分别进行说明：

① 在第 1 组演示所使用的解码模式下,只需要向 MAX6950/51 写入字符的 ASCII 码即可,其优点是不需要提供任何字符段码,不足之处是有些特殊字符在这种模式下无法显示,例如温度符号就是其内置编码表中没有的。

② 在第 2 组混合解码模式演示中,发送给 MAX6951 的一部分是字符 ASCII 码,一部分则是字符段码。

③ 在第 3 种模式下,由于全部不解码,因此需要提供所有待发送字符的段码。需要注意的是,这里的段码顺序与此前数码管由 A～DP 进行逆向编码的顺序是不同的,这是因为驱动线的连接不同,具体编码顺序可参考源程序中的相关说明。

阅读 MAX6950/51 初始化函数 Init_MAX695X 时,可进一步参阅 6951 的技术手册文件,该初始化函数分别设置了亮度、扫描范围及非关断模式。

2. 实训要求

① 6951 最多可驱动 8 位独立数码管,完成本例调试后,在电路中再添加 3 只独立数码管,并进一步改写程序,选用不同解码模式显示自定义数据内容。

② 同时使用 6950/51 两块芯片,驱动更多数据信息显示。

3. 源程序代码

```
001 //-------------------------------------------------------------------
002 //  名称：8 位数码管段位复用串行驱动芯片 MAX6951 应用
003 //-------------------------------------------------------------------
004 //  说明：本例程序仅占用 PD 端口 3 只引脚即实现了多位数码管的显示控制
005 //
006 //-------------------------------------------------------------------
007 #include <avr/io.h>
008 #include <util/delay.h>
009 #define INT8U   unsigned char
010 #define INT16U  unsigned int
011
012 //MAX695X 引脚操作定义
013 #define CLK_1() PORTD |=   _BV(PD5)
014 #define CLK_0() PORTD &= ~_BV(PD5)
015 #define DIN_1() PORTD |=   _BV(PD7)
016 #define DIN_0() PORTD &= ~_BV(PD7)
017 #define CS_1()  PORTD |=   _BV(PD6)
018 #define CS_0()  PORTD &= ~_BV(PD6)
019
020 //695X 待显示的几组数据-----------------------------------------
```

```
021  //1.显示 A、C、2、2、0,全解码(直接发送)
022  const INT8U Test1[] = {0x0A,0x0C,0x02,0x02,0x00};
023
024  //2.显示温度:-32℃,其中第0位0x01,第3位0x63不解码,
025  //它们分别是"-"的段码及"℃"中小圆圈的段码
026  const INT8U Test2[] = {0x01,0x03,0x02,0x63,0x0C};
027
028  //3.显示C000.0递增,全部不解码
029  //显示此数组时要使用MAX695X的段码表
030  INT8U Test3[] = {0x0C,0,0,0,0};
031
032  //在非解码模式下MAX6950/1对应的段码表,此表不同于直接驱动时所使用的段码表
033  //原来的各段顺序是:      DP,G,F,E,D,C,B,A
034  //MAX6950/1的驱动顺序是:DP,A,B,C,D,E,F,G
035  //除小数点位未改变外,其他位是逆向排列的
036  const INT8U SEG_CODE_695X[] =
037  { 0x7E,0x30,0x6D,0x79,0x33,0x5B,0x5F,0x70,
038    0x7F,0x7B,0x77,0x1F,0x4E,0x3D,0x4F,0x47
039  };
040  void Count_Demo();
041  //-------------------------------------------------------------
042  // 向MAX695X写数据
043  //-------------------------------------------------------------
044  void Write(INT8U Addr,INT8U Dat)
045  {
046      INT8U i;
047      CS_0();
048      for(i = 0; i<8; i ++ )              //串行写入8位地址Addr
049      {
050         CLK_0();
051         if (Addr & 0x80) DIN_1(); else DIN_0();
052         CLK_1(); _delay_us(20);          //时钟上升沿移入数据
053         Addr <<= 1;
054      }
055      for(i = 0; i<8; i ++ )              //串行写入8位数据Dat
056      {
057         CLK_0();
058         if (Dat & 0x80)  DIN_1(); else DIN_0();
059         CLK_1(); _delay_us(20);          //时钟上升沿移入数据
060         Dat <<= 1;
061      }
062      CS_1();
063  }
064
```

```
065 //-------------------------------------------------------------------
066 // MAX695X 初始化
067 //-------------------------------------------------------------------
068 void Init_MAX695X()
069 {
070     Write(0x02,0x07);                //设置亮度:中等亮度
071     Write(0x03,0x05);                //扫描所有数码管
072     Write(0x04,0x01);                //非关断 0x01;关断:0x00
073 }
074
075 //-------------------------------------------------------------------
076 // 主程序
077 //-------------------------------------------------------------------
078 int main()
079 {
080     INT8U i;
081     DDRD = 0xFF; PORTD = 0xFF;
082     Init_MAX695X();                  //695X 初始化
083     while (1)
084     {
085         //1 - 显示 A、C、2、2、0(全解码)-------------------------------
086         Write(0x01,0B00011111); //解码模式:对 0～4 位全部解码
087         for(i = 0; i<5; i ++ ) Write( 0x60 | i, Test1[i]);
088         _delay_ms(1000);
089         //2 - 显示温度: - 32 ℃(部分解码)-----------------------------
090         Write(0x01,0B00010110); //解码模式:对 1、2、4 解码,第 0、3 位不解码
091         for(i = 0; i<5; i ++ ) Write( 0x60 | i, Test2[i]);
092         _delay_ms(1000);
093         //3 - C000.0 递增演示(全部不解码,发送段码)-------------------
094         Write(0x01,0B00000000); //解码模式:全部不解码
095         Count_Demo();
096         _delay_ms(2000);
097     }
098 }
099
100 //-------------------------------------------------------------------
101 // 数码管数码递增演示 C000.0～C999.9(本例实际演示到 C015.0 时停止)
102 //-------------------------------------------------------------------
103 void Count_Demo()
104 {
105     INT8U i,j,k,l;
106     Write( 0x60, SEG_CODE_695X[Test3[0]]); //显示第一个字符 C
107     //以下分别显示 3 个整数位和 1 个小数位
108     for (i = 0; i<10; i ++ )
```

```
109     {
110         //显示百位数
111         Test3[1] = i; Write( 0x61, SEG_CODE_695X[Test3[1]]);
112         for (j = 0; j<10; j ++ )
113         {
114             //显示十位数
115             Test3[2] = j;     Write( 0x62, SEG_CODE_695X[Test3[2]]);
116             for (k = 0; k<10; k ++ )
117             {
118                 //显示个位数,小数点显示个位数旁边
119                 Test3[3] = k; Write( 0x63, SEG_CODE_695X[Test3[3]] | 0x80);
120                 for (l = 0; l<10; l ++ )
121                 {
122                     //显示小数位
123                     Test3[4] = l; Write( 0x64, SEG_CODE_695X[Test3[4]]);
124                     _delay_ms(80);
125                     //为提前结束演示,这里添加演示到15.0时退出的语句
126                     if (i == 0 && j == 1 && k == 5) return;
127                 }
128             }
129         }
130     }
131 }
```

4.11 串行共阴显示驱动器MAX7219与7221应用

本例所使用的MAX7219/7221是串行集成式共阴显示驱动器,它用来连接单片机与8位七段数码管,也可以连接条形LED或者8×8 LED点阵屏。本例中的两片MAX7219/7221驱动两组8位七段共阴数码管,两块芯片的串行数据输入线(DIN)与时钟线(CLK)分别共用PC0与PC2引脚,片选线($\overline{CS}$)与数据加载线(LOAD)独立。案例电路及部分运行效果如图4-13所示。

1. 程序设计与调试

本例用共阴显示驱动芯片MAX7219/7221控制数码管显示,每只仅占用单片机3只引脚。本例中通过复用串行数据线(DIN)与时钟线(CLK),两者共占用了PC端口的4只引脚。

除了对单片机端口的占用很少外,使用MAX7219/7221最大的优点还在于单片机向它输出所要显示的内容以后,不再需要用动态扫描法高速刷新数码管显示,这显然大大节省了对单片机时间的占用。

表4-9给出了MAX7219/7221的引脚功能说明,表4-10给出了MAX7219/7221的串行数据格式(16位)及寄存器地址表,阅读源程序中的初始化函数Init_MAX72XX和Write函数时可参考这些表格。

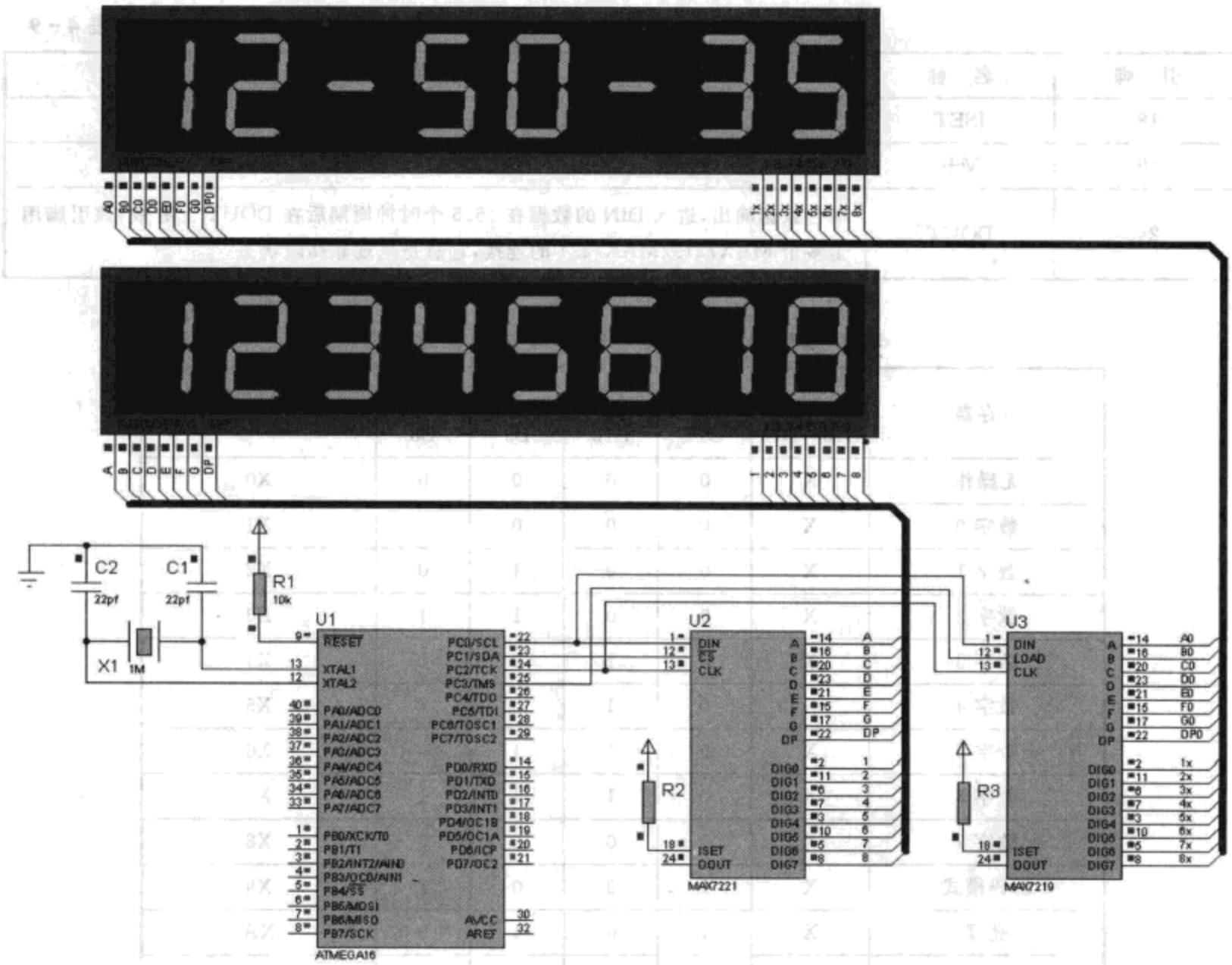


图 4-13　串行共阴显示驱动器 MAX7219 与 7221 应用

表 4-9　MAX7219/MAX7221 引脚功能表

引　脚	名　称	功　能
1	DIN	串行数据输入，数据在时钟上升沿进入内部 16 位的移位寄存器
2,3,5～8,10,11	DIG0～DIG7	接入 8 位共阴数码管反向电流的驱动线。在关闭时，MAX7219 将数字输出拉到 V+，而 MAX7221 的数字驱动线呈现高阻状态
4,9	GND	地端(两个 GND 都必须接地)
12	LOAD (MAX7219)	加载数据输入，最近的 16 位串行数据在 LOAD 的上升沿锁存
	CS/ (MAX7221)	片选输入，当 CS 为低电平时，串行数据加载到移位寄存器，最近的 16 位串行数据在 CS 的上升沿锁存
13	CLK	串行时钟输入，最大速率为 10 MHz，在时钟上升沿时数据移入内部移位寄存器，下降沿时数据由 DOUT 移出，对于 MAX7221，时钟仅在 CS 为低电平时有效
14～17,20～23	SEGA～SEGG,DP	数码管段驱动线(含小数位)，它们为数码管显示提供驱动电流，在关闭时，MAX7219 段驱动被拉到地，而 MAX7221 为高阻状态

续表 4-9

引 脚	名 称	功 能
18	ISET	通过电阻连接 VDD(R_{SET})以控制最高段电流
19	V+	正电源电压,连接+5 V
24	DOUT	串行数据输出,进入 DIN 的数据在 16.5 个时钟周期后在 DOUT 上有效,该引脚用于多个 MAX7219/MAX7221 的连接,它总是呈现非高阻状态

表 4-10 MAX7219/MAX7221 寄存器地址表

寄存器	地 址					十六进制编码
	D15～D12	D11	D10	D9	D8	
无操作	X	0	0	0	0	X0
数字 0	X	0	0	0	1	X1
数字 1	X	0	0	1	0	X2
数字 2	X	0	0	1	1	X3
数字 3	X	0	1	0	0	X4
数字 4	X	0	1	0	1	X5
数字 5	X	0	1	1	0	X6
数字 6	X	0	1	1	1	X7
数字 7	X	1	0	0	0	X8
解码模式	X	1	0	0	1	X9
亮度	X	1	0	1	0	XA
扫描范围	X	1	0	1	1	XB
关闭	X	1	1	0	0	XC
显示测试	X	1	1	1	1	XF

注:16 个数据位中,D11～D8 位为寄存器地址,D7～D0 为发送的数据。

2. 实训要求

① 将 MAX7219 或 7221 设为全部不解码,使数码管可同时显示普通字符与特殊字符。

② 重新设计电路,用两片 MAX7219 或 7221 控制两片 8×8 LED 点阵显示屏显示图文信息。

③ 重新设计电路,使两片 MAX7219 或 7221 的$\overline{CS}$(或 LOAD)、CLK 分别并联到 PC1 与 PC2 引脚,串行数据通过单片机 PC0 引脚传入第一片的 DIN 引脚,第一片的 DOUT 引脚则连接第二片的 DIN 引脚,重新编程在两片 MAX7219/7221 控制的数码管上显示两组数据信息。

④ 在完成上一设计后可再添加多片 MAX7219/7221,控制更多组数码管的显示。显然,采用 DOUT 与 DIN 级联的方法可使多片芯片只占用单片机的 3 只引脚。

3. 源程序代码

```
01  //-------------------------------------------------------------
02  //  名称:串行共阴显示驱动器 MAX7219/7221 控制数码管显示
03  //-------------------------------------------------------------
04  //  说明:本例用 MAX7219/7221 控制 8 只数码管动态显示,每组数字输出后
```