

```

53     }
54 }
55
56 //-----
57 // 生成正弦波
58 //-----
59 void Sin_Wave()
60 {
61     float i;
62     for (i = 0; i <= 2 * PI; i += 0.02)
63     {
64         * DAC0832 = 128 + sin(i) * 127;
65         _delay_us(100);
66         if (!S3_ON()) return;
67     }
68 }
69
70 //-----
71 // 主程序
72 //-----
73 int main()
74 {
75     DDRA = 0xFF;                      //PA 端口输出
76     DDRB = 0x00; PORTB = 0xFF;          //PB 端口输入,内部上拉
77     MCUCR |= 0x80;                   //允许访问外部接口
78     while (1)
79     {
80         if (S1_ON()) SawTooth_Wave();    //锯齿波
81         else if (S2_ON()) Triangle_Wave(); //三角波
82         else if (S3_ON()) Sin_Wave();      //正弦波
83         else if (S4_ON()) * DAC0832 = 0xFF;
84         else _delay_ms(100);
85     }
86 }

```

## 4.25 用带 SPI 接口的数/模转换芯片 MAX515 调节 LED 亮度

MAX515 是 Maxim 公司生产的一种低功耗的电压输出型 10 位串行 D/A 转换器,兼容 SPI 接口,MAX515 固定增益为 2,用 +5 V 单电源工作。本例运行时,通过调节 RV1 向单片机输入模拟电压,单片机将 A/D 转换后的数字量输出给 MAX515,经 D/A 转换后所输出的模拟电压控制 LED 亮度变化。本例电路及运行效果如图 4-37 所示。

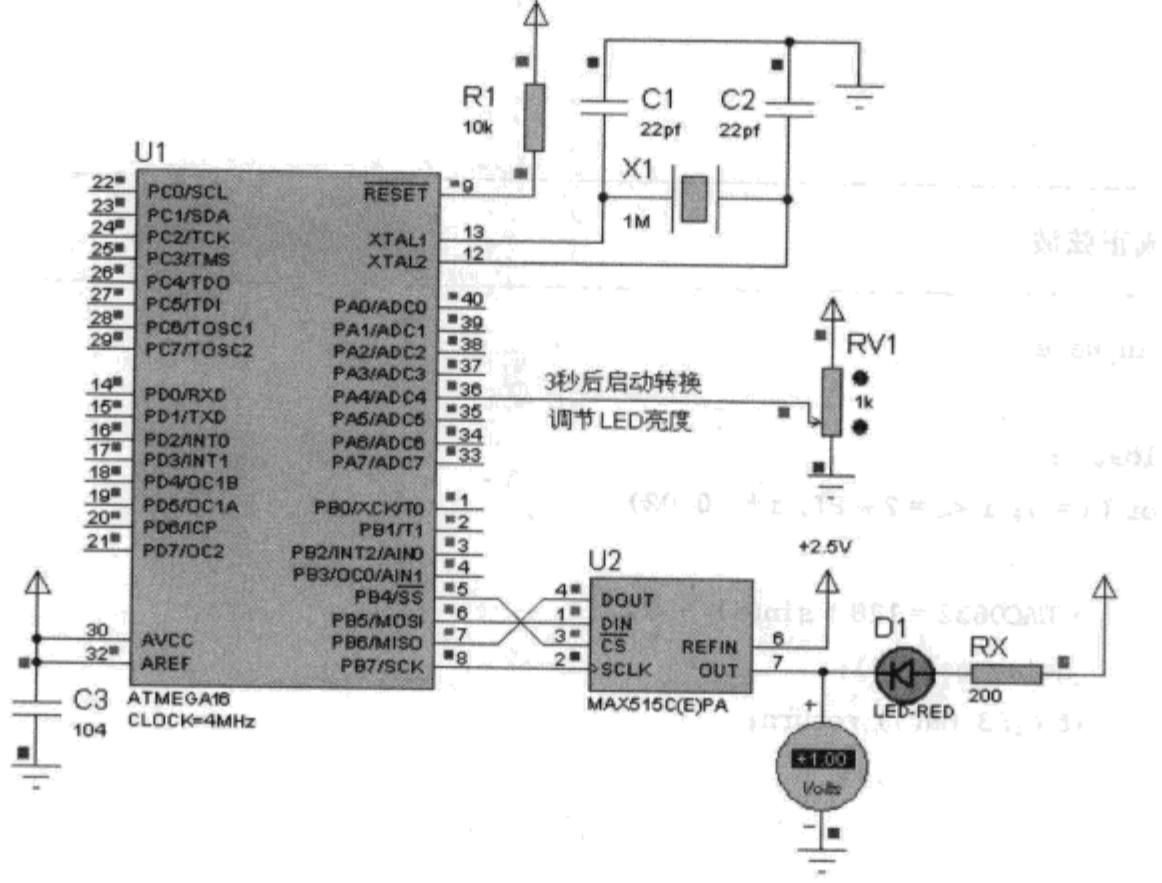


图 4-37 用带 SPI 接口的数/模转换芯片 MAX515 调节 LED 亮度

## 1. 程序设计与调试

AVR 单片机 A/D 转换精度为 10 位,以 A/D 转换输出的最大值为例:

对于 0B0000001111111111(即 0x03FF),这 10 位数据通过 SPI 接口写入 MAX515 时(实际写入 2 字节,共 16 位),其格式为 0B00001111111100(即 0x0FFC),高位填充了 4Bit 0,接着是 10Bit 待写入数位,最后是 2Bit 0,串行写入 MAX515 时从高位开始,写入 DIN 引脚的比特序列为 DIN→00111111110000。

比较 A/D 转换结果 0x03FF 和实际写入的 2 字节 0x0FFC 可知,在获取 A/D 转换结果后,向 MAX515 写入之前要先将其左移 2 位(<<2)。

AVR 单片机 A/D 转换输出的 10 位结果默认是右对齐的。如果将 ADMUX 寄存器中的 ADLAR(Left Adjust Result)位置 1,将输出结果设为左对齐,A/D 转换函数中原有语句:“ADMUX=ch;”(ch 为通道号,它影响 ADMUX 的低 4 位 MUX3~MUX0)相应修改为:“ADMUX= ch | \_BV(ADLAR);”,则 A/D 转换的最大值 0B000000111111111 将变为 0B111111111100000,即由 0x03FF 变为 0xFFC0,对于这样的输出结果,在写入 MAX515 进行 D/A 转换之前则要先将其右移 4 位(>>4)。

由于 MAX515 转换增益为 2,故输出电压公式为:

$$V_{out}=2 \times REFIN \times x / 1024 \quad (\text{其中 } x \text{ 为输入值})$$

本例电路中输入参考电压 REFIN 设为 +2.5 V,故输出电压最大值约为 +5 V,如果将 REFIN 设为 +1.25 V,则输出电压最大值约为 +2.5 V。

## 2. 实训要求

- ① 从 Proteus 元件库中选用与 MAX515 兼容的 TLC5615 重新设计本例。
- ② 根据 MAX515 技术手册模拟 SPI 操作时序重新编写本例程序。

- ③ 删除本例中的 RV1, 改写程序, 通过循环语句使 LED 反复由亮到暗变化。  
 ④ 设置 T/C1 工作于快速 PWM 模式, 利用低通滤波电路实现 DAC 输出。

### 3. 源程序代码

```

01 //-----
02 // 名称: 用带 SPI 接口的数/模转换芯片 MAX515 调节 LED 亮度
03 //-----
04 // 说明: 本例模拟信号由 PA4 引入, 转换为数字信号后由 PB4 写入 MAX515,
05 //         MAX515 进行数/模数转换后控制 LED 亮度变化
06 //
07 //-----
08 #include <avr/io.h>
09 #include <util/delay.h>
10 #define INT8U unsigned char
11 #define INT16U unsigned int
12
13 //SPI 使能与禁用
14 #define SPI_EN() (PORTB &= ~_BV(PB4))
15 #define SPI_DI() (PORTB |= _BV(PB4))
16 //-----
17 // SPI 主机初始化
18 //-----
19 void SPI_MasterInit()
20 {
21     //配置 SPI 端口方向
22     DDRB = 0B10110000; PORTB = 0xFF;
23     //SPI 使能, 主机模式, 16 分频
24     SPCR |= _BV(SPE) | _BV(MSTR) | _BV(SPR0);
25 }
26
27 //-----
28 // SPI 数据传输
29 //-----
30 INT8U SPI_Transmit(INT8U d)
31 {
32     SPDR = d;                      //启动数据传输
33     while(!(SPSR & _BV(SPIF)));   //等待结束
34     SPSR |= _BV(SPIF);            //清中断标志
35     return SPDR;
36 }
37
38 //-----
39 // 对通道 CH 进行模/数转换

```



```
40 //-----
41 INT16U ADC_Convert(INT8U CH)
42 {
43     ADMUX = CH ;           //ADC 通道选择
44     return ADC;           //返回转换结果
45 }
46
47 //-----
48 // 主程序
49 //-----
50 int main()
51 {
52     INT16U dat;
53     DDRA = 0x00;           //模/数转换端口设为输入
54     SPI_MasterInit();      //SPI 主机初始化
55     ADCSRA = 0xE6;         //ADC 转换置位,启动转换,64 分频
56     _delay_ms(3000);       //延时 3 s 等待系统稳定
57     while (1)
58     {
59         dat = ADC_Convert(4); //获取通道 AD4 的模/数转换结果
60         //向 MAX515 写入 16 位的 2 字节数据时,由高位到低位,其格式如下:
61         //0000 - XXXXXXXXXX - 00
62         //其中:高 4 位为填充 0,接着的 10 位 X 为输入值,最后是 2 位 0
63         //对右对齐输出的 10 位 A/D 转换结果,其左端共有 6 个 0,即:
64         //000000 - XXXXXXXXXX
65         //为适应写入 MAX515 的格式,dat 要先左移 2 位
66         dat <<= 2;
67
68         //通过 SPI 接口向 MAX515 发送 10 位的转换结果
69         SPI_EN();             //使能 SPI
70         SPI_Transmit(dat>>8); //发送高 8 位(其中低 4 位有效)
71         SPI_Transmit(dat);   //发送低 8 位(其中高 6 位有效)
72         SPI_DI();             //禁用 SPI
73         _delay_ms(100);
74     }
75 }
```

## 4.26 正反转可控的直流电机

运行本例时,按下 K1 可使直流电机正转,按下 K2 可使直流电机反转,按下 K3 时停止,在进行相应操作时,对应 LED 将被点亮。本例电路及部分运行效果如图 4-38 所示。

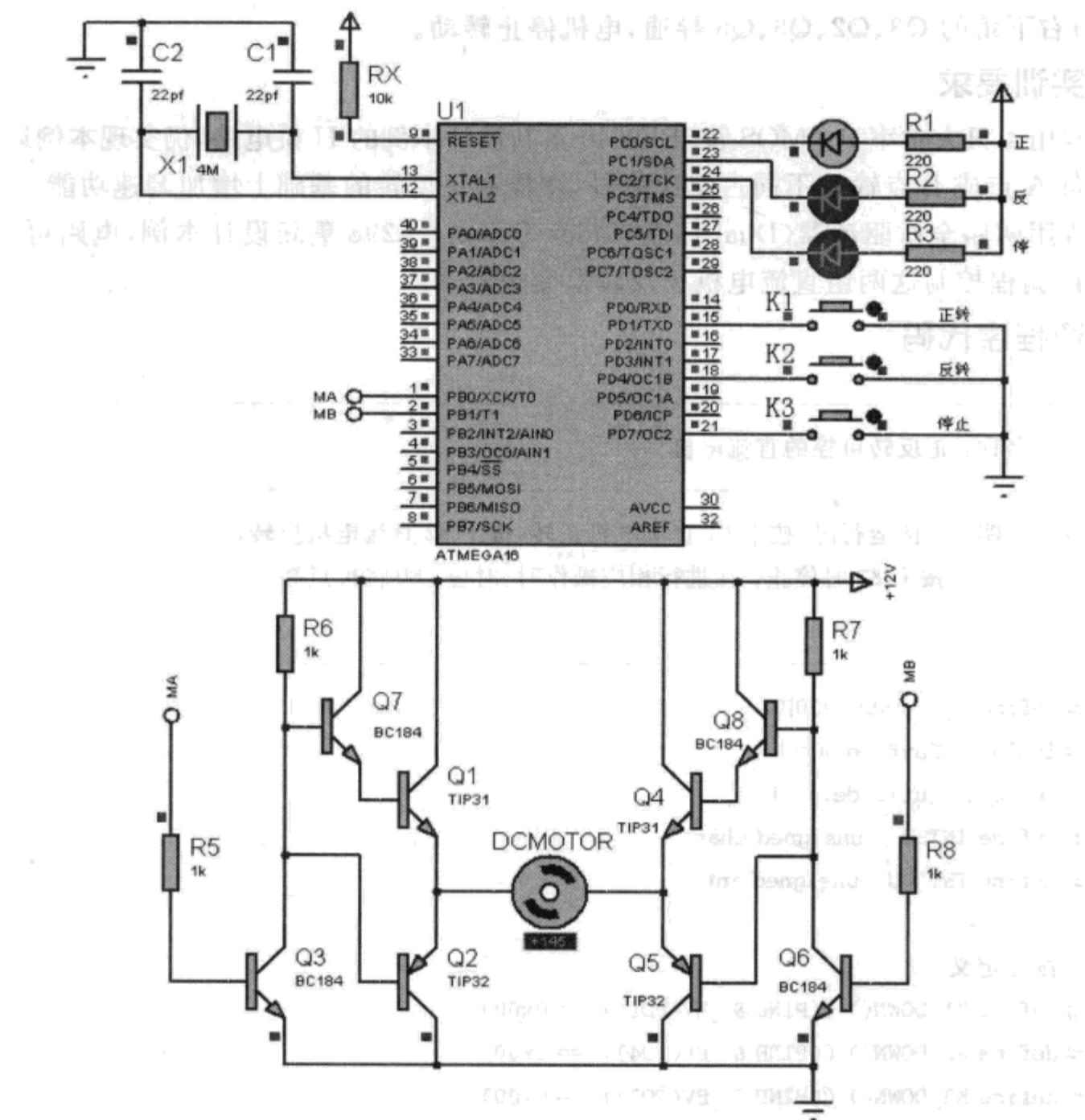


图 4-38 正反转可控的直流电机

## 1. 程序设计与调试

本例代码编写非常简单,案例关键在于电路的搭建,本例给出的这种直流电机驱动电路称为H桥驱动电路。对于图4-38所示电路:

当MA点为低电平时,Q3、Q2截止,Q7、Q1导通,电机左端呈现高电平;当MB点为高电平时,Q8、Q4截止,Q6、Q5导通,电机右端呈现低电平。因此,在MA为0、MB为1时电机正转。

反之,当MA点为高电平时,Q3、Q2导通,Q7、Q1截止,电机左端呈现低电平;当B点为低电平时,Q8、Q4导通,Q6、Q5截止,电机右端呈现高电平。因此,在MA为1、MB为0时电机反转。

当MA点和MB点同为低电平时,电机两端均为高电平,电机停止转动。同样,当MA点和MB点同为高电平时,电机两端均为低电平,电机停止转动。

由以上分析可知:电路中左上角Q7、Q1与右下角Q6、Q5导通时电机正转;反之,当右上角Q8、Q4与左下角Q3、Q2导通时电机反转。如果左上角与右上角的Q7、Q1、Q8、Q4导通或

左下角与右下角的 Q3、Q2、Q6、Q5 导通，电机停止转动。

## 2. 实训要求

- ① 改用 4 只大功率 P-MOS 管 IRF9540 重新设计本例的 H 桥电路，仍实现本例运行效果。
- ② 向 A 点或 B 点输入不同占空比信号，在控制正反转的基础上增加调速功能。
- ③ 改用两路全桥驱动器(Dual Full-Bridge Driver) L298 重新设计本例，电路可加入两组直流电机，编程控制这两组直流电机正反转及变速转动。

## 3. 源程序代码

```

01 //-----
02 // 名称：正反转可控的直流电机
03 //-----
04 // 说明：本例运行时，按下 K1 直流电机正转，按下 K2 直流电机反转，
05 //       按下 K3 时停止。在进行相应操作时，对应 LED 将被点亮
06 //
07 //-----
08 #define F_CPU 4000000UL
09 #include <avr/io.h>
10 #include <util/delay.h>
11 #define INT8U unsigned char
12 #define INT16U unsigned int
13
14 //按键定义
15 #define K1_DOWN() ((PIND & _BV(PD1)) == 0x00)
16 #define K2_DOWN() ((PIND & _BV(PD4)) == 0x00)
17 #define K3_DOWN() ((PIND & _BV(PD7)) == 0x00)
18 //LED 定义
19 #define LED1_ON() ( PORTC = 0B11111110 )
20 #define LED2_ON() ( PORTC = 0B11111101 )
21 #define LED3_ON() ( PORTC = 0B11111011 )
22 //电机控制端 A,B 操作定义
23 #define MA_0() ( PORTB &= ~_BV(PB0) )
24 #define MA_1() ( PORTB |= _BV(PB0) )
25 #define MB_0() ( PORTB &= ~_BV(PB1) )
26 #define MB_1() ( PORTB |= _BV(PB1) )
27 //-----
28 // 主程序
29 //-----
30 int main(void)
31 {
32     DDRB = 0xFF; PORTB = 0xFF;          //配置端口
33     DDRC = 0xFF; PORTC = 0xFF;
34     DDRD = 0x00; PORTD = 0xFF;

```

```

35     LED3_ON();                                //停转指示灯亮
36     while (1)
37     {
38         if (K1_DOWN())                         //正转
39         {
40             while (K1_DOWN());
41             LED1_ON(); MA_0(); MB_1();
42         }
43         if (K2_DOWN())                         //反转
44         {
45             while (K2_DOWN());
46             LED2_ON(); MA_1(); MB_0();
47         }
48         if (K3_DOWN())                         //停止
49         {
50             while (K3_DOWN());
51             LED3_ON(); MA_0(); MB_0();
52         }
53     }
54 }

```

## 4.27 正反转可控的步进电机

ULN2003 是高耐压、大电流达林顿阵列，由 7 个硅 NPN 达林顿管组成。ULN2003 灌电流可达 500 mA，并且能够在关态时承受 50 V 的电压，输出还可以在高负载电流并行运行。本例使用 ULN2003 驱动步进电机，在运行过程中，按下 K1 将使步进电机正转 3 圈，按下 K2 时反转 3 圈，在转动过程中按下 K3 时可使步进电机停止转动。本例电路及部分运行效果如图 4-39 所示。

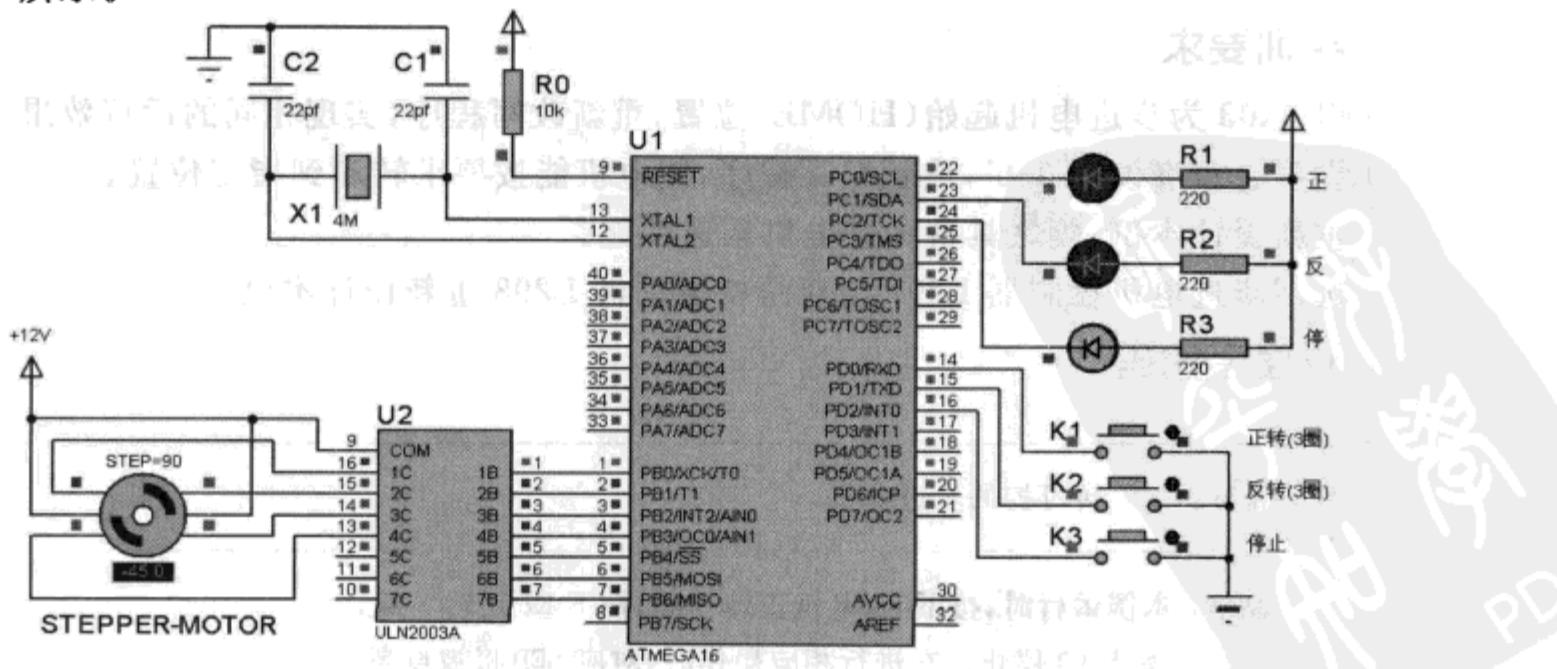


图 4-39 正反转可控的步进电机



## 1. 程序设计与调试

本例关键在于励磁序列的定义,表 4-18 给出了四相步进电机的 3 种励磁方式,本例四相步进电机工作于 8 拍方式,参考该表可得出步进电机正转与反转控制序列数组如下:

```
//正转励磁序列为 A→AB→B→BC→C→CD→D→DA  
const INT8U FFW[] = {0x01,0x03,0x02,0x06,0x04,0x0C,0x08,0x09};  
//反转励磁序列为 A→AD→D→CD→C→BC→B→AB  
const INT8U REV[] = {0x01,0x09,0x08,0x0C,0x04,0x06,0x02,0x03};
```

表 4-18 四相步进电机的 3 种励磁方式

STEP	单 4 拍				双 4 拍				8 拍			
	A	B	C	D	A	B	C	D	A	B	C	D
1	1	0	0	0	1	1	0	0	1	0	0	0
2	0	1	0	0	0	1	1	0	1	1	0	0
3	0	0	1	0	0	0	1	1	0	1	0	0
4	0	0	0	1	1	0	0	1	0	1	1	0
5	1	0	0	0	1	1	0	0	0	0	1	0
6	0	1	0	0	0	1	1	0	0	0	1	1
7	0	0	1	0	0	0	1	1	0	0	0	1
8	0	0	0	1	1	0	0	1	1	0	0	1

本例电机步进角度为 90°(步电机组件中 Step Angle 属性默认值为 90)。在四相八拍方式下,按上述励磁序列,每拍可步进 45°角。对于 FFW 数组,当选用 0x01 使电机归位时,电机处于 -45°角位置(选用 0x03 时,电机处于 0°角位置),循环输出 FFW 数组的每一拍后,电机总共步进 7 步,下一趟循环的第一拍 0x01 将使其到达最初的起点位置(HOME),这时才刚好形成完整的一圈。可见,每一圈的 8 拍都使其步进 7 步,下一圈的第一拍走完后才是完整的一圈。因此,本例程序在控制其正转 n 圈时,最后一圈的后面单独补上了 0x01 这一拍。反转 n 圈亦如此。

## 2. 实训要求

- ① 以 0x03 为步进电机起始(HOME)位置,重新改写程序,实现相同的运行效果。
- ② 设步进角度设为 3.6°,重新编写程序,使电机能按要求转动到指定位置。
- ③ 重新设计本例,使之具备步进电机调速功能。
- ④ 选用步进电机控制器 L297 与双全桥驱动器 L298 重新设计本例。

## 3. 源程序代码

```
01 //-----  
02 // 名称: 正反转可控的步进电机  
03 //-----  
04 // 说明: 本例运行时,按下 K1 电机正转 3 圈,按下 K2 反转 3 圈,  
05 // 按下 K3 停止。在进行相应操作时,对应 LED 将被点亮  
06 //-----
```

```

07 //-----
08 #define F_CPU 4000000UL
09 #include <avr/io.h>
10 #include <util/delay.h>
11 #define INT8U unsigned char
12 #define INT16U unsigned int
13
14 //本例四相步进电机工作于 8 拍方式
15 //正转励磁序列为 A->AB->B->BC->C->CD->D->DA
16 const INT8U FFW[] = {0x01,0x03,0x02,0x06,0x04,0x0C,0x08,0x09};
17 //反转励磁序列为 A->AD->D->CD->C->BC->B->AB
18 const INT8U REV[] = {0x01,0x09,0x08,0x0C,0x04,0x06,0x02,0x03};
19
20 //按键定义
21 #define K1_DOWN() ((PIND & _BV(PD0)) == 0x00) //K1:正
22 #define K2_DOWN() ((PIND & _BV(PD1)) == 0x00) //K2:反
23 #define KX_DOWN() ( PIND != 0xFF ) //KX:任意键按下
24 //-----
25 // 步进电机正转或反转 n 圈(本例步进电机在 8 拍方式下每次步进 45°角)
26 //-----
27 void STEP_MOTOR_RUN(INT8U Direction, INT8U n)
28 {
29     INT8U i,j;
30     for (i = 0; i<n; i++)          //转动 n 圈
31     {
32         for (j = 0; j<8; j++)      //循环输出 8 拍
33         {
34             if(KX_DOWN()) return; //中途按下 KX 时电机停止转动
35             if (Direction == 0)
36                 PORTB = FFW[j]; //方向为 0 时正转
37             else
38                 PORTB = REV[j]; //方向为 1 时反转
39             _delay_ms(200);
40         }
41     }
42     PORTB = 0x01;                //最后一圈之后输出 0x01 这一拍,电机回到起点
43 }
44
45 //-----
46 // 主程序
47 //-----
48 int main()
49 {

```



```
50     INT8U r = 3;           //转动圈数
51     DDRB = 0xFF; PORTB = FFW[0]; //控制输出,电机归位
52     DDRC = 0xFF; PORTC = 0xFF;   //LED 输出
53     DDRD = 0x00; PORTD = 0xFF;   //按键输入
54     while(1)
55     {
56         if(K1_DOWN())
57         {
58             while (K1_DOWN()); //等待 K1 释放
59             PORTC = 0xFE;      //LED1 点亮
60             STEP_MOTOR_RUN(0,r); //电机正转 r 圈
61         }
62         if(K2_DOWN())
63         {
64             while (K2_DOWN()); //等待 K2 释放
65             PORTC = 0xFD;      //LED2 点亮
66             STEP_MOTOR_RUN(1,r); //电机反转 r 圈
67         }
68         PORTC = 0xFB;        //LED3 点亮
69     }
70 }
```

## 4.28 DS18B20 温度传感器测试

本例的 DS18B20 数字温度计是 DALLAS 公司生产的 1-Wire 式单总线器件, 每个器件都有唯一的序列号。DS18B20 体积很小, 用它来组成的温度测量系统线路非常简单, 只要求一个端口即可实现通信, 其温度测量范围为 -55~+125 ℃, 数字温度计的分辨率可以从 9 位到 12 位选择, 内部可设置报警温度上、下限。

运行本例时, 1602LCD 将显示 DS18B20 所测量的外部温度, 调节 DS18B20 模拟改变外界温度时, 新的温度值将刷新显示在 LCD 上。

案例电路及部分运行效果如图 4-40 所示。

### 1. 程序设计与调试

通过本例学习调试后要熟悉 DS18B20 的应用技术要点, 包括 DQ 引脚的操作时序、温度传感器命令应用、所读取温度数据的格式转换、摄氏度(℃)符号的液晶显示等。

在阅读本例时需要参考图 4-41 所示的 DS18B20 内存结构表 4-19 所列的 DS18B20RAM 操作命令集及表 4-20 所列的温度寄存器字节格式。由图 4-41 可知, 传感器初始上电时温度寄存器初值为 0x0550(表示 85 ℃)。

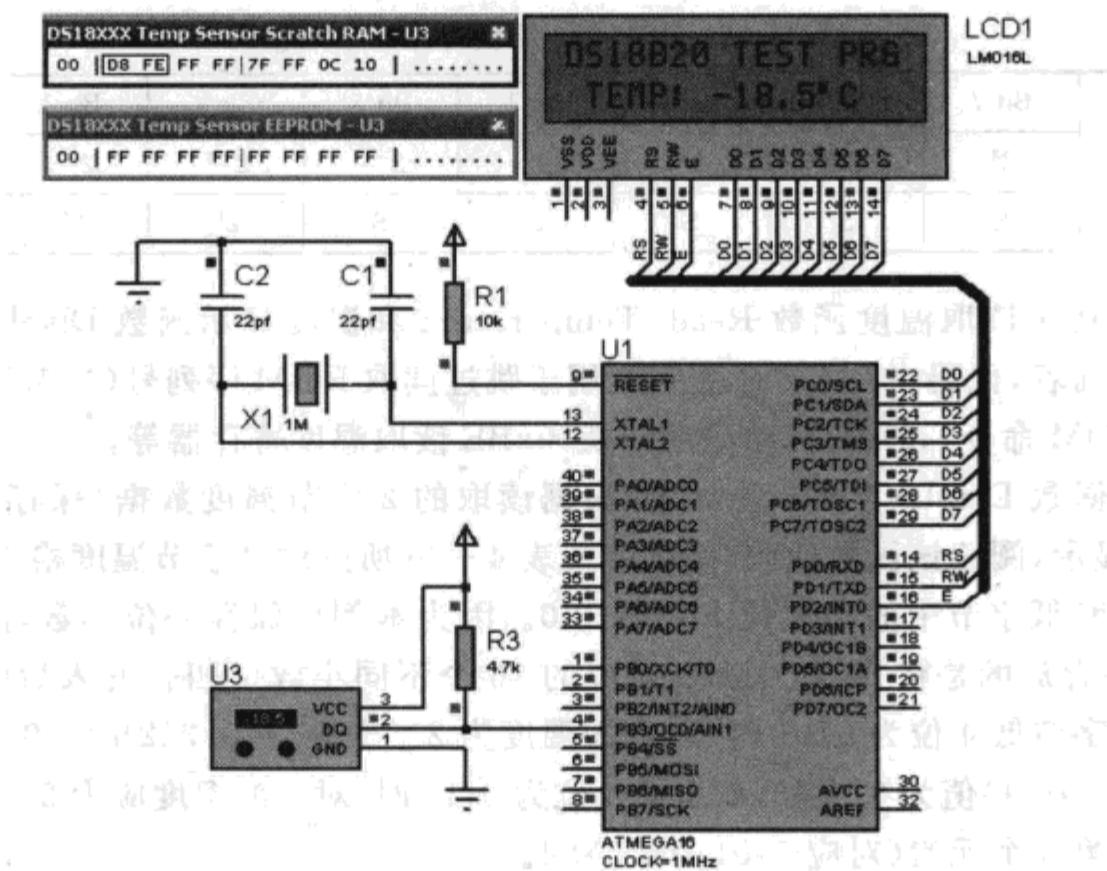


图 4-40 DS18B20 温度传感器测试

Byte 0	温度低字节 (50h)
Byte 1	温度高字节 (05h)
Byte 2	TH 寄存器或用户字节 1
Byte 3	TL 寄存器或用户字节 2
Byte 4	配置寄存器
Byte 5	保留 (FFh)
Byte 6	保留 (0Ch)
Byte 7	保留 (10h)
Byte 8	CRC

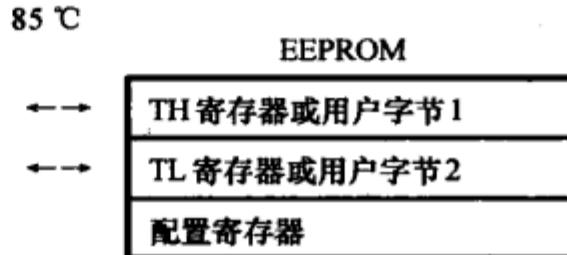


图 4-41 DS18B20 内存结构图

表 4-19 DS18B20 功能命令集

命令	说明	协议	总线数据操作
温度转换	开始温度转换	44H	DS18B20 将转换状态发送给主设备
读寄存器	读所有寄存器, 包括 CRC 字节	BEH	DS18B20 将 9 个字节的数据发送给主设备
写寄存器	将数据写入寄存器 2,3,4 字节(即 TH, TL 和配置寄存器)	4EH	主设备向 DS18B20 发送 3 个字节数
复制	将寄存器 TH, TL 和配置寄存器数据复制到 EEPROM	48H	无
回调	由 EEPROM 向寄存器恢复 TH, TL 和配置寄存器数据	B8H	DS18B20 将恢复状态发送给主设备
读电源	主设备读取 DS18B20 电源模式	B4H	DS18B20 向主设备发送电源状态



表 4-20 DS18B20 温度寄存器字节格式

位	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
LSB	$2^3$	$2^2$	$2^1$	$2^0$	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$
MSB	S	S	S	S	S	$2^6$	$2^5$	$2^4$

本例要点在于读取温度函数 Read\_Temperature 和温度显示函数 Display\_Temperature 的编写。对于前者，代码中 0xCC 命令字节用于跳过读取 ROM 序列号（参考 DS18B20 手册中的 DS18B20ROM 命令），0x44 启动温度转换，0xBE 读取温度寄存器等。

温度显示函数 Display\_Temperature 根据读取的 2 字节温度数据（保存在 Temp\_Value 数组中）进行显示，阅读该函数代码时，可参考表 4-20 所列的 2 字节温度格式，特别是高字节中的符号位 S 和低字节中的小数位 Bit3~Bit0。因为本例仅保存一位小数，温度小数位对照表 df\_Table 中存放的是将 0000~1111 对应的 16 个不同小数位四舍五入后的结果。假设当前温度数据低字节低 4 位为 0101 时，对应的温度为  $2^{-2} + 2^{-4} = 0.3125 \approx 0.3$ ，因此数组第 5 个元素（对应于 0101）值为 3；又如，如果低 4 位为 0110 时，对应的温度应为  $2^{-2} + 2^{-3} = 0.375 \approx 0.4$ ，因此数组第 6 个元素（对应于 0110）值为 4。

本例除通过查表法获取温度小数位以外，还在源程序中提供了通过计算法获取温度小数位的代码，为便于阅读分析，代码附上了详细的说明与注释。

对于 DS18B20 的 ROM 操作命令、时序及其他技术细节，可进一步参考 DS18B20 的技术手册。

运行本例并显示温度以后，暂停程序，单击 Debug 菜单中的 DS18B20 菜单，打开 RAM 和 EEPROM 菜单，所显示的两个小窗口如图 4-38 左上角所示。当前读取的 2 字节温度值为 FED8，即 11111110 11011000，由表 4-20 所示的两字节温度寄存器格式可知该温度为负数，将该值取反加 1 后可得补码：00000001 00101000。这 16 位中最低 4 位 1000 对应于小数位，查温度小数表 df\_Table 中第 8(1000)项可得小数位 5，即 0.5 °C，温度整数部分由高字节中的低 4 位与低字节中的高 4 位构成，即 00010010，转换为十进制数是 18，两数相加并添加“-”可得 -18.5 °C。

为显示摄氏度(°C)符号，本例提供了以下两种方法：

方法一：

本例液晶字符编码表共 16 行 16 列(256 个)，其中固定字符有 192 个。通过查阅本例液晶技术手册可知度(°)的符号编码是 0xDF(1101 1111)。根据编码 0xDF 可直接显示度(°)的符号，然后再在其后附加摄氏符号(C)，缺点是度(°)的符号与摄氏符号(C)之间会一些间隔，因为它更靠近温度数据的右上角，而不是 C 的左上角。

方法二：

在本例液晶的 16 行 16 列(256 个)字符编码中，前 16 个编码被分配给自定义 CGRAM 点阵字符编码，编码依次为 0x00~0x0F。

为输出自定义的度(°)的符号，可先获取度(°)的符号点阵数据，然后将其写入液晶 CGRAM。本例写入的 CGRAM 地址空间是 0x40~0x47(共 8 字节点阵数据)，其对应的字符编码为 0x00，如果要重新写入第二个自定义字符，则 CGRAM 地址区间为 0x48~0x4F，其对应字符编码为 0x01，以此类推。如果连续写入多个自定字符的点阵数据，则 CGRAM 起始地

址不需要重新指定。

将度(°)的符号点阵写入 CGRAM 后,即可通过输出编码 0x00 来显示该符号了,其后再附加显示摄氏符号(C)即可。使用这种方法时可以自行调整度(°)的符号点阵数据,使其显示时更靠近摄氏符号(C)的左上角。绘制液晶字符点阵或用软件生成点阵时,注意选择横向取模,且左边为高位。

当然,使用方法二时还可以将摄氏度符号(℃)看成一个整体来获取点阵,然后再写入 CGRAM,这时的摄氏度符号输出只需使用一字节编码。

## 2. 实训要求

- ① 修改程序,使温度显示精确到 2 位小数。
- ② 重新设计本例,实现多点温度的测量与显示。

## 3. 源程序代码

```

01 //----- main.c -----
02 // 名称: DS18B20 温度传感器测试
03 //-----
04 // 说明: 运行本例时,外界温度将显示在 1602 LCD 上
05 // 调节 DS18B20 时,所模拟的外界温度值将刷新显示在液晶显示屏上,
06 // 包括正负温度及小数位
07 //
08 //-----
09 # include <avr/io.h>
10 # include <util/delay.h>
11 # include <string.h>
12 # define INT8U unsigned char
13 # define INT16U unsigned int
14
15 //液晶相关函数
16 extern void Initialize_LCD();
17 extern void Set_LCD_POS(INT8U x, INT8U y);
18 extern void Write_LCD_Data(INT8U dat);
19 extern void LCD_ShowString(INT8U x, INT8U y, char * str);
20
21 //温度传感器相关函数
22 extern void Read_Temperature();
23 extern void Temperature_Convert();
24 extern char Current_Temp_Display_Buffer[];
25 extern INT8U DS18B20_ERROR;
26 //-----
27 // 显示温度信息
28 //-----
29 void Disp_Temperature()
30 {

```



```
31 //在第 2 行显示当前温度
32 LCD_ShowString(0,1,Current_Temp_Display_Buffer);
33 //摄氏度符号从第 2 行 12 列开始显示
34 Set_LCD_POS(12,1);
35 //显示摄氏温度符号℃ 中的度符号(°)时有两种方法:
36 //方法 1:查询本例液晶手册可知该符号的编码为 0xDF,直接显示该编码即可
37 Write_LCD_Data(0xDF);
38 //方法 2:将该符号点阵数据写入 CGRAM,0x00~0x0F 全部是自定义字符的编
39 //      码。本例将该符号写入 CGRAM 地址 0x40~0x47,对应字符编码为 0x00,
40 //      因此也可以用下面的语句来度符号(°)
41 //Write_LCD_Data(0x00);
42 //最后附加显示摄氏符号 C
43 Set_LCD_POS(13,1); Write_LCD_Data('C');
44 }
45
46 //-----
47 // 主程序
48 //-----
49 int main()
50 {
51     DDRB = 0x00; PORTB = 0x00;           //端口配置
52     DDRC = 0xFF;
53     DDRD = 0xFF;
54     Initialize_LCD();                 //初始化 LCD
55     LCD_ShowString(0,0,"DS18B20 TEST PRG"); //输出两行提示信息
56     LCD_ShowString(0,1,"    Wait...    ");
57     Read_Temperature();              //预读温度
58     _delay_ms(1000);                //等待 1 s
59     LCD_ShowString(0,1,"          ");
60     while(1)
61     {
62         Read_Temperature();          //读取温度
63         if (!DS18B20_ERROR)
64         {
65             Temperature_Convert();    //温度转换为所需要的格式
66             Disp_Temperature();      //LCD 显示温度
67             _delay_ms(100);
68         }
69     }
70 }

001 //----- DS18B20.c -----
002 // 名称: DS18B20 温度传感器程序
```

```

003 //-----
004 #define F_CPU 1000000UL
005 #include <avr/io.h>
006 #include <util/delay.h>
007 #define INT8U unsigned char
008 #define INT16U unsigned int
009
010 //DS18B20 引脚定义
011 #define DQ PB3
012 //设置数据方向
013 #define DQ_DDR_0()    DDRB &= ~_BV(DQ)
014 #define DQ_DDR_1()    DDRB |= _BV(DQ)
015 //温度管引脚操作定义
016 #define DQ_1()         PORTB |= _BV(DQ)
017 #define DQ_0()         PORTB &= ~_BV(DQ)
018 #define RD_DQ_VAL()   (PINB & _BV(DQ)) //注意保留这一行的括号
019
020 //温度小数位对照表
021 //如果不使用此表,也可以使用本例后面代码中提供的小数位计算程序
022 const INT8U df_Table[] = {0,1,1,2,3,3,4,4,5,6,6,7,8,8,9,9}; //四舍五入表
023
024 //当前读取的温度整数部分
025 INT8U CurrentT = 0 ;
026 //从 DS18B20 读取的温度值
027 INT8U Temp_Value[] = {0x00,0x00};
028 //待显示的各温度数位
029 INT8U Display_Digit[] = {0,0,0,0};
030 //传感器状态标志
031 INT8U DS18B20_ERROR = 0;
032 //当前温度显示缓冲
033 char Current_Temp_Display_Buffer[] = {" TEMP:      "};
034 //-----
035 // 初始化 DS18B20
036 //-----
037 INT8U Init_DS18B20()
038 {
039     INT8U status;
040     DQ_DDR_1(); DQ_0(); _delay_us(500); //主机拉低 DQ, 占领总线
041     DQ_DDR_0();           _delay_us(50); //DQ 设为输入
042     status = RD_DQ_VAL(); _delay_us(500); //读总线, 为 0 时器件在线
043     DQ_1();                  //释放总线
044     return status;          //返回器件状态(0 为正常)
045 }

```



```
046
047 //-----
048 // 读 1 字节
049 //-----
050 INT8U ReadOneByte()
051 {
052     INT8U i, dat = 0;
053     for (i = 0; i < 8; i++)           //串行读取 8 位
054     {
055         DQ_DDR_1(); DQ_0();          //写 0 拉低 DQ 占领总线
056         DQ_DDR_0();                //读 DQ 引脚
057         if(RD_DQ_VAL()) dat |= _BV(i); //读取的第 i 位放入 dat 内对应位置
058         _delay_us(80);             //延时
059     }
060     return dat;                  //返回读取的 1 字节数据
061 }
062
063 //-----
064 // 写 1 字节
065 //-----
066 void WriteOneByte(INT8U dat)
067 {
068     INT8U i ;
069     for (i = 0x01; i != 0x00; i <<= 1)    //串行写入 8 位
070     {
071         DQ_DDR_1(); DQ_0();          //写 0 拉低 DQ 占领总线
072         if (dat & i) DQ_1(); else DQ_0(); //向 DQ 数据线写 0/1
073         _delay_us(80);             //延时
074         DQ_1();                    //释放总线
075     }
076 }
077
078 //-----
079 // 读取温度值
080 //-----
081 void Read_Temperature()
082 {
083     if( Init_DS18B20() != 0x00 )        //DS18B20 故障
084         DS18B20_ERROR = 1;
085     {
086         WriteOneByte(0xCC);            //跳过序列号
087         WriteOneByte(0x44);            //启动温度转换
088         Init_DS18B20();              //重新启动 DS18B20
```

```

089     WriteOneByte(0xCC);           //跳过序列号
090     WriteOneByte(0xBE);           //读取温度寄存器
091     Temp_Value[0] = ReadOneByte(); //温度低 8 位
092     Temp_Value[1] = ReadOneByte(); //温度高 8 位
093     DS18B20_ERROR = 0;
094 }
095 }
096
097 //-----
098 // 温度值转换
099 //-----
100 void Temperature_Convert()
101 {
102     INT8U ng = 0; //负数标识
103     //如果不查表法换算小数时可使用下面两行
104     //INT8U i; float Temp_Df = 0.0;
105     //如果为负数则取反加 1，并设置负数标识
106     //按技术手册说明，高 5 位为符号位，应与上 0xF8 进行 + / - 判断
107     if ((Temp_Value[1] & 0xF0) == 0xF0)
108     {
109         Temp_Value[1] = ~Temp_Value[1];
110         Temp_Value[0] = ~Temp_Value[0] + 1;
111         if (Temp_Value[0] == 0x00) Temp_Value[1]++;
112         //负数标识置为 1
113         ng = 1;
114     }
115     //查表得到温度小数部分
116     Display_Digit[0] = df_Table[Temp_Value[0] & 0x0F];
117     //-----不使用查表法获取温度小数部分时可使用下面的代码-----
118     //for (i = 0; i<4; i++)
119     //{
120         //根据低 4 位是否为 1，分别累加：1.0/2, 1.0/4, 1.0/8, 1.0/16
121         //if (Temp_Value[0] & _BV(i)) Temp_Df += 1.0 / _BV(4 - i);
122     //}
123     //Display_Digit[0] = (INT8U)((Temp_Df + 0.05) * 10); //第 2 位小数四舍五入
124
125     //获取温度整数部分(无符号)
126     CurrentT = (Temp_Value[0]>>4)|(Temp_Value[1]<<4);
127
128     //将整数部分分解为 3 位待显示数字
129     Display_Digit[3] = CurrentT / 100;
130     Display_Digit[2] = CurrentT % 100 / 10;
131     Display_Digit[1] = CurrentT % 10;

```

```

132 //刷新 LCD 显示缓冲
133 Current_Temp_Display_Buffer[11] = Display_Digit[0] + '0';
134 Current_Temp_Display_Buffer[10] = '.';
135 Current_Temp_Display_Buffer[9] = Display_Digit[1] + '0';
136 Current_Temp_Display_Buffer[8] = Display_Digit[2] + '0';
137 Current_Temp_Display_Buffer[7] = Display_Digit[3] + '0';
138
139 //高位为 0 时不显示
140 if (Display_Digit[3] == 0) Current_Temp_Display_Buffer[7] = '';
141 //高位为 0 且次高位为 0 时,次高位不显示
142 if (Display_Digit[2] == 0 && Display_Digit[3] == 0)
143 Current_Temp_Display_Buffer[8] = '';
144
145 //负数符号显示在恰当位置
146 if (ng)
147 {
148     if (Current_Temp_Display_Buffer[8] == '-')
149         Current_Temp_Display_Buffer[8] = '--';
150     else
151         if (Current_Temp_Display_Buffer[7] == '-')
152             Current_Temp_Display_Buffer[7] = '--';
153     else
154         Current_Temp_Display_Buffer[6] = '--';
155 }
156 }

```

```

001 //----- LCD1602.c -----
002 // 名称: 液晶控制与显示程序
003 //-----
004 #include <avr/io.h>
005 #include <util/delay.h>
006 #define INT8U unsigned char
007 #define INT16U unsigned int
008
009 //LCD 控制引脚定义
010 #define RS_1() PORTD |= (1<<PD0)
011 #define RS_0() PORTD &= ~(1<<PD0)
012 #define RW_1() PORTD |= (1<<PD1)
013 #define RW_0() PORTD &= ~(1<<PD1)
014 #define EN_1() PORTD |= (1<<PD2)
015 #define EN_0() PORTD &= ~(1<<PD2)
016
017 //LCD 端口定义

```

```

018 #define LCD_PORT  PORTC           //发送 LCD 数据端口
019 #define LCD_PIN   PINC           //接收 LCD 数据端口
020 #define LCD_DDR   DDRC           //LCD 端口方向数据
021 //-----
022 // LCD 忙等待
023 //-----
024 void LCD_BUSY_WAIT()
025 {
026     RS_0();  RW_1();           //读状态寄存器
027     LCD_DDR = 0x00;          //将端口设为输入
028     EN_1(); _delay_us(10);
029     loop_until_bit_is_clear(LCD_PIN,7); //LCD 忙等待,直到 LCD_PIN 最高位为 0
030     EN_0();
031     LCD_DDR = 0xFF;          //还原 LCD 端口为输出
032 }
033
034 //-----
035 // 写 LCD 命令寄存器
036 //-----
037 void Write_LCD_Command(INT8U cmd)
038 {
039     LCD_BUSY_WAIT();
040     RS_0();  RW_0();           //写命令寄存器
041     LCD_PORT = cmd;          //发送命令
042     EN_1();  EN_0();          //写入
043 }
044
045 //-----
046 // 写 LCD 数据寄存器
047 //-----
048 void Write_LCD_Data(INT8U dat)
049 {
050     LCD_BUSY_WAIT();
051     RS_1();  RW_0();           //写数据寄存器
052     LCD_PORT = dat;          //发送数据
053     EN_1();  EN_0();          //写入
054 }
055
056 //-----
057 // 自定义字符写 CGRAM
058 //-----
059 void Write_NEW_LCD_Char()
060 {

```



```
061     INT8U i;
062     const INT8U Temperature_Char[8] =          //自定义温度符号点阵
063     { 0x06,0x09,0x09,0x06,0x00,0x00,0x00,0x00 };
064     Write_LCD_Command(0x40);                  //从 CGRAM 地址 0x40 开始写入
065     for (i = 0; i < 8; i++)                   //8 字节点阵写入 CGRAM 0x40~0x47
066     Write_LCD_Data(Temperature_Char[i]);
067 }
068
069 //-----
070 // LCD 初始化
071 //-----
072 void Initialize_LCD()
073 {
074     Write_LCD_Command(0x38); _delay_ms(15); //置功能,8 位,双行,5 * 7
075     Write_LCD_Command(0x01); _delay_ms(15); //清屏
076     Write_LCD_Command(0x06); _delay_ms(15); //字符进入模式:屏幕不动,字符后移
077     Write_LCD_Command(0x0C); _delay_ms(15); //显示开,光标
078     Write_NEW_LCD_Char();                  //将℃ 中的度(°)的点阵数据写入 CGRAM
079 }
080
081 //-----
082 // 设置显示位置
083 //-----
084 void Set_LCD_POS(INT8U x, INT8U y)
085 {
086     //设置显示起始位置
087     if(y == 0) Write_LCD_Command(0x80 | x); else
088     if(y == 1) Write_LCD_Command(0xC0 | x);
089 }
090
091 //-----
092 // 显示字符串
093 //-----
094 void LCD_ShowString(INT8U x, INT8U y,char * str)
095 {
096     INT8U i = 0;
097     //设置显示起始位置
098     Set_LCD_POS(x, y);
099     //输出字符串
100     for (i = 0; i < 16 && str[i]!='\0';i++)
101     Write_LCD_Data(str[i]);
102 }
```

## 4.29 SPI 接口温度传感器 TC72 应用测试

Microchip 公司生产的温度传感器 TC72 兼容 SPI 接口, 温度测量范围为  $-55 \sim +125^{\circ}\text{C}$ 。使用 TC72 时不需要附加任何外部电路, 它可以工作于连续的温度转换模式(Continuous Conversion mode)或单次转换模式(One-Shot mode)。在连续转换模式下, TC72 约每隔 150 ms 进行一次温度转换, 并将获取的数据保存于温度寄存器中, 后者在一次转换后即进入省电模式。本例电路及部分运行效果如图 4-42 所示。

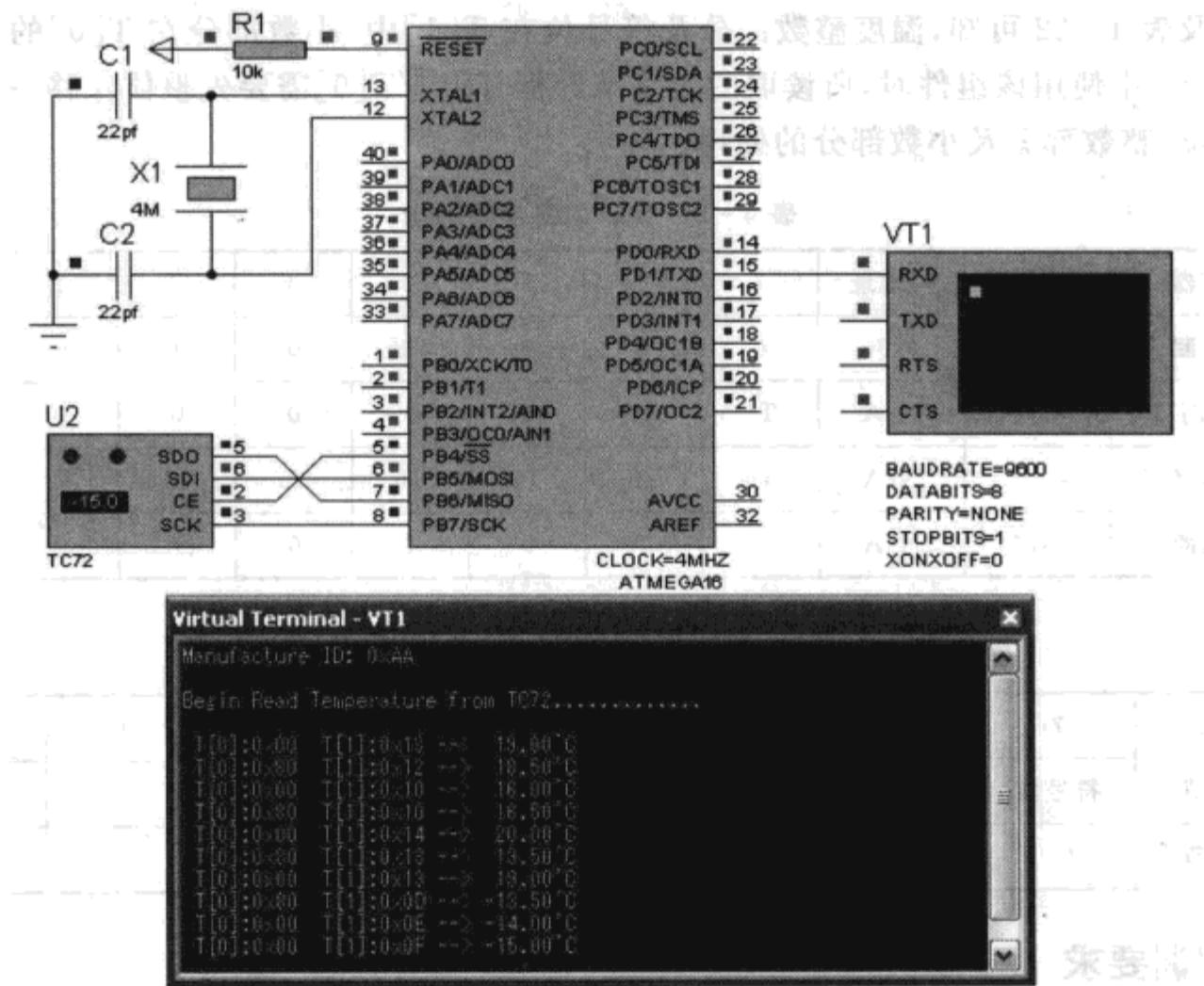


图 4-42 SPI 接口温度传感器 TC72 应用测试

### 1. 程序设计与调试

由于 TC72 兼容 SPI 接口, 此前案例中对 SPI 接口器件进行操作的程序在本例中同样适用, 编写本例程序时, 只需要弄清 TC72 的寄存器地址及温度寄存器数据格式即可。

根据表 4-21, 本例对 TC72 寄存器地址给出如下定义:

```

#define TC72_CTRL      0x80    //控制寄存器
#define TC72_TEMP_LSB  0x01    //温度低字节
#define TC72_TEMP_MSB  0x02    //温度高字节
#define TC72_MANU_ID   0x03    //制造商 ID

```

为将 TC72 配置为单次检测与关断省电模式, 程序设置 TC72\_CTRL 为 0x15, 即 00010101。

为读取温度寄存器数据, 程序访问寄存器地址 0x01 与 0x02, 即 TC72\_TEMP\_LSB 与 TC72\_TEMP\_MSB, 分别读取温度低字节和温度高字节。相关语句如下:

```

SPI_Transmit(TC72_TEMP_LSB); //发送读温度低字节命令
T[0] = SPI_Transmit(0xFF); //读 LSB
SPI_Transmit(TC72_TEMP_MSB); //发送读温度高字节命令
T[1] = SPI_Transmit(0xFF); //读 MSB

```

上述语句使用的是单字节读取的方法，本例代码中还添加了连续读取 2 字节温度数据的方法，阅读分析时可参考 TC72 的技术手册。

将 2 字节温度数据读取到 T[0]与 T[1]后，还需要将这 2 字节转换为温度数据显示。本例函数 Convert\_Temperature 给出了完整的转换代码及详细说明。需要说明的是，根据 TC72 技术手册及表 4-22 可知，温度整数部分及符号位在 T[1]中，小数部分在 T[0]的高 2 位中，但在 Proteus 中使用该组件时，所读取的 2 字节数据 T[1]/T[0]需要先整体左移一位，然后再进行符号位、整数部分及小数部分的处理。

表 4-21 TC72 寄存器地址

寄存器	读地址	写地址	7	6	5	4	3	2	1	0
控制	0x00	0x80	0	0	0	单次	0	1	0	关断
温度低字节	0x01	N/A	T1	T0	0	0	0	0	0	0
温度高字节	0x02	N/A	T9	T8	T7	T6	T5	T4	T3	T2
制造商	0x03	N/A	0	1	0	1	0	1	0	0

表 4-22 TC72 温度传感器的 2 字节温度数据寄存器格式

数位	7	6	5	4	3	2	1	0
高字节	符号位	$2^6$	$2^5$	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$
低字节	$2^{-1}$	$2^{-2}$	0	0	0	0	0	0

## 2. 实训要求

- ① 修改本例代码，将 TC72 设置为工作于连续转换模式。
- ② 重新设计本例，用数码管显示所读取的温度数据。

## 3. 源程序代码

```

001 //----- SPI 接口温度传感 TC72 应用测试.c -----
002 // 名称：SPI 接口温度传感 TC72 应用测试
003 //-
004 // 说明：本例运行时，单片机将持续从 TC72 传感器读取温度数据并转换为
005 // 十进制温度值送串口显示
006 //
007 //-
008 #define F_CPU 4000000UL
009 #include <avr/io.h>
010 #include <util/delay.h>

```

```

011 # include <stdio.h>
012 # include <math.h>
013 # define INT8U unsigned char
014 # define INT16U unsigned int
015
016 //串口相关函数
017 extern void Init_USART();
018 extern void PutChar(char c);
019 extern void PutStr(char * s);
020
021 //SPI 使能与禁用(注意 TC72 是高电平使能,低电平禁用)
022 # define SPI_EN() PORTB |= _BV(PB4)
023 # define SPI_DI() PORTB &= ~_BV(PB4)
024
025 //TC72 寄存器地址定义
026 # define TC72_CTRL 0x80 //控制寄存器
027 # define TC72_TEMP_LSB 0x01 //温度低字节
028 # define TC72_TEMP_MSB 0x02 //温度高字节
029 # define TC72_MANU_ID 0x03 //制造商 ID
030 //-----
031 // SPI 主机初始化
032 //-----
033 void SPI_MasterInit()
034 {
035     //PB4、PB5、PB7(SS,MOSI,SCK)为输出,PB6 为输入(MISO)
036     DDRB = 0B10110000; PORTB = 0xFF;
037     //SPI 使能,主机模式,16 分频
038     SPCR |= _BV(SPE) | _BV(MSTR) | _BV(SPR0);
039 }
040
041 //-----
042 // SPI 数据传输
043 //-----
044 INT8U SPI_Transmit(INT8U dat)
045 {
046     SPDR = dat;                                //启动数据传输
047     while(!(SPSR & _BV(SPIF)));                //等待结束
048     SPSR |= _BV(SPIF);                          //清中断标志
049     return SPDR;
050 }
051
052 //-----
053 // 向 TC72 写入 2 字节(地址,数据)

```



```
054 //-----  
055 void Write_TC72_Aaddr_Dat(INT8U addr, INT8U dat)  
056 {  
057     SPI_EN();  
058     SPI_Transmit(addr);  
059     SPI_Transmit(dat);  
060     SPI_DI();  
061 }  
062  
063 //-----  
064 // 写 TC72 配置数据  
065 //-----  
066 void Config_TC72()  
067 {  
068     //配置为单次转换与关断模式  
069     Write_TC72_Aaddr_Dat(TC72_CTRL,0x15);  
070 }  
071  
072 //-----  
073 // 从 TC72 读取 2 字节温度数据  
074 //-----  
075 void Read_TC72_Temperature(INT8U T[])  
076 {  
077     Config_TC72(); _delay_ms(200);  
078     SPI_EN();  
079     SPI_Transmit(TC72_TEMP_MSB);           //发送读温度高字节命令  
080     //连续读取 2 字节(连续读取时先得到的是高字节,后得到的是低字节)  
081     T[1] = SPI_Transmit(0xFF);           //读高字节  
082     T[0] = SPI_Transmit(0xFF);           //读低字节  
083     SPI_DI();  
084  
085     //还可以使用以下单字节读取的方法  
086     //SPI_EN();  
087     //SPI_Transmit(TC72_TEMP_LSB);        //发送读温度低字节命令  
088     //T[0] = SPI_Transmit(0xFF);          //读 LSB  
089     //SPI_DI();  
090     //SPI_EN();  
091     //SPI_Transmit(TC72_TEMP_MSB);        //发送读温度高字节命令  
092     //T[1] = SPI_Transmit(0xFF);          //读 MSB  
093     //SPI_DI();  
094 }  
095  
096 //-----
```

```

097 // 读 TC72 制造商 ID
098 //-----
099 INT8U Read_Manufacture_ID()
100 {
101     INT8U d;
102     SPI_EN();
103     SPI_Transmit(TC72_MANU_ID);           //发送读制造商 ID 命令
104     d = SPI_Transmit(0xFF);               //读取 MID
105     SPI_DI();
106     return d;
107 }
108
109 //-----
110 // 将读取的 2 字节温度数据转换为十进制温度
111 //-----
112 float Convert_Temperature(INT8U T[])
113 {
114     float TempX = 0.0;                   //浮点温度值
115     int Sign = 1;                      //正负标识,默认为正数
116     //将 16 位的两字节数据 T[1] T[0]整体左移 1 位
117     //根据 TC72 技术手册,T[1]与 T[0]不需要整体左移
118     //但 Proteus7.5 版的 TC72 器件读取的两字节温度数据
119     //需要左移 1 位后再进行转换
120     T[1] <<= 1;
121     if (T[0] & 0x80) T[1] |= 0x01;
122     T[0] <<= 1;
123     //如果高位为 1 则为负,将两字节取反加 1
124     if (T[1] & 0x80)
125     {
126         T[1] = ~T[1];                  //T[1]取反
127         T[0] = ~T[0] + 0x01;          //T[0]取反加 1
128         if (T[0] == 0x00) T[1] += 0x01; //如果 T[0]取反加 1 后为 0x00 则进位
129         Sign = -1;                  //设"- "符号
130     }
131     //得到整数部分
132     TempX = (float)T[1];
133     //分别累加两位小数部分
134     if (T[0] & 0x80) TempX += 0.5;
135     if (T[0] & 0x40) TempX += 0.25;
136     //附加符号位并返回温度
137     return Sign * TempX;
138 }
139

```



```
140 //-----  
141 // 主程序  
142 //-----  
143 int main()  
144 {  
145     INT8U Temp[2];  
146     int d,f;  
147     float Current_Temp = 0.00,Pre_Temp = 0.00;  
148     char DisplayBuffer[50];  
149     DDRD = 0xFF;                                //配置端口  
150     Init_USART();  
151     SPI_MasterInit();                          //SPI 主机初始化  
152     Config_TC72();  
153     _delay_ms(300);  
154     //读取并显示制造商 ID  
155     sprintf(DisplayBuffer,"Manufacture ID: 0x % 02X\r\n",Read_Manufacture_ID());  
156     PutStr(DisplayBuffer);  
157     //提示开始读取温度  
158     PutStr("Begin Read Temperature from TC72.....\n\n");  
159     while(1)  
160     {  
161         //第 1 次读取温度  
162         Read_TC72_Temperature(Temp);  
163         //格式转换  
164         Current_Temp = Convert_Temperature(Temp);  
165         //第 2 次读取温度  
166         Read_TC72_Temperature(Temp);  
167  
168         //如果连续 2 次获取的温度不一致则继续  
169         if (Current_Temp != Convert_Temperature(Temp)) continue;  
170         //如果温度未变化则继续  
171         if (Current_Temp == Pre_Temp) continue;  
172  
173         Pre_Temp = Current_Temp;  
174         //生成待显示字符串(由于 GCC 的 sprintf 不支持 %f,因此需要转换)  
175         d = (int)Current_Temp;                      //整数部分  
176         f = (int)(fabs(Current_Temp - d) * 100); //小数部分  
177         sprintf(DisplayBuffer, " T[0]:0x % 02X  T[1]:0x % 02X --> % 3d. % 02d'C\n",  
178             Temp[0],Temp[1],d,f);  
179         //串口输出温度转换结果  
180         PutStr(DisplayBuffer);  
181     }  
182 }
```

```

01 //----- usart.c -----
02 // 名称：串口程序
03 //-----
04 #define F_CPU 4000000UL           //4 MHz 晶振
05 #include <avr/io.h>
06 #include <util/delay.h>
07 #define INT8U unsigned char
08 #define INT16U unsigned int
09
10 //-----
11 // USART 初始化
12 //-----
13 void Init_USART()
14 {
15     UCSRB = _BV(RXEN) | _BV(TXEN) | _BV(RXCIE); //允许接收和发送,接收中断使能
16     UCSRC = _BV(URSEL) | _BV(UCSZ1) | _BV(UCSZ0); //8 位数据位,1 位停止位
17     UBRRH = (F_CPU / 9600 / 16 - 1) % 256;        //波特率:9600
18     UBRRL = (F_CPU / 9600 / 16 - 1) / 256;
19 }
20
21 //-----
22 // 发送 1 个字符
23 //-----
24 void PutChar(char c)
25 {
26     if(c == '\n') PutChar('\r');
27     UDR = c;
28     while(!(UCSRA & _BV(UDRE)));
29 }
30
31 //-----
32 // 发送字符串
33 //-----
34 void PutStr(char * s)
35 {
36     while (*s) PutChar(*s++);
37 }

```

## 4.30 SHT75 温、湿度传感器测试

SHT75 是瑞士 SENSIRION 生产的一种高度集成的温、湿度传感器，具有 14 位的温度和 12 位的湿度全量程标定数字输出。传感器包含 1 个电容性聚合体相对湿度传感器和 1 个带



隙(bandgap)温度传感器,14 位 A/D 转换器以及 1 个 2-Wires 式串行接口电路。湿度在 0~100%RH 范围内能达到士 1.8% 的高精度,温度能在 25 ℃时把误差控制在士 0.3 ℃的范围内。SHT75 工作电压为 2.4~5.5 V,体积小、功耗低,使用电池供电可以长期稳定运行,防浸泡特性使其在高湿环境下也能长期正常工作,它是各类温湿度测量系统应用设计的首选传感器。本例电路及部分运行效果如图 4-43 所示。

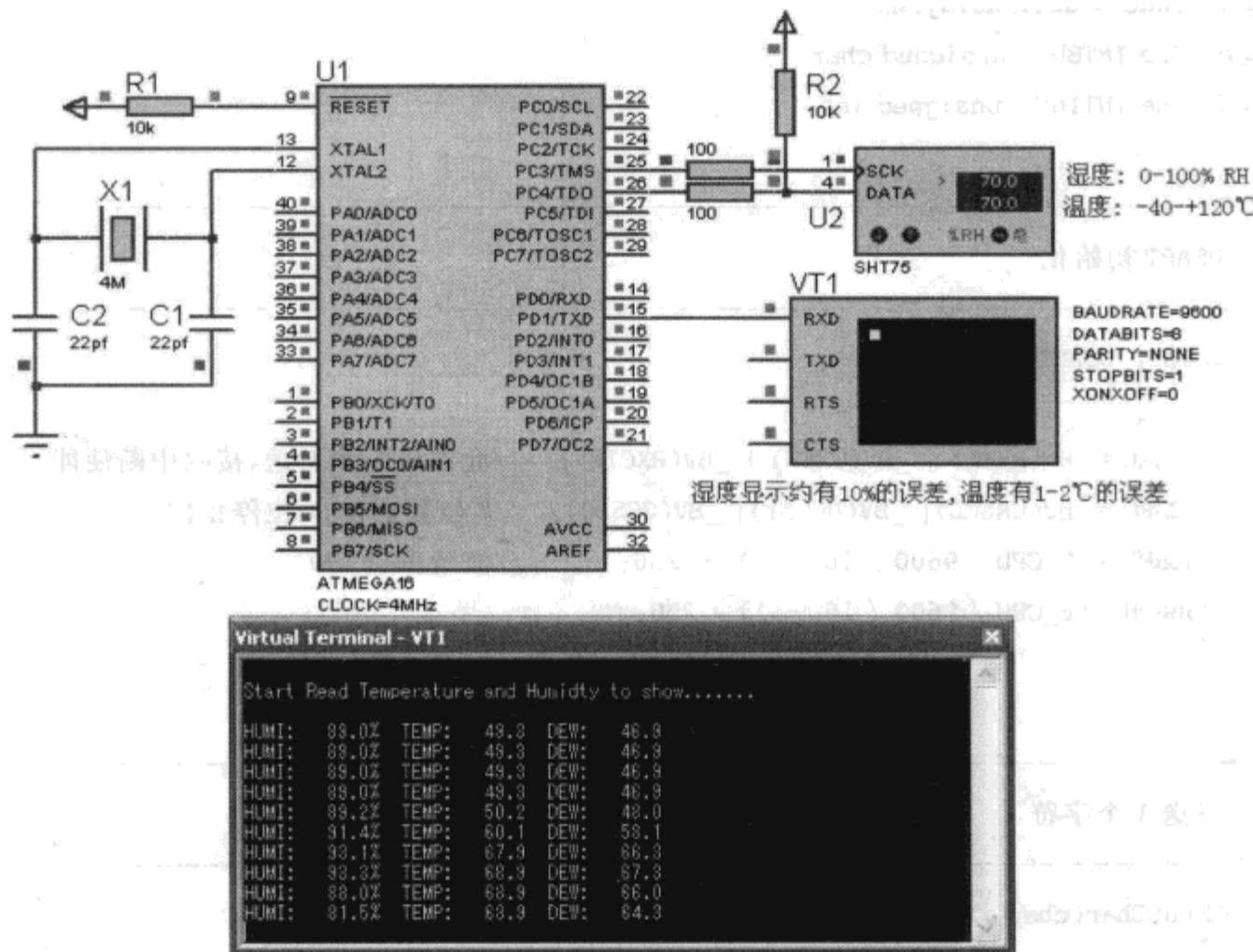


图 4-43 SHT75 温、湿度传感器测试

## 1. 程序设计与调试

空气湿度(Humidity)有绝对湿度和相对湿度之分,日常生活中所指的湿度为相对湿度 RH (Relative Humidity),单位为%RH,空气湿度可通俗地理解为空气的潮湿程度。下面对有关湿度的概念作简单要说明:

绝对湿度——空气的湿度可以用空气中所含水蒸汽的密度,即单位体积的空气中所含水蒸汽的质量来表示。由于直接测量空气中水蒸汽的密度比较困难,而水蒸汽的压强随水蒸汽密度的增大而增大,所以通常用空气中水蒸汽的压强  $p$  来表示空气的绝对湿度。

相对湿度——相对湿度的概念用于表示空气中的水蒸汽离饱和状态的远近程度,某温度时空气的绝对湿度  $p$  与同一温度下水的饱和汽压  $p_s$  的百分比称为此时空气的相对湿度,不同温度下水的饱和汽压可以查表得到,在绝对湿度  $p$  不变而降低温度时,水的饱和汽压减小使空气的相对湿度增大。

本例中还有一个概念是露点(Dew Point),它是指水蒸汽凝结开始出现时的温度(也就是空气达到饱和的温度)。

SHT75 的分辨率可以根据对现场的采集速率进行调整,一般情况下默认的测量分辨率分

别为 14-bit(温度)、12-bit(湿度)。在高速采集时可通过状态寄存器将其分别降至 12-bit 和 8-bit, 它对温度的测量范围为: -40~123.8 °C, 对湿度的测量范围为: 0~100%RH。

为 SHT75 传感器编写程序时, 需要参考 SHT75 的命令表及操作时序等。

本例的 SHT75 传感器命令集参考表 4-23 进行定义。

表 4-23 SHT75 命令表

功 能	命 令			
	地 址	命 令	读/写	十六进制命令字节
测 量 温 度	000	0001	1	0x03
测 量 湿 度	000	0010	1	0x05
读 状态 寄 存 器	000	0011	1	0x07
写 状态 寄 存 器	000	0011	0	0x06
软 件 复 位, 复 位 接 口, 将 状 态 寄 存 器 清 为默 认 值, 在 执 行 下 一 命 令 前 等 待 11 ms	000	1111	0	0x1E

图 4-44 是传感器连接复位时序, 传感器连接复位函数 s\_ConnectionReset()根据该时序图编写。

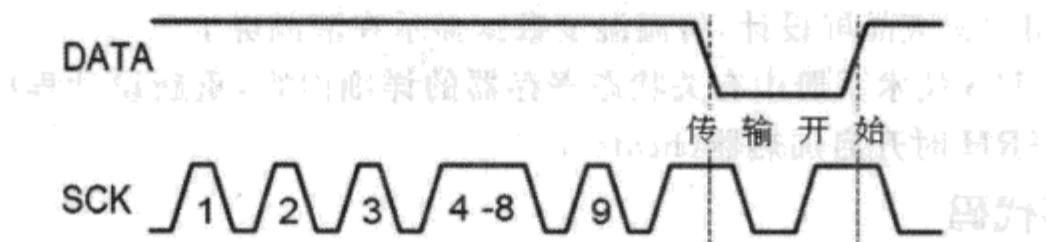


图 4-44 SHT75 连接复位时序

图 4-45 是 SHT75 的状态寄存器写时序(左)与读时序(右), 写状态寄存器函数 s\_Write\_StatusReg(INT8U \* p\_value)与读状态寄存器函数 s\_Read\_StatusReg(INT8U \* p\_value, INT8U \* p\_checksum)分别根据该时序图编写。状态寄存器的最低位为精度选择位, 取 0 时表示 14 位温度精度与 12 位湿度精度, 取 1 时为 12 位温度精度与 8 位湿度精度。默认值为 0。

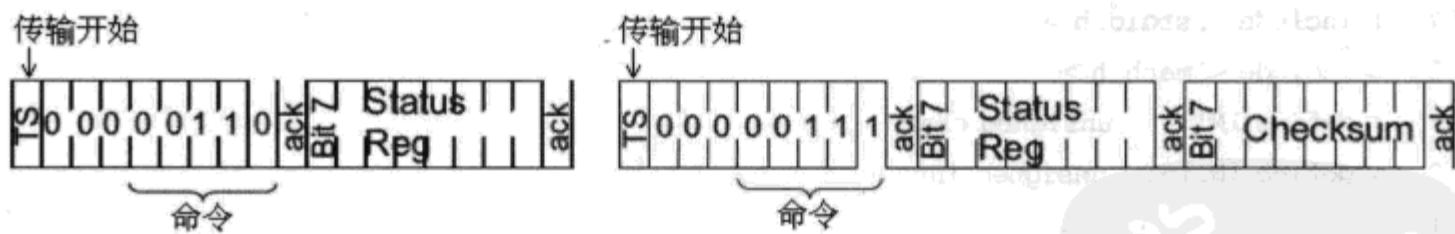


图 4-45 SHT75 状态寄存器写时序(左)与读时序(右)

图 4-46 是 SHT75 温湿度数据测量时序, 函数 INT8U s\_Measure(INT8U \* p\_value, INT8U \* p\_checksum, INT8U mode)根据该时序图编写。

在计算温度与湿度值时, 可参考 SHT75 技术手册中第 3 部分: 转换输出到物理值(Converting Output to Physical Values), 该部分给了表 Table6~8 及相应的计算公式(即表 4-24), 其中  $SO_{RH}$  为传感器输出的相对湿度,  $SO_T$  为传感器输出温度。最后的结果由表 4-24 后面的公式计算。本例计算温湿度的函数 Calc\_SHT75(float \* p\_humidity, float \* p\_temperature)即根据该表格系数与相应公式编写。

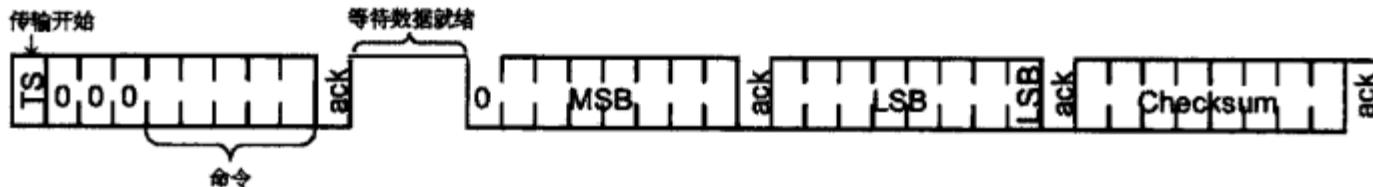


图 4-46 SHT75 温湿度数据测量时序

表 4-24 温湿度转换系数与计算公式

湿度转换系数				温度补偿系数			温度转换系数	
SO <sub>RH</sub>	c1	c2	c3	SO <sub>T</sub>	t1	t2	d1(@5V)	d2
12bit	-4.0	0.0405	$-2.8 \times 10^{-6}$	14-bit	0.01	0.00008	-40	0.01
8bit	-4.0	0.648	$-7.2 \times 10^{-4}$	12-bit	0.01	0.00128	-40	0.04

计算公式:  $RH_{linear} = c1 + c2 \cdot SO_{RH} + c3 \cdot SO_{RH}^2$   
 $RH_{true} = (T_c - 25) \cdot (t1 + t2 \cdot SO_{RH}) + RH_{linear}$   
 $Temperature = d1 + d2 \cdot SO_T \quad (@5 V)$

## 2. 实训要求

- ① 重新修改本例,用数码管分别显示温湿度数据。
- ② 重新改用中文液晶屏设计,将温湿度数据显示在液晶屏上。
- ③ 阅读 SHT75 技术手册中有关状态寄存器的详细内容,重新设计程序使系统能够在相对湿度大于 95%RH 时开启加热器(heater)。

## 3. 源程序代码

```

001 //----- SHT75.c -----
002 // 名称: SHT75 传感器程序(参照 SENSIRION 公司提供的 8051 版代码改编)
003 //-----
004 #define F_CPU 4000000UL
005 #include <avr/io.h>
006 #include <util/delay.h>
007 #include <stdio.h>
008 #include <math.h>
009 #define INT8U unsigned char
010 #define INT16U unsigned int
011
012 //传感器引脚定义
013 #define SCL PC3
014 #define SDA PC4
015
016 //传感器引脚操作定义
017 #define SCL_0() PORTC &= ~_BV(SCL) //串行时钟
018 #define SCL_1() PORTC |= _BV(SCL)
019 #define SDA_0() PORTC &= ~_BV(SDA) //串行数据
020 #define SDA_1() PORTC |= _BV(SDA)
021 #define SDA_DDR_0() DDRC &= ~_BV(SDA) //SDA 数据方向

```

```

022 #define SDA_DDR_1() DDRC |= _BV(SDA)
023 #define Get_SDA_Bit() (PINC & _BV(SDA)) //获取 SDA 引脚数据(保留括号)
024
025 //SHT75 传感器命令集                                //地址 命令 读/写
026 #define MEASURE_TEMP 0x03                          //000 0001 1
027 #define MEASURE_HUMI 0x05                          //000 0010 1
028 #define STATUS_REG_W 0x06                         //000 0011 0
029 #define STATUS_REG_R 0x07                         //000 0011 1
030 #define RESET          0x1E                         //000 1111 0
031
032 //是否应答
033 #define NACK 0
034 #define ACK 1
035
036 //温湿度信息显示缓冲
037 char HT_Display_Buffer[20];
038 //定义温度与湿度符号
039 enum {TEMP,HUMI};
040 extern void PutStr(char * s);
041 //-----
042 // 写 1 字节到 SHT75 并检查应答
043 //-----
044 INT8U s_Write_Byte(INT8U dat)
045 {
046     INT8U i,error = 0;
047     SDA_DDR_1();
048     for (i = 0x80; i > 0; i >>= 1)           //从字节高位开始向 SDA 写入 8 位
049     {
050         if (i & dat) SDA_1(); else SDA_0();
051         SCL_1(); _delay_us(5); SCL_0();      //模拟传感器总线约 5 μs 脉宽时钟
052     }
053     SDA_1();                                //释放数据线
054     SDA_DDR_0();                            //SDA 设为读
055     SCL_1();
056     error = Get_SDA_Bit() ? 1 : 0;          //正常时 SDA 将被 SHT75 拉低
057     SCL_0();                                //无应答时返回 1
058     return error;
059 }
060
061 //-----
062 // 从传感器读 1 字节,在 ack = 1 时发送应答
063 //-----
064 INT8U s_Read_Byte(INT8U ack)
065 {
066     INT8U i,val = 0x00;

```



```
067     SDA_DDR_1();                                //SDA 方向设为写
068     SDA_1();                                    //拉高 SDA 释放数据线
069     SDA_DDR_0();                                //SDA 方向设为读
070     for (i = 0x80; i > 0; i >>= 1)           //读取 8 位,先读取的为高位
071     {
072         SCL_1();                                //模拟总线时钟
073         if (Get_SDA_Bit()) val |= i;            //读取 1 位
074         SCL_0();
075     }
076     SDA_DDR_1();
077     if (!ack) SDA_1(); else SDA_0();             //根据参数 ack 拉高或拉低 SDA
078     SCL_1();                                    //第 9 个时钟读取应答
079     _delay_us(5);                             //脉宽约 5 μs
080     SCL_0();
081     SDA_1();                                    //释放数据线
082     return val;
083 }
084
085 //-----
086 // 传输开始
087 //-----
088 void s_TransStart()
089 {
090     SDA_1();
091     SCL_0();    _delay_us(1);
092     SCL_1();    _delay_us(1);
093     SDA_0();    _delay_us(1);
094     SCL_0();    _delay_us(3);
095     SCL_1();    _delay_us(1);
096     SDA_1();    _delay_us(1);
097     SCL_0();
098 }
099
100 //-----
101 // 传感器连接复位
102 //-----
103 void s_ConnectionReset()
104 {
105     INT8U i;
106     SDA_1(); SCL_0();                          //初始状态
107     for(i = 0; i<9; i++)                     //模拟 9 个时钟周期
108     {
109         SCL_1(); SCL_0();
110     }
111     s_TransStart();                           //传输开始
```

```

112 }
113
114 //-----
115 // 传感器软复位
116 //-----
117 INT8U s_SoftReset()
118 {
119     INT8U error = 0;
120     s_ConnectionReset();           //连接通信复位
121     error += s_Write_Byte(RESET); //向传感器发送复位命令
122     return error;               //传感器无响应时返回 1
123 }
124
125 //-----
126 // 读状态寄存器
127 //-----
128 INT8U s_Read_StatusReg(INT8U * p_value, INT8U * p_checksum)
129 {
130     INT8U error = 0;
131     s_TransStart();             //传输开始
132     error = s_Write_Byte(STATUS_REG_R); //向传感器发送命令 STATUS_REG_R
133     * p_value = s_Read_Byte(ACK);   //读状态寄存器(8 位)
134     * p_checksum = s_Read_Byte(NACK); //读取校验和(8 位)
135     return error;                //传感器无响应时返回 1
136 }
137
138 //-----
139 // 写状态寄存器
140 //-----
141 INT8U s_Write_StatusReg(INT8U * p_value)
142 {
143     INT8U error = 0;
144     s_TransStart();             //传输开始
145     error += s_Write_Byte(STATUS_REG_W); //向传感器发送命令 STATUS_REG_W
146     error += s_Write_Byte(* p_value); //发送状态寄存器的值
147     return error;                //传感器无响应时返回 1
148 }
149
150 //-----
151 // 带检验码的温度与湿度测量
152 //-----
153 INT8U s_Measure(INT8U * p_value, INT8U * p_checksum, INT8U mode)
154 {
155     INT8U i = 0, error = 0;
156     s_TransStart();             //传输开始

```



```
157     switch(mode)                                //向传感器发送命令
158     {
159         case TEMP : error += s_Write_Byte(MEASURE_TEMP); break;
160         case HUMI : error += s_Write_Byte(MEASURE_HUMI); break;
161         default   : break;
162     }
163     SDA_DDR_0();
164     while (Get_SDA_Bit() != 0x00 && ++ i<40) _delay_ms(100);
165     if(Get_SDA_Bit()) error += 1;                  //超时
166     * (p_value)      = s_Read_Byte(ACK);          //读第 1 字节(MSB)
167     * (p_value + 1) = s_Read_Byte(ACK);          //读第 2 字节(LSB)
168     * p_checksum    = s_Read_Byte(NACK);          //读检验和
169     return error;
170 }
171
172 //-----
173 // 计算温湿度
174 //-----
175 void Calc_STH75(float * p_humidity ,float * p_temperature)
176 {
177     const float C1 = - 4.0;                      //12 位,系数 C1
178     const float C2 = + 0.0405;                     //12 位,系数 C2
179     const float C3 = - 0.0000028;                  //12 位,系数 C3
180     const float T1 = + 0.01;                      //14 位 @ 5 V ,系数 T1
181     const float T2 = + 0.00008;                    //14 位 @ 5 V ,系数 T2
182     float rh= * p_humidity;                      // rh:      湿度 12-Bit
183     float t   = * p_temperature;                 // t:       温度 14-Bit
184     float rh_lin;                               // rh_lin:  线性湿度
185     float rh_true;                             // rh_true: 温度补偿湿度
186     float t_C;                                 // t_C :    温度(℃)
187     t_C=t * 0.01 - 40;                         //计算温度
188     rh_lin=C3 * rh * rh+C2 * rh+C1;           //计算湿度
189     rh_true=(t_C - 25) * (T1 + T2 * rh) + rh_lin;//计算:温度补偿湿度
190     if(rh_true > 100) rh_true = 100;            //将湿度数据限制在正常范围之内
191     if(rh_true<0.1) rh_true = 0.1;              //即 0.1 % ~ 100 %
192     * p_temperature = t_C;                      //返回温度[℃]
193     * p_humidity = rh_true;                     //返回湿度[% RH]
194 }
195
196 //-----
197 // 根据输入的湿度与温度计算露点
198 //-----
199 float Calc_Dew_point(float h,float t)
200 {
201     float logEx,dew_point;
```

```

202     logEx = 0.66077 + 7.5 * t / (237.3 + t) + (log10(h) - 2);
203     dew_point = (logEx - 0.66077) * 237.3 / (0.66077 + 7.5 - logEx);
204     return dew_point;
205 }
206
207 //-----
208 // 传感器测试(读取湿度与温度数据并进行转换计算,送虚拟终端显示)
209 //-----
210 void Temp_and_Humi_Sensors_Test()
211 {
212     INT8U a[2], b[2];           //读传感器湿度与温度,各两字节
213     float x, y, d;             //计算转换后的湿度、温度、露点
214     INT8U error, checksum;      //错误及校验和
215     s_ConnectionReset();       //连接复位
216     while(1)
217     {
218         error = 0;
219         error += s_Measure((INT8U *)a, &checksum, HUMI); //测量湿度
220         error += s_Measure((INT8U *)b, &checksum, TEMP); //测量温度
221         if(error != 0) s_ConnectionReset(); //出错时传感器连接复位
222         else
223         {
224             x = (float)(a[0] * 256 + a[1]); //将两字节温度转换为 float 类型
225             y = (float)(b[0] * 256 + b[1]); //将两字节湿度转换为 float 类型
226             Calc_STH75(&x, &y);          //计算湿度与温度
227             d = Calc_Dew_point(x, y);    //计算露点
228
229             //生成指定格式的待输出字符串(GCC 中 sprintf 不支持 %f 输出,因此需要转换)
230             sprintf(HT_Display_Buffer, "HUMI: %5d. %1d % % TEMP: %5d. %1d DEW: %5d. %1d\n",
231                     (int)x, (int)(fabs(x - (int)x) * 10),
232                     (int)y, (int)(fabs(y - (int)y) * 10),
233                     (int)d, (int)(fabs(d - (int)d) * 10));
234
235             PutStr(HT_Display_Buffer); //向虚拟终端输出结果
236         }
237         _delay_ms(800); //延时近 0.8 s,以免传感器过热
238     }
239 }

01 //----- uart.c -----
02 // 名称:串口程序
03 //-----
04 #define F_CPU 4000000UL
05 #include <avr/io.h>
06 #include <util/delay.h>

```



```
07 #define INT8U unsigned char
08 #define INT16U unsigned int
09 //---
10 // USART 初始化
11 //---
12 void Init_USART()
13 {
14     UCSRB = _BV(RXEN) | _BV(TXEN) | _BV(RXCIE);      //允许接收和发送,接收中断使能
15     UCSRC = _BV(URSEL) | _BV(UCSZ1) | _BV(UCSZ0);    //8 位数据位,1 位停止位
16     UBRRH = (F_CPU / 9600 / 16 - 1) % 256;          //波特率:9600
17     UBRRL = (F_CPU / 9600 / 16 - 1) / 256;
18 }
19
20 //---
21 // 发送 1 个字符
22 //---
23 void PutChar(char c)
24 {
25     if(c == '\n') PutChar('\r');
26     UDR = c;
27     while(!(UCSRA & _BV(UDRE)));
28 }
29
30 //---
31 // 发送字符串
32 //---
33 void PutStr(char * s)
34 {
35     while (*s) PutChar(*s++);
36 }

01 //----- main.c -----
02 // 名称: SHT75 温湿度传感器测试
03 //---
04 // 说明: 本例运行时,虚拟终端中将持续刷新显示温、湿度信息
05 //
06 //---
07 #define F_CPU 4000000UL
08 #include <avr/io.h>
09 #include <util/delay.h>
10 #include <stdio.h>
11 #define INT8U unsigned char
12 #define INT16U unsigned int
13 extern void Init_USART();
14 extern void Temp_and_Humi_Sensors_Test();
```

```

15 extern void PutStr(char * s);
16 //-----
17 // 主程序
18 //-----
19 int main()
20 {
21     DDRC = 0xFF; PORTC = 0xFF;           //配置端口
22     Init_USART();                      //初始化串口
23     PutStr("\nStart Read Temperature and Humidity to show.....\n\n");
24     Temp_and_Humi_Sensors_Test();      //传感器测试(读取温湿度数据并转换显示)
25     while (1);
26 }

```

## 4.31 用 SPI 接口读/写 AT25F1024

兼容 SPI 接口的 AT25F1024、2048、4096 分别是 1-Mbit、2-Mbit 和 4-Mbit 的串行可编程 Flash 存储器。本例使用的 1-Mbits(128-KBytes)AT25F1024 分为 4 个区(sector)，每个区 32 KB，各区又分为 128 页，每页有 256 个字节空间。本例运行时，按下 K1 将清除芯片内所有数据，并在存储空间最前面和最后面分别写入 256 个有序字节和随机字节，按下 K2、K3 可以分别读取和显示这些字节，按下 K4 时可显示制造商 ID。案例电路及部分运行效果如图 4-47 所示。

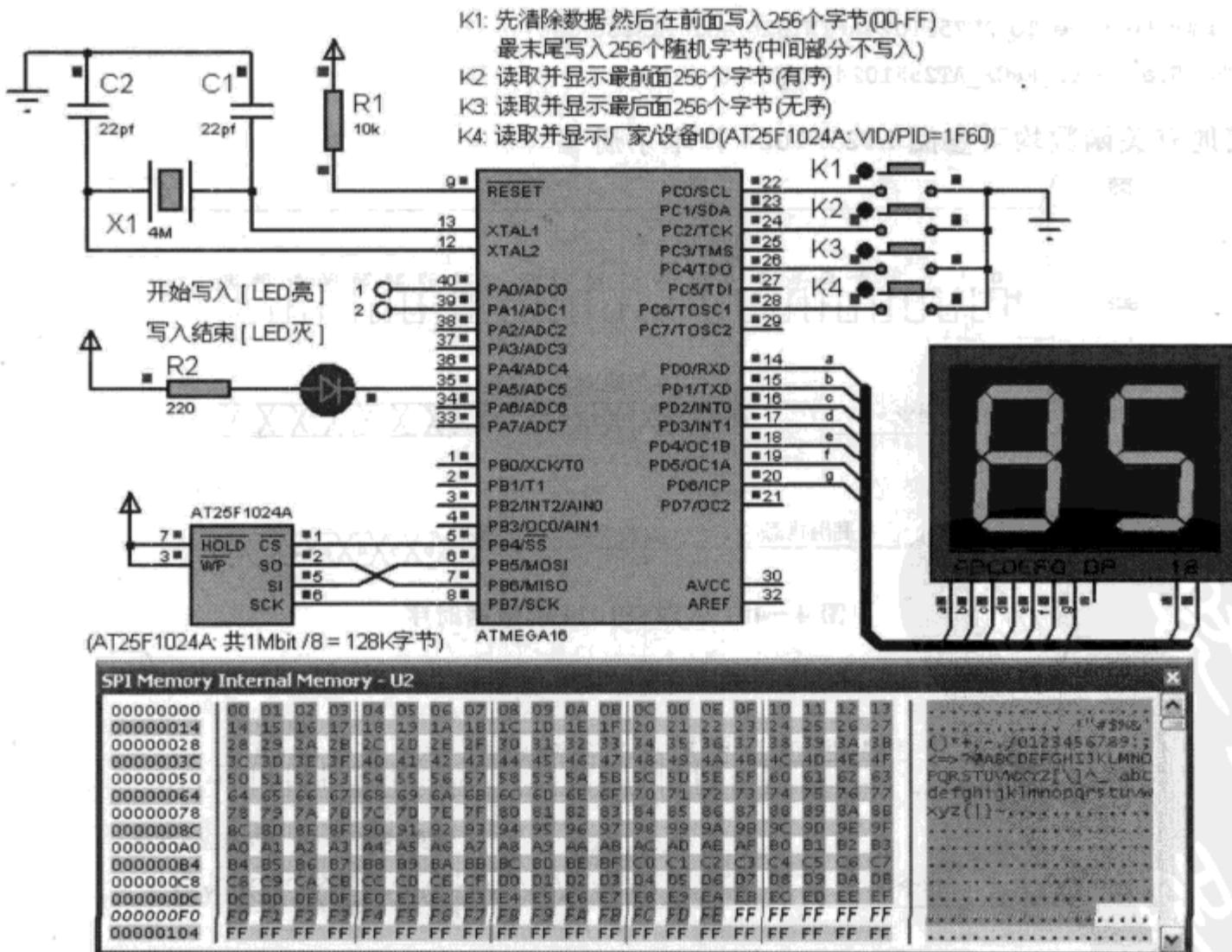


图 4-47 用 SPI 接口读/写 AT25F1024



## 1. 程序设计与调试

设计访问 AT25F1024 的程序时,参考其技术手册文件,可查到表 4-25 所列的相关操作指令,包括使能与禁止写、读/写状态寄存器、读/写数据字节、删除区域数据或芯片数据、读取厂商与产品 ID 等。源程序最前面的 AF25F1024 操作指令集即根据该表格编写。

表 4-25 AT25F1024 指令集

指令名称	指令格式	操作
WREN	0000 * 110	使能写
WRDI	0000 * 110	禁止写
RDSR	0000 * 101	读状态
WRSR	0000 * 001	写状态
READ	0000 * 011	读字节
PROGRAM	0000 * 010	写字节
SECTOR ERASE	0101 * 010	删除区域数据
CHIP ERASE	0110 * 010	删除内存中所有区域数据
RDID	0001 * 101	读取厂商与产品 ID

图 4-48 与图 4-49 给出了 AT25F1024 的数据读/写时序。本例的写字节函数与读字节函数就是分别根据这两个时序图编写完成的:

```
void Write_Byte_TO_AT25F1024A(INT32U addr, INT8U dat)  
INT8U Read_Byte_FROM_AT25F1024A(INT32U addr)
```

其他有关函数均可参阅 AT25F1024 技术手册编写。

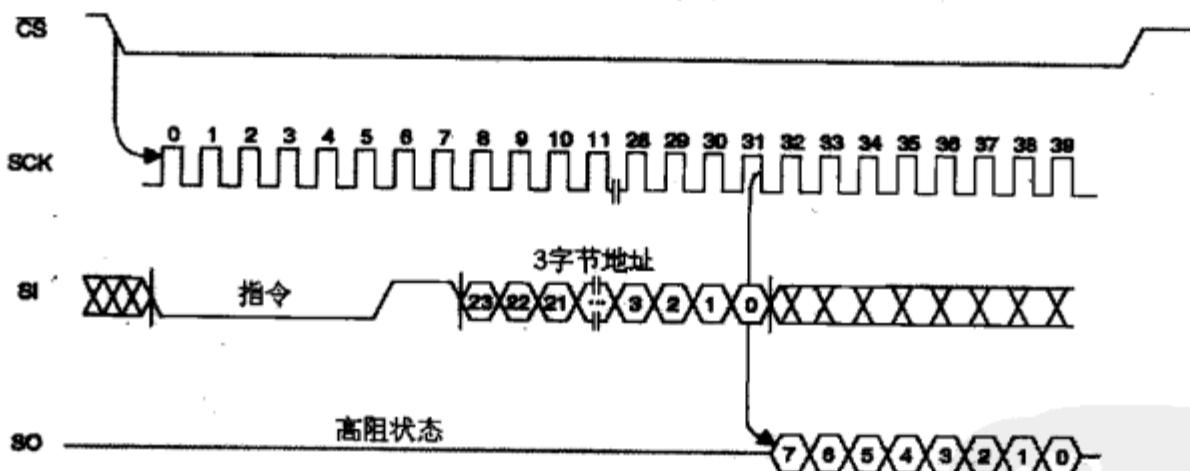


图 4-48 AT25F1024 读数据时序

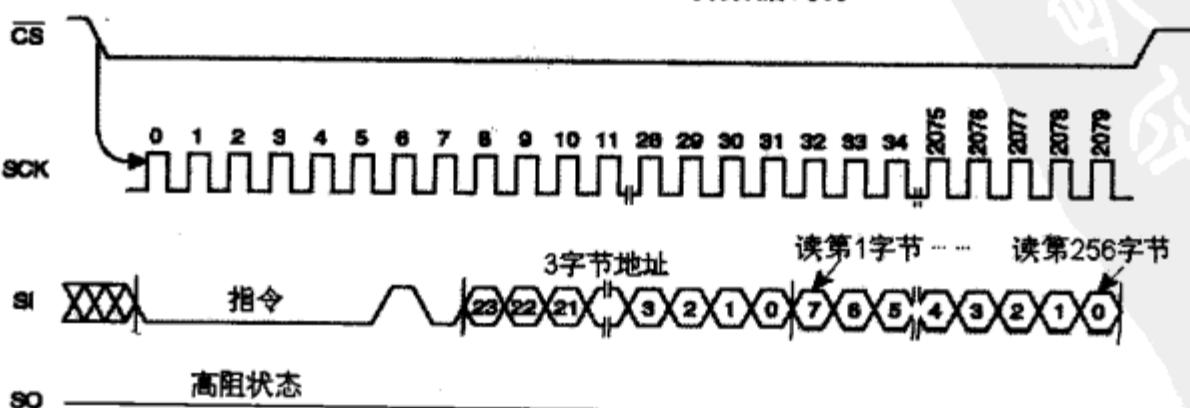


图 4-49 AT25F1024 写数据时序

## 2. 实训要求

- ① 查阅本例存储芯片技术手册文件, 编程对该芯片前 1/4、1/2 或整个区域数据设置写保护。
- ② 查阅 AT25F2048 与 AT25F4096 的技术手册文件, 编程对这两种存储器数据进行读/写操作。

## 3. 源程序代码

```

001 //-----
002 // 名称: 用 SPI 接口读/写 AT25F1024A
003 //-----
004 // 说明: 本例以 AVR 单片机为 SPI 主机,AT25F1024A 为 SPI 从机
005 //       按下 K1 时先删除芯片中未保护区的所有数据,然后从开始处写入
006 //       256 个字节(0x00~0xFF),最末尾 256 字节空间写入随机数
007 //       (其余部分不写入)
008 //       按下 K2 时读取最前面的 256 个字节并显示
009 //       按下 K3 时读取最后面的 256 个字节并显示
010 //       按下 K4 时显示 VID/PID
011 //       (显示格式均为十六进制数)
012 //
013 //-----
014 #define F_CPU 4000000UL
015 #include <avr/io.h>
016 #include <util/delay.h>
017 #define INT8U unsigned char
018 #define INT16U unsigned int
019 #define INT32U unsigned long
020
021 //AT25F1024A 指令集
022 #define WREN 0x06 //使能写
023 #define WRDI 0x04 //禁止写
024 #define RDSR 0x05 //读状态
025 #define WRSR 0x01 //写状态
026 #define READ 0x03 //读字节
027 #define PROGRAM 0x02 //写字节
028 #define SECTOR_ERASE 0x52 //删除区域数据
029 #define CHIP_ERASE 0x62 //删除芯片数据
030 #define RDID 0x15 //读厂商与产品 ID
031
032 //按键定义
033 #define K1_DOWN() ((PIN_C & _BV(PC0)) == 0x00) //写入两组字节
034 #define K2_DOWN() ((PIN_C & _BV(PC2)) == 0x00) //读前 256 字节并显示
035 #define K3_DOWN() ((PIN_C & _BV(PC4)) == 0x00) //读后 256 字节并显示
036 #define K4_DOWN() ((PIN_C & _BV(PC6)) == 0x00) //显示厂家和产品 ID

```



```
037
038 //LED 开关定义
039 #define LED_ON() ( PORTA &= ~_BV(PA5) )
040 #define LED_OFF() ( PORTA |= _BV(PA5) )
041
042 //SPI 使能与禁用
043 #define SPI_EN() (PORTB &= 0xEF)
044 #define SPI_DI() (PORTB |= 0x10)
045
046 //0~F 的数码管段码表(共阴数码管)
047 INT8U SEG_CODE[] =
048 {
049     0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,
050     0x6F,0x77,0x7F,0x7C,0x39,0x5E,0x79,0x71
051 };
052 //读/写字节数据的临时存放空间及有效数据长度
053 INT8U TMP_Buffer[256];
054 INT16U Buffer_LEN = 256;
055 //分解后的待显示数位
056 INT8U Display_Buffer[] = {0,0};
057 //-----
058 // 数码管显示 1 字节(十六进制)
059 //-----
060 void Show_Count_ON_DSY()
061 {
062     PORTA = 0xFF;
063     PORTD = SEG_CODE[Display_Buffer[1]];
064     PORTA = 0xFE;
065     _delay_ms(2);
066     PORTA = 0xFF;
067     PORTD = SEG_CODE[Display_Buffer[0]];
068     PORTA = 0xFD;
069     _delay_ms(2);
070 }
071
072 //-----
073 // SPI 主机初始化
074 //-----
075 void SPI_MasterInit()
076 {
077     //第 4、5、7 位/SS、MOSI、SCK 为输出,第 6 位 MISO 为输入
078     DDRB = 0B10110000; PORTB = 0xFF;
079     //SPI 使能,主机模式,16 分频
```

```
080     SPCR |= _BV(SPE) | _BV(MSTR) | _BV(SPR0);
081 }
082
083 //-----
084 // SPI 数据传输
085 //-----
086 INT8U SPI_Transmit(INT8U dat)
087 {
088     SPDR = dat;                                //启动数据传输
089     while(!(SPSR & _BV(SPIF)));                //等待结束
090     SPSR |= _BV(SPIF);                          //清中断标志
091     return SPDR;
092 }
093
094 //-----
095 // 读 AT25F1024A 芯片状态
096 //-----
097 INT8U Read_SPI_Status()
098 {
099     INT8U status;
100     SPI_EN();
101     SPI_Transmit(RDSR);                      //发送读状态指令
102     status = SPI_Transmit(0xFF);              //读取状态寄存器
103     SPI_DI();
104     return status;
105 }
106
107 //-----
108 // 读取 AT25F1024A 的厂商和产品 ID
109 //-----
110 void Get_VID_PID()
111 {
112     SPI_EN();                                 //发送读取 ID 指令
113     SPI_Transmit(RDID);                      //读取厂商代码 VID
114     TMP_Buffer[0] = SPI_Transmit(0xFF);        //读取产品代码 PID
115     TMP_Buffer[1] = SPI_Transmit(0xFF);
116     SPI_DI();
117 }
118
119 //-----
120 // AT25F1024A 忙等待
121 //-----
122 void Busy_Wait()
```



```
123 {
124     while(Read_SPI_Status() & 0x01);           //忙等待
125 }
126
127 //-----
128 // 删除 AT25F1024A 芯片未加保护的所有区域数据
129 //-----
130 void ChipErase()
131 {
132     SPI_EN();
133     SPI_Transmit(WREN);                      //使能写
134     SPI_DI();
135     Busy_Wait();
136
137     SPI_EN();
138     SPI_Transmit(CHIP_ERASE);                //清除芯片数据指令
139     SPI_DI();
140     Busy_Wait();
141 }
142
143 //-----
144 // 向 AT25F1024A 写入 3 个字节的地址 0x000000~0x01FFFF
145 //-----
146 void Write_3_Bytes_AT25F1024A_Address(INT32U addr)
147 {
148     SPI_Transmit((INT8U)(addr >> 16 & 0xFF)); //先发送最高的地址字节
149     SPI_Transmit((INT8U)(addr >> 8  & 0xFF)); //再发送次高的地址字节
150     SPI_Transmit((INT8U)(addr & 0xFF));        //最后发送低位地址字节
151 }
152
153 //-----
154 // 从指定地址读单字节
155 //-----
156 INT8U Read_Byte_FROM_AT25F1024A(INT32U addr)
157 {
158     INT8U dat;
159     SPI_EN();
160     SPI_Transmit(READ);                     //发送读指令
161     Write_3_Bytes_AT25F1024A_Address(addr); //发送 3 字节地址
162     dat = SPI_Transmit(0xFF);               //读取字节数据
163     SPI_DI();
164     return dat;
165 }
```

```
166
167 //-----
168 // 从指定地址读多字节到缓冲
169 //-----
170 void Read_Some_Bytes_FROM_AT25F1024A(INT32U addr, INT8U * p, INT16U len)
171 {
172     INT16U i;
173     SPI_EN();
174     SPI_Transmit(READ); //发送读指令
175     Write_3_Bytes_AT25F1024A_Address(addr); //发送 3 字节地址
176     for( i = 0; i < len; i ++ ) //读取多字节数据
177         p[i] = SPI_Transmit(0xFF);
178     SPI_DI();
179 }
180
181 //-----
182 // 向 AT25F1024A 指定地址写入单字节数据
183 //-----
184 void Write_BytE_TO_AT25F1024A(INT32U addr, INT8U dat)
185 {
186     SPI_EN();
187     SPI_Transmit(WREN); //使能写
188     SPI_DI();
189     Busy_Wait();
190     SPI_EN();
191     SPI_Transmit(PROGRAM); //写指令
192     Write_3_Bytes_AT25F1024A_Address(addr); //发送 3 字节地址
193     SPI_Transmit(dat); //写字节数据
194     SPI_DI();
195     Busy_Wait();
196 }
197
198 //-----
199 // 向 AT25F1024A 指定地址开始写入多字节数据
200 //-----
201 void Write_Some_Bytes_TO_AT25F1024A(INT32U addr, INT8U * p, INT16U len)
202 {
203     INT16U i;
204     SPI_EN();
205     SPI_Transmit(WREN); //使能写
206     SPI_DI();
207     Busy_Wait();
208     SPI_EN();
```

```

209     SPI_Transmit(PROGRAM);           //写指令
210     Write_3_Bytes_AT25F1024A_Address(addr); //发送 3 字节地址
211     for (i = 0; i<len; i++)
212         SPI_Transmit(p[i]);           //写多字节数据
213     SPI_DI();
214     Busy_Wait();
215 }
216
217 //-----
218 // 主程序
219 //-----
220 int main()
221 {
222     INT32U i;
223     INT8U Current_Data,Current_Displ_Index = 0,LOOP_SHOW_FLAG = 0,
224     DDRA = 0xFF; PORTA = 0xFF;           //配置端口
225     DDRD = 0xFF; PORTD = 0xFF;
226     DDRC = 0x00; PORTC = 0xFF;
227     TCCR0 = 0x01;                      //未分频
228     TIMSK = _BV(TOIE0);               //允许 T/C0 定时器溢出中断
229     SPI_MasterInit();                //SPI 主机初始化
230     while(1)
231     {
232         Begin:
233         //K1:向 AT25F1024A 写入数据
234         if(K1_DOWN()) //-----
235         {
236             //先点亮 LED
237             LED_ON();
238             //删除芯片全部内容
239             ChipErase();
240             // 下面两个循环使用的是单字节逐个写入的方法.....
241             // 前面 256 字节空间写入 0x00~0xFF
242             // for (i = 0x000000; i<= 0x0000FF; i++)
243             // {
244                 //     Write_Byt_TO_AT25F1024A(i,(INT8U)i);
245                 //     Busy_Wait();
246             // }
247             // 最末尾 256 个字节空间全部写入随机字节(随机字节由 T/C0 提供)
248             // for (i = 0xFFFF00; i<= 0xFFFFFFF; i++)
249             // {
250                 //     Write_Byt_TO_AT25F1024A(i,(TCNT0>>6) | (TCNT0<<2));
251                 //     Busy_Wait();

```

```

252     // }
253     //下面使用的是顺序写入的方法..... .
254     //先准备待写入有序字节数组
255     for (i = 0; i < 256; i++) TMP_Buffer[i] = i;
256     //从指定地址开始顺序写入
257     Write_Some_Bytes_TO_AT25F1024A(0x000000, TMP_Buffer, 256);
258     //先准备待写入随机字节数组(随机字节由 T/C0 提供)
259     for (i = 0; i < 256; i++) TMP_Buffer[i] = (TCNT0 >> 6) | (TCNT0 << 2);
260     //从指定地址开始顺序写入
261     Write_Some_Bytes_TO_AT25F1024A(0x1FFF00, TMP_Buffer, 256);
262     while (K1_DOWN());
263     //完成后关闭 LED
264     LED_OFF();
265 }
266 //K2:读取并显示前 256 个字节
267 if (K2_DOWN()) //-----
268 {
269     //从指定地址顺序读取多字节到缓冲
270     Read_Some_Bytes_FROM_AT25F1024A(0x000000, TMP_Buffer, 256);
271     //下面是单字节逐个读取的代码
272     // for (i = 0x000000; i <= 0x0000FF; i++)
273     // TMP_Buffer[(INT8U)i] = Read_Byte_FROM_AT25F1024A(i);
274     while (K2_DOWN());
275     Current_Displ_Index = 0;
276     Buffer_LEN = 256;
277     LOOP_SHOW_FLAG = 1;
278 }
279 //K3:读取并显示后 256 个字节
280 if (K3_DOWN()) //-----
281 {
282     //从指定地址顺序读取多字节到缓冲
283     Read_Some_Bytes_FROM_AT25F1024A(0x1FFF00, TMP_Buffer, 256);
284     //下面是单字节逐个读取的代码
285     // for (i = 0x1FFF00; i <= 0xFFFF; i++)
286     // TMP_Buffer[(INT8U)(i - 0x1FFF00)] = Read_Byte_FROM_AT25F1024A(i);
287     while (K3_DOWN());
288     Current_Displ_Index = 0;
289     Buffer_LEN = 256;
290     LOOP_SHOW_FLAG = 1;
291 }
292 //K4:显示厂家和设备 ID
293 if (K4_DOWN()) //-----
294 {

```



```
295     Get_VID_PID();           //读取厂商与产品 ID(VID/PID)
296     Buffer_LEN = 2;          //设置显示缓冲有效数据长度为 2
297     Current_Displ_Index = 0;
298     LOOP_SHOW_FLAG = 1;
299     while (K4_DOWN());
300 }
301 //循环显示数据
302 if (LOOP_SHOW_FLAG) //-----
303 {
304     Current_Data = TMP_Buffer[Current_Displ_Index];
305     Display_Buffer[1] = Current_Data / 16;
306     Display_Buffer[0] = Current_Data % 16;
307     //每个数显示保持一段时间
308     for (i = 0 ; i<150; i++)
309     {
310         //数码管显示 2 位十六进制数
311         Show_Count_ON_DSY();
312         //显示过程中如果某键按下则停止显示
313         if (PIN_C != 0xFF)
314         {
315             LOOP_SHOW_FLAG = 0;
316             PORTD = 0x00;
317             goto Begin;
318         }
319     }
320     //显示索引循环递增( 0~Buffer_LEN - 1)
321     Current_Displ_Index = (Current_Displ_Index + 1) % Buffer_LEN;
322 }
323 }
324 }
```

## 4.32 用 TWI 接口读/写 24C04

本例用到的 AT24C04 是 4-Kbit(512-Byte)的 EEPROM，兼容 I<sup>2</sup>C 接口总线，它与单片机通信时只需要通过两根串行线，一根是双向的串行数据线 SDA，另一根是时钟线 SCL。该存储器件占用很少的资源和 I/O 引脚，且具有工作电源宽、抗干扰能力强、功耗低、数据不易丢失和支持在线编程等特点。本例运行时，按下 K1 将向 AT24C04 写入 512 个字节数据，按下 K2 或 K3 时将分别读取并显示前后各 256 个字节数据。电路及运行效果如图 4-50 所示。

### 1. 程序设计与调试

本例程序要点在于字节读/写函数设计，这两个函数分别参考时序图 4-51 和图 4-52

编写：

```
INT8U I2C_Write(INT16U addr, INT8U dat)
INT8U I2C_Read(INT16U addr)
```

在参阅时序图时,其中 START 后的 DEVICE ADDRESS(器件地址)可参考表 4-26 的 24C04 器件地址字节格式,高 4 位固定为 1010,本例程序最前面定义了从器件(或称子器件)地址:

```
#define DEV_SUB_ADDR 0xA0
```

定义中 0xA0 的高 4 位即 1010,0xA0 的后 4 位为 0000,分别对应于该表格中的 A2/A1/页选择位/读/写位。其中 A2/A1 用于在 DEVICE ADDRESS 中指定共用 SDA 与 SCL 的外部多达 4 片 24C04 中的一片,因为 24C04 硬地址配置引脚有 A2/A1,硬地址有 4 种组合,因此 A2/A1 对应的取值有 00、01、10、11。本例中 A2 与 A1 全部接地,这两位为 00。

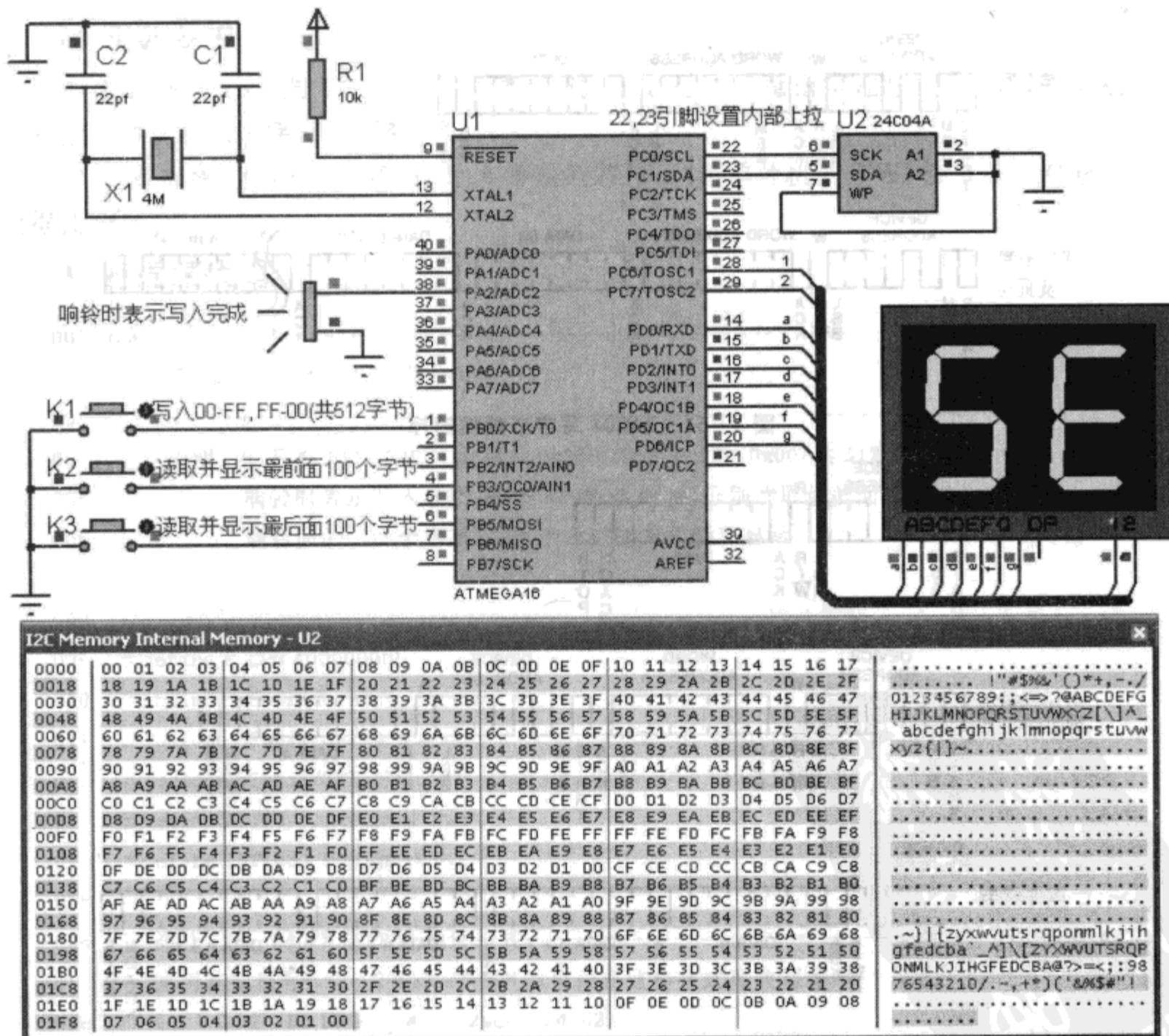


图 4-50 用 TWI 接口读/写 24C04

最后讨论一下 P0 位,对于 512 KB 的 24C04 串行存储器,其地址位共 9 位( $2^9 = 512$ )。图

4-51、图 4-52 所示多个时序图中,在 DEVICE ADDRESS 后接着需要给出待读/写的字节地址 BYTE ADDR(时序图中为 WORD ADDRESS),该字节仅 8 位(A7~A0),而 9 位地址的最高位 A8 则附加在 DEVICE ADDRESS 中提供,这一位就是 P0。显然,发送器件地址的第一字节(也称控制字节)“携带”了 24C04 内部空间字节地址的最高位。明白这一点后再来分析字节读/写函数中的以下语句就很清楚了:

```
WriteByte(DEV_SUB_ADDR | (INT8U)(addr >> 8 << 1))
```

该语句中 addr 为 16 位的无符号整数(对于 24C04,这 16 位中仅有低 9 位是有效的地址位),为将 9 位地址的高位 A8 填充到 DEVICE ADDRESS 的倒数第 2 位 P0,可先将最高的第 9 位移到最低位,即  $addr >> 8$ ,然后再将其左移 1 位,最后将得到的 8 位字节与 DEV\_SUB\_ADDR(即 0xA0)作或操作即可。要注意这里不要将  $addr >> 8 << 1$  简写为  $addr >> 7$ ,因为这样不能保证最低位为 0,由于要向指定地址写字节,因此控制字节的最低位 R/W 必须为 0。

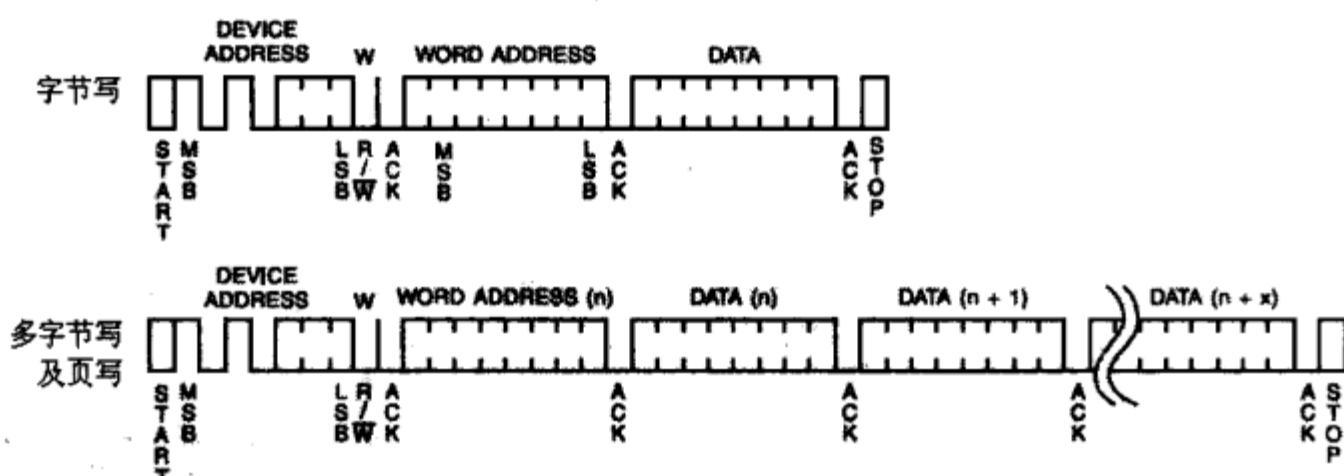


图 4-51 24C04 写字节数据时序

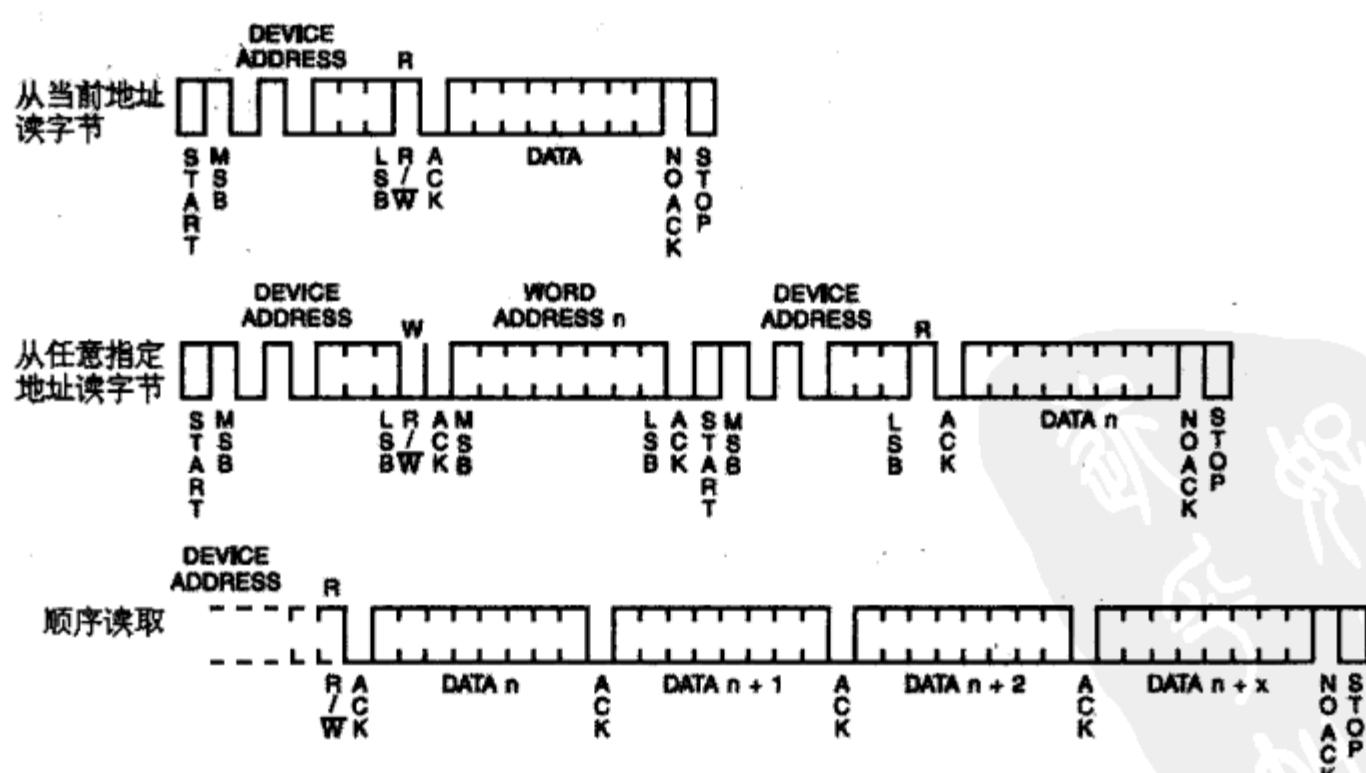


图 4-52 24C04 各类读字节时序

同样,根据读字节操作时序编写的函数中有如下语句:

```
WriteByte(DEV_SUB_ADDR | (INT8U)(addr >> 8 << 1) | 0x01)
```

它将控制字节的最低位置为 1,接下来的每 8 个时钟周期都将从相应地址读取 1 字节数据。twi.h 中将符号 TWI\_READ 定义为 1,因此上述语句还可以写成:

```
WriteByte(DEV_SUB_ADDR | (INT8U)(addr >> 8 << 1) | TWI_READ);
```

阅读本例程序的其他编写细节时,可进一步参考阅读 AT24C04 的技术手册文件。

表 4-26 24C04 器件地址字节格式

	器件编码				硬地址位		块选择	读/写
位	7	.6	5	4	3	2	1	0
器件选择	1	0	1	0	A2	A1	P0	R/W

说明:该地址高 4 位固定为 1010,A2/A1 与器件 A2/A1 引脚对应,用于选择器件硬地址,P0 为 24C04 共 9 位地址中的最高位(即 A0~A8 中的第 A8 位)。表中 7~1 位对应时序图中的 DEVICE ADDRESS。

## 2. 实训要求

① 参考图 4-51、图 4-52 提供的读/写时序,编写从任意指定地址读取多字的函数及从任意指定地址写入多字节的函数。

② 重新选择 24C08/16/32/64/128 等系列芯片,利用单片机的 TWI 接口编程对它们进行数据读/写。

## 3. 源程序代码

```

001 //-----
002 // 名称: 用 TWI 接口读/写 24C04
003 //-----
004 // 说明: 按下 K1 时向 24C04 中写入 0x00~0xFF,0xFF~0x00(共 512 个字节),
005 //       响铃时表示写入结束,按下 K2 或 K3 时,可分别读取最前面的 100 个或
006 //       最后面的 100 个字节,并以十六进制方式循环显示
007 //
008 //-----
009 #define F_CPU 4000000UL
010 #include <avr/io.h>
011 #include <util/delay.h>
012 #include <util/TWI.h>
013 #define INT8U unsigned char
014 #define INT16U unsigned int
015
016 // 部分 IIC-24CXX EEPROM 地址格式-----
017 // 使用其他型号 IIC-24CXX EEPROM 时可根据这些格式可改写程序
018 // 1 0 1 0 E2 E1 E0 R/~W 24C01/24C02
019 // 1 0 1 0 E2 E1 A8 R/~W 24C04
020 // 1 0 1 0 E2 A9 A8 R/~W 24C08
021 // 1 0 1 0 A10 A9 A8 R/~W 24C16
022 //-----

```



```
023
024 //按键定义
025 #define Write_512_Key_DOWN() ((PINB & 0x01) == 0x00) //写入
026 #define Read_F100_Key_DOWN() ((PINB & 0x08) == 0x00) //读前 100 个显示
027 #define Read_R100_Key_DOWN() ((PINB & 0x40) == 0x00) //读后 100 个显示
028
029 //蜂鸣器定义
030 #define BEEP() (PORTA ^= _BV(PA2))
031
032 //定义子器件地址
033 #define DEV_SUB_ADDR 0xA0
034
035 //TWI 通用操作
036 #define Wait()           while ((TWCR & _BV(TWINT)) == 0)
037 #define START()          {TWCR = _BV(TWINT) | _BV(TWSTA) | _BV(TWEN); Wait(); }
038 #define STOP()           {TWCR = _BV(TWINT) | _BV(TWSTO) | _BV(TWEN)}
039 #define WriteByte(x)     {TWDR = (x); TWCR = _BV(TWINT) | _BV(TWEN); Wait(); }
040 #define ACK()            (TWCR |= _BV(TWEA))
041 #define NACK()           (TWCR &= ~_BV(TWEA))
042 #define TWI()             {TWCR = _BV(TWINT) | _BV(TWEN); Wait(); }
043
044 //0~F 的数码管段码表(共阴数码管)
045 const INT8U SEG_CODE[] =
046 {
047     0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x07
048     0x6F, 0x77, 0x7F, 0x7C, 0x39, 0x5E, 0x79, 0x71
049 };
050
051 //读取 100 个字节数据的临时存放空间
052 INT8U TMP_Buffer[100];
053 //分解后的待显示数位
054 INT8U Display_Buffer[] = {0,0};
055 //-----
056 // 响铃子程序
057 //-----
058 void Play_BEEP()
059 {
060     INT16U i;
061     for (i = 0 ; i < 300; i++)
062     {
063         BEEP(); _delay_us(200);
064     }
065 }
```

```

066
067 //-----
068 // 数码管上显示(PC6、PC7 分别连接 2 只数码管共阴引脚)
069 //-----
070 void Show_Count_ON_DSY()
071 {
072     PORTD = 0x00;                                //暂时关闭段码
073     PORTD = SEG_CODE[Display_Buffer[1]];           //发送段码
074     PORTC = (PORTC | _BV(PC7)) & ~_BV(PC6);       //发送位码
075     _delay_ms(2);
076     PORTD = 0x00;                                //显示第 2 位数,操作同上
077     PORTD = SEG_CODE[Display_Buffer[0]];
078     PORTC = (PORTC | _BV(PC6)) & ~_BV(PC7);
079     _delay_ms(2);
080 }
081
082 //-----
083 // 从 IIC 中指定地址读 1 字节
084 //-----
085 INT8U I2C_Read(INT16U addr)
086 {
087     INT8U dat;
088     START();
089     if (TW_STATUS != TW_START)      return 0;
090     //下面地址部分的(INT8U)(addr >> 8 << 1)对应于 A8 位,其他部分代码同此
091     WriteByte(DEV_SUB_ADDR | (INT8U)(addr >> 8 << 1)); //写控制字节(含最高位 A8)
092     if (TW_STATUS != TW_MT_SLA_ACK)  return 0;
093     WriteByte((INT8U)addr);          //发送 EEPROM 地址
094     if (TW_STATUS != TW_MT_DATA_ACK) return 0;
095     START();                         //重新启动
096     if (TW_STATUS != TW REP START)   return 0;
097     WriteByte(DEV_SUB_ADDR | (INT8U)(addr >> 8 << 1) | 0x01); //发送读操作命令
098     if (TW_STATUS != TW_MR_SLA_ACK) return 0;
099     TWI();
100     if (TW_STATUS != TW_MR_DATA_NACK) return 0;
101     dat = TWDR;                      //读取 1 字节
102     STOP();
103     return dat;
104 }
105
106 //-----
107 // 向 IIC24C04A 中指定地址写 1 字节
108 //-----

```

```

109  INT8U I2C_Write(INT16U addr, INT8U dat)
110 {
111     START();
112     Wait();
113     if(TW_STATUS != TW_START)      return 0;
114     WriteByte(DEV_SUB_ADDR | (INT8U)(addr >> 8 << 1)); //写控制字节(含最高位 A8)
115     Wait();
116     if(TW_STATUS != TW_MT_SLA_ACK) return 0;
117     WriteByte((INT8U)addr);           //发送 EEPROM 地址
118     Wait();
119     if(TW_STATUS != TW_MT_DATA_ACK) return 0;
120     WriteByte(dat);                //向该地址写 1 字节数据
121     Wait();
122     if(TW_STATUS != TW_MT_DATA_ACK) return 0;
123     STOP();
124     return 1;
125 }
126
127 //-----
128 // 主程序
129 //-----
130 int main()
131 {
132     INT16U i, Current_Dispc_Index = 0;
133     INT8U dat = 0x00, Current_Data, LOOP_SHOW_FLAG = 0;
134     DDRC = 0xF0; PORTC = 0xFF;                      //配置端口
135     DDRA = 0xFF; PORTA = 0xFF;
136     DDRD = 0xFF; PORTD = 0xFF;
137     DDRB = 0x00; PORTB = 0xFF;
138     while(1)
139     {
140         Begin:
141         //K1:写入 512 个字节(00~FF, FF~00)
142         if(Write_512_Key_DOWN()) //-----
143         {
144             //关闭 2 只数码管
145             PORTC |= _BV(PB6) | _BV(PB7);
146             //在 24C04 前半部分空间写入 0x00~0xFF
147             for (i = 0x0000; i <= 0x00FF; i++)
148             {
149                 I2C_Write(i,dat++);
150                 _delay_ms(2);
151             }
}

```

```

152 //在 24C04 后半部分空间写入 0xFF~0x00
153 for (i = 0x0100; i <= 0x01FF; i++)
154 {
155     I2C_Write(i, --dat);
156     _delay_ms(2);
157 }
158 Play_BEEP();
159 while (Write_512_Key_DOWN()); //等待释放
160 LOOP_SHOW_FLAG = 0;
161 }
162
163 //K2:读取并显示前 100 个字节
164 if (Read_F100_Key_DOWN()) //-----
165 {
166     for (i = 0x0000; i <= 0x0063; i++)
167     {
168         TMP_Buffer[(INT8U)i] = I2C_Read(i);
169         _delay_ms(2);
170     }
171     while (Read_F100_Key_DOWN()); //等待释放
172     Current_Displ_Index = 0;
173     LOOP_SHOW_FLAG = 1;
174 }
175 //K3:读取并显示后 100 个字节
176 if (Read_R100_Key_DOWN()) //-----
177 {
178     for (i = 0x01FF; i > 0x01FF - 100; i--)
179     {
180         TMP_Buffer[99 - (INT8U)(0x01FF - i)] = I2C_Read(i);
181         _delay_ms(4);
182     }
183     while (Read_R100_Key_DOWN()); //等待释放
184     Current_Displ_Index = 0;
185     LOOP_SHOW_FLAG = 1;
186 }
187 if (LOOP_SHOW_FLAG) //-----
188 {
189     Current_Data = TMP_Buffer[Current_Displ_Index];
190     Display_Buffer[1] = Current_Data / 16;
191     Display_Buffer[0] = Current_Data % 16;
192     //每个数显示保持一段时间
193     for (i = 0; i < 150; i++)
194     {

```



```
195 //数码管显示 2 位十六进制数  
196 Show_Count_ON_DSY();  
197 //显示过程中如果有键按下则停止显示  
198 if (Write_512_Key_DOWN() || Read_F100_Key_DOWN() || Read_R100_Key_DOWN())  
199 {  
200     LOOP_SHOW_FLAG = 0;  
201     PORTA = 0xFF;  
202     goto Begin;  
203 }  
204 }  
205 //显示索引循环递增(0~99)  
206 Current_Displ_Index = (Current_Displ_Index + 1) % 100;  
207 }  
208 }  
209 }
```

## 4.33 MPX4250 压力传感器测试

本例使用了 Motorola 公司生产的压力传感器 MPX4250。程序运行时,传感器向单片机输入模拟电压信号,经 A/D 转换后,再根据技术手册提供的公式进行换算,最后将压力值显示在 4 位数码管上。案例电路及部分运行效果如图 4-53 所示。

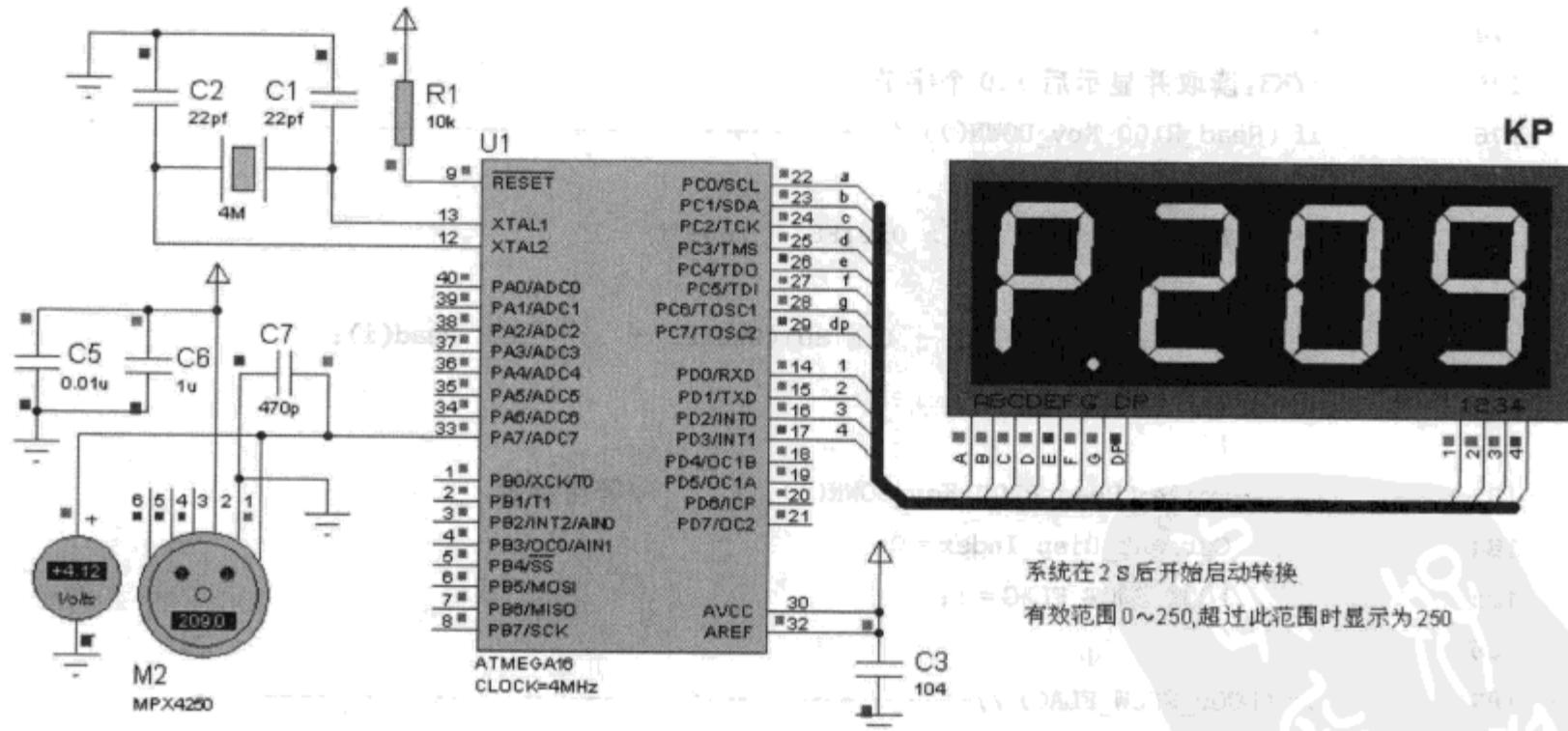


图 4-53 MPX4250 压力传感器测试

### 1. 程序设计与调试

有了 A/D 转换案例程序的学习与调试,本例的编写就显得很简单了,剩下的唯一关键仅在于压力传感器的输出电压与压力值之间的换算关系。根据图 4-54 所示的压力传感器的压力/电压输出曲线与转换函数,程序中第 50 行语句的编写就很容易了:

```
Pressure_Value = (AD_Result * 5.0 / 1023.0 / 5.1 - 0.04) / 0.00369 + 1.99;
```

该语句中,AD\_Result 为 10 位的模/数转换结果,+1.99 为本例所设置的 Error 值。根据上述公式和 Error 设置,在实际测试过程中,对于较低的压力转换,其误差在 1~4 kPa 以内,在较高的压力下则误差很小。

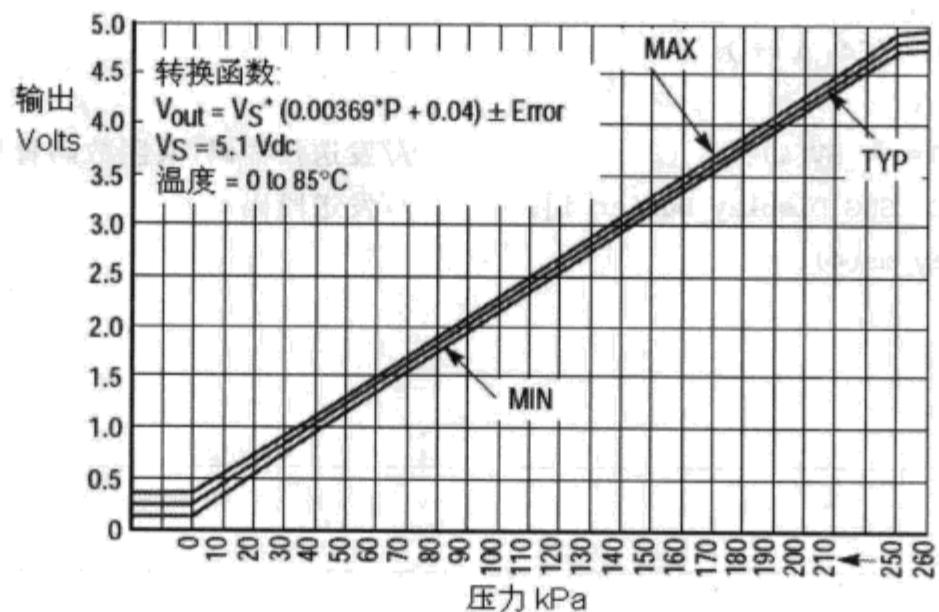


图 4-54 MPX4250 压力传感器电压输出曲线与转换函数

## 2. 实训要求

- ① 重新设置 Error 值, 观察在较高压力和较低压力下的显示误差情况。
- ② 改用液晶显示屏, 将压力值实时显示在液晶显示屏上, 为第 5 章电子秤的仿真设计打下基础。

## 3. 源程序代码

```

01 //-----
02 // 名称: MPX4250 压力传感器测试
03 //-----
04 // 说明: 本例运行时,MPX4250 通过 PA7 向单片机输入模拟电压,经模/数
05 //        转换并通过公式换算后,数码管将显示出当前压力值
06 //
07 //-----
08 #define F_CPU 4000000UL
09 #include <avr/io.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 的数字编码,最后一位为黑屏
15 const INT8U SEG_CODE[] =
16 { 0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x00 };
17 //分解后的待显示数位段码缓冲(第一位是大写字母 P)
18 INT8U SEG_Display_Buffer[] = {0xF3,0,0,0};
19 //-----

```



```
20 // 数码管显示压力
21 //-----
22 void Show_PRESS_ON_DSY()
23 {
24     INT8U i = 0;
25     for (i = 0; i<4; i++)
26     {
27         PORTD = ~_BV(i);           //发送扫描码(共阴数码管加~)
28         PORTC = SEG_Display_Buffer[i]; //发送段码
29         _delay_ms(4);
30     }
31 }
32
33 //-----
34 // 主程序
35 //-----
36 int main()
37 {
38     int AD_Result,Pressure_Value;          //模/数转换结果及压力换算结果
39     DDRA = 0x7F; PORTA = 0xFF;           //配置端口(PA7 为模拟输入)
40     DDRC = 0xFF; PORTC = 0xFF;
41     DDRD = 0xFF; PORTD = 0xFF;
42     ADCSRA = 0xE6;                      //ADC 转换置位,启动转换,64 分频
43     _delay_ms(2000);                   //延时等待系统稳定
44     ADMUX = 0x07;                      //选择模/数转换通道 AD7
45     while(1)
46     {
47         //读取转换结果(10 位精度,获取的值为 0~1023)
48         AD_Result = ADCL + (ADCH << 8);
49         //根据 MPX4250 技术手册,经下面的公式换算出当前压力
50         Pressure_Value = (AD_Result * 5.0 / 1023.0 / 5.1 - 0.04) / 0.00369 + 1.99;
51         //将压力值转换到显示缓冲,并对高位及次高位的 0 作相应的屏蔽处理
52         SEG_Display_Buffer[1] = SEG_CODE[ Pressure_Value / 100 ];
53         SEG_Display_Buffer[2] = SEG_CODE[ Pressure_Value / 10 % 10 ];
54         SEG_Display_Buffer[3] = SEG_CODE[ Pressure_Value % 10 ];
55         if (SEG_Display_Buffer[1] == 0x3F)
56         {
57             SEG_Display_Buffer[1] = 0x00;
58             if (SEG_Display_Buffer[2] == 0x3F) SEG_Display_Buffer[2] = 0x00;
59         }
60         //数码管显示
61         Show_PRESS_ON_DSY();
62     }
63 }
```

## 4.34 MMC 存储卡测试

MMC 卡即多媒体卡(Multimedia Card),由西门子公司和 SanDisk 于 1997 年推出。1998 年 1 月,14 家公司联合成立了 MMC 协会(MultiMediaCard Association,即 MMCA)。MMC 的发展目标主要是针对数码影像、音乐、手机、PDA、电子书、玩具等产品。MMC 的工作电压为 2.7~3.6 V,写/读电流分别为 27 mA 和 23 mA,功耗很低。MMC 卡把存储单元和控制器设计到一张卡上,智能控制单元使 MMC 保证了兼容性和灵活性。

本例对 MMC 存储卡进行读/写仿真测试。在系统运行时,按下 K1、K2 可分别向 MMC 卡的第 0 块与第 1 块中分别写入 512 字节数据。按下 K3、K4 可分别读取这些字节数据并发送到虚拟终端显示。本例电路及部分运行效果如图 4-55 所示。

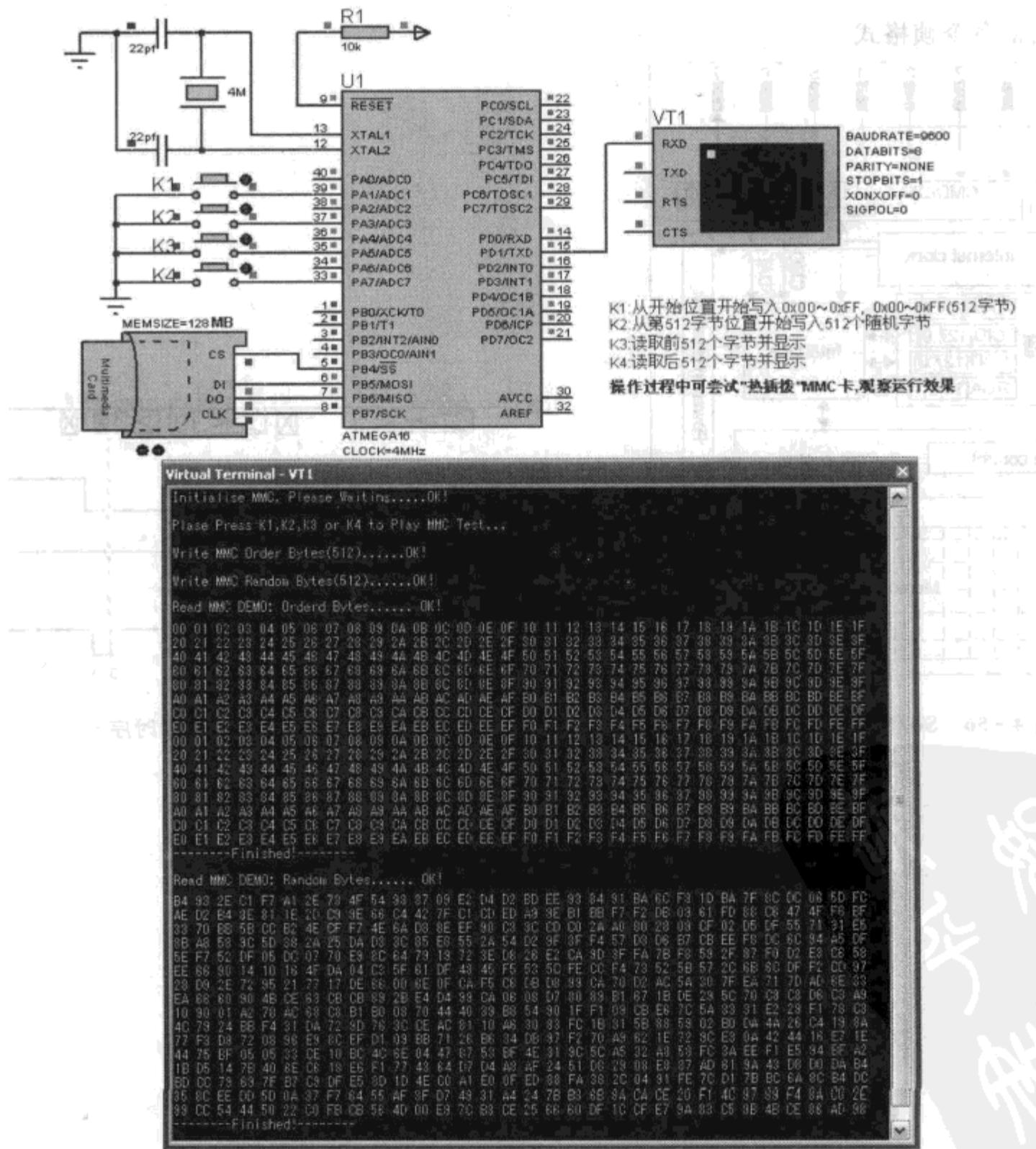


图 4-55 MMC 存储卡测试



## 1. 程序设计与调试

MMC 存储卡可工作于 MMC 模式和 SPI 模式,前者是标准的默认模式,具有 MMC 的全部特性。SPI 模式则是 MMC 存储卡可选的第二种模式,它是 MMC 协议的一个子集,主要用于仅需要少量卡(一般是 1 块卡)和低速数据传输的场合。本例 MMC 卡工作于 SPI 模式。

图 4-56 给出了 SD/MMC 卡的引脚名称与内部结构,其中 CS、CMD/DI、CLK/SCLK、DAT/DO 分别与单片机的 PB4(/SS)、PB5(MOSI)、PB6(MISO)、PB7(SCK)相连。

MMC 卡的读/写模式包括流式(Stream)、多块(Multi-Block)和单块(Single-Block)。数据传送以块为单位时,默认的块大小为 512 字节。

通过 SPI 接口读/写 MMC 卡时,需要查阅 MMC 技术资料,特别是各命令字节、命令格式、操作时序等。图 4-57~图 4-60 分别给出了复位命令时序、初始化命令时序、单块读时序及单块写时序,各时序图中同时给出了对应的命令字节 CMDx。表 4-27 给出了 48 位(6 字节)的完整命令帧格式。

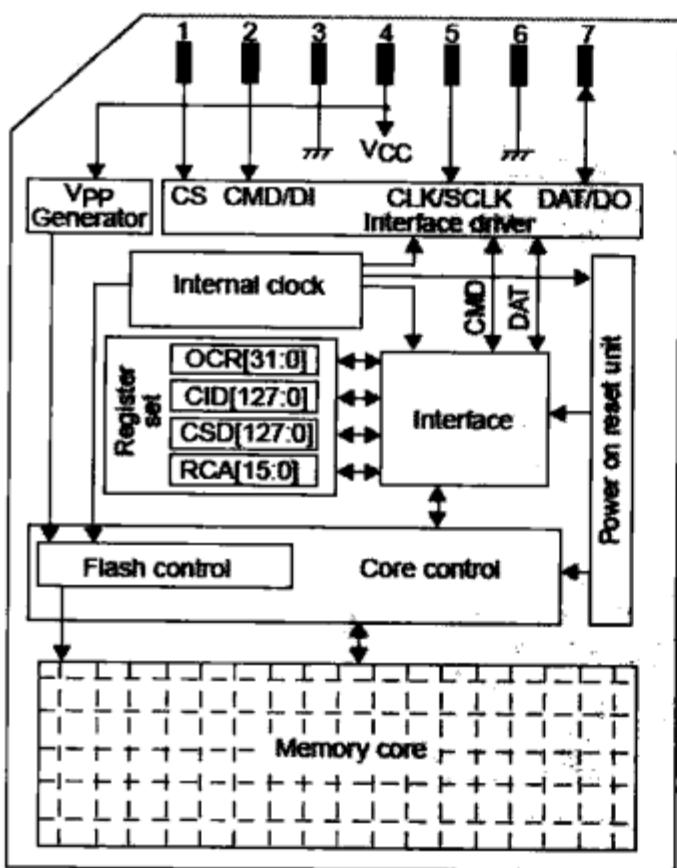


图 4-56 SD/MMC 卡结构图

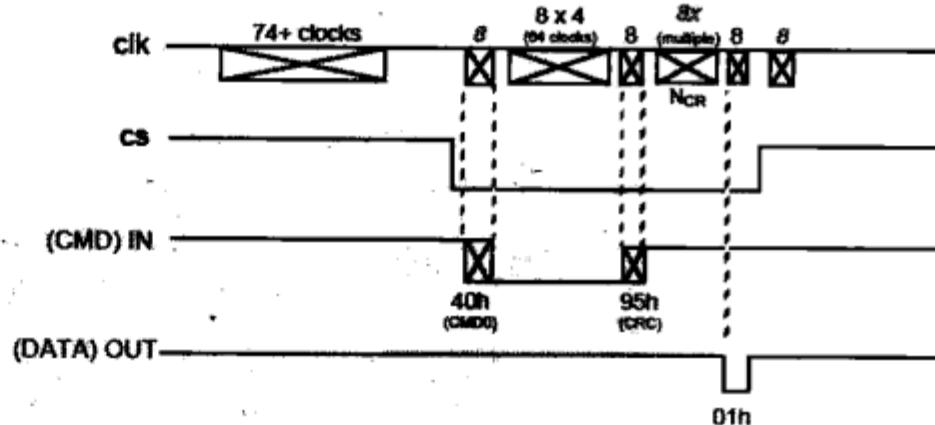


图 4-57 MMC 卡复位命令时序

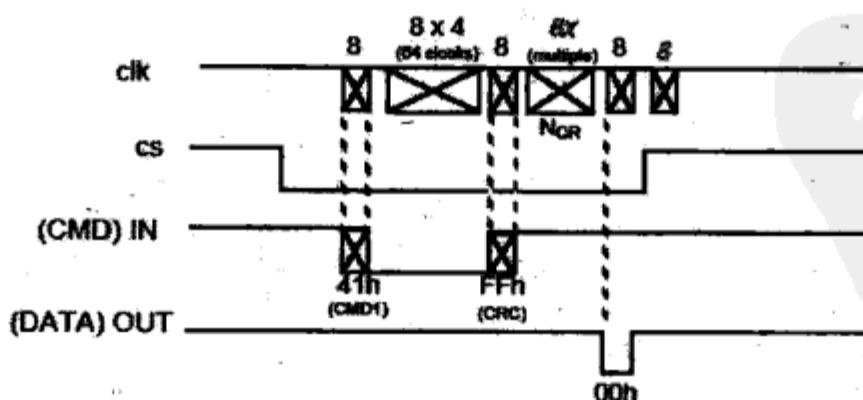


图 4-58 MMC 卡初始化命令时序

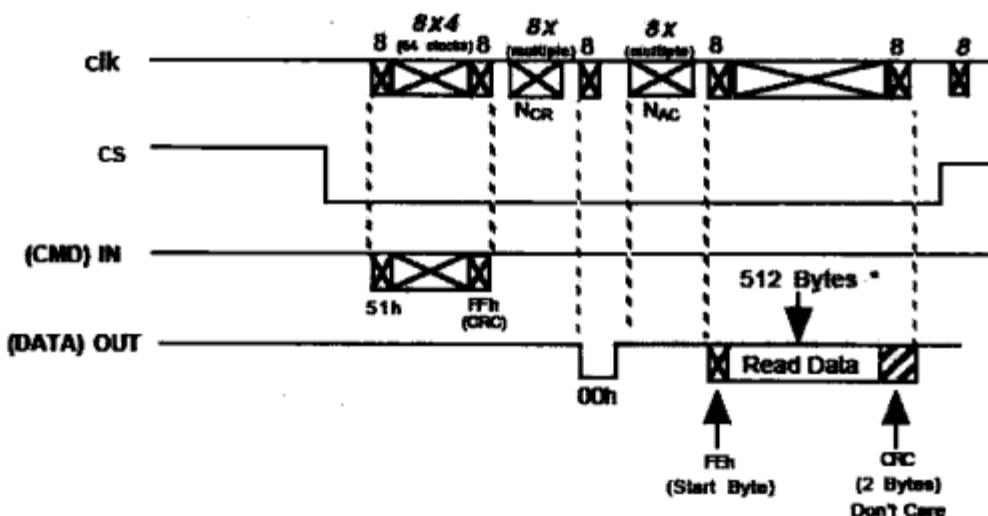


图 4-59 读 MMC 卡单个块字节命令时序

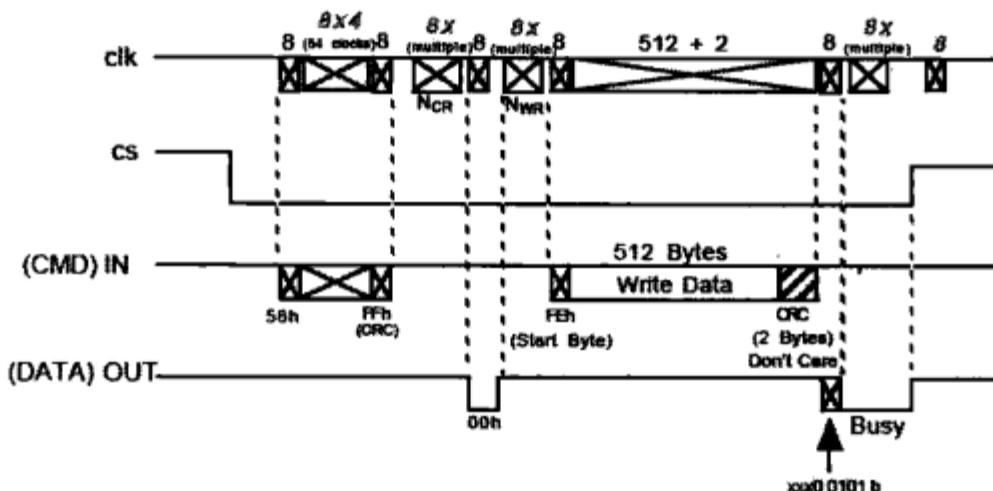


图 4-60 写 MMC 卡单个块字节命令时序

表 4-27 MMC 卡 48 位的命令帧格式

0	1	Bit5~Bit0	Bit31 ~ Bit0	Bit6~0	1
起始位	主机	命令(CMD)	参数(Argument)	CRC	结束位

发送 6 字节的 MMC 卡命令帧时总是以一个起始位开始,以 7 位的冗余校验字段 CRC 及一个停止位结束。命令帧起始 2 位固定为“01”,接下来是 6 位的二进制命令编码,这 8 位构成的字节是发送给 MMC 卡的第一个 CMD 字节。在第一个字节后面是 4 个字节的参数(Argument),参数的具体含义与当前命令相关。

有了上述说明及相关时序图与命令帧格式表,本例 3 个重要的 MMC 卡相关函数就很容易编写与阅读了:

- ① MMC 卡初始化函数:INT8U MMC\_Initialise();
- ② 从 address 地址读取单个块字节数据:INT8U MMC\_Read\_Block(INT32U address);
- ③ 向 address 地址开始写入单个块字节数据函数:INT8U MMC\_Write\_Block(INT32U address, INT8U \* buffer)。

在 Proteus 仿真环境下,读/写 MMC 卡存储介质时,实际上是在访问 MMC 卡的映象文件(Card Image File)。本例将 MMC 卡映象文件设为 my.mmc,该文件名直接填写在属性对话框中即可。运行仿真案例时,如果 my.mmc 文件尚不存在,则 Proteus 将自动创建该文件,在写 MMC 卡后,如果要观察写入情况,除了可通过虚拟终端显示 MMC 卡所保存的字节数据

以外,还可以直接用 UltraEdit 软件打开 my mmc 文件,然后在十六进制方式下查看。

有关 MMC 卡的更多技术细节,大家可进一步参阅 MMC 卡技术手册文件。

## 2. 实训要求

- ① 观察读/写过程中“热插拔”MMC 卡时的运行效果。
- ② 改写程序,使函数能向任意地址位置开始写入指定字节序列。
- ③ 进一步研究 MMC 卡的技术手册文件及 FAT16 文件分配表格式,重新设计本例,使 MMC 卡数据文件既能被单片机读/写,也能被 PC 机的文件管理器直接读/写。

## 3. 源程序代码

```

001 //----- mmc.c -----
002 // 名称: MMC 卡块读/写程序
003 //-----
004 #define F_CPU 4000000UL
005 #include <avr/io.h>
006 #include <util/delay.h>
007 #define INT8U unsigned char
008 #define INT16U unsigned int
009 #define INT32U unsigned long
010
011 //MMC 卡使能与禁止操作
012 #define MMC_CS_EN() PORTB &= ~ _BV(PB4)
013 #define MMC_CS_DI() PORTB |= _BV(PB4)
014
015 //块字节读写缓冲(512 字节)
016 extern INT8U Block_bytes[512];
017 //写 SPI 接口函数
018 extern INT8U SPI_Send(INT8U data);
019
020 //MMC 卡操作命令帧(6 字节,48 位)
021 INT8U cmd[6] = { 0x00,0x00,0x00,0x00,0x00,0x00 };
022 //-----
023 // MMC 命令帧清零
024 //-----
025 void clear_cmd_frame()
026 {
027     INT8U i = 0;
028     for (i = 0; i<6; i++) cmd[i] = 0x00;      //6 字节 MMC 命令帧清零
029 }
030
031 //-----
032 // 写 MMC 命令帧
033 //-----
```

```

034 INT8U MMC_Write_Command(INT8U * cmd_frame)
035 {
036     INT8U i = 0, k = 0, temp = 0xFF;
037     MMC_CS_DI();                                //禁止片选
038     SPI_Send(0xFF);                            //发送 8 个时钟
039     MMC_CS_EN();                             //片选有效
040     //发送 6 字节命令帧(48 位)
041     for(i = 0; i < 6; i++) SPI_Send(cmd_frame[i]);
042     while(temp == 0xFF)
043     {
044         temp = SPI_Send(0xFF);                  //等待响应
045         if(k++ > 200) return temp;           //超时返回
046     }
047     return temp;
048 }
049
050 //-----
051 // MMC 初始化
052 //-----
053 INT8U MMC Initialise()
054 {
055     INT16U timeout = 0;
056     INT8U i = 0;
057     clear_cmd_frame();                         //命令帧清零
058     cmd[0] = 0x40;                            //设置 CMD0(0x40...0x95)(复位)
059     cmd[5] = 0x95;
060     _delay_ms(500);
061     for(i = 0; i < 16; i++) SPI_Send(0xFF);    //发送时钟脉冲
062     if(MMC_Write_Command(cmd) != 0x01) return 0; //发送 MMC 复位命令 CMD0
063     cmd[0] = 0x41;                            //设置 CMD1(0x41...0xFF)(初始化)
064     cmd[5] = 0xFF;
065     while(MMC_Write_Command(cmd) != 0x00)       //发送 MMC 初始化命令 CMD1
066         if(timeout++ > 0xFFFFE) return 0;        //等待初始化完成
067                                         //容量大的 MMC 卡需要较长时间
068     return 1;
069 }
070
071 //-----
072 // 从 address 地址读取单个块字节数据
073 //-----
074 INT8U MMC_Read_Block(INT32U address)
075 {
076     INT16U i;

```

```

077     clear_cmd_frame();                                //命令帧清零
078     cmd[0] = 0x51;                                    //设置 CMD17(0x51...0xFF)(读单个块)
079     cmd[5] = 0xFF;
080     address = address<<9;                          //地址<<9 位,取 512 的整数倍
081     cmd[1] = address>>24;                          //将 address 分解到
082     cmd[2] = address>>16;                          //4 字节的命令帧参数中
083     cmd[3] = address>>8;
084     cmd[4] = address>>0;
085     if(MMC_Write_Command(cmd) != 0x00) return 0;    //发送 CMD17
086     while(SPI_Send(0xFF) != 0xFE) _delay_us(1);    //等待数据接受开始(0xFE)
087     for(i = 0; i < 512; i++)                      //读取块数据(512 字节)
088         Block_bytes[i] = SPI_Send(0xFF);
089     SPI_Send(0xFF);                                //取走 2 字节的 CRC
090     SPI_Send(0xFF);
091     return 1;
092 }
093
094 //-----
095 // 向 address 地址开始写入单个块字节数据(buffer 为数据缓冲指针)
096 //-----
097 INT8U MMC_Write_Block(INT32U address, INT8U * buffer)
098 {
099     INT16U i,Dout;
100     clear_cmd_frame();                            //命令帧清零
101     cmd[0] = 0x58;                                //设置 CMD24(0x58...0xFF)(写单个块)
102     cmd[5] = 0xFF;
103     address = address<<9;                        //地址<<9 位,取 512 的整数倍
104     cmd[1] = address>>24;                        //将 address 分解到
105     cmd[2] = address>>16;                        //4 字节的命令帧参数中
106     cmd[3] = address>>8;
107     cmd[4] = address>>0;
108     if(MMC_Write_Command(cmd) != 0x00) return 0;  //发送 CMD24
109     SPI_Send(0xFF);                                //发送填充字节
110     SPI_Send(0xFE);                                //发送数据开始标志 0xFE
111     //将块读写缓冲 Block_bytes 中的 512 字节数据写入 MMC
112     for(i = 0; i < 512; i++) SPI_Send(Block_bytes[i]);
113     SPI_Send(0xFF);                                //写入 2 字节 CRC
114     SPI_Send(0xFF);
115     Dout = SPI_Send(0xFF) & 0x1F;                  //读取 XXX0 0101 字节
116     if(Dout != 0x05) return 0;                      //如果未能读到 XXX0, 则 0101 写入失败
117     while(SPI_Send(0xFF) == 0x00) _delay_us(1);   //忙等待
118     return 1;
119 }

```

```

01 //----- usart.c -----
02 //名称:串口程序
03 //-----
04 #define F_CPU 4000000UL           //4 MHz 晶振
05 #include <avr/io.h>
06 #include <util/delay.h>
07 #define INT8U unsigned char
08 #define INT16U unsigned int
09 //-----
10 // USART 初始化
11 //-----
12 void Init_USART()
13 {
14     UCSRB = _BV(RXEN) | _BV(TXEN) | _BV(RXCIE);    //允许接收和发送,接收中断使能
15     UCSRC = _BV(URSEL) | _BV(UCSZ1) | _BV(UCSZ0);  //8 位数据位,1 位停止位
16     UBRRH = (F_CPU / 9600 / 16 - 1) % 256;        //波特率:9600
17     UBRLR = (F_CPU / 9600 / 16 - 1) / 256;
18 }
19
20 //-----
21 // 发送 1 个字符
22 //-----
23 void PutChar(char c)
24 {
25     if(c == '\n') PutChar('\r');
26     UDR = c;
27     while(!(UCSRA & _BV(UDRE)));
28 }
29
30 //-----
31 // 发送字符串
32 //-----
33 void PutStr(char * s)
34 {
35     while (*s) PutChar(*s++);
36 }

01 //----- spi.c -----
02 // 名称: SPI 接口程序
03 //-----
04 #define F_CPU 4000000UL
05 #include <avr/io.h>

```



```
06 # include <util/delay.h>
07 # define INT8U unsigned char
08 # define INT16U unsigned int
09 //-----
10 // SPI 初始化
11 //-----
12 void SPI_Initialise()
13 {
14     //4、5、6、7 位分别对应单片机 SS:输出, MOSI:输出, MISO:输入, SCK:输出
15     DDRB = 0B10111111;
16     SPCR |= _BV(SPE) | _BV(MSTR) | _BV(SPR1) | _BV(SPR0);
17     PORTB |= _BV(PB4);
18 }
19
20 //-----
21 // SPI 发送数据
22 //-----
23 INT8U SPI_Send(INT8U data)
24 {
25     INT8U t;
26     SPDR = data;
27     while(!(SPSR & _BV(SPIF))) _delay_us(1);
28     t = SPDR;
29     return t;
30 }
```

```
001 //----- main.c -----
002 // 名称: MMC 存储卡测试
003 //-----
004 // 说明: 本例运行时,按下 K1 将向 MMC 卡第 0 块写入 512 个有序字节,按下 K2 时
005 //       将向第 1 块写入 512 个随机字节,按下 K3 与 K4 时将分别读取并通过虚
006 //       拟终端显示这些字节数据
007 //
008 //-----
009 # define F_CPU 4000000UL
010 # include <avr/io.h>
011 # include <util/delay.h>
012 # include <stdio.h>
013 # include <stdlib.h>
014 # define INT8U unsigned char
015 # define INT16U unsigned int
016 # define INT32U unsigned long
017
```

```

018 //定义按键操作
019 #define K1_DOWN() (PIN_A == (INT8U)_BV(PA1))
020 #define K2_DOWN() (PIN_A == (INT8U)_BV(PA3))
021 #define K3_DOWN() (PIN_A == (INT8U)_BV(PA5))
022 #define K4_DOWN() (PIN_A == (INT8U)_BV(PA7))
023
024 //MMC 相关函数
025 extern void SPI_Initialise();
026 extern INT8U MMC_Initialise();
027 extern INT8U MMC_Read_Block(INT32U address);
028 extern INT8U MMC_Write_Block(INT32U address, INT8U * buffer);
029 //串口相关函数
030 extern void Init_USART();
031 extern void PutChar(char c);
032 extern void PutStr(char * s);
033
034 //当前按键操作代号
035 INT8U OP = 0;
036 //MMC 块字节读/写缓冲
037 INT8U Block_bytes[512];
038 //MMC 卡操作错误标识(为 1 表示正常,为 0 表示出错)
039 INT8U ERROR_Flag = 1;
040 //-----
041 // 以十六进制形式显示所读取的字节
042 //-----
043 void Show_Byt_by_HEX(INT8U * Buffer, INT32U Len)
044 {
045     INT32U i; char s[4];
046     for (i = 0; i < Len; i++)
047     {
048         //每行显示 32 个字节
049         if (i % 32 == 0) PutChar('\r');
050         //将第 i 个字节的十六进制数据转换为字符串 s,注意在 x 后面添加一个空格
051         sprintf(s, "%02X", Buffer[i]);
052         PutStr(s);
053     }
054     PutStr("\r----- Finished! ----- \r");
055 }
056
057 //-----
058 // 主程序
059 //-----
060 int main()

```

```

061  {
062      INT32U i;
063      DDRA = 0x00; PORTA = 0xFF;
064      DDRD = 0xFF; PORTD = 0xFF;
065      //SPI,USART 初始化
066      SPI_Initialise(); Init_USART(); _delay_ms(100);
067      //初始化 MMC
068      PutStr("Initialise MMC, Please Waiting.....");
069      ERROR_Flag = MMC_Initialise();
070      if (ERROR_Flag) PutStr("OK! \r\r"); else PutStr("ERROR! \r\r");
071
072      //提示进行 K1~K4 操作
073      PutStr("Plase Press K1,K2,K3 or K4 to Play MMC Test... \r\r");
074      //设置随机种子
075      srand(300);
076      while(1)
077      {
078          while (PINA == 0xFF); //未按键则等待-----
079          if (K1_DOWN()) OP = 1;
080          else if (K2_DOWN()) OP = 2;
081          else if (K3_DOWN()) OP = 3;
082          else if (K4_DOWN()) OP = 4;
083          //如果上次 MMC 出错则重新初始化 SPI 接口与 MMC 卡
084          if (ERROR_Flag == 0) //-----
085          {
086              PutStr("Re- Initialise MMC, Please Waiting.....");
087              SPI_Initialise();
088              _delay_ms(100);
089              ERROR_Flag = MMC_Initialise();
090              if (ERROR_Flag) PutStr("OK! \r\r");
091              else
092              {
093                  PutStr("ERROR! \r\r"); goto next;
094              }
095          }
096          //根据按键操作代号分别进行操作,因为上述可能的重新初始化会耗费较多时间
097          //如果在这里仍用 K1~K4 的 DOWN 判断时,按键可能已经释放,从而导致判断失效
098          //因此这里使用的是提前获取的按键操作代号
099          if (OP == 1) //-----
100          {
101              PutStr("Write MMC Order Bytes(512).....");
102              //写入 512 字节,0x00~0xFF,0x00~0xFF
103              for(i = 0; i < 512; i++) Block_bytes[i] = (INT8U)i;

```

```

104         ERROR_Flag = MMC_Write_Block(0,Block_bytes);
105         if (ERROR_Flag) PutStr("OK! \r\r");
106         else PutStr("ERROR! \r\r");
107     }
108     else if (OP == 2) //-----
109     {
110         PutStr("Write MMC Random Bytes(512).....");
111         //在 MMC 第 512 字节位置开始写入 512 个随机字节
112         for(i = 0; i < 512; i++) Block_bytes[i] = (INT8U)rand();
113         ERROR_Flag = MMC_Write_Block(512,Block_bytes);
114         if (ERROR_Flag) PutStr("OK! \r\r");
115         else PutStr("ERROR! \r\r");
116     }
117     else if (OP == 3) //-----
118     {
119         PutStr("\r\rRead MMC DEMO: Orderd Bytes.....");
120         //读 MMC 第 0 块数据
121         ERROR_Flag = MMC_Read_Block(0);
122         if (ERROR_Flag)
123         {
124             PutStr(" OK! \r\r");
125             //显示所读取的 512 个字节
126             Show_Byt_by_HEX(Block_bytes,512);
127         }
128         else PutStr(" ERROR * * * ! \r\r");
129     }
130     else if (OP == 4) //-----
131     {
132         PutStr("\r\rRead MMC DEMO: Random Bytes.....");
133         //从 512 字节位置开始读 1 个扇区数据
134         ERROR_Flag = MMC_Read_Block(512);
135         if (ERROR_Flag)
136         {
137             PutStr(" OK! \r\r");
138             //显示所读取的 512 个字节
139             Show_Byt_by_HEX(Block_bytes,512);
140         }
141         else PutStr(" ERROR * * * ! \r\r");
142     }
143     next: while (PIN_A != 0xFF); //等待释放按键
144 }
145 }
```



## 4.35 红外遥控发射与解码仿真

Proteus 提供了兼容 SIRC(索尼红外编码格式)的 IRLINK 组件,这使得本例在虚拟环境下仿真红外遥控收发成为可能。本例运行时,按下发射器上的任何一按键,对应的 12 位编码将被“发送”到接收端的红外接收头,经程序解码后,12-Bit 的编码值将显示在 3 只数码管上。案例电路及部分运行效果如图 4-61 所示。

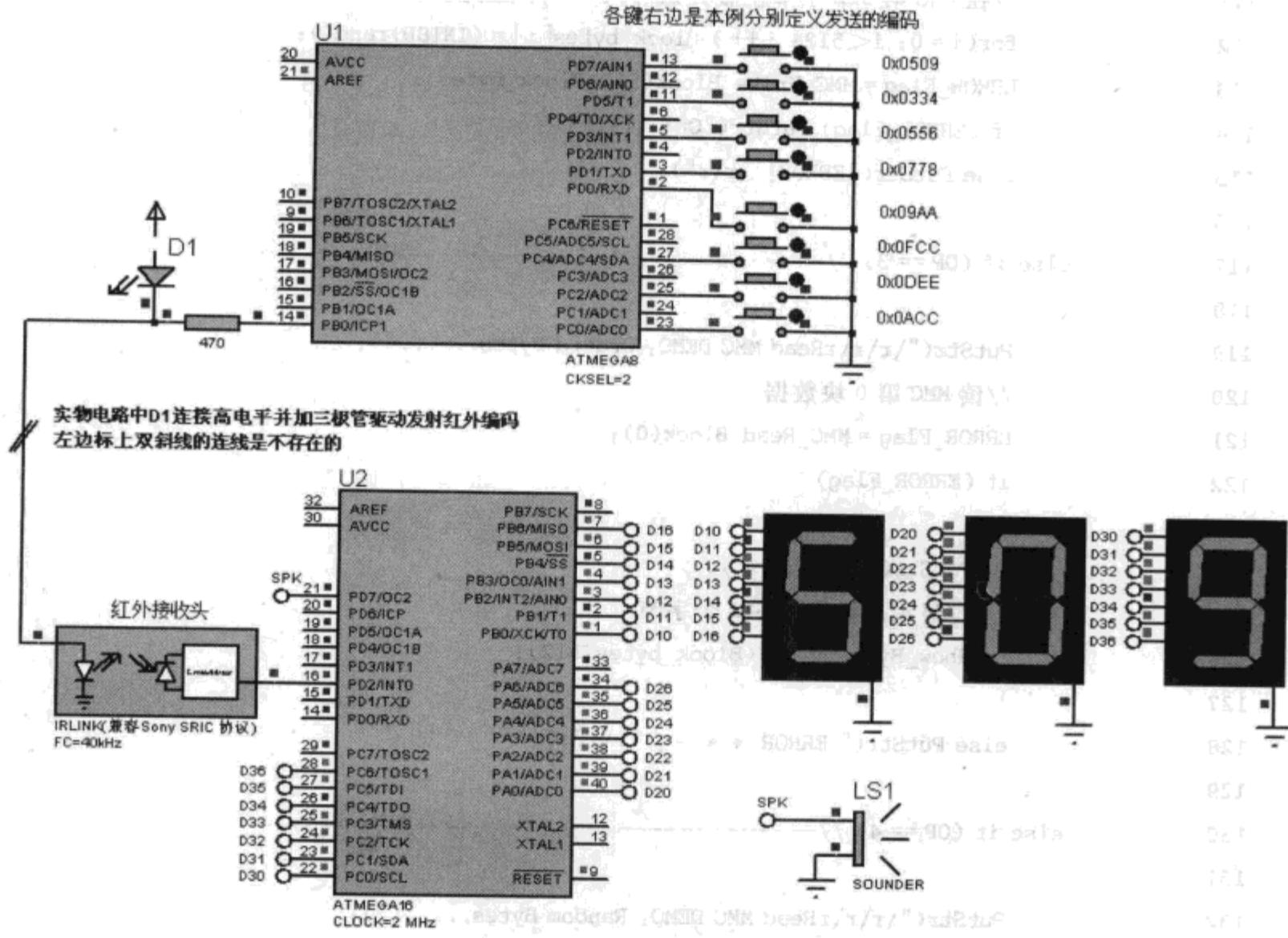


图 4-61 红外遥控发射与解码仿真

### 1. 程序设计与调试

红外光的波长为 950 nm, 低于人眼的可见光谱, 因此我们是看不见这种光线的。

在大量的消费类电子产品中都使用红外遥控器对受控设备进行非接触式的操作控制, 生活中能发出红外光的物体很多, 甚至人体也是能发出红外光的, 为使红外遥控器发出的红外信号能将指定的编码信号发给接收端, 因此接收端所能接收的信号必须区别于噪音信号。

为解决这个问题, 在发送编码时需要将待发送编码进行调制(Modulation)。图 4-62 所示的红外信号发射与接收示意图中, 发送端的 LED 按一定频率闪烁, 而接收端被调谐到这个频率, 因此它仅能够被这个频率“引起注意”而忽略该频率以外的其他信号。这个频率就是收发双方所使用的载波频率。

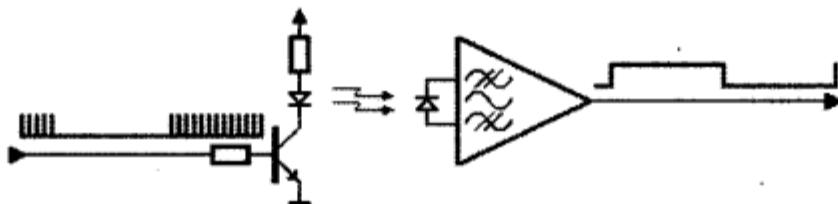


图 4-62 红外信号发射与接收示意图

通过红外信号发送编码时,不同的公司使用了不同的编码格式与协议,例如:JVC Protocol、NEC Protocol、Nokia NRC17、Sharp Protocol、Sony SIRC、Philips RC-5、Philips RC-6、Philips RECS80 等。本例使用的 Proteus 组件 IRLINK 兼容 SONY 的 SIRC 协议。

SIRC 红外控制协议有 3 个版本:12 位版本、15 位版本及 20 位版本。本例程序使用的是 12 位的版本,其中 5 位为地址编码,7 位为命令编码,使用载波频率为 40 kHz。其中地址编码(例如 TV、VCR)与命令编码(例如十频道、一音量等)是预定义的。

SONY 的 SIRC 协议使用脉宽调制(Pulse Width Modulation),使用不同的脉冲宽度来对比特位进行编码。由图 4-63 可知,对于 40 kHz 的载波,它用 1.2 ms 载波脉冲宽度表示逻辑“1”,用 0.6 ms 载波宽度表示逻辑“0”,载波脉冲之间用 0.6 ms 的固定空闲周期进行分隔。

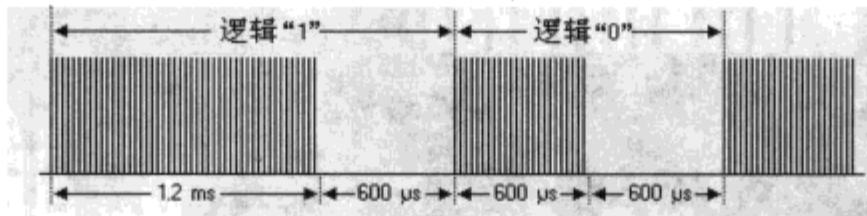


图 4-63 SIRC 分别用 1.2 ms 与 0.6 ms 宽度表示逻辑 1 与逻辑 0

由图 4-64 可知,在发送 12 位的编码之前要先发送 2.4 ms 宽度的脉冲信号作为起始信号,随后是 0.6 ms 的标准空闲间隔周期,接下来再发送 7 位命令与 5 位地址,且都是从低位开始发送。该示意图中所发送的 7 位命令为 19(0010011),5 位地址为 1(00001)。

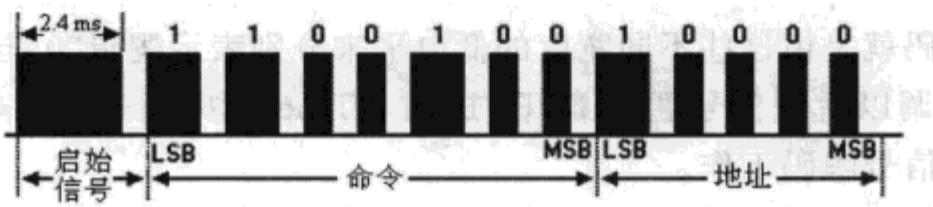


图 4-64 SIRC 红外数据信号格式示例

有了上述知识准备后,开始编写调试本例程序就比较容易了。由于当前版本的 Proteus 中尚没有调制发送 SIRC 载波与编码的仿真器件,本例使用了两片 AVR 单片机,其中 ATmega8 用于生成载波信号,调制(Modulate)发送自定义的 SIRC 编码,另一片单片机 ATmega16 则通过兼容 SIRC 的 IRLINK 组件接收红外信号并进行解调(Demodulate)。前者充当了“红外遥控器”的角色,后者则是受红外遥控器控制的设备。

下面首先讨论为 ATmega8 编写的程序,由于图 4-63 中的载波脉冲宽度有 3 种,即 2.4 ms、1.2 ms、0.6 ms,它们分别是 600 μs 的 4 倍、2 倍、1 倍,为简化设计,程序中首先编写了输出 600 μs 红外载波的子程序 Emit\_IR\_Carrier\_Nx600us(INT8U N),调用时分别给出参数值 4、2、1 即可输出 3 种不同宽度的载波,它们将分别表示起始信号,逻辑“1”与逻辑“0”。

有了子程序 Emit\_IR\_Carrier\_Nx600us 以后,在发送 12 位(7+5)红外编码数据的函数 Emit\_D12(INT16U D12)中就可以很方便地调用它了。函数中首先发送 2.4 ms 起始信号,然



后发送 12 位编码,发送时从低位开始,for 循环语句:

```
for (i = 0x0001; i < 0x1000; i <<= 1)
```

控制了这 12 位编码由低位到高位的逐比特发送过程,每遇到 1 时发送 1.2 ms 宽度载波脉冲,每遇到 0 时则发送 0.6 ms 宽度载波脉冲,每发送完一位后接着送出 0.6 ms 的空闲区,该空间区可分隔所调制的各比特位。

运行本例时,用虚拟示波器的 A 通道和 B 通道观察 IRLINK 的输入/输出端信号时,可观察到如图 4-65 所示的两组波形,上面是发送的调制信号,前面最宽的“白色区域”是 2.4 ms 的载波信号,所有高频率脉冲对应的“白色区域”之间的相间的“黑色区域”是 0.6 ms 的间隔区域,对上面这一组信号从后向前观察,可得到了 010100001001,它就是按下 K1 时发送的编码“509”。虚拟示波器中的第二组波形是通过 IRLINK 解调的结果,其中载波已被去掉,在 ATmega16 接收到的一组脉冲中已经可以清晰的观察到“0”和“1”两种逻辑状态了。

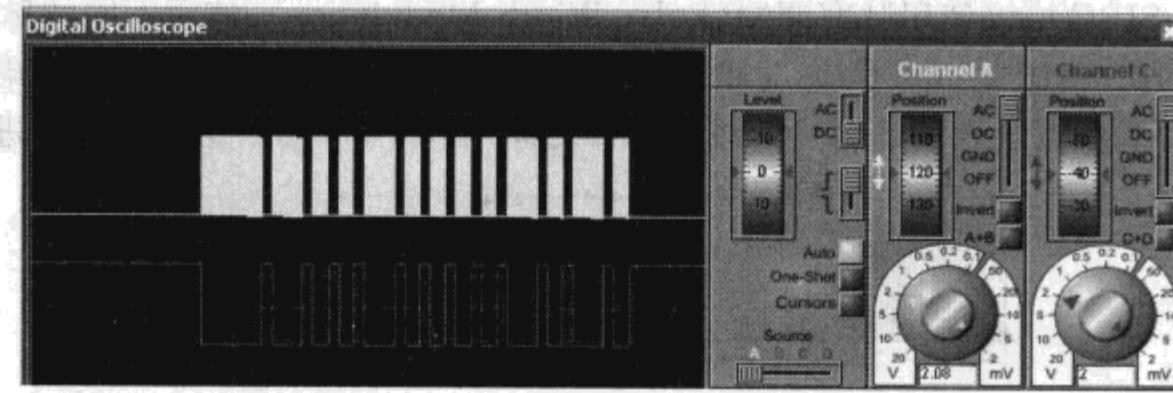


图 4-65 用虚拟示波观察编码“509”的波形

下面再来讨论第二块单片机以接收到第二组信号后应如何解析出对应的 12 位的 SIRC 编码。在第二组波形中,低电平与高电平并非是 SIRC 编码中的“0”和“1”,仔细观察就会发现,所有的高电平与 600  $\mu$ s 的间隔区域对应,它们的宽度全部相同,而所有的低电平则具有不同的宽度,SIRC 编码就是由这些不同宽度的低电平来分别表示逻辑“0”与逻辑“1”的。

由 IRLINK 解调以后的信号通过 INT0 送入 ATmega16,通过编写 INT0 中断程序可进一步完成解调后的信号解码工作。

ATmega16 的程序中第 60 行开始提供了中断函数:ISR (INT0\_vect)。

中断函数在跳过 2.4 ms 的起始信号区域以后开始“收集”12 位的 SIRC 编码,函数中的 for 循环完成了这项工作:

```
for (i = 0; i < 12; i++)  
{...}
```

中断函数内对相关语句给出了非常详细的注释与说明,阅读时可参照图 4-64 仔细分析。

对于解析出来的 12 位编码,本例将其看成 3 个独立字节,将其分别显示在 3 只数码管上,根据 SIRC 编码格式,还可将这 12 位分离为 7 位命令与 5 位地址再进行显示或相应处理。

## 2. 实训要求

- ① 将本例发射与解码程序分别进行适当修改,在实物电路上完成遥控测试。
- ② 分析研究 Nokia NRC17 协议的技术资料,重新编写本例程序,实现 Nokia NRC17 红外编码的发送与接收。

### 3. 源程序代码

```

01 //-----
02 // 名称：红外遥控仿真发射器
03 //-----
04 // 说明：本例运行时，按键键值以 40 kHz 红外线载波发射出去，所模拟的载波
05 // 数据格式符合索尼红外遥控编码格式(SIRC)
06 //
07 //-----
08 #define F_CPU 2000000UL
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 //按键定义
16 #define K1_DOWN() (PIND & _BV(PD7)) == 0x00
17 #define K2_DOWN() (PIND & _BV(PD5)) == 0x00
18 #define K3_DOWN() (PIND & _BV(PD3)) == 0x00
19 #define K4_DOWN() (PIND & _BV(PD1)) == 0x00
20 #define K5_DOWN() (PIND & _BV(PD0)) == 0x00
21 #define K6_DOWN() (PINC & _BV(PC4)) == 0x00
22 #define K7_DOWN() (PINC & _BV(PC2)) == 0x00
23 #define K8_DOWN() (PINC & _BV(PC0)) == 0x00
24
25 //红外发射管定义
26 #define IRLED_BLINK() PORTB ^= _BV(PB0)
27 #define IRLED_1() PORTB |= _BV(PB0)
28 #define IRLED_0() PORTB &= ~_BV(PB0)
29 //-----
30 // 发送 N 倍的 600 μs 载波(1/40K/2≈12 μs)
31 //-----
32 void Emit_IR_Carrier_Nx600us(INT8U N)
33 {
34     INT8U i;
35     for (i = 0; i < N * 50; i++)
36     {
37         _delay_us(12); IRLED_BLINK(); //通过 LED 输出载波脉冲 010101...
38     }
39 }
40
41 //-----

```



```
42 // 发送 12 位数据
43 //-----
44 void Emit_D12(INT16U D12)
45 {
46     INT16U i;
47     //首先发送引导部分 2.4 ms 载波(4 * 600 μs = 2.4 ms)
48     Emit_IR_Carrier_Nx600us(4);
49     IRLED_0(); _delay_us(600); //输出 600 μs 空白区
50     //接着发送 12 位的命令与数据码(7 + 5)
51     for (i = 0x0001; i < 0x1000; i <<= 1)
52     {
53         //从低位开始,每遇到 1/0 时分别输出 1.2 ms/600 μs 载波
54         if (D12 & i)
55             Emit_IR_Carrier_Nx600us(2); //输出 1.2 ms 载波
56         else
57             Emit_IR_Carrier_Nx600us(1); //输出 600 μs 载波
58         //接着输出 600 μs 的低电平
59         IRLED_0(); _delay_us(600);
60     }
61 }
62
63 //-----
64 // 主程序
65 //-----
66 int main()
67 {
68     DDRC = 0x00; PORTC = 0xFF; //配置端口
69     DDRD = 0x00; PORTD = 0xFF;
70     DDRB = 0xFF;
71     while(1) //按键发射编码
72     {
73         if (K1_DOWN()) Emit_D12(0x0509);
74         else if (K2_DOWN()) Emit_D12(0x0334);
75         else if (K3_DOWN()) Emit_D12(0x0556);
76         else if (K4_DOWN()) Emit_D12(0x0778);
77         else if (K5_DOWN()) Emit_D12(0x09AA);
78         else if (K6_DOWN()) Emit_D12(0x0FCC);
79         else if (K7_DOWN()) Emit_D12(0x0DEE);
80         else if (K8_DOWN()) Emit_D12(0x0AAC);
81         _delay_ms(10);
82     }
83 }
```

```

001 //-----
002 // 名称：红外遥控器受控端程序
003 //-----
004 // 说明：程序运行时，根据 SONY 红外协议接收数据并解码，然后将 12 位编码
005 //      以十六进制数形式显示在 3 只数码管上
006 //
007 //-----
008 #define F_CPU 2000000UL
009 #include <avr/io.h>
010 #include <avr/interrupt.h>
011 #include <util/delay.h>
012 #define INT8U unsigned char
013 #define INT16U unsigned int
014
015 //蜂鸣器定义
016 #define BEEP() PORTD ^= _BV(PD7)
017 //读取红外输入信号
018 #define Read_IR() (PIND & _BV(PD2))
019
020 //0~9,A~F 的数码管段码
021 const INT8U SEG_CODE[] =
022 { 0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,
023   0x7F,0x6F,0x77,0x7C,0x39,0x5E,0x79,0x71
024 };
025
026 //接收到的 12 位红外数据及上次接收的数据
027 INT16U IR_D12 = 0x0000, Old_IR_D12 = 0x0000;
028 //12 位二进制编码分解为 3 个十六进制数位
029 INT8U Digit_Buffer[] = {0,0,0};
030 //-----
031 // 输出提示音
032 //-----
033 void Sounder()
034 {
035     INT8U i;
036     for (i = 0; i<200; i++)
037     {
038         BEEP(); _delay_us(280);
039     }
040 }
041
042 //-----
043 // 主程序

```



```
044 //-----  
045 int main()  
046 {  
047     DDRD = ~_BV(PD2); PORTC = 0xFF;      //配置端口  
048     DDRA = 0xFF;                      PORTA = 0x40;  
049     DDRB = 0xFF;                      PORTB = 0x40;  
050     DDRC = 0xFF;                      PORTC = 0x40;  
051     MCUCR = 0x02;                     //INT0 为下降沿触发  
052     GICR |= _BV(INT0);                //INT0 中断使能  
053     SREG  = 0x80;                     //使能总中断  
054     while(1);  
055 }  
056  
057 //-----  
058 // INT0 中断函数 (通过实测,以 122、242 为两个时长的上限)  
059 //-----  
060 ISR (INT0_vect)  
061 {  
062     INT8U i;  
063     INT16U IR_us = 0;                  //红外载波时长  
064     GICR &= ~_BV(INT0);                //禁止外部中断  
065     _delay_ms(2);                    //红外信号引导部分宽度为 2.4 ms  
066     if (Read_IR() != 0x00) goto end;  //如果 2 ms 后已经变为高则退出  
067  
068     //2.4 ms 的前导信号还未接收完成则继续,  
069     //如果红外前导信号确实出现则继续延时,直到出现高电平  
070     //跳过总计 2.4 ms 的前导信号  
071     while (Read_IR() == 0x00)  
072     {  
073         _delay_us(1);  
074         if (++IR_us > 2400) goto end; //异常时退出  
075     }  
076     //收集 12 位编码  
077     for (i = 0; i < 12; i++)  
078     {  
079         //等待 IR 变为低电平,跳过 600 μs 空白区  
080         while (Read_IR() != 0x00)  
081         {  
082             _delay_us(1);  
083             if (++IR_us > 600) goto end; //异常时退出  
084         }  
085         //计算低电平时长  
086         IR_us = 0;
```

```

087     while (Read_IR() == 0x00)
088     {
089         _delay_us(1);
090         if (++IR_us > 300) goto end; //超过该值时异常退出
091     }
092     //12位红外数据的高位默认补0
093     IR_D12 >>= 1;
094     //如果时长为 1200 μs 则在高位补1
095     //通过对本代码检测,两者计时上限分别为:122,242,
096     //故这里选择 150 为 0/1 的分界值
097     if (IR_us > 150) IR_D12 |= 0x0800;
098 }
099 //本例定义的发射端发送来的编码为下列之一:
100 //0x0509/0x0334/0x0556/0x0778/0x09AA/0x0FCC/0x0DEE/0x0AAC
101 //下面将当前编码分解 3 个十六进制数位并显示
102 if (Old_IR_D12 != IR_D12)
103 {
104     Old_IR_D12 = IR_D12;           //保存本次编码
105     PORTC = SEG_CODE[IR_D12 >> 0 & 0x000F]; //分解显示第 3 位
106     PORTA = SEG_CODE[IR_D12 >> 4 & 0x000F]; //分解显示第 2 位
107     PORTB = SEG_CODE[IR_D12 >> 8 & 0x000F]; //分解显示第 1 位
108     Sounder();                  //输出提示音
109 }
110 //重新允许 INT0 中断
111 end: GICR |= _BV(INT0);
112 }

```

# 第5章

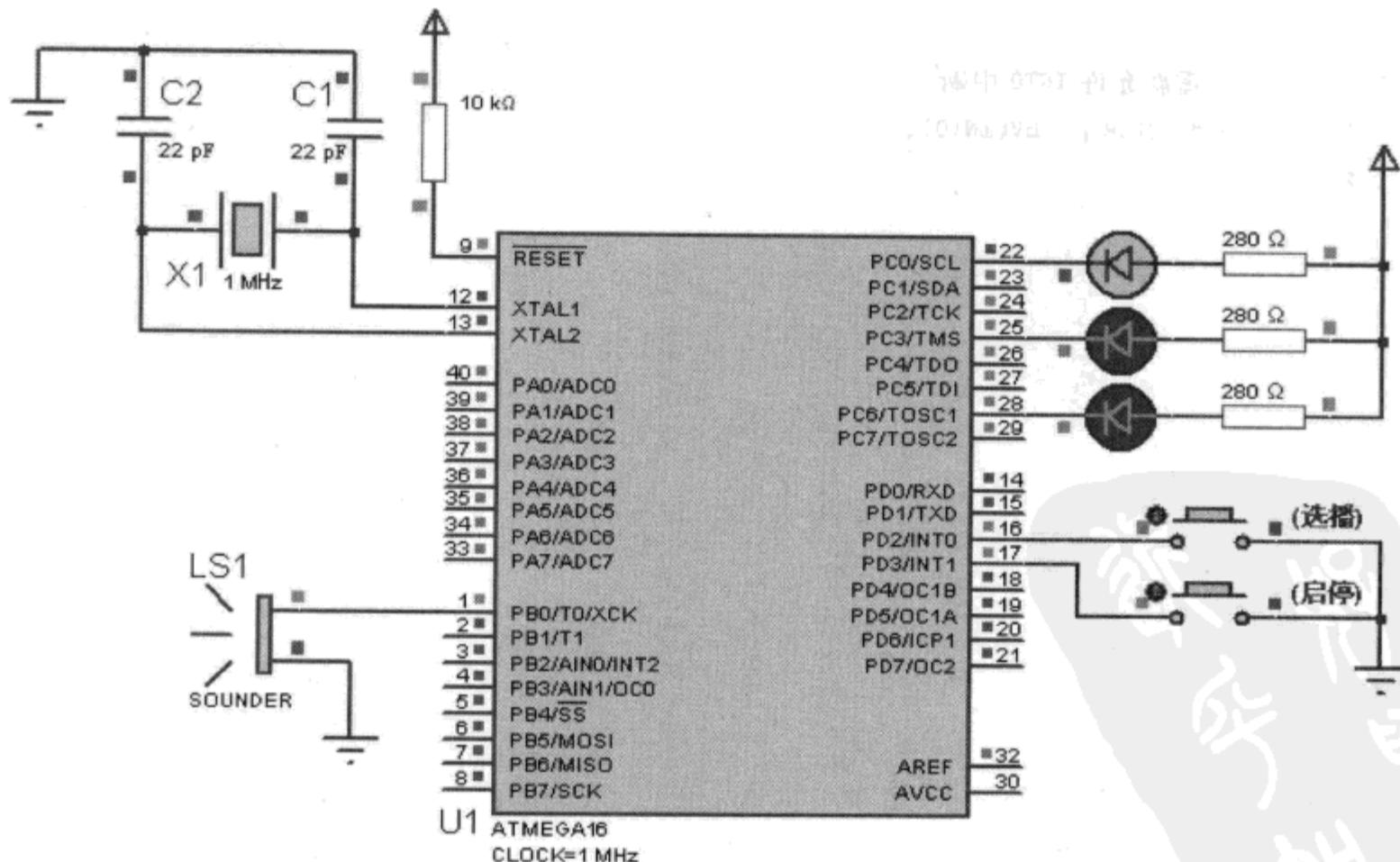
## 综合设计

通过前几章的学习实践,读者已经在 AVR 单片机基础程序设计及外围扩展硬件的程序设计方面打下了很好的基础。本章将在此基础上进一步提出数十个综合设计案例,这些案例有的综合应用了更多的程序设计技术,有的整合应用了更多的外围硬件。通过对本章案例的学习调试与研究,对各案例设计实训要求的独立实践,读者 AVR 单片机 C 程序开发能力会得到进一步锻炼,C 程序设计水平会得到很大提高。

### 5.1 多首电子音乐的选播

本例单片机内置了三段自定义的电子音乐,按下连接 PD 端口的“选播”键时将触发 INT0 中断,中断例程控制切换播放另一段音乐,在选播指定音乐后,按下“启停”键可启动播放,在播放过程中再次按下该键时可终止播放。

本例电路及部分运行效果如图 5-1 所示。



本例三段电子音乐是任意编写,读者可自行修改代码  
每段音乐输出后将停顿2 s,然后继续

图 5-1 多首电子音乐的选播

## 1. 程序设计与调试

本例提供了标准的 0~15 号音符延时表 Tone\_Delay\_Table，输出各音符频率所需要的定时初值推算方法已经在第 3 章有关案例中讨论过了，本例给出的 Tone\_Delay\_Table 数组内容如下：

```
INT16U Tone_Delay_Table[] =
{
    64021, 64103, 64260, 64400, 64524, 64580, 64684, 64777,
    64820, 64898, 64968, 65030, 65058, 65110, 65157, 65178
};
```

各段音乐则分别用 2 个独立数组 MusicX\_Tone、MusicX\_Time 提供。2 个数组分别给出了音符表和节拍表，音符索引由主程序控制，T/C1 定时器根据当前音符索引获取不同的延时初值，从而实现不同声音频率的输出。由于音符索引等全局变量在中断程序中被使用，编写程序时注意添加 volatile 关键字。

为简化 3 段音乐的选播设计，本例程序将 3 个音符数组和 3 个节拍数组名称分别“收集”在两个指针数组 Music\_Tone\_Ptr 与 Music\_Time\_Ptr 中，代码如下：

```
volatile INT8U * Music_Tone_Ptr[] = { Music1_Tone, Music2_Tone, Music3_Tone },
* Music_Time_Ptr[] = { Music1_Time, Music2_Time, Music3_Time };
```

这样处理后，选播按钮驱动的代码设计就可以大大简化了。

另外，本例程序综合应用了 INT0 和 INT1 两个外部中断以及 T/C1 定时器溢出中断。这些中断的综合应用方法已经在第 3 章有关中断案例中出现过，通过本例的调试学习，综合应用多种中断的程序设计能力会得到进一步提高。

## 2. 实训要求

① 用 2 个二维数组分别提供 3 段音乐的音符表与节拍表，并自编一段新的音乐添加到数组中，然后再向电路中添加条形 LED，重新编程使选播不同音乐段时能根据不同声音频率显示不同长度的 LED。

② 本例程序中启动或停止音符输出时都是通过设置 TIMSK 完成的，完成本例调试后，改用设置 TCCR1B 来启停音符输出。

③ 在电路中使用音频功放芯片 LM386 驱动扬声器输出所设计的音乐。

## 3. 源程序代码

```
001 //-
002 // 名称：多首电子音乐的选播
003 //-
004 // 说明：本例运行时，每次按下 K1 将切换播放下一首电子音乐，对应的
005 //       LED 指示灯将被点亮
006 //
007 //-
008 #define F_CPU 1000000UL
009 #include <avr/io.h>
```



```
010 # include <avr/interrupt.h>
011 # include <util/delay.h>
012 # define INT8U unsigned char
013 # define INT16U unsigned int
014
015 //指示灯控制(任一 LED 点亮时都会关闭其他指示灯)
016 # define LED1_ON() PORTC = 0xFE
017 # define LED2_ON() PORTC = 0xF7
018 # define LED3_ON() PORTC = 0xBF
019 //蜂鸣器
020 # define BEEP() PORTB ^= _BV(PB0)
021 //音符延时表,它们分别对应于 0~15 号音符的输出频率
022 INT16U Tone_Delay_Table[] =
023 {
024     64021,64103,64260,64400,64524,64580,64684,64777,
025     64820,64898,64968,65030,65058,65110,65157,65178
026 };
027
028 //第一段(Tone 为音符,Time 为节拍)
029 INT8U Music1_Tone[] =
030 { 3,5,5,3,2,1,2,3,5,3,2,3,5,5,3,2,1,2,3,2,1,1,0xFF };
031 INT8U Music1_Time[] =
032 { 2,1,1,2,1,1,1,2,1,1,1,2,1,1,2,1,1,1,2,1,1,1,0xFF };
033
034 //第二段
035 INT8U Music2_Tone[] =
036 { 1,3,3,3,3,5,4,2,5,3,7,6,5,5,7,4,4,3,6,7,2,1,0xFF };
037 INT8U Music2_Time[] =
038 { 2,1,1,2,1,1,1,2,1,1,3,2,1,1,2,4,1,1,2,1,1,1,0xFF };
039
040 //第三段
041 INT8U Music3_Tone[] =
042 { 0,1,2,3,4,5,5,6,7,8,9,10,11,12,13,14,15,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0xFF };
043 INT8U Music3_Time[] =
044 { 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0xFF };
045
046 //音符与延时指针数组
047 volatile INT8U * Music_Tone_Ptr[] = { Music1_Tone, Music2_Tone, Music3_Tone },
048             * Music_Time_Ptr[] = { Music1_Time, Music2_Time, Music3_Time };
049 //音乐片段索引,音符索引
050 volatile INT8U Music_Idx = 2, Tone_Idx = 0;
051 //从当前数组中取音符的位置
052 volatile INT8U i = 0;
```

```

053 //暂停控制
054 volatile enum bool { FALSE = 0, TRUE = 1 } Pause = TRUE;
055 //-----
056 // 主程序
057 //-----
058 int main()
059 {
060     DDRB = 0xFF;                                //端口配置
061     DDRC = 0xFF;
062     DDRD = ~(_BV(PD2) | _BV(PD3));           //中断引脚设为输入
063     PORTC = 0xFF;                             //LED 初始时全部关闭
064     PORTD = _BV(PD2) | _BV(PD3);             //中断输入引脚设为内部上拉
065     TCCR1B = 0x01;                            //T1 预设分频:1(未分频)
066     MCUCR = 0x82;                            //INT0,INT1 均为下降沿触发
067     GICR = _BV(INT0) | _BV(INT1);           //INT0,INT1 中断许可
068     SREG = 0x80;                            //开中断
069     while (1)
070     {
071         //暂停控制
072         if (Pause) { _delay_ms(200); continue; }
073         //Tone_Idx 是当前音乐片段中的第 i 个音符的序号(取值为 0~15 中的某一个)
074         //它将用于获取对应的延时,以便输出对应的频率
075         Tone_Idx = Music_Tone_Ptr[Music_Idx][i];
076         if (Tone_Idx == 0xFF)
077         {
078             _delay_ms(2000);                      //每段音乐播放结束后停顿一段时间
079             i = 0;                               //回到当前音乐片段的第 0 个音符
080             continue;                          //继续播放
081         }
082         TIMSK = _BV(TOIE1);                  //启动定时器溢出中断,开始输出当前音符
083         //音符输出时长(节拍)由各段音乐中 MusicX_Time 数组中对应音符的延时值决定
084         _delay_ms( Music_Time_Ptr[Music_Idx][Tone_Idx] * 200 );
085         TIMSK = 0x00;                        //禁止定时器溢出中断,停止当前音符输出
086         i++;                                //取音符位置变量 i 递增
087     }
088 }
089
090 //-----
091 // T1 定时器溢出中断控制音符输出
092 //-----
093 ISR (TIMER1_OVF_vect)
094 {
095     //如果遇到音乐片段结束标志则返回

```



```
096     if ( Tone_Idx == 0xFF ) return;
097     //根据 Tone_Delay_Table[ Tone_Idx ]设置定时初值
098     //该初值即决定了输出的频率
099     TCNT1 = Tone_Delay_Table[ Tone_Idx ];
100     BEEP();
101 }
102
103 //-----
104 // 按键触发 INT0 中断,控制音乐段切换
105 //-----
106 ISR (INT0_vect)
107 {
108     TIMSK = 0x00;           //禁止定时器溢出中断,音符输出停止
109     //切换到另一段音乐
110     if (Music_Idx == 2) Music_Idx = 0; else Music_Idx++;
111     //切换到新的一段音乐后总是从其第 0 个音符开始输出
112     i = 0;
113     //打开对应的指示灯
114     switch (Music_Idx)
115     {
116         case 0: LED1_ON(); break;
117         case 1: LED2_ON(); break;
118         case 2: LED3_ON(); break;
119     }
120     _delay_ms(1000);        //在开始另一段音乐输出前暂停 1 s
121     Pause = FALSE;          //取消暂停
122     TIMSK = _BV(TOIE1);    //允许定时器溢出中断,音符输出继续
123 }
124
125 //-----
126 // 播放启/停控制
127 //-----
128 ISR (INT1_vect)
129 {
130     Pause = ! Pause;
131     if (Pause)
132     {
133         PORTC = 0xFF;      //如果停止则关闭所有 LED
134         TIMSK = 0x00;      //停止音符输出
135     }
136     else
137     {
138         //否则打开对应指示灯

```



```

139     switch (Music_Idx)
140     {
141         case 0: LED1_ON(); break;
142         case 1: LED2_ON(); break;
143         case 2: LED3_ON(); break;
144     }
145     //允许定时器溢出中断,输出音符
146     TIMSK = _BV(TOIE1);
147 }
148

```

## 5.2 电子琴仿真

本例程序运行过程中,数码管将显示键盘矩阵按键,蜂鸣器输出按键对应的音符,按键保持时间越长则音符输出时间也越长。调试本例后要熟练掌握键盘矩阵程序与具体应用环境的整合设计方法。本例电路及部分运行效果如图 5-2 所示。

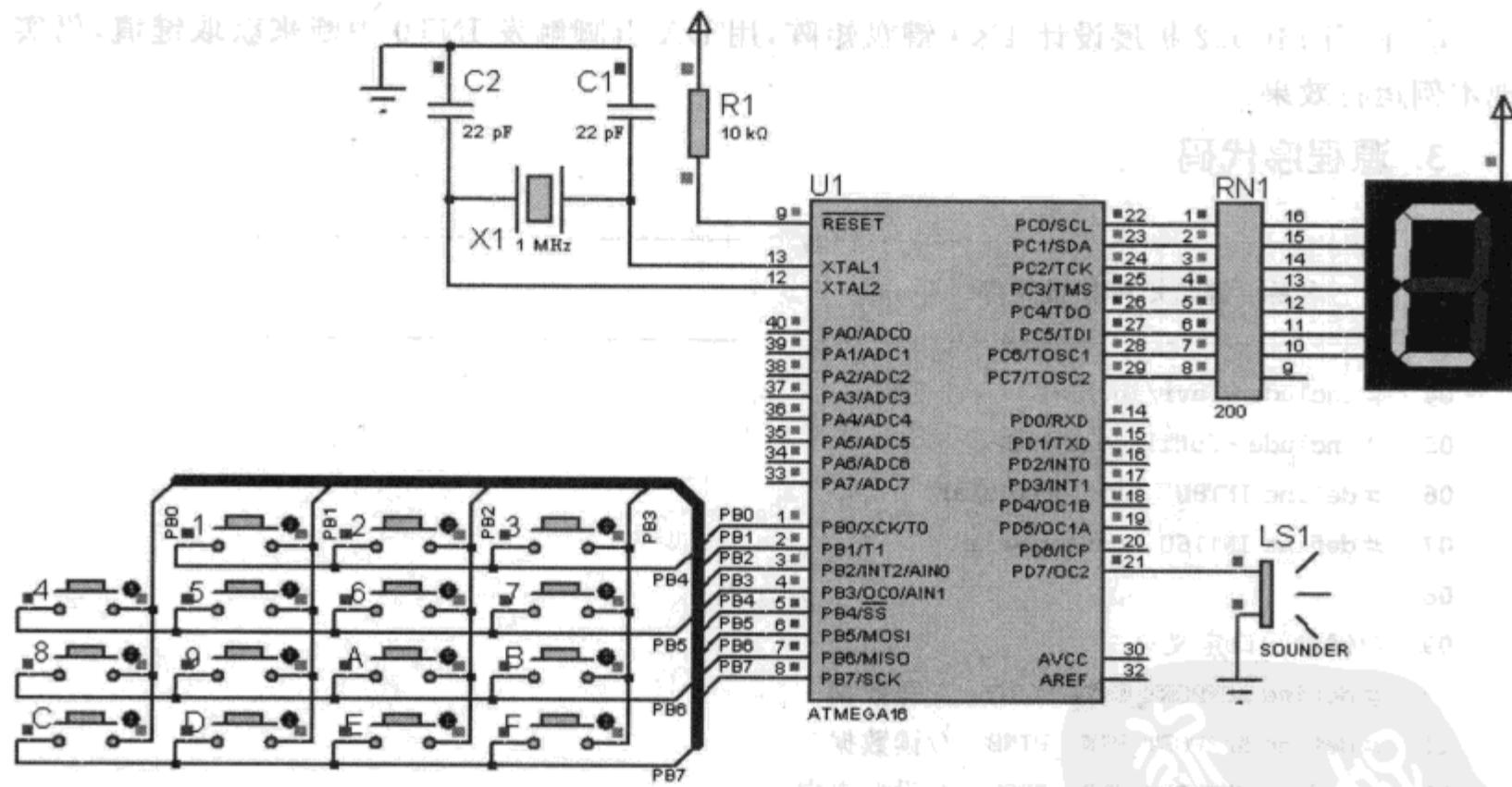


图 5-2 电子琴仿真

### 1. 程序设计与调试

$4 \times 4$  键盘矩阵扫描程序在第 3 章中已经讨论过了,这里不再讨论 Key.c 的程序设计方法。本例主程序持续扫描键盘矩阵,在获取按键号 KeyNo 后,根据该键值可从频率表 TONE\_FRQ 中获取相应频率 TONE\_FRQ[KeyNo],由该频率即可计算出延时值。由于本例 T/C1 预设为 1 分频,因此可得如下延时计算公式:

$$\text{OCR1A} = \text{F\_CPU} / 2 / \text{TONE\_FRQ}[\text{KeyNo}]$$



该延时值直接赋给了 16 位的输出比较寄存器 OCR1A，每次设置 OCR1A 后，紧接着将 TCNT1 置为 0，TCNT1 在 1 分频计数时钟的驱动下从 0 开始不断累加，TCNT1 在每次累加到与 OCR1A 匹配时自动归 0，并由 0 开始重新继续累加，匹配 OCR1A 后再次归 0，如此往复。

主程序在判断按键并设置 OCR1A 与 TCNT1 后，TCNT1 反复累加计数（计时），不断与 OCR1A 匹配，如果允许比较匹配中断，中断程序中的 SPK() 输出 010101… 序列即形成了音符输出，按下不同按键时，OCR1A 的值不同，SPK() 被调用的时间间隔不同，因此输出的频率也不同。主程序通过以下 3 行实现对声音输出的控制：

```
Enable_TIMER1_OCIE();           // 允许 T/C1 比较匹配中断，播放当前音符
while (KeyMatrix_Down());       // 等待释放
Disable_TIMER1_OCIE();          // 释放当前按键后禁止 T/C1 比较匹配中断，停止播放
```

上述 3 行中，第 2 行持续扫描键盘，矩阵按键一旦释放即导致第 3 行被执行，比较匹配中断被禁止，虽然禁止比较匹配中断后，TCNT1 与 OCR1A 的匹配仍不断地出现，但声音却不会输出。

## 2. 实训要求

- ① 修改程序，通过设置 TCCR1B 分频的方法允许或禁止声音输出。
- ② 将蜂鸣器改接在 PD5(OC1A)引脚，重新编写程序，使用非中断方式输出按键音符。
- ③ 使用 74C922 扩展设计 4×4 键盘矩阵，用 DA 引脚触发 INT0 中断来获取键值，仍实现本例运行效果。

## 3. 源程序代码

```
01 //----- Key.c -----
02 // 名称：键盘矩阵扫描程序
03 //-----
04 #include <avr/io.h>
05 #include <util/delay.h>
06 #define INT8U unsigned char
07 #define INT16U unsigned int
08
09 // 键盘端口定义
10 #define KEYPORT_DATA PORTB // 写数据
11 #define KEYPORT_PIN PINB // 读数据
12 #define KEYPORT_DDR DDRB // 设置方向
13 // 当前按键序号，该矩阵中序号范围为 0~15, 16 表示无按键
14 INT8U KeyNo = 16 ;
15 //-----
16 // 判断键盘矩阵是否有键按下
17 //-----
18 INT8U KeyMatrix_Down()
19 {
20     // 高 4 位输出，低 4 位输入，高 4 位先置 0，放入 4 行
```

```

21     KEYPORT_DDR = 0xFO; KEYPORT_DATA = 0xOF; _delay_ms(1);
22     return KEYPORT_PIN != 0xOF ? 1 : 0;
23 }
24
25 //-----
26 // 键盘矩阵扫描子程序
27 //-----
28 void Keys_Scan()
29 {
30     //在判断是否有键按下的函数 KeyMatrix_Down 中,
31     //高 4 位输出,低 4 位输入,高 4 位先置 0,放入 4 行
32     //按键后 00001111 将变成 0000XXXX,X 中有 1 个为 0,3 个仍为 1
33     //下面判断按键发生于 0~3 列中的哪一列
34     switch (KEYPORT_PIN)
35     {
36         case 0B00001110: KeyNo = 0; break;
37         case 0B00001101: KeyNo = 1; break;
38         case 0B00001011: KeyNo = 2; break;
39         case 0B00000111: KeyNo = 3; break;
40         default: KeyNo = 0xFF;
41     }
42
43     //高 4 位输入,低 4 位输出,低 4 位先置 0,放入 4 列
44     KEYPORT_DDR = 0xOF; KEYPORT_DATA = 0xF0; _delay_ms(1);
45
46     //按键后 11110000 将变成 XXXX0000,X 中 1 个为 0,3 个仍为 1
47     //下面对 0~3 行分别附加起始值 0、4、8、12
48     switch (KEYPORT_PIN)
49     {
50         case 0B11100000: KeyNo += 0; break;
51         case 0B11010000: KeyNo += 4; break;
52         case 0B10110000: KeyNo += 8; break;
53         case 0B01110000: KeyNo += 12; break;
54         default: KeyNo = 0xFF;
55     }
56 }
01 //----- main.c -----
02 // 名称: 电子琴仿真
03 //-----
04 // 说明: 本例在键盘矩阵上模拟演奏电子琴,数码管显示键号。
05 //        按下不同按键时将输出不同频率音符,按键长按时发出长音,
06 //        短按时发出短音
07 //

```



```
08 //-----  
09 #define F_CPU 1000000UL           //1 MHz 晶振  
10 #include <avr/io.h>  
11 #include <avr/interrupt.h>  
12 #define INT8U unsigned char  
13 #define INT16U unsigned int  
14  
15 //蜂鸣器定义  
16 #define SPK() (PORTD ^= _BV(PD7))  
17 //定时器比较中断启停定义  
18 #define Enable_TIMER1_OCIE() (TIMSK |= _BV(OCIE1A))  
19 #define Disable_TIMER1_OCIE() (TIMSK &= ~_BV(OCIE1A))  
20  
21 //C 调音符频率表(部分)  
22 const INT16U TONE_FRQ[] =  
23 { 0,262,294,330,349,392,440,494,523,587,659,698,784,880,988,1046 };  
24 //共阳数码管段码表(0~F)  
25 const INT8U SEG_CODE[] =  
26 {  
27     0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8, //0 1 2 3 4 5 6 7  
28     0x80,0x90,0x88,0x83,0xC6,0xA1,0x86,0x8E //8 9 A B C D E F  
29 };  
30 //键盘矩阵相关变量与程序  
31 extern INT8U KeyNo;  
32 extern INT8U KeyMatrix_Down();  
33 extern void Keys_Scan();  
34 //-----  
35 // 主程序  
36 //-----  
37 int main()  
38 {  
39     DDRB = 0x00; PORTB = 0xFF;           //配置端口  
40     DDRC = 0xFF; PORTC = 0xFF;  
41     DDRD = 0xFF; PORTD = 0xFF;  
42     PORTC = 0xBF;                      //数码管初始显示"-"  
43     TCCR1A = 0x00;                     //TC1 与 OC1A 不连接,禁止 PWM 功能  
44     TCCR1B = 0x09;                     //T1 预设分频:1  
45                                         //CTC 模式(比较匹配时 TC1 自动清零)  
46     sei();                            //开中断  
47     while(1)  
48     {  
49         if (KeyMatrix_Down()) Keys_Scan(); //有键按下则扫描按键  
50         else continue;
```

```

51     if (KeyNo == 0 || KeyNo > = 16)          //按键无效或按下 0 键则继续
52         continue;
53     PORTC = SEG_CODE[KeyNo];                //数码管显示键值
54     OCR1A = F_CPU / 2 / TONE_FRQ[KeyNo];    //根据键值对应频率计算延时
55     TCNT1 = 0;
56     Enable_TIMER1_OCIE();                  //允许 T/C1 比较匹配中断,输出当前音符
57     while (KeyMatrix_Down());              //等待释放
58     Disable_TIMER1_OCIE();                //释放当前按键后禁止匹配中断,停止输出
59 }
60 }
61
62 //-----
63 // T1 定时器比较匹配中断程序,控制音符频率输出
64 //-----
65 ISR (TIMER1_COMPA_vect)
66 {
67     SPK();
68 }

```

## 5.3 普通电话机拨号键盘应用

本例设计综合应用了 1602 液晶显示程序与  $4 \times 3$  键盘矩阵扫描程序,案例运行时,所按下的电话号码将显示在 1602 液晶屏上。案例电路及部分运行效果如图 5-3 所示。

### 1. 程序设计与调试

本例设计综合应用了键盘矩阵扫描程序与 1602 字符液晶显示程序。

液晶显示程序可参考阅读此前讨论过的其他有关液晶案例,此略。

下面主要讨论案例中  $4 \times 3$  键盘矩阵的扫描程序,本例给出了 2 个  $4 \times 3$  键盘矩阵扫描函数 GetKey,第一个扫描函数单独给出了键盘扫描码表与键盘特征码表:

```

INT8U KeyScanCode[] = {0xEF, 0xDF, 0xBF, 0x7F};
INT8U KeyCodeTable[] =
{ 0xEE, 0xED, 0xEB, 0xDE, 0xDD, 0xDB, 0xBE, 0xBD, 0xBB, 0x7E, 0x7D, 0x7B };

```

4 个键盘扫描码 0xEF、0xDF、0xBF、0x7F 的高 4 位分别为 1110、1101、1011、0111,它们分别用于扫描键盘的第 0、1、2、3 行。由于每行各有 3 个按键,当其中任何一个按键按下时,各扫描码的返回值对应的也有 3 种,键盘特征码表 KeyCodeTable 给出了共  $4 \times 3$  种特征码。显然,该码表中每 3 个特征码为一组,总共有 4 组,它们分别与 4 个扫描码对应。

有了上述码表后,通过 4 次大循环发送扫描码,再通过 3 次小循环分别检查当前读取的特征码,并将其与 12 个特征码进行比对,如此即可得到当前按键键值。

对于程序中给出的  $4 \times 3$  键盘矩阵的第 2 种扫描函数 GetKey,其设计思想与前一种方法是相同的,只是去掉了扫描码表与特征码表数组及双重循环语句,改用 4 组 switch 来分别扫

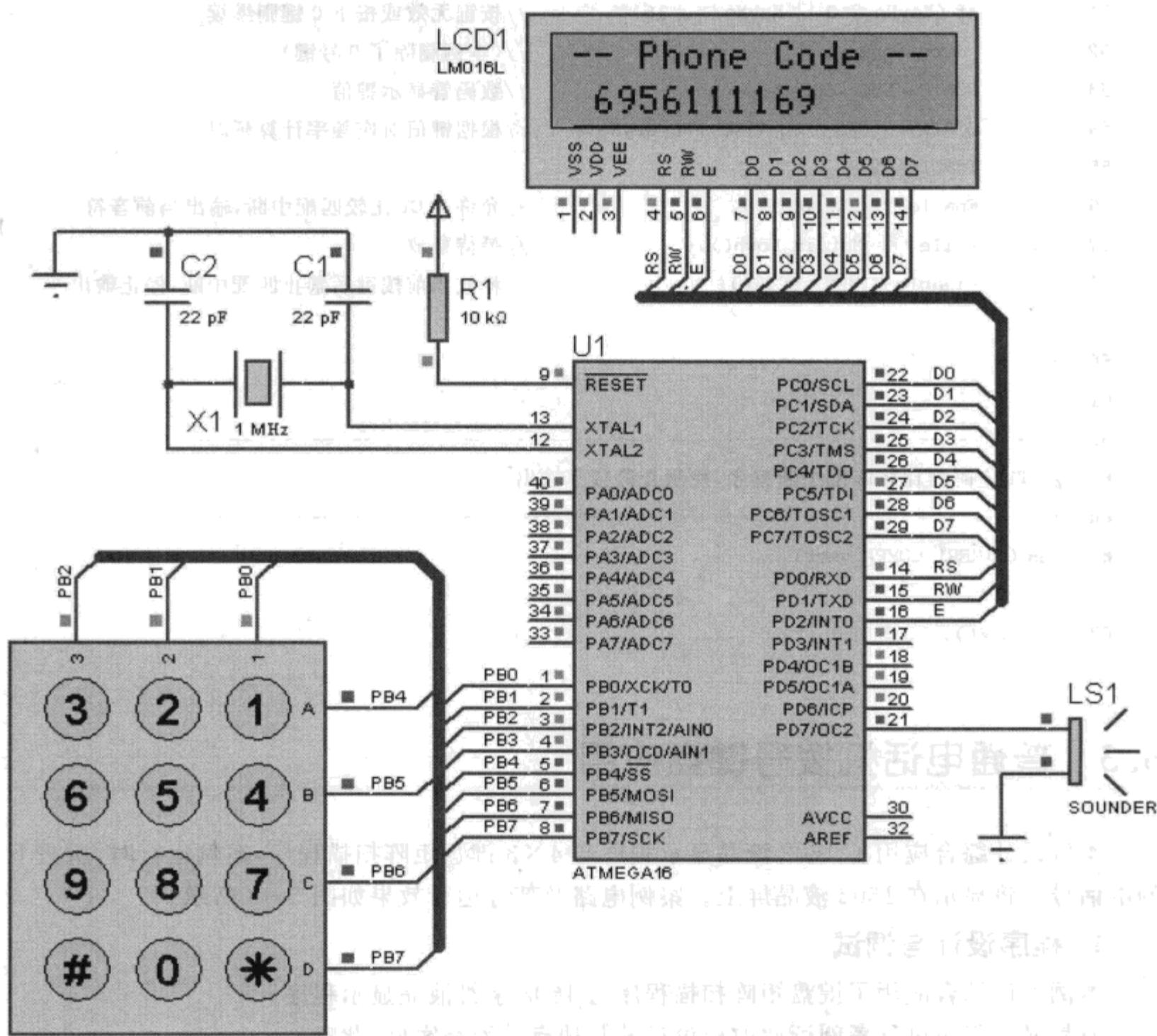


图 5-3 1602LCD 显示电话拨号键盘实验

描 4 行, 再对每行各 3 列中某键按下时的返回特征码进行比对, 比对吻合时即可得到当前键值。

## 2. 实训要求

- ① 修改电路, 将键盘 4 行与 PB0~PB3 连接, 3 列与 PB5~PB7 连接, 重新编写键盘扫描程序实现本例功能。
- ② 在电话拨号键盘中的“\*”号键上实现退格功能, 在“#”号键上实现清除功能。

## 3. 源程序代码

```

01 //----- Key.c -----
02 // 名称: 电话键盘矩阵扫描程序(4 * 3)
03 //-----
04 #define F_CPU 1000000UL

```

```

05 # include <avr/io.h>
06 # include <util/delay.h>
07 # define INT8U unsigned char
08 # define INT16U unsigned int
09
10 //键盘端口定义
11 # define KEYPORT_DATA PORTB
12 # define KEYPORT_DDR DDRB
13 # define KEYPORT_PIN PINB
14 /*
15 //-----
16 // 键盘扫描
17 //-----
18 INT8U GetKey()
19 {
20     INT8U i,j,k = 0;
21     //键盘扫描码
22     INT8U KeyScanCode[] = {0xEF,0xDF,0xBF,0x7F};
23     //键盘特征码
24     INT8U KeyCodeTable[] =
25     { 0xEE,0xED,0xEB,0xDE,0xDD,0xDB,0xBE,0xBD,0xBB,0x7E,0x7D,0x7B };
26     //低 4 位设为输入,高 4 位先放入 4 个 0
27     KEYPORT_DDR = 0xF0; KEYPORT_DATA = 0x0F; _delay_ms(1);
28     //扫描键盘获取按键序号
29     if(KEYPORT_PIN != 0x0F)
30     {
31         for (i = 0; i < 4; i++)
32         {
33             //输出扫描码
34             KEYPORT_DDR = 0xFF;
35             KEYPORT_DATA = KeyScanCode[i];
36             for (j = 0; j < 3; j++)
37
38                 k = i * 3 + j; //i 行 j 列键号为 k
39             //检查低 4 位的变化
40             KEYPORT_DDR = 0xF0;
41             //如果读取的端口值与第 k 个特征码吻合则返回键值 k
42             if (KEYPORT_PIN == KeyCodeTable[k]) return k;
43         }
44     }
45 }
46 else return 0xFF;
47 } */

```



```
48
49 //-----
50 // 本例键盘扫描还可用下面的代码代替
51 //-----
52 INT8U GetKey()
53 {
54     //高 4 位输出,低 4 位输入,高 4 位先放入 4 个 0,低 4 位内部上拉
55     KEYPORT_DDR = 0xF0; KEYPORT_DATA = 0x0F;
56     _delay_ms(1);
57     //扫描键盘获取按键序号
58     if(KEYPORT_PIN != 0x0F)
59     {
60         KEYPORT_DDR = 0xF0;           //高 4 位(4)行设为输出,低 4 位(仅连接 3 列)设为输入
61         KEYPORT_DATA = 0xEF;        //4 行放置扫描码 1110,列设内部上拉
62         _delay_ms(1);
63         switch (KEYPORT_PIN)      //读低 4 位
64         {
65             case 0xEE: return 0;    //低 4 位为 1110,-->0
66             case 0xED: return 1;    //低 4 位为 1101,-->1
67             case 0xEB: return 2;    //低 4 位为 1011,-->2
68         }
69         KEYPORT_DDR = 0xF0;       //高 4 位输出,低 4 位输入
70         KEYPORT_DATA = 0xDF;     //4 行放置扫描码 1101,列设内部上拉
71         _delay_ms(1);
72         switch (KEYPORT_PIN)      //读低 4 位
73         {
74             case 0xDE: return 3;    //低 4 位为 1110,-->3
75             case 0xDD: return 4;    //低 4 位为 1101,-->4
76             case 0xDB: return 5;    //低 4 位为 1011,-->5
77         }
78         KEYPORT_DDR = 0xF0;       //高 4 位输出,低 4 位输入
79         KEYPORT_DATA = 0xBF;     //4 行放置扫描码 1011,列设内部上拉
80         _delay_ms(1);
81         switch (KEYPORT_PIN)      //读低 4 位
82         {
83             case 0xBE: return 6;    //低 4 位为 1110,-->6
84             case 0xBD: return 7;    //低 4 位为 1101,-->7
85             case 0xBB: return 8;    //低 4 位为 1011,-->8
86         }
87         KEYPORT_DDR = 0xF0;       //高 4 位输出,低 4 位输入
88         KEYPORT_DATA = 0x7F;     //4 行放置扫描码 0111,列设内部上拉
89         _delay_ms(1);
90         switch (KEYPORT_PIN)      //读低 4 位
```

```

91     {
92         case 0x7E: return 9;      //低4位为1110,-->9
93         case 0x7D: return 10;    //低4位为1101,-->10
94         case 0x7B: return 11;    //低4位为1011,-->11
95     }
96     return 0xFF;
97 }
98 else return 0xFF;
99 }

01 //----- main.c -----
02 // 名称：普通电话拨号键盘应用
03 //-----
04 // 说明：本例将电话拨号键盘上所拨号码显示在1602液晶屏上
05 //
06 //-----
07 #define F_CPU 1000000UL
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 //液晶及键盘相关函数
15 extern void Initialize_LCD();
16 extern void Write_LCD_Command(INT8U cmd);
17 extern void Write_LCD_Data(INT8U dat);
18 extern void LCD_ShowString(INT8U x, INT8U y, char * str);
19 extern INT8U GetKey();
20
21 //蜂鸣器操作定义
22 #define SPK() PORTD ^= _BV(PD7)
23 //键盘序号与键盘符号映射表
24 const char Key_Table[] = {'1','2','3','4','5','6','7','8','9','*','0','#'};
25 //或写成：
26 //const char Key_Table[] = "123456789 * 0#";
27 //键盘拨号数字缓冲(初始时为17个空格)
28 char Dial_Code_Str[] = {"                 "};
29 //计时累加变量
30 INT16U tCount = 0;
31 //按键键值
32 INT8U KeyNo;
33 //-----
34 // T0 控制按键声音

```

```

35 //-----
36 ISR (TIMER0_OVF_vect)
37 {
38     TCNT0 = 256 - F_CPU / 8.0 * 0.0006;           //重设 600 μs 定时初值
39     SPK();
40     if ( ++tCount == 200)                         //累计 120 ms(600 * 200)后关闭声音
41     {
42         tCount = 0;
43         TIMSK = 0x00;
44     }
45 }
46
47 //-----
48 // 主程序
49 //-----
50 int main()
51 {
52     INT8U i = 0, j;
53     DDRC = 0xFF;                                //配置端口
54     DDRD = 0xFF;
55     DDRB = 0x00; PORTB = 0xFF;
56
57     TCCR0 = 0x02;                               //预设分频:8
58     TCNT0 = 256 - F_CPU / 8.0 * 0.0006;          //晶振 1 MHz,600 μs 定时初值
59     TIMSK = 0x01;                               //允许 T0 定时器溢出中断
60     sei();                                     //开中断
61
62     Initialize_LCD();                          //初始化 LCD
63     LCD_ShowString(0,0, "--- Phone Code ---"); //显示固定信息部分
64     while(1)
65     {
66         //获取按键,无按键时继续扫描
67         if ((KeyNo = GetKey()) == 0xFF) continue;
68         //本例不处理 * 号键与 # 号键
69         if ( KeyNo == 9 || KeyNo == 11) continue;
70         //超过 11 位时清空
71         if ( ++i == 11)
72         {
73             i = 0;
74             for (j = 0; j < 16; j++) Dial_Code_Str[j] = ' ';
75         }
76         //将待显示字符放入待显示的拨号串中
77         Dial_Code_Str[i] = Key_Table[KeyNo];

```

```

78     //在第 2 行显示号码
79     LCD_ShowString(0, 1, Dial_Code_Str);
80     //启动 T0 中断输出按键音
81     TIMSK = 0x01;
82     //等待释放
83     while (GetKey() != 0xFF);
84 }
85 }

```

## 5.4 1602 LCD 显示仿手机键盘按键字符

本例键盘矩阵仿照手机键盘，在每个按键上集成了多个按键字符，可选择输入电话号码或英文字符。案例运行过程中，通过“\*”号键可切换英文/电话号码数字输入，当选择号码输入时，屏幕提示“TEL>”，直接按下各按键时，各键位对应的数字字符将显示在 LCD 上。当选择英文输入时，屏幕提示“ENG>”，多数按键上都安排有多个字符，当依次按下不同按键时，各按键的第一个英文字母将直接显示在 LCD 上；在一个按键上连续按下时，如果时间间隔 $<1.5\text{ s}$ ，程序会允许在该键所有字符中循环选择输入某个字符；如果同一按键按下的时间间隔 $>1.5\text{ s}$ ，则最近显示的字符将被确认显示在 LCD 上；如果在某键上连续快速按下( $<1.5\text{ s}$ )选择了某个字符；当快速按下其他按键( $<1.5\text{ s}$ )时，此键上最后选择的字符也将被确认显示在 LCD 上。本例电路及部分运行效果如图 5-4 所示。

### 1. 程序设计与调试

本例综合了键盘矩阵扫描与 1602 字符液晶显示功能，案例中  $4 \times 3$  的键盘矩阵扫描程序与第 4 章中  $4 \times 4$  键盘矩阵扫描程序设计方法相同，唯一的差别是键盘矩阵减少了一列，因此扫描程序中第一个 switch 语句内也相应减少了一个 case 语句。

与上一案例不同的是本例在同一按键上仿照手机键盘集成了多个按键字符，案例重点在于输入英文字母时，同键 $<1.5\text{ s}$  的连按处理。为完成规定的设计要求，主程序发现所输入的是英文字母且是在同一按键上操作时随即启动定时器开始计时，每次计时超过  $1.5\text{ s}$  时停止计时；主程序中探测到同一按键再次按下时，代码判断 2 次连按的时间间隔是否在  $1.5\text{ s}$  以内，如果在该时间以内则认为是循环选择同一按键上的多个字符，否则将确认输入最后选择的字符。主程序会在每次确认输入一个字符后停止定时器溢出中断且将计时变量 tSpan 归 0，只有遇到同键按下时才启动定时器溢出中断。

在定时器溢出中断程序中，每当计时变量 tSpan 累加超过 30，即超过  $1.5\text{ s}$  时禁止溢出中断，如果此时不禁止中断，这可能使某次连按过程中一次较长的暂停使计时变量 tSpan 不断累加而超过 255 后，再次从 0 开始累加计时，本来是一次较长的停顿就会被误判为一次较短的时间间隔。

对于本例键盘扫描程序的相关细节，大家可以参考程序后面所附带的详细注释仔细分析研究，并思考进一步对源程序设计加以改进。

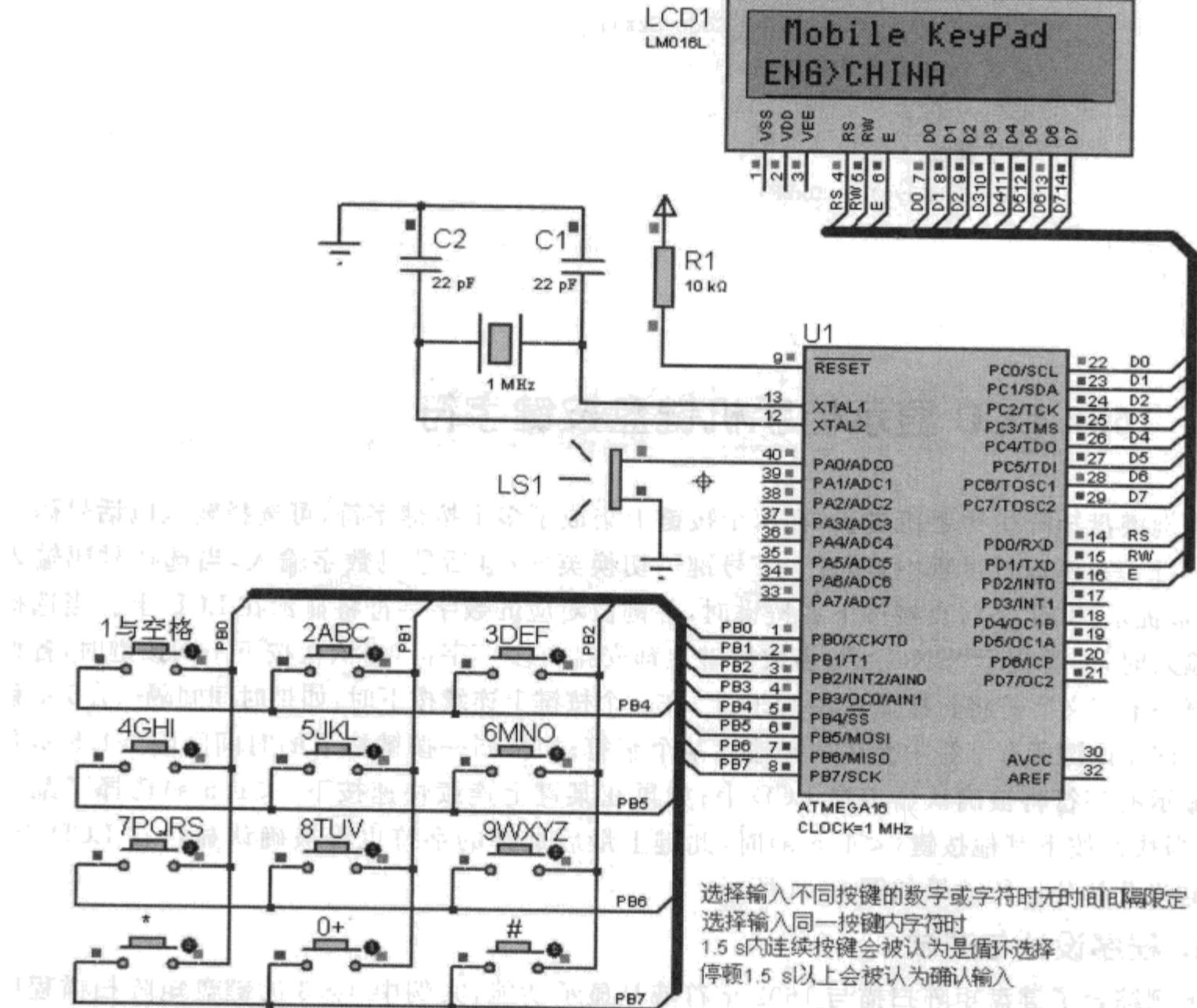


图 5-4 1602 LCD 显示仿手机键盘按键字符

## 2. 实训要求

- ① 进一步修改程序，使本例具备大小写字母输入功能。
- ② 改用 4×4 键盘矩阵解码芯片 74C922 重新设计本例，仍实现仿手机键盘功能。

## 3. 源程序代码

```

01 //----- Key.c -----
02 // 名称：键盘矩阵扫描程序(4x3)
03 //-----
04 #include <avr/io.h>
05 #include <util/delay.h>
06 #define INT8U unsigned char
07 #define INT16U unsigned int
08
09 //键盘端口定义

```

```

10 #define KEYPORT_DATA PORTB
11 #define KEYPORT_DDR DDRB
12 #define KEYPORT_PIN PINB
13 //当前按键序号,该矩阵中序号范围为0~15,0xFF表示无按键
14 INT8U KeyNo = 0xFF;
15 //-----
16 // 判断键盘矩阵是否有键按下
17 //-----
18 INT8U KeyMatrix_Down()
19 {
20     //高4位输出,低4位输入,高4位先置0,放入4行
21     KEYPORT_DDR = 0xF0; KEYPORT_DATA = 0x0F; _delay_ms(1);
22     return KEYPORT_PIN != 0x0F ? 1 : 0;
23 }
24
25 //-----
26 // 4x3 键盘矩阵扫描
27 //-----
28 void Keys_Scan()
29 {
30     //在判断是否有键按下的函数 KeyMatrix_Down 中,
31     //高4位输出,低4位输入,高4位先置0,放入4行
32     //按键后 00001111 将变成 0000XXXX,X中有1个为0,3个仍为1
33     //下面判断按键发生于0~2列中的哪一列
34     switch (KEYPORT_PIN)
35     {
36         case 0B00001110: KeyNo = 0; break;
37         case 0B00001101: KeyNo = 1; break;
38         case 0B00001011: KeyNo = 2; break;
39         default: KeyNo = 0xFF;
40     }
41     //高4位输入,低4位输出,低4位先置0,放入4列
42     KEYPORT_DDR = 0x0F; KEYPORT_DATA = 0xF0; _delay_ms(1);
43     //按键后 11110000 将变成 XXXX0000,X中1个为0,3个仍为1
44     //下面对0~3行分别附加起始值0、3、6、9
45     switch (KEYPORT_PIN)
46     {
47         case 0B11100000: KeyNo += 0; break; //此行可省,这里为了对称而保留
48         case 0B11010000: KeyNo += 3; break;
49         case 0B10110000: KeyNo += 6; break;
50         case 0B01110000: KeyNo += 9; break;
51         default: KeyNo = 0xFF;
52     }

```



```
53 }
001 //----- main.c -----
002 // 名称：手机键盘仿真
003 //-----
004 // 说明：按下仿手机键盘矩阵按键时，对应按键字符显示在 1602 LCD 上
005 // 本例可选择输入电话号码或英语字符序列，实现的效果仿真
006 // 手机的电话或字符串输入(例如使用拼音输入法)效果
007 //
008 //-----
009 #define F_CPU 1000000UL
010 #include <avr/io.h>
011 #include <avr/interrupt.h>
012 #include <util/delay.h>
013 #include <string.h>
014 #define INT8U unsigned char
015 #define INT16U unsigned int
016
017 //液晶及键盘相关函数,相关变量
018 extern void Initialize_LCD();
019 extern void Write_LCD_Command(INT8U cmd);
020 extern void Write_LCD_Data(INT8U dat);
021 extern void LCD_ShowString(INT8U x, INT8U y, char * str);
022 extern void Keys_Scan();
023 extern INT8U KeyMatrix_Down();
024 extern INT8U KeyNo;
025
026 //蜂鸣器定义
027 #define SPK() PORTA ^= _BV(PA0)
028 //12 个键盘按键字符总表(每个按键有 1~5 个字符)
029 //注意串长应设为 6,因为实际最大串长为 5,设为 6 时才能使串尾附带结束标志'\0'
030 //另外,其中第一个字符串中"1"的后面有一个空格
031 const char KeyPad_Chars[12][6] =
032 {"1 ","2ABC","3DEF","4GHI","5JKL","6MNO","7PQRS","8TUV","9WXYZ","* ","0 + ", "#"};
033
034 INT8U Inner_Idx = 0;           //同键位的内部索引
035 INT8U tSpan = 0;               //同键位连续按键的时间间隔
036 char Input_Buffer[16];         //输入缓冲
037 INT8U Buffer_Index = 0;         //缓冲索引
038 INT8U ENG_TEL = 1;             //输入内容切换标识(ENG: 英文输入, TEL: 电话输入)
039 //-----
040 // 蜂鸣器
041 //-----
042 void Beep()
```

```

043 {
044     INT8U i;
045     for(i = 0; i < 30; i++)
046     {
047         SPK(); _delay_us(300);
048     }
049 }
050
051 //-----
052 // 定时器 0 跟踪同位按键的时间间隔 (30 × 50 ms = 1.5 s)
053 //-----
054 ISR (TIMERO_OVF_vect)
055 {
056     //设置定时初始 50 ms
057     TCNT0 = 256 - F_CPU / 1024 * 0.05;
058     //tSpan 最大值限制在 31 以内即可
059     //不加限制时会使某次较长的延时累加使 tSpan 超过 255 后
060     //累加又从 0 开始,而程序判断时它可能刚好还在 30 以内
061     //从而导致较长的延时却被误判为较短的延时
062     if (tSpan < 31) tSpan++; else TIMSK = 0x00;
063 }
064
065 //-----
066 // 功能键处理 *(9): 切换输入, #(11)键清除内容
067 //-----
068 void Function_Key_Process()
069 {
070     if (KeyNo == 9)           //输入内容标识切换
071     {
072         ENG_TEL = !ENG_TEL;
073         Inner_Idx = ENG_TEL ? 1:0; //如果是输入英文,内部索引为 1,否则设为 0
074     }
075     Buffer_Index = 0;        //输入缓冲索引归 0
076     Input_Buffer[0] = '\0';   //将输入缓冲设为空串
077     if (ENG_TEL)
078         LCD_ShowString(0, 1, "ENG>"); //显示输入英文
079     else
080         LCD_ShowString(0, 1, "TEL>"); //显示输入电话
081     while (KeyMatrix_Down()); //等待释放按键
082 }
083 //-----
084 // 主程序
085 //-----

```

```

086 int main()
087 {
088     INT8U Pre_KeyNo = 0xFF;           //上一按键特征码
089     DDRA = 0xFF; PORTA = 0xFF;       //配置端口
090     DDRB = 0x00; PORTB = 0xFF;
091     DDRC = 0xFF;
092     DDRD = 0xFF;
093     TCCR0 = 0x05;                  //预设分频:1024
094     TCNT0 = 256 - F_CPU / 1024.0 * 0.05; //晶振1MHz,50ms定时初值
095     TIMSK = 0x01;                  //允许T0定时器溢出中断
096     sei();                         //开中断
097
098     Initialize_LCD();             //初始化LCD
099     //显示固定信息部分(初始显示ENG>表示输入英文字符序列)
100    LCD_ShowString(0, 0, "Mobile KeyPad ");
101    LCD_ShowString(0, 1, "ENG> ");
102    while(1)
103    {
104        //有键按下则扫描,否则不作任何处理
105        if (KeyMatrix_Down()) Keys_Scan(); else continue;
106        if (KeyNo == 0xFF) continue;
107        //功能键处理(9[*]:切换英文/数字,11[#]:清除所有输入)
108        if (KeyNo == 9 || KeyNo == 11)
109        {
110            Function_Key_Process(); Beep(); continue;
111        }
112
113        //如果是输入数字则直接显示
114        if (!ENG_TEL) goto SHOW_MOBILE_KEY;
115        //如果输入的不是英文字母则继续(英文字符在1~8号键)
116        if (KeyNo < 1 || KeyNo > 8) continue;
117
118        //否则输入的是英文字符序列,以下代码将根据是否为同位按键进行相应处理
119        if (Pre_KeyNo != KeyNo) //按下新按键-----
120        {
121            Pre_KeyNo = KeyNo;           //保存当前按键
122            Inner_Idx = 1;             //输入英文时内部索引起点为1
123        }
124        else //否则按下的是相同位置按键-----
125        {
126            //同位按键时间间隔在50ms * 30 = 1.5s以内则认为是连续按键
127            if (tSpan < 31)
128            {

```

```

129         //连续按键时在键内循环递增字符索引
130         if (++Inner_Idx == strlen(KeyPad_Chars[KeyNo])) Inner_Idx = 1;
131         //因为是连续短按,故将每次显示后递增的输入缓冲索引后退一格,
132         //以便替换此前输入的字符
133         — Buffer_Index;
134     }
135     else Inner_Idx = 1;           //否则按键内英文字符索引回归起点索引 1
136 }
137 tSpan = 0;   TIMSK = 0x01;      //时间间隔归 0,计时开始
138
139 SHOW_MOBILE_KEY:             //显示按键字符
140     if (Buffer_Index >= 12) continue; //输入缓冲限制在 12 个字符以内
141     //更新输入缓冲字符串并送 LCD 显示
142     Input_Buffer[Buffer_Index ++ ] = KeyPad_Chars[KeyNo][Inner_Idx];
143     Input_Buffer[Buffer_Index] = '\0';
144     LCD_ShowString(4, 1, Input_Buffer);
145     Beep();                     //输出提示音
146     while (KeyMatrix_Down());    //等待释放按键
147 }
148 }

```

## 5.5 数码管模拟显示乘法口诀

运行本例时,每次按下 K1 按键可使数码管随机显示乘法口诀,再次按下时显示口诀结果。为使数码管获得较高亮度,本例使用非反向驱动器 7407 驱动数码管显示。本例电路及部分运行效果如图 5-5 所示。

### 1. 程序设计与调试

本例用随机函数随机生成被乘数与乘数,使用随机函数 rand 时注意添加头文件<stdlib.h>。除可以使用随机函数生成口诀题的被乘数与乘数外,还可以使用定时器计数寄存器来获取随机数。

本例数码管用定时器溢出中断刷新显示,编写程序时注意将数码管位索引变量 i 设为静态变量(static),变量 i 在 0~5 的范围内循环取值,定时器溢出中断每隔 4 ms 刷新显示下一位数码管。由于本例使用的是共阴数码管,所输入的位码字节中只能有一位是 0,因此要注意在\_BV(i)的前面添加按位取反运算符“~”。

对于本例电路,在 7407 驱动芯片输出端要注意添加上拉电阻。

### 2. 实训要求

- ① 在电路中添加 4×4 键盘矩阵,用于输入每道口诀题的答案,输入错误时能提示重新输入。
- ② 改用 12864 液晶屏与 4×4 键盘矩阵重新设计本例,液晶屏每屏能随机显示 3 道口诀

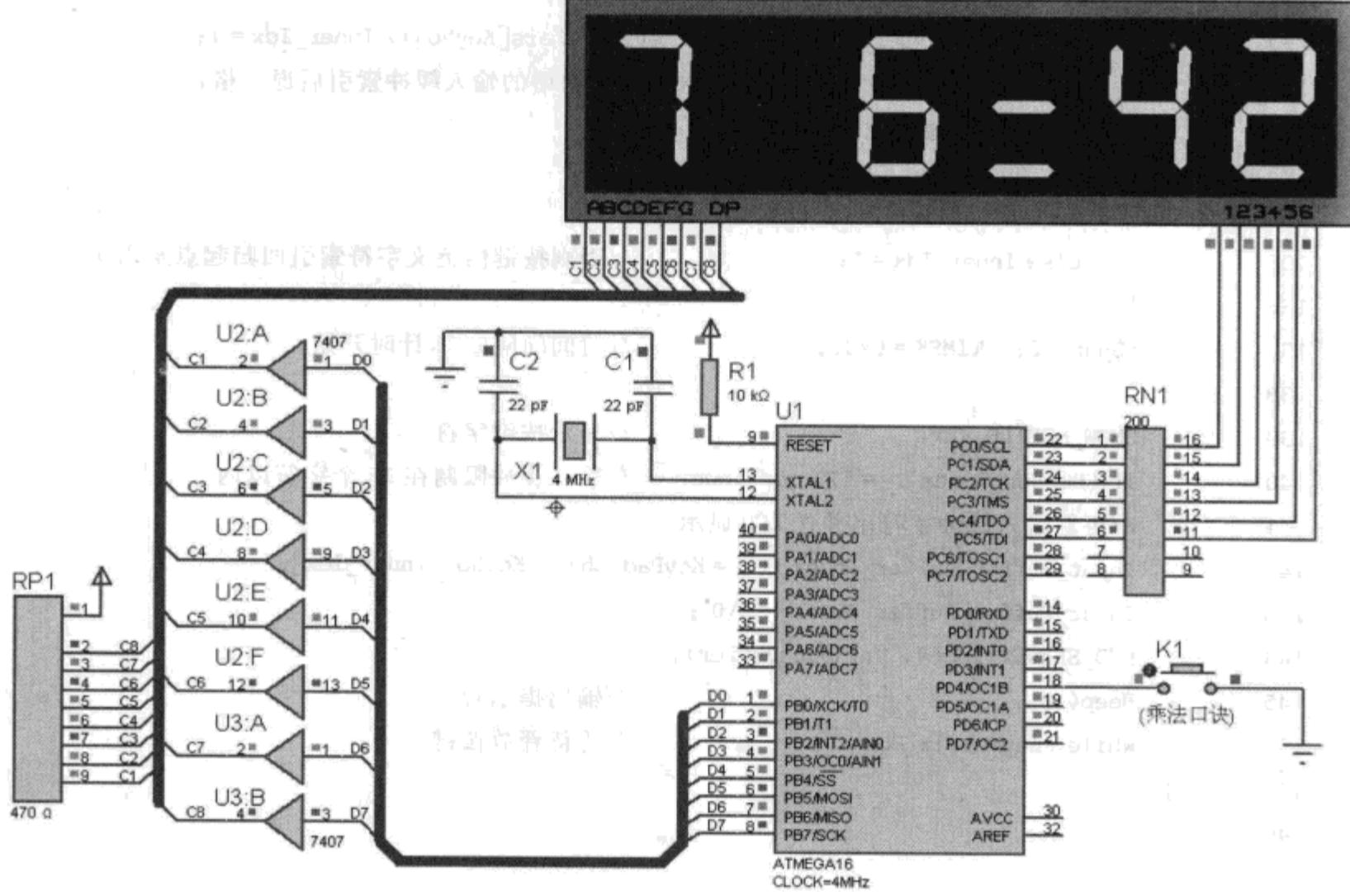


图 5-5 数码管模拟显示乘法口诀

题，通过键盘可分别输入 3 道题的答案。在完成 10 道题后能显示最后成绩，并滚动显示答错的口诀题。

### 3. 源程序代码

```

01 //-----
02 // 名称：数码管随机模拟显示乘法口诀
03 //-----
04 // 说明：每按一下 K1 时会模拟显示一道乘法口诀
05 // 第 1、3 位数码管显示被乘数与乘数
06 // 第 4 位数码管显示等号
07 // 第 5、6 位数码管显示乘积(第 2 次按下 K1 时才显示乘积)
08 //
09 //-----
10 #define F_CPU 4000000UL
11 #include <avr/io.h>
12 #include <avr/interrupt.h>
13 #include <stdlib.h>
14 #define INT8U unsigned char
15 #define INT16U unsigned int
16

```

```

17 //K1 按键判断
18 #define K1_DOWN() ((PIND & _BV(PD4)) == 0x00)
19 //0~9 的共阴数码管段码,最后 3 位 0x00、0x48、0x08 分别是黑屏,等号,下划线
20 //其索引分别为 10,11,12
21 const INT8U SEG_CODE[] =
22 { 0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F,0x00,0x48,0x08 };
23
24 //存放被乘数,乘数,乘积(乘积前面的 11 表示显示的是等号)初始显示"0 0 = 0"
25 INT8U M_ABC[] = {0,10,0,11,10,0};
26 INT8U Result;                                //两数乘积
27 INT8U i = 0;                                 //数码管待显示数字索引
28 //-----
29 // 随机生成被乘数与乘数,计算结果但不显示
30 //-----
31 void Get_Random_Num_A_B()
32 {
33     //随机生成被乘数、乘数
34     M_ABC[0] = rand() % 9 + 1;
35     M_ABC[2] = rand() % 9 + 1;
36     //计算乘积
37     Result = M_ABC[0] * M_ABC[2];
38     //显示结果的 4、5 两位时,先显示下划线,12 是下划线"_"的段码索引
39     M_ABC[4] = 12;
40     M_ABC[5] = 12;
41 }
42
43 //-----
44 // 主程序
45 //-----
46 int main()
47 {
48     INT8U Show_Answer_Flag = 0;                //口诀结果显示标识
49     DDRB = 0xFF;    PORTB = 0xFF;              //配置端口
50     DDRC = 0xFF;    PORTC = 0xFF;
51     DDRD = 0x00;    PORTD = 0xFF;
52     srand(87);                                //设置随机种子
53     TCCR0 = 0x03;                                //预设分频:64
54     TCNT0 = 256 - F_CPU / 64.0 * 0.004;        //晶振 4 MHz,4 ms 定时初值
55     TIMSK = 0x01;                                //允许 T0 定时器溢出中断
56     sei();                                     //开中断
57     while(1)
58     {
59         while (!K1_DOWN());                      //未按下时等待

```



```
60     while ( K1_DOWN() );           //等待释放
61     if ( ! Show_Answer_Flag )      //显示新的乘数、被乘数,不显示答案
62         Get_Random_Num_A_B();
63     else
64     {
65         //分解乘积(在 4、5 位中显示)
66         M_ABC[4] = Result / 10;
67         M_ABC[5] = Result % 10;
68         //当乘积的十位数(即数组中的第 4 位)为 0 时不显示
69         if ( M_ABC[4] == 0 ) M_ABC[4] = 10;
70     }
71     Show_Answer_Flag = ! Show_Answer_Flag; //切换标识
72 }
73 }
74
75 //-----
76 // T0 定时器溢出中断程序(控制数码管扫描显示)
77 //-----
78 ISR (TIMERO0_OVF_vect )
79 {
80     static INT8U i = 0;           //数码管位索引
81     TCNT0 = 256 - F_CPU / 64.0 * 0.004; //位间延时 4 ms
82     PORTC = ~_BV(i);           //发送位扫描码
83     PORTB = SEG_CODE[ M_ABC[i] ]; //发送段码
84     if ( ++i == 6 ) i = 0;       //i 指向 6 位数码管中的下一位
85 }
```

## 5.6 用 DS1302 与数码管设计的可调电子钟

本例用时钟芯片 DS1302 设计可调式电子时钟,当前日期时间用两组 8 位集成式数码管显示,数码管选用 MAX7219 驱动,按键调节部分使用了 74HC148 编码芯片。当有键按下时 GS 引脚触发 INT1 中断(下降沿触发),INT1 中断程序根据不同按键分别实现对日期时间的调节,其中最后 2 个按键用于保存或取消调节。案例电路及部分运行效果如图 5-6 所示。

### 1. 程序设计与调试

本例添加了日期时间的调节与写入功能,程序运行过程中按下任何按键时,编码芯片 74HC148 的 GS 引脚都将触发 INT1 中断,中断程序通过读取编码芯片的 A2~A0 引脚即可判断是哪一按键被按下。

当按下“年、月、日、时、分”按键时,程序进入调整状态,变量 Run\_or\_Adjust 被设为 0;按下“保存”或“取消”按键时,Run\_or\_Adjust 被重新设为 1,程序恢复到正常状态,不断读取并刷新显示当前日期时间。

在调节日期时,为保证日期的合法性,每次修改年、月、日 3 项中的任何一项时,都要调用 Validate\_Date 函数对所调节后的日期进行合法性判断并对数据进行校正。

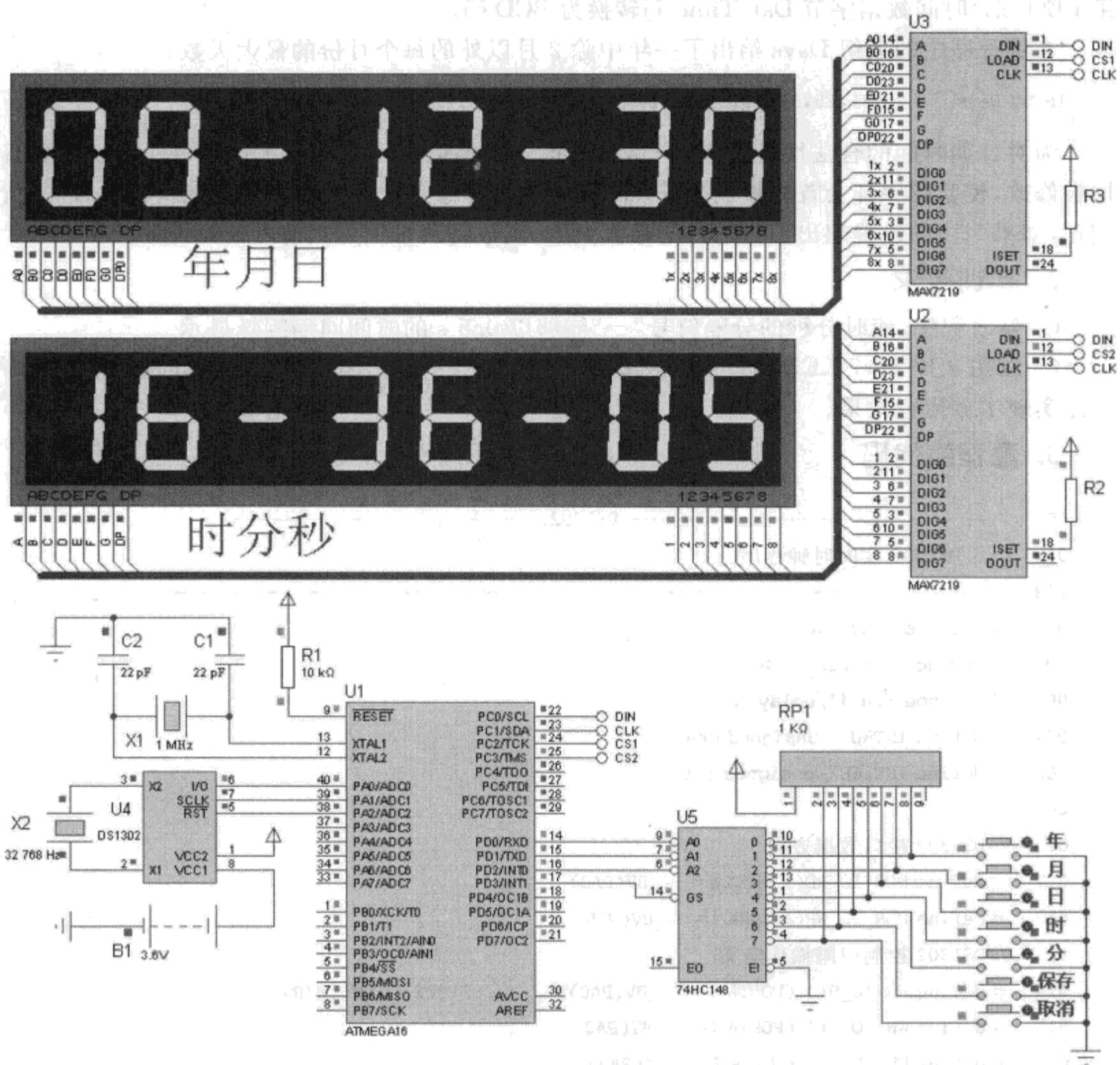


图 5-6 用 DS1302 与数码管设计的可调电子钟

完成日期时间调节后,为将数据写入 DS1302,函数 SetDateTime 首先对 DS1302 解除写保护,这可通过写 DS1302 的控制字节地址 0x8E(10001110)实现,写入字节的最高位为 WP 位(写保护,write-protect),设为 1 为写保护,设为 0 则解除保护。SetDateTime 函数中,第一行解除保存和最后一行加保护的语句如下:

```
Write_DS1302(0x8E,0x00); //写控制字,取消写保护
Write_DS1302(0x8E,0x80); //加保护
```

由于调节后的数据被保存在 DateTime 数组中,SetDateTime 函数通过 for 循环将数组中的值逐一写入 DS1302,写秒、分、时…的地址依次为 0x80、0x82、0x84…,本例中星期数据可不



写入。需要注意的是,DateTime 数组中存放的是十进制的日期时间数据,而 DS1302 的时钟数据是 BCD 码,因此在写入时要通过表达式(DateTime[i]/10<<4) | (DateTime[i] % 10) 将第 i 项日期/时间数据字节 DateTime[i] 转换为 BCD 码。

本例程序中的数组 Days 给出了一年中除 2 月以外的每个月份的最大天数:

```
INT8U Days[] = {0,31,28,31,30,31,30,31,31,30,31,30,31};
```

为对日期时间的合法性进行判断与处理,程序中提供了函数 Validate\_Date(),只要是日期被修改,校验程序都会首先根据当前“年”是否为闰年来获取 2 月的天数,然后再对“日”进行校正,如果“日”的设置超出了当前月的最大天数,函数将其修正为当前月最大天数。

## 2. 实训要求

- ① 修改程序,使时分秒的分隔符号“—”能够以 0.5 s 的时间间隔闪烁显示。
- ② 改用 8 片 5×7 LED 点阵屏显示日期,另外 8 片 5×7 LED 点阵屏显示时间,重新编写程序实现本例运行效果。

## 3. 源程序代码

```
001 //----- DS1302.c -----  
002 // DS1302 实时时钟程序  
003 //-----  
004 #include <avr/io.h>  
005 #include <string.h>  
006 #include <util/delay.h>  
007 #define INT8U unsigned char  
008 #define INT16U unsigned int  
009  
010 //DS1302 接口数据方向  
011 #define DDR_IO_RD() DDRA &= ~_BV(PA0)  
012 #define DDR_IO_WR() DDRA |= _BV(PA0)  
013 //DS1302 控制引脚操作定义  
014 #define WR_IO_0() (PORTA &= ~_BV(PA0)) //DS1302 I/O 线(W/R)  
015 #define WR_IO_1() (PORTA |= _BV(PA0))  
016 #define RD_IO() (PINB &= _BV(PA0))  
017 #define SCLK_1() (PORTA |= _BV(PA1)) //DS1302 时钟线  
018 #define SCLK_0() (PORTA &= ~_BV(PA1))  
019 #define RST_1() (PORTA |= _BV(PA2)) //DS1302 复位线  
020 #define RST_0() (PORTA &= ~_BV(PA2))  
021  
022 //0,1,2,3,4,5,6 分别对应周日,周一~周六(本例未使用)  
023 char * WEEK[] = {"SUN", "MON", "TUS", "WEN", "THU", "FRI", "SAT"};  
024 //所读取的日期时间  
025 INT8U DateTime[7];  
026 //-----  
027 // 向 DS1302 写入 1 字节
```

```

028 //-----
029 void Write_Byte_TO_DS1302(INT8U x)
030 {
031     INT8U i;
032     DDR_IO_WR();                                //写 DS1302 I/O 口
033     for(i = 0x01; i != 0x00 ; i <<= 1)          //写 1 字节(上升沿写入)
034     {
035         if (x & i) WR_IO_1(); else WR_IO_0(); SCLK_0();SCLK_1();
036     }
037 }
038
039 //-----
040 // 从 DS1302 读取 1 字节
041 //-----
042 INT8U Get_Byte_FROM_DS1302()
043 {
044     INT8U i,dat = 0x00;
045     DDR_IO_RD();                                //读 DS1302 I/O 口
046     for(i = 0; i < 8 ; i++)                      //串行读取 1 字节(下降沿读取)
047     {
048         SCLK_1(); SCLK_0();if (RD_IO()) dat |= _BV(i);
049     }
050     return dat / 16 * 10 + dat % 16;              //将 BCD 码转换为十进制数并返回
051     //或  return (dat >> 4) * 10 + (dat & 0x0F); //括号不能省略
052 }
053
054 //-----
055 // 从 DS1302 指定地址读数据
056 //-----
057 INT8U Read_Data(INT8U addr)
058 {
059     INT8U dat;
060     RST_1();                                    //将 RST 拉高
061     Write_Byte_TO_DS1302(addr);                 //向 DS1302 写地址
062     dat = Get_Byte_FROM_DS1302();                //从指定地址读字节
063     RST_0();                                    //将 RST 拉低
064     return dat;
065 }
066
067 //-----
068 // 向 DS1302 指定地址写数据
069 //-----
070 void Write_DS1302(INT8U addr,INT8U dat)

```

```
071  {
072      RST_1();
073      Write_Byte_TO_DS1302(addr);           //写地址
074      Write_Byte_TO_DS1302(dat);           //写数据
075      RST_0();
076  }
077
078 //-----
079 // 读取当前日期时间
080 //-----
081 void GetDateTime()
082 {
083     INT8U i,addr = 0x81;                  //从读秒地址 0x81 开始
084     for (i = 0; i < 7; i++)              //依次读取 7 字节,分别是秒、分、时、日、月、周、年
085     {
086         DateTime[i] = Read_Data(addr);
087         addr += 2;                      //读日期时间地址依次为 0x81、0x83、0x85...
088     }
089 }
090
091 //-----
092 // 设置时间
093 //-----
094 void SetDateTime()
095 {
096     INT8U i;
097     Write_DS1302(0x8E,0x00);           //写控制字节,取消写保护
098     //秒、分、时、日、月、周、年依次写入(因为本例未涉及星期调节,第 5 项可不写入)
099     for(i = 0; i < 7 ; i++)
100    {
101        //秒的起始地址 10000000(0x80),
102        //后续依次是分、时、日、月、周、年,写入地址每次递增 2(即 0x80、0x82、0x84...)
103        //2 位的十进制日期时间数据要转换为 BCD 码后再写入
104        Write_DS1302(0x80 + 2 * i, (DateTime[i]/10<<4) | (DateTime[i] % 10));
105    }
106    Write_DS1302(0x8E,0x80);           //加保护
107 }

1
001 //----- 用 DS1302 与数码管设计的可调式电子钟.c -----
002 // 名称: 用 DS1302 与数码管设计的可调式电子钟
003 //-----
004 // 说明: 本例运行时,当前日期时间将显示在两组数码管上,本例还添加了
005 //       日期时间的调节功能
```

```

006 //-----  

007 //-----  

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 //引脚操作定义  

015 #define DIN_1() PORTC |= (INT8U)_BV(PC0)  

016 #define DIN_0() PORTC &= ~(INT8U)_BV(PC0)  

017 #define CLK_1() PORTC |= (INT8U)_BV(PC1)  

018 #define CLK_0() PORTC &= ~(INT8U)_BV(PC1)  

019 #define CS1_1() PORTC |= (INT8U)_BV(PC2)  

020 #define CS1_0() PORTC &= ~(INT8U)_BV(PC2)  

021 #define CS2_1() PORTC |= (INT8U)_BV(PC3)  

022 #define CS2_0() PORTC &= ~(INT8U)_BV(PC3)  

023  

024 //读/写DS1302日期时间函数及缓存变量  

025 extern void GetDateTime();  

026 extern void SetDateTime();  

027 extern INT8U DateTime[];  

028  

029 //年月日显示缓冲  

030 INT8U YMD_Disp_Buffer[] = {0,0,10,0,0,10,0,0};  

031 //时分秒显示缓冲  

032 INT8U HMS_Dispatcher[] = {0,0,10,0,0,10,0,0};  

033 //继续显示时间或调节时间(1为正常运行,0为调节时间)  

034 volatile INT8U Run_or_Adjust = 1;  

035 //1~12月每月的天数,其中2月天数通过闰年判断再进行调整  

036 volatile INT8U Days[] = {0,31,28,31,30,31,30,31,31,30,31,30,31};  

037 //-----  

038 //向片号为Clip_NO的MAX7219写数据  

039 //-----  

040 void Write(INT8U Addr, INT8U Dat, INT8U Clip_NO)  

041 {  

042     INT8U i;  

043     if (Clip_NO == 1) CS1_0(); else CS2_0();      //第0片或第1片MAX7219片选  

044     for(i = 0; i < 8; i++)                      //串行写入8位地址Addr  

045     {  

046         CLK_0(); if (Addr & 0x80) DIN_1(); else DIN_0();  

047         CLK_1(); _delay_us(2);  

048         CLK_0(); Addr <<= 1;

```



```
049     }
050     for(i = 0; i < 8; i++)           //串行写入 8 位数据 Dat
051     {
052         CLK_0(); if (Dat & 0x80) DIN_1(); else DIN_0();
053         CLK_1(); _delay_us(2);
054         CLK_0(); Dat <<= 1;
055     }
056     if (Clip_N0 == 1) CS1_1(); else CS2_1(); //禁止片选
057 }
058
059 //-----
060 // MAX7221 初始化
061 //-----
062 void Init_MAX72XX(INT8U i)
063 {
064     Write(0x09,0xFF, i);           //编码模式地址 0x09,0x00~0xFF,1 为解码,0 为不解码
065     Write(0x0A,0x07, i);           //亮度地址 0x0A,0x00~0x0F,0x0F 最亮
066     Write(0x0B,0x07, i);           //扫描数码管个数地址 0BH,最多扫描 8 只数码管
067     Write(0x0C,0x01, i);           //工作模式地址 0x0C,0x00:关闭;0x01:正常
068 }
069
070 //-----
071 // 刷新日期时间显示缓冲
072 //-----
073 void Refresh_DateTime_Buffer()
074 {
075     YMD_Disp_Buffer[0] = DateTime[6] / 10; //年
076     YMD_Disp_Buffer[1] = DateTime[6] % 10;
077     YMD_Disp_Buffer[3] = DateTime[4] / 10; //月
078     YMD_Disp_Buffer[4] = DateTime[4] % 10;
079     YMD_Disp_Buffer[6] = DateTime[3] / 10; //日
080     YMD_Disp_Buffer[7] = DateTime[3] % 10;
081     HMS_Disp_Buffer[0] = DateTime[2] / 10; //时
082     HMS_Disp_Buffer[1] = DateTime[2] % 10;
083     HMS_Disp_Buffer[3] = DateTime[1] / 10; //分
084     HMS_Disp_Buffer[4] = DateTime[1] % 10;
085     HMS_Disp_Buffer[6] = DateTime[0] / 10; //秒
086     HMS_Disp_Buffer[7] = DateTime[0] % 10;
087 }
088
089 //-----
090 // 主程序
091 //-----
```

```

092 int main()
093 {
094     INT8U i;
095     DDRD = 0x00; PORTD = 0xFF;           //配置端口
096     DDRA = 0xFF;
097     DDRC = 0xFF;
098     Init_MAX72XX(0);                  //初始化两片 MAX7219
099     Init_MAX72XX(1);
100    MCUCR = 0x08;                     //INT1 下降沿触发
101    GICR = _BV(INT1);                //允许 INT1 中断
102    sei();                           //开中断
103    while (1)
104    {
105        if (Run_or_Adjust) GetDateTime(); //读 DS1302 实时时钟
106        Refresh_DateTime_Buffer();    //刷新日期时间显示缓冲
107
108        for(i = 0; i < 8; i++)         //显示时分秒
109            Write( i + 1, HMS_Displ_Buffer[i],0);
110        for(i = 0; i < 8; i++)         //显示年月日
111            Write( i + 1, YMD_Displ_Buffer[i],1);
112        _delay_ms(200);
113    }
114 }
115
116 //-----
117 // 日期合法性判断及校正
118 //-----
119 void Validate_Date()
120 {
121     //判断是否为闰年,更新 Days 数组中 2 月的天数
122     if ((DateTime[6] / 4 == 0 && DateTime[6] / 100 != 0) ||
123         (DateTime[6] / 400 == 0))
124         Days[2] = 29; else Days[2] = 28;
125     //如果当前月份中设置的天数大于合法的最大天数则将其限制在最大值以内
126     if (DateTime[3] > Days[DateTime[4]]) DateTime[3] = Days[DateTime[4]];
127 }
128
129 //-----
130 // INT1 中断控制日期时间调节
131 //-----
132 ISR (INT1_vect)
133 {
134     //获取按键序号 1~7(0 号键未用)

```



```
135 //由于 74HC148 的 0~7 号输入引脚对应的输出编码是 111~000,而不是 000~111,  
136 //因此要对 PIND 取反,获取的编码是取反后的低 3 位  
137 INT8U i = ~PIND & 0x07;  
138 //如果按下的不是"确认"与"取消"键则进入调节状态,时间停止运行  
139 if (i != 6 && i != 7) Run_or_Adjust = 0;  
140 switch (i) //年、月、日、时、分的调节/保存/取消  
141 {  
142     case 1: if ( ++ DateTime[6] == 99) DateTime[6] = 0; //年(00~99)  
143         Validate_Date();  
144         break;  
145     case 2: if ( ++ DateTime[4] == 13) DateTime[4] = 1; //月(1~12)  
146         Validate_Date();  
147         break;  
148     case 3: if ( ++ DateTime[3] == Days[DateTime[4]] + 1)  
149         DateTime[3] = 1; //日(1~28/29/30/31)  
150         Validate_Date();  
151         break;  
152     case 4: if ( ++ DateTime[2] == 24) DateTime[2] = 0; //时(0~23)  
153         break;  
154     case 5: if ( ++ DateTime[1] == 60) DateTime[1] = 0; //分(00~59)  
155         break;  
156     case 6: SetDateTime(); //将所设置的日期时间写入 DS1302  
157         Run_or_Adjust = 1; //恢复运行  
158         break;  
159     case 7: Run_or_Adjust = 1; //恢复运行,取消所调节的日期时间  
160         break;  
161 }  
162 }
```

## 5.7 用 DS1302 与 LGM12864 设计的可调式中文电子日历

本例运行时,12864 液晶屏将同时显示当前日期、星期及时间信息。调节日期时间数据时,当前所选中的调节对象将被反相显示,日期时间的调节与保存和此前案例设计类似。本例电路及部分运行效果如图 5-7 所示。

### 1. 程序设计与调试

本例程序设计要点在于调节日期时如何使星期能够自动刷新显示。本例程序中,该任务由函数 RefreshWeekDay()完成。

为简化计算,RefreshWeekDay()函数从 1999-12-31 开始推算,已知 1999 年最后一天是星期五,程序从 2000-1-1 开始向后推算,假设当前日期是 yyyy-mm-dd,程序中第一个 for 循环从 2000-1-1 开始推出 yyyy-1-1 是星期几(w),为避免累加天数时出现溢出,本例

采取了边累加边对 7 取余的方法,而不是先累加出总天数,最后再对 7 进行一次取余。

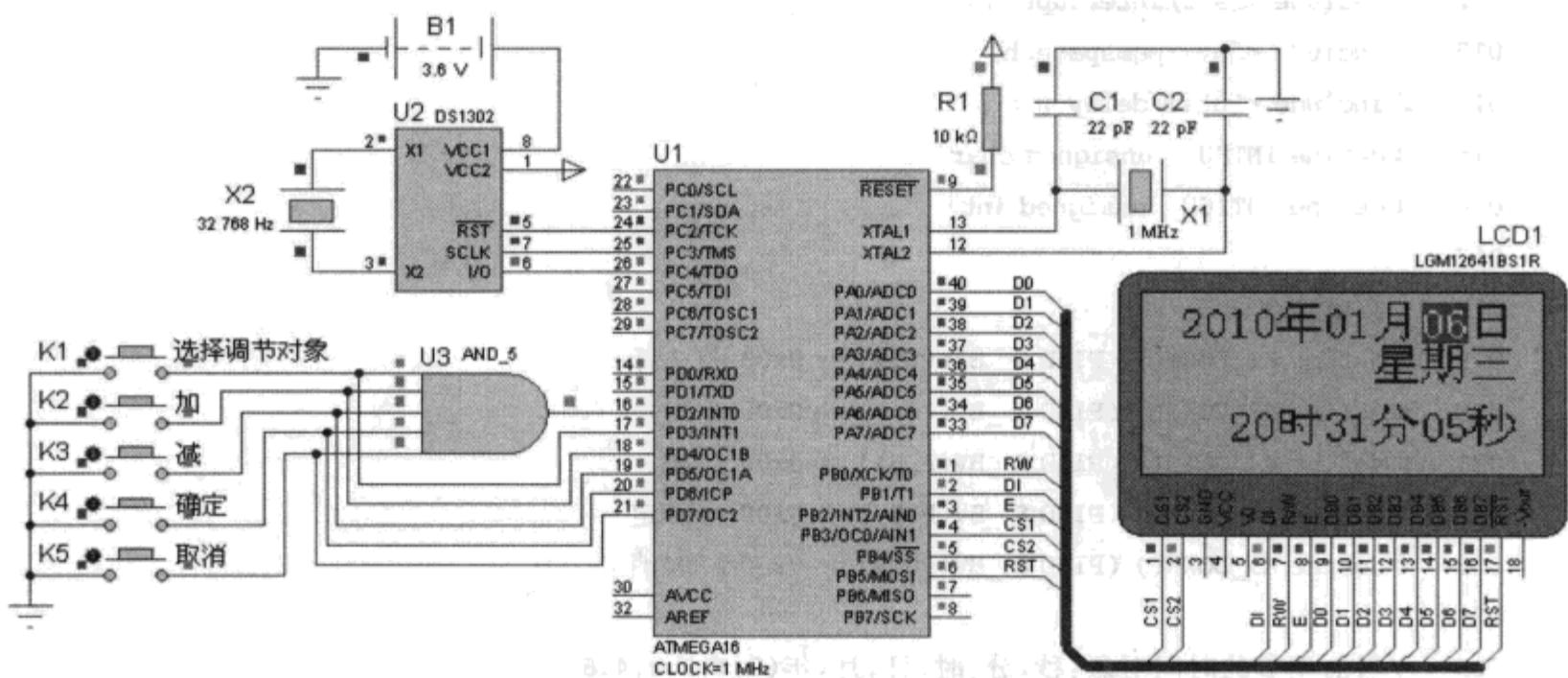


图 5-7 用 DS1302 与 LGM12864 设计的可调式中文电子日历

在推算出  $yyyy - 1 - 1$  是星期几以后,接着先算出从  $yyyy - 1 - 1$  至  $yyyy - mm - dd$  共有多少天,这个天数由变量 d 保存,在得出  $yyyy - 1 - 1$  是星期 w 及  $yyyy - mm - dd$  是当前年的第 d 天以后,由表达式:  $(w+d) \% 7$  可得出  $yyyy - mm - dd$  是星期几,该表达式所得出的值 0 ~ 6 分别对应于星期日、星期一~星期六。由于 DS1302 中的星期日、星期一~星期六对应于数字 1~7,因此实际返回的星期为:  $(w+d) \% 7 + 1$ 。

另外,由于本例所使用的汉字字模点阵数据需要占用较多的内存空间,本例将它们存放于 Flash 程序空间,定义点阵数组时使用了 `prog_uchar` 类型,LCD12864.c 中的程序也作了相应修改。在阅读与调试本例时,要注意与此前其他案例作对比分析。

## 2. 实训要求

- ① 修改程序,使当前被选中的调节对象能闪烁显示。
- ② 将本例扩展中断的与门芯片改成编码芯片 74HC148,仍实现本例所设定的功能。
- ③ 为本例添加闹铃功能,使时钟运行到所设定的时间时能输出自定义的闹铃声音。

## 3. 源程序代码

```

001 //----- main.c -----
002 // 名称: 用 DS1302 与 12864LCD 设计的可调式电子日历与时钟
003 //
004 // 说明: 本例运行时会以 PC 时间为默认时间开始显示,运行过程中可以通过
005 // K1 选择调节对象,所选中的调节对象反相显示,K2、K3 进行加减
006 // K4 保存,K5 则用于取消保存。
007 // 本例自动将日期时间调节控制在合法范围内,星期调节会在调整
008 // 年月日时自动完成,闰年问题也能自动处理
009 //
010 //

```



```
011 # include <avr/io.h>
012 # include <avr/interrupt.h>
013 # include <avr/pgmspace.h>
014 # include <util/delay.h>
015 # define INT8U unsigned char
016 # define INT16U unsigned int
017
018 //按键定义
019 # define K1_DOWN() (PIND & _BV(PD3)) == 0x00 //选择
020 # define K2_DOWN() (PIND & _BV(PD4)) == 0x00 //加
021 # define K3_DOWN() (PIND & _BV(PD5)) == 0x00 //减
022 # define K4_DOWN() (PIND & _BV(PD6)) == 0x00 //确定
023 # define K5_DOWN() (PIND & _BV(PD7)) == 0x00 //取消
024
025 //当前调节的时间对象:秒,分,时,日,月,年(0,1,2,3,4,6)
026 //5 对应星期,星期调节由年月日调节自动完成
027 char Adjust_Index = -1;
028
029 //一年中每个月的天数,2月的天数由年份决定
030 INT8U MonthsDays[] = {0,31,0,31,30,31,30,31,31,30,31,30,31};
031
032 //所读取的日期时间(分别是秒,分,时,日,月,周,年)
033 extern INT8U DateTime[7];
034 //在调节日期时间时,用该位决定是否反相显示
035 extern INT8U Reverse_Display ;
036
037 //12864LCD 及 DS1302 相关函数
038 extern void LCD_Initialize();
039 extern void Display_A_Char_8X16(INT8U P, INT8U L, INT8U * M);
040 extern void Display_A_WORD(INT8U P, INT8U L, INT8U * M);
041 extern void Display_A_WORD_String(INT8U P, INT8U L, INT8U C, INT8U * M);
042 extern void GetDateTime();
043 extern void SetDateTime();
044
045 //以下点阵均用 Zimo 软件提取
046 //年月日,星期,时分秒汉字点阵(16 * 16)-----
047 prog_uchar DATE_TIME_WORDS[] =
048 {
049 /*-----年-----*/
050 0x40,0x20,0x10,0x0C,0xE3,0x22,0x22,0x22,0xFE,0x22,0x22,0x22,0x22,0x02,0x00,0x00,
051 0x04,0x04,0x04,0x04,0x07,0x04,0x04,0x04,0xFF,0x04,0x04,0x04,0x04,0x04,0x04,0x00,
052 /*-----月-----*/
053 0x00,0x00,0x00,0x00,0x00,0xFF,0x11,0x11,0x11,0x11,0x11,0x11,0x11,0x00,0x00,0x00,0x00,
```





```
097 0x10,0x10,0x10,0x10,0x10,0x91,0x12,0xE,0x94,0x10,0x10,0x10,0x10,0x10,0x00,  
098 0x00,0x40,0x20,0x10,0x0C,0x03,0x01,0x00,0x00,0x01,0x02,0x0C,0x78,0x30,0x00,0x00  
099 };  
100  
101 //半角数字点阵(8 × 16) -----  
102 prog_uchar DIGITS[] =  
103 {  
104 0x00,0xE0,0x10,0x08,0x08,0x10,0xE0,0x00,0x00,0xF,0x10,0x20,0x20,0x10,0xF,0x00,//0  
105 0x00,0x10,0x10,0xF8,0x00,0x00,0x00,0x00,0x20,0x20,0x3F,0x20,0x20,0x00,0x00,//1  
106 0x00,0x70,0x08,0x08,0x08,0x88,0x70,0x00,0x00,0x30,0x28,0x24,0x22,0x21,0x30,0x00,//2  
107 0x00,0x30,0x08,0x88,0x88,0x48,0x30,0x00,0x00,0x18,0x20,0x20,0x20,0x11,0x0E,0x00,//3  
108 0x00,0x00,0xC0,0x20,0x10,0xF8,0x00,0x00,0x00,0x07,0x04,0x24,0x24,0x3F,0x24,0x00,//4  
109 0x00,0xF8,0x08,0x88,0x88,0x08,0x08,0x00,0x00,0x19,0x21,0x20,0x20,0x11,0x0E,0x00,//5  
110 0x00,0xE0,0x10,0x88,0x88,0x18,0x00,0x00,0x00,0xF,0x11,0x20,0x20,0x11,0x0E,0x00,//6  
111 0x00,0x38,0x08,0x08,0xC8,0x38,0x08,0x00,0x00,0x00,0x3F,0x00,0x00,0x00,0x00,//7  
112 0x00,0x70,0x88,0x08,0x08,0x88,0x70,0x00,0x00,0x1C,0x22,0x21,0x21,0x22,0x1C,0x00,//8  
113 0x00,0xE0,0x10,0x08,0x08,0x10,0xE0,0x00,0x00,0x31,0x22,0x22,0x11,0xF,0x00 //9  
114 };  
115  
116 INT8U H_Offset = 10, V_Page_Offset = 0; //水平与垂直偏移  
117 //-----  
118 // 判断是否为闰年  
119 //-----  
120 INT8U isLeapYear(INT16U y)  
121 {  
122     return (y % 4 == 0 && y % 100 != 0) || (y % 400 == 0);  
123 }  
124  
125 //-----  
126 // 求自 2000.1.1 开始的任何一天是星期几  
127 // 函数没有通过求出总天数后再求星期几，  
128 // 因为求总天数可能会越出 INT16U 的范围  
129 //-----  
130 void RefreshWeekDay()  
131 {  
132     INT16U i, d, w = 5; //已知 1999.12.31 是周五  
133     //从 2000.1.1 开始推算出当前年 - 1 年.12.31 是星期几(w)  
134     for (i = 2000; i < 2000 + DateTime[6]; i++)  
135     {  
136         d = isLeapYear(i) ? 366 : 365;  
137         w = (w + d) % 7;  
138     }  
139 }
```

```

140 //计算出当前所设置的年月日是该年的第几天(d).
141 for (d = 0, i = 1; i < DateTime[4] ; i++)
142     d += MonthsDays[i];
143 d += DateTime[3];
144
145 //根据 w 与 d 计算出当前年/月/日是星期几
146 //0~6 表示星期日,星期一,二,...,六,为了与 DS1302 的星期匹配
147 //返回值需要加 1,由计算的 0~6 对应返回 1~7
148 DateTime[5] = (w + d) % 7 + 1;
149 }
150
151 //-----
152 // 年月日时分 ++ / --
153 //-----
154 void DateTime_Adjust(char x)
155 {
156     switch (Adjust_Index)
157     {
158         case 6: //年 00~99
159             if (x == 1 && DateTime[6] < 99) DateTime[6]++;
160             if (x == -1 && DateTime[6] > 0) DateTime[6]--;
161             //获取 2 月天数
162             MonthsDays[2] = isLeapYear(2000 + DateTime[6]) ? 29 : 28;
163             //如果年份变化后当前月份的天数大于上限则设为上限
164             if (DateTime[3] > MonthsDays[DateTime[4]])
165                 DateTime[3] = MonthsDays[DateTime[4]];
166             RefreshWeekDay(); //刷新星期
167             break;
168         case 4: //月 01~12
169             if (x == 1 && DateTime[4] < 12) DateTime[4]++;
170             if (x == -1 && DateTime[4] > 1) DateTime[4]--;
171             //获取 2 月天数
172             MonthsDays[2] = isLeapYear(2000 + DateTime[6]) ? 29 : 28;
173             //如果月份变化后当前月份的天数大于上限则设为上限
174             if (DateTime[3] > MonthsDays[DateTime[4]])
175                 DateTime[3] = MonthsDays[DateTime[4]];
176             RefreshWeekDay(); //刷新星期
177             break;
178         case 3: //日 01~28/29/30/31; 调节之前首先根据年份得出该年中 2 月的天数
179             MonthsDays[2] = isLeapYear(2000 + DateTime[6]) ? 29 : 28;
180             //根据当前月份决定调节日期的上限
181             if (x == 1 && DateTime[3] < MonthsDays[DateTime[4]]) DateTime[3]++;
182             if (x == -1 && DateTime[3] > 1) DateTime[3]--;

```

```

183             RefreshWeekDay(); //刷新星期
184         break;
185     case 2: //时
186         if (x == 1 && DateTime[2] < 23) DateTime[2]++;
187         if (x == -1 && DateTime[2] > 0) DateTime[2]--;
188         break;
189     case 1: //分
190         if (x == 1 && DateTime[1] < 59) DateTime[1]++;
191         if (x == -1 && DateTime[1] > 0) DateTime[1]--;
192         break;
193     case 0: //秒
194         if (x == 1 && DateTime[0] < 59) DateTime[0]++;
195         if (x == -1 && DateTime[0] > 0) DateTime[0]--;
196         break;
197     }
198 }
199
200 //-----
201 // 主程序
202 //-----
203 int main()
204 {
205     DDRA = 0xFF; PORTA = 0xFF; //配置端口
206     DDRB = 0xFF; PORTB = 0xFF;
207     DDRC = 0xFF; PORTC = 0xFF;
208     DDRD = 0x00; PORTD = 0xFF;
209     LCD_Initialize(); //初始化 LCD
210     //显示年的固定前两位(20XX)
211     //-----
212     Display_A_Char_8X16(V_Page_Offset, 0 + H_Offset,
213                           (prog_uchar *) (DIGITS + 2 * 16));
214     Display_A_Char_8X16(V_Page_Offset, 8 + H_Offset,
215                           (prog_uchar *) (DIGITS));
216     //显示固定汉字:年月日
217     //-----
218     Display_A_WORD(V_Page_Offset, 32 + H_Offset,
219                     (prog_uchar *) (DATE_TIME_WORDS + 0 * 32));
220     Display_A_WORD(V_Page_Offset, 64 + H_Offset,
221                     (prog_uchar *) (DATE_TIME_WORDS + 1 * 32));
222     Display_A_WORD(V_Page_Offset, 96 + H_Offset,
223                     (prog_uchar *) (DATE_TIME_WORDS + 2 * 32));
224     //显示固定汉字:星期
225     //-----

```

```

226     Display_A_WORD(V_Page_Offset + 2, 64 + H_Offset,
227                         (prog_uchar *) (DATE_TIME_WORDS + 3 * 32));
228     Display_A_WORD(V_Page_Offset + 2, 80 + H_Offset,
229                         (prog_uchar *) (DATE_TIME_WORDS + 4 * 32));
230 //显示固定汉字:时分秒
231 //-----
232     Display_A_WORD(V_Page_Offset + 5, 32 + H_Offset,
233                         (prog_uchar *) (DATE_TIME_WORDS + 5 * 32));
234     Display_A_WORD(V_Page_Offset + 5, 64 + H_Offset,
235                         (prog_uchar *) (DATE_TIME_WORDS + 6 * 32));
236     Display_A_WORD(V_Page_Offset + 5, 96 + H_Offset,
237                         (prog_uchar *) (DATE_TIME_WORDS + 7 * 32));
238
239     TCCR0 = 0x05;                      //预分频:1024
240     TCNT0 = 256 - F_CPU / 1024.0 * 0.05;    //晶振 4 MHz,0.05 s 定时
241     TIMSK = 0x01;                      //使能 T0 中断
242     MCUCR = 0x02;                      //INT0 为下降沿触发
243     GICR = 0x40;                      //INT0 中断使能
244     sei();                            //使能中断
245     while(1)
246     {
247         //如果未执行调节操作则正常读取当前时间
248         if (Adjust_Index == -1) GetDateTime(); else _delay_ms(100);
249     }
250 }
251
252 //-----
253 // 定时器 0 中断刷新 LCD 显示,Reverse_Display 决定当前显示的内容是否反相
254 //-----
255 ISR (TIMERO_OVF_vect)
256 {
257     TCNT0 = 256 - F_CPU / 1024.0 * 0.05;    //晶振 4 MHz,0.05 s 定时
258     //年(后两位)
259     Reverse_Display = Adjust_Index == 6;
260     Display_A_Char_8X16(V_Page_Offset,16 + H_Offset,
261                         (prog_uchar *) (DIGITS + DateTime[6] / 10 * 16));
262     Display_A_Char_8X16(V_Page_Offset,24 + H_Offset,
263                         (prog_uchar *) (DIGITS + DateTime[6] % 10 * 16));
264     //月
265     Reverse_Display = Adjust_Index == 4;
266     Display_A_Char_8X16(V_Page_Offset,48 + H_Offset,
267                         (prog_uchar *) (DIGITS + DateTime[4] / 10 * 16));
268     Display_A_Char_8X16(V_Page_Offset,56 + H_Offset,

```

```

269                         (prog_uchar *) (DIGITS + DateTime[4] % 10 * 16));
270     //日
271     Reverse_Display = Adjust_Index == 3;
272     Display_A_Char_8X16(V_Page_Offset, 80 + H_Offset,
273                         (prog_uchar *) (DIGITS + DateTime[3] / 10 * 16));
274     Display_A_Char_8X16(V_Page_Offset, 88 + H_Offset,
275                         (prog_uchar *) (DIGITS + DateTime[3] % 10 * 16));
276     //星期
277     Reverse_Display = 0;
278     Display_A_WORD(V_Page_Offset + 2, 96 + H_Offset,
279                     (prog_uchar *) (WEEKDAY_WORDS + (DateTime[5] - 1) * 32));
280     //时
281     Reverse_Display = Adjust_Index == 2;
282     Display_A_Char_8X16(V_Page_Offset + 5, 16 + H_Offset,
283                         (prog_uchar *) (DIGITS + DateTime[2] / 10 * 16));
284     Display_A_Char_8X16(V_Page_Offset + 5, 24 + H_Offset,
285                         (prog_uchar *) (DIGITS + DateTime[2] % 10 * 16));
286     //分
287     Reverse_Display = Adjust_Index == 1;
288     Display_A_Char_8X16(V_Page_Offset + 5, 48 + H_Offset,
289                         (prog_uchar *) (DIGITS + DateTime[1] / 10 * 16));
290     Display_A_Char_8X16(V_Page_Offset + 5, 56 + H_Offset,
291                         (prog_uchar *) (DIGITS + DateTime[1] % 10 * 16));
292     //秒
293     Reverse_Display = Adjust_Index == 0;
294     Display_A_Char_8X16(V_Page_Offset + 5, 80 + H_Offset,
295                         (prog_uchar *) (DIGITS + DateTime[0] / 10 * 16));
296     Display_A_Char_8X16(V_Page_Offset + 5, 88 + H_Offset,
297                         (prog_uchar *) (DIGITS + DateTime[0] % 10 * 16));
298 }
299
300 //-----
301 // 键盘中断(INT0)
302 //-----
303 ISR (INT0_vect)
304 {
305     if (K1_DOWN())                                //选择调节对象
306     {
307         if (Adjust_Index == -1 || Adjust_Index == 0) Adjust_Index = 7;
308         Adjust_Index--;
309         //跳过对星期的调节,星期由年/月/日共同决定,不能单独调节
310         if (Adjust_Index == 5) Adjust_Index = 4;
311     }

```

```

312     else if (K2_DOWN() && Adjust_Index != -1)      //加
313         DateTime_Adjust( 1 );
314
315     else if (K3_DOWN() && Adjust_Index != -1)      //减
316         DateTime_Adjust( -1 );
317
318     else if (K4_DOWN() && Adjust_Index != -1)      //确定
319     {
320         //将调节后的时间写入 DS1302 后时间继续正常显示
321         SetDateTime();
322         Adjust_Index = -1;
323     }
324     else if (K5_DOWN() && Adjust_Index != -1)      //取消
325     {
326         //操作索引重设为 -1, 时间继续正常显示
327         Adjust_Index = -1;
328     }
329 }
```

```

001 //----- LCD12864.c -----
002 // 名称: LCD12864 显示驱动程序 (不带字库)
003 //-----
004 #include <avr/io.h>
005 #include <avr/pgmspace.h>
006 #include <util/delay.h>
007 #include <string.h>
008 #define INT8U unsigned char
009 #define INT16U unsigned int
010 //液晶起始行,页,列命令定义
011 #define LCD_START_ROW 0xC0          //起始行
012 #define LCD_PAGE        0xB8          //页指令
013 #define LCD_COL         0x40          //列指令
014 //液晶控制引脚
015 #define RW             PB0           //读/写
016 #define DI             PB1           //数据/命令选择
017 #define E              PB2           //使能
018 #define CS1            PB3           //左半屏选择
019 #define CS2            PB4           //右半屏选择
020 #define RST            PB5           //复位
021 //液晶端口
022 #define LCD_PORT       PORTA          //液晶 DB0~DB7
023 #define LCD_DDR        DDRA          //设置数据方向
024 #define LCD_PIN        PINA          //读状态数据
```



```
025 #define LCD_CTRL PORTB //液晶控制端口
026 //液晶引脚操作定义
027 #define RW_1() LCD_CTRL |= _BV(RW)
028 #define RW_0() LCD_CTRL &= ~_BV(RW)
029 #define DI_1() LCD_CTRL |= _BV(DI)
030 #define DI_0() LCD_CTRL &= ~_BV(DI)
031 #define E_1() LCD_CTRL |= _BV(E)
032 #define E_0() LCD_CTRL &= ~_BV(E)
033 #define CS1_1() LCD_CTRL |= _BV(CS1)
034 #define CS1_0() LCD_CTRL &= ~_BV(CS1)
035 #define CS2_1() LCD_CTRL |= _BV(CS2)
036 #define CS2_0() LCD_CTRL &= ~_BV(CS2)
037 #define RST_1() LCD_CTRL |= _BV(RST)
038 #define RST_0() LCD_CTRL &= ~_BV(RST)
039 //是否反相显示(白底黑字/黑底白字,不同背光的液晶会有不同)
040 INT8U Reverse_Display = 0;
041 //-----
042 // 等待液晶就绪
043 //-----
044 void Wait_LCD_Ready()
045 {
046     Check_Busy:
047     LCD_DDR = 0x00; //设置数据方向为输入
048     LCD_PORT = 0xFF; //内部上拉
049     RW_1(); asm("nop"); DI_0(); //读状态寄存器
050     E_1(); asm("nop"); E_0();
051     if (LCD_PIN & 0x80) goto Check_Busy;
052 }
053
054 //-----
055 // 向 LCD 发送命令
056 //-----
057 void LCD_Write_Command(INT8U cmd)
058 {
059     Wait_LCD_Ready(); //等待 LCD 就绪
060     LCD_DDR = 0xFF; //设置方向为输出
061     LCD_PORT = 0xFF; //初始输出高电平
062     RW_0(); asm("nop"); DI_0(); //写命令寄存器
063     LCD_PORT = cmd; //发送命令
064     E_1(); asm("nop"); E_0();
065 }
066
067 //-----
```

```

068 // 向 LCD 发送数据
069 //-----
070 void LCD_Write_Data(INT8U dat)
071 {
072     Wait_LCD_Ready();           //等待 LCD 就绪
073     LCD_DDR = 0xFF;            //设置方向为输出
074     LCD_PORT = 0xFF;           //初始输出高电平
075     RW_0(); asm("nop"); DI_1(); //写数据寄存器
076     //发送数据,根据 Reverse_Display 决定是否反相显示
077     if ( !Reverse_Display) LCD_PORT = dat; else LCD_PORT = ~dat;
078     E_1(); asm("nop"); E_0();
079 }
080
081 //-----
082 // 初始化 LCD
083 //-----
084 void LCD_Initialize()
085 {
086     CS1_1(); CS2_1();
087     LCD_Write_Command(0x38); _delay_ms(15);
088     LCD_Write_Command(0x0F); _delay_ms(15);
089     LCD_Write_Command(0x01); _delay_ms(15);
090     LCD_Write_Command(0x06); _delay_ms(15);
091     LCD_Write_Command(LCD_START_ROW); _delay_ms(15);
092 }
093
094 //-----
095 //
096 // 通用显示函数
097 //
098 // 从第 P 页第 L 列开始显示 W 个字节数据,数据在 r 所指向的缓冲
099 // 每字节 8 位是垂直显示的,高位在下,低位在上
100 // 每个 8 * 128 的矩形区域为一页
101 // 整个 LCD 又由 64 * 64 的左半屏和 64 * 64 的右半屏构成
102 //-----
103 void Common_Show(INT8U P, INT8U L, INT8U W, prog_uchar * r)
104 {
105     INT8U i;
106     //显示在左半屏或左右半屏
107     if( L < 64 )
108     {
109         CS1_1(); CS2_0();
110         LCD_Write_Command( LCD_PAGE + P );

```



```
111     LCD_Write_Command( LCD_COL + L );
112     //全部显示在左半屏
113     if( L + W < 64 )
114     {
115         for(i = 0;i < W;i++) LCD_Write_Data(pgm_read_byte(r + i));
116     }
117     //如果越界则跨越左右半屏显示
118     else
119     {
120         //左半屏显示
121         for(i = 0;i < 64 - L;i++) LCD_Write_Data(pgm_read_byte(r + i));
122         //右半屏显示
123         CS1_0(); CS2_1();
124         LCD_Write_Command( LCD_PAGE + P );
125         LCD_Write_Command( LCD_COL );
126         for(i = 64 - L;i < W;i++) LCD_Write_Data(pgm_read_byte(r + i));
127     }
128 }
129 //全部显示在右半屏
130 else
131 {
132     CS1_0(); CS2_1();
133     LCD_Write_Command( LCD_PAGE + P );
134     LCD_Write_Command( LCD_COL + L - 64 );
135     for( i = 0;i < W; i++) LCD_Write_Data(pgm_read_byte(r + i));
136 }
137 }
138 //-----
139 // 显示一个 8 * 16 点阵字符
140 //-----
141 //-----
142 void Display_A_Char_8X16(INT8U P, INT8U L,prog_uchar * M)
143 {
144     Common_Show( P,      L, 8, M );           //显示上半部分 8 * 8
145     Common_Show( P + 1, L, 8, M + 8 );        //显示下半部分 8 * 8
146 }
147 //-----
148 //-----
149 // 显示一个 16 * 16 点阵汉字
150 //-----
151 void Display_A_WORD(INT8U P, INT8U L,prog_uchar * M)
152 {
153     Common_Show( P,      L, 16, M );          //显示汉字上半部分 16 * 8
```

```

154     Common_Show( P + 1, L, 16, M + 16);           //显示汉字下半部分 16 * 8
155 }
156
157 //-----
158 // 显示一串 16 * 16 点阵汉字
159 //-----
160 void Display_A_WORD_String(INT8U P, INT8U L, INT8U C, prog_uchar * M)
161 {
162     INT8U i;
163     for (i = 0; i < C; i++)
164     {
165         Display_A_WORD(P, L + i * 16, M + i * 32);
166     }
167 }

```

## 5.8 用 PG12864LCD 设计的指针式电子钟

本例采用 PG12864 液晶屏作为显示元件,液晶屏模拟表盘与时分秒指针显示当前时钟,液晶屏右边同时以数字方式显示当前时间,在调节时间时,当前被选中的调节对象下面将出现下划线,在保存或取消调节后下划线消失,时钟继续正常运行。本例电路及部分运行效果如图 5-8 所示。

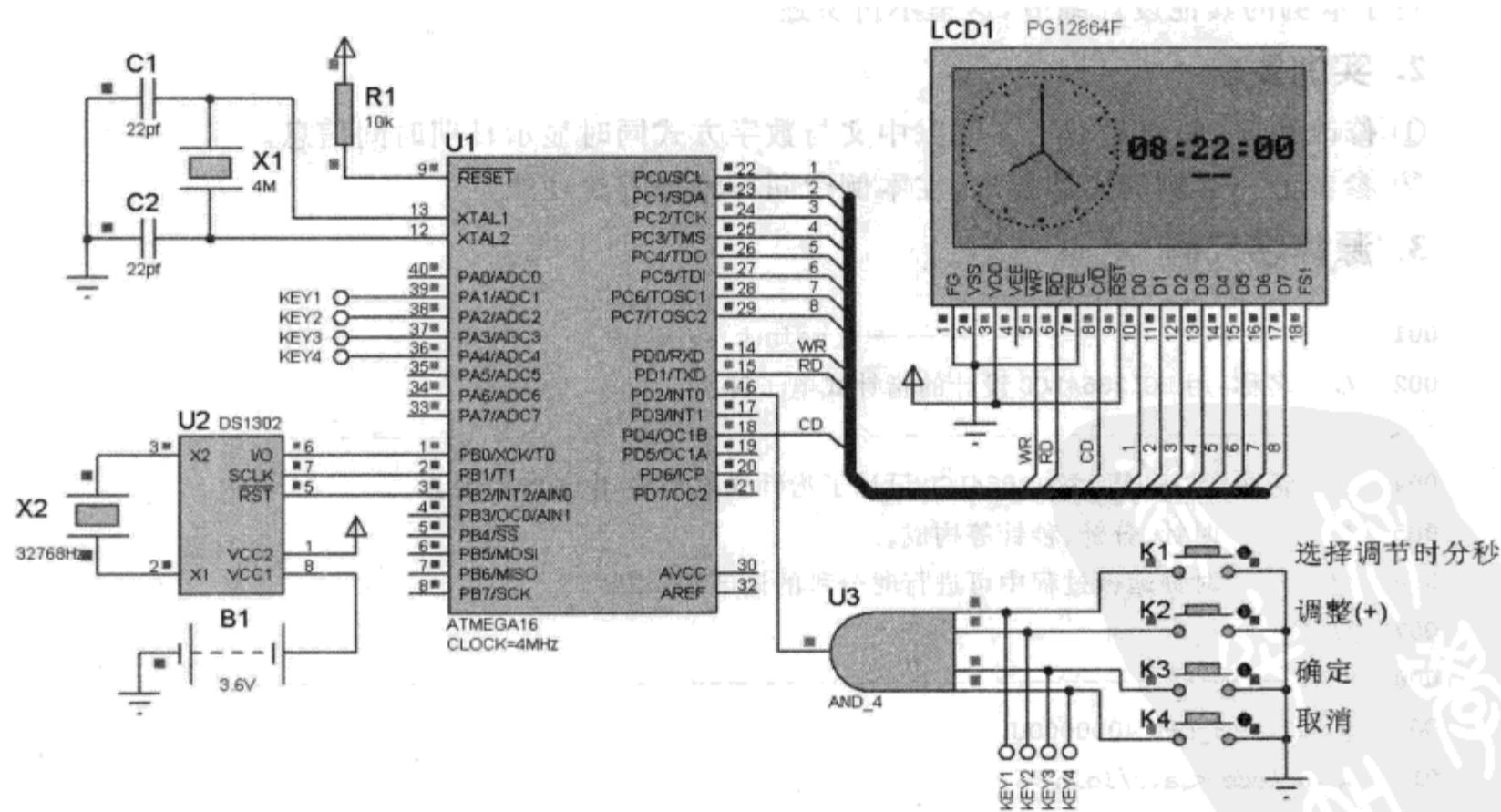


图 5-8 用 PG12864LCD 设计的指针式电子钟



## 1. 程序设计与调试

本例程序包含 main.c、PG12864.c、PG12864.h、DS1302.c 共 4 个文件,读者在阅读调试本例时可重点关注表盘绘制及指针移动代码。

函数 Clock\_Plate() 通过 2 个 for 循环分别完成表盘外圈及 12 个刻度的绘制。在绘制表盘外圈时,Clock\_Plate() 函数的第一个 for 循环以  $0.1$  个弧度的间隔绘点形成外圈,如果需要绘制更连续的外圈,可进一步减少步长或者在相邻两个点之间绘制直线,这样都可形成更平滑的外圈。第二个 for 循环步长  $2 \times \pi / 12$ ,通过 12 次循环即可绘制出 12 个刻度。

在移动指针时,主程序中提供了重绘指针函数 Repaint\_A\_Hand(INT8U i),在绘制移动的秒针时( $i=2$ ),每次转过  $2 \times \pi$  弧度的  $1/60$ ,当绘制移动的时针与分针( $i=0/1$ )时,每次转过  $2 \times \pi$  弧度的  $1/12$ ,变量 m 根据 i 分别取值 60 与 12。下面的语句根据当前指针 i 对应的时间 DateTime[i](秒/分/时)即可得出第 i 只指针的当前弧度 r:

$$r = \text{DateTime}[i] / m \times 2 \times \pi + 1.5 \times \pi;$$

注意:DateTime[i]/m×2×PI 后面要 + $1.5 \times \pi$  或  $-0.5 \times \pi$ ,因为时钟的 0 刻度位置在垂直向上的方向,而后续使用的弧度 r 以水平向右方向为 0 刻度,因而需要添加偏移弧度。

为绘制不同长度的指针,数组 HMS\_Hand\_Length 设置秒针、分针、时针长度分别为 24、20、15,根据当前指针所指向的弧度及指针长度即可得到指针终点坐标(x,y),下面 3 行代码即可完成当前指针的绘制:

```
x = HMS_Hand_Length[i] * cos(r);  
y = HMS_Hand_Length[i] * sin(r);  
Line (30,30, x + 30, y + 30, 1);
```

对于本例的其他设计细节,这里不再赘述。

## 2. 实训要求

- ① 修改程序,使液晶右半屏能以中文与数字方式同时显示日期时间信息。
- ② 参考上一案例的实训要求,在本例中同样添加闹铃功能。

## 3. 源程序代码

```
001 //----- main.c -----  
002 // 名称:用 PG12864LCD 设计的指针式电子钟  
003 //-----  
004 // 说明:本例利用 PG12864LCD 设计了指针式电子钟,电子钟由表盘、  
005 // 时针、分针、秒针等构成。  
006 // 时钟运行过程中可进行时分秒的调节和保存  
007 //  
008 //-----  
009 #define F_CPU 4000000UL  
010 #include <avr/io.h>  
011 #include <avr/pgmspace.h>  
012 #include <avr/interrupt.h>  
013 #include <util/delay.h>
```

```

014 # include <stdio.h>
015 # include "PG12864.h"
016 # define INT8U unsigned char
017 # define INT16U unsigned int
018 //如果引入头文件 math.h 则可以直接使用 PI 的符号常量定义: M_PI
019 # define PI 3.1415926
020 extern void cls();                                //清屏
021 extern INT8U LCD_Initialise();                  //LCD 初始化
022 extern INT8U LCD_Write_Command(INT8U cmd);      //写无参数的命令
023                                         //写双参数命令
024 extern INT8U LCD_Write_Command_P2(INT8U cmd, INT8U para1, INT8U para2);
025 extern INT8U LCD_Write_Data(INT8U dat);          //写数据
026 extern void Set_LCD_POS(INT8U row, INT8U col);  //设置当前地址
027                                         //绘制线条
028 extern void Line(INT8U x1, INT8U y1, INT8U x2, INT8U y2, INT8U Mode);
029 extern void Pixel(INT8U x, INT8U y, INT8U Mode); //画点函数
030                                         //显示字符串
031 extern INT8U Display_Str_at_xy(INT8U x, INT8U y, char * fmt);
032 extern INT8U LCD_WIDTH;
033 extern INT8U LCD_HEIGHT;
034 extern void GetDateTime();                        //从 DS1302 获取时间
035 extern void SetDateTime();                      //设置时间
036
037 //按键定义
038 # define K1_ON() (PIN_A & _BV(PA1)) == 0x00      //选择调整时、分、秒
039 # define K2_ON() (PIN_A & _BV(PA2)) == 0x00      //调整(+)
040 # define K3_ON() (PIN_A & _BV(PA3)) == 0x00      //确定(写入 DS1302)
041 # define K4_ON() (PIN_A & _BV(PA4)) == 0x00      //取消
042
043 //所读取的日期时间
044 extern INT8U DateTime[7];
045 //当前调节的时间对象:秒,分,时(0,1,2),为 -1 时表示时钟正常运行
046 char Adjust_Index = -1;
047 //保存前一秒、分、时数据,用于在绘制当前新的指针时擦除上次绘制的指针
048 INT8U TimeBack[] = { -1, -1, -1 };
049 //秒,分,时针的长度
050 INT8U HMS_Hand_Length[] = { 24, 20, 15 };
051 //数字式时钟的显示串缓冲
052 char DisplayBuffer[] = "00:00:00";
053 //-----
054 // 显示数字式时钟
055 //-----
056 void ShowDigitTime()

```



```
057 {
058     DisplayBuffer[0] = DateTime[2] / 10 + '0';
059     DisplayBuffer[1] = DateTime[2] % 10 + '0';
060     DisplayBuffer[3] = DateTime[1] / 10 + '0';
061     DisplayBuffer[4] = DateTime[1] % 10 + '0';
062     DisplayBuffer[6] = DateTime[0] / 10 + '0';
063     DisplayBuffer[7] = DateTime[0] % 10 + '0';
064     Display_Str_at_xy(64,23,DisplayBuffer);
065 }
066
067 //-----
068 // 绘制电子钟圆形面板
069 //-----
070 void Clock_Plate()
071 {
072     float sta,x,y;
073     //绘制表盘外围圆圈,如果希望绘制较连续的圆圈线条,
074     //可减少步长,将 0.1 改为 0.02
075     for ( sta = 0; sta <= 2 * PI; sta += 0.1 )
076     {
077         x = sin(sta);
078         y = cos(sta);
079         Pixel(30 + 30 * x, 30 + 30 * y ,1);
080     }
081     //绘制 1~12 点的刻度,每隔 2 * PI/12 绘制一段
082     for ( sta = 0; sta <= 2 * PI; sta += 2 * PI / 12 )
083     {
084         x = sin(sta);
085         y = cos(sta);
086         Pixel(30 + 27 * x, 30 + 27 * y ,1);
087         Pixel(30 + 26 * x, 30 + 26 * y ,1);
088     }
089 }
090
091 //-----
092 // 重绘 HMS 中的某一指针(参数 0、1、2 分别为秒、分、时)
093 //-----
094 void Repaint_A_Hand(INT8U i)
095 {
096     float r,m; INT16U x,y;
097     //指针 2 对应于 12,指针 0,1 对应于 60,该值用于弧度划分
098     //无论使用 12 小时制还是 24 小时制,这里都要用 12
099     m = (i == 2)? 12.0 : 60.0;
```

```

100 //擦除指针 i
101 r = TimeBack[i] / m * 2 * PI + 1.5 * PI;
102 x = HMS_Hand_Length[i] * cos(r);
103 y = HMS_Hand_Length[i] * sin(r);
104 Line (30,30, x + 30, y + 30, 0);
105 //重绘新的指针 i
106 r = DateTime[i] / m * 2 * PI + 1.5 * PI;
107 x = HMS_Hand_Length[i] * cos(r);
108 y = HMS_Hand_Length[i] * sin(r);
109 Line (30,30, x + 30, y + 30, 1);
110 //时间备份,以便用于下次绘制新指针时擦除原指针
111 TimeBack[i] = DateTime[i];
112 }
113
114 //-----
115 // 时间变化时重绘
116 // 秒针与分针时钟接近重叠,或分钟与时针接近重叠时也要重绘
117 //-----
118 void Display_HMS_Hand()
119 {
120     Repaint_A_Hand(0);                      //绘制秒针
121     Repaint_A_Hand(1);                      //绘制分针
122     Repaint_A_Hand(2);                      //绘制时针
123 }
124
125 //-----
126 // 主程序
127 //-----
128 int main()
129 {
130     DDRA = 0x00; PORTA = 0xFF;                //配置端口
131     DDRB = 0xFF;
132     DDRC = 0xFF;
133     DDRD = 0xFF & ~_BV(PD2); PORTD |= _BV(PD2);
134     TCCR0 = 0x05;                            //预分频:1024
135     TCNT0 = 256 - F_CPU / 1024.0 * 0.05;    //晶振4MHz,0.05s定时
136     TIMSK = 0x01;                            //使能T0中断
137     MCUCR = 0x02;                            //INT0为下降沿触发
138     GICR = 0x40;                            //INT0中断使能
139     LCD Initialise();                      //初始化LCD
140     Set_LCD_POS(0,0);  cls();               //从LCD左上角开始清屏
141     Clock_Plate();                         //绘制时钟面板
142     sei();                                //使能中断

```

```

143     while(1)
144     {
145         //如果未执行调节操作则正常读取当前时间
146         if (Adjust_Index == -1) GetDateTime(); else _delay_ms(1000);
147     }
148 }
149
150 //-----
151 // T0 定时器刷新 LCD 时间显示
152 //-----
153 ISR (TIMERO_OVF_vect)
154 {
155     TCNT0 = 256 - F_CPU / 1024.0 * 0.05;           //恢复 T0 定时初值
156     //定时器不能每秒显示一次 DS1302 时钟,因为定时器程序自身执行需要时间
157     //每秒刷新显示时间时会出现掉秒现象,例如显示了 12:09:04 后出现 12:09:06
158     //因此需要在 1 s 内显示时间,但这样又可能取到相同时间,如果遇到相同时间
159     //时仍在 LCD 上刷新指针,这时会出现抖动感
160     //下面的语句使程序仅在取得的时间变化时才刷新显示,否则返回
161     if ( DateTime[0] == TimeBack[0] &&
162         DateTime[1] == TimeBack[1] &&
163         DateTime[2] == TimeBack[2] ) return;
164
165     Display_HMS_Hand();                         //显示当前时间指针
166     ShowDigitTime();                           //同时显示数字式时间
167 }
168
169 //-----
170 // 键盘中断(INT0)
171 //-----
172 ISR (INT0_vect)
173 {
174     if (K1_ON()) //选择调节对象
175     {
176         if ( Adjust_Index == -1 || Adjust_Index == 0) Adjust_Index = 3;
177         Adjust_Index--; //在 2,1,0 中循环选择,对应调节时,分,秒
178         //在数字时钟 XX:XX:XX 下对应位置显示横线,标识当前调节对象
179         if ( Adjust_Index == 2) Display_Str_at_xy(64,34, "—");
180         else if (Adjust_Index == 1) Display_Str_at_xy(64,34, "—");
181         else if (Adjust_Index == 0) Display_Str_at_xy(64,34, "—");
182     }
183     else if (K2_ON() && Adjust_Index != -1) //调整(递增,溢出时回 0 继续)
184     {
185         if (Adjust_Index == 2) DateTime[2] = (DateTime[2] + 1) % 24; //时

```

```

186     else if (Adjust_Index == 1) DateTime[1] = (DateTime[1] + 1) % 60; //分
187     else if (Adjust_Index == 0) DateTime[0] = (DateTime[0] + 1) % 60; //秒
188 }
189 else if (K3_ON() && Adjust_Index != -1) //确定
190 {
191     SetDateTime(); //将调节后的时间写入 DS1302
192     Display_Str_at_xy(64, 34, ""); //擦除标识调节对象的横线
193     Adjust_Index = -1; //操作索引重设为 -1, 时间继续正常显示
194 }
195 else if (K4_ON() && Adjust_Index != -1) //取消
196 {
197     Display_Str_at_xy(64, 34, "");
198     Adjust_Index = -1;
199 }
200 }

```

```

001 //----- DS1302.c -----
002 // DS1302 实时时钟程序
003 //-----
004 #include <avr/io.h>
005 #include <string.h>
006 #include <util/delay.h>
007 #define INT8U unsigned char
008 #define INT16U unsigned int
009
010 //DS1302 引脚定义
011 #define IO PB0
012 #define SCLK PB1
013 #define RST PB2
014 //DS1302 接口数据方向
015 #define DDR_IO_RD() DDRB &= ~_BV(IO)
016 #define DDR_IO_WR() DDRB |= _BV(IO)
017 //DS1302 控制引脚操作定义
018 #define WR_IO_0() (PORTB &= ~_BV(IO)) //DS1302 IO 线(W/R)
019 #define WR_IO_1() (PORTB |= _BV(IO))
020 #define RD_IO() (PINB & _BV(IO))
021 #define SCLK_1() (PORTB |= _BV(SCLK)) //DS1302 时钟线
022 #define SCLK_0() (PORTB &= ~_BV(SCLK))
023 #define RST_1() (PORTB |= _BV(RST)) //DS1302 复位线
024 #define RST_0() (PORTB &= ~_BV(RST))
025
026 //星期字符串表(本例未使用)
027 char *WEEK[] = {"SUN", "MON", "TUS", "WEN", "THU", "FRI", "SAT"};

```



```
028 //所读取的日期时间
029 INT8U DateTime[7];
030 //-----
031 // 向 DS1302 写入 1 字节
032 //-----
033 void Write_Byte_TO_DS1302(INT8U x)
034 {
035     INT8U i;
036     DDR_IO_WR();                                //写 DS1302 I/O 口
037     for(i = 0x01; i != 0x00 ; i <<= 1)          //写 1 字节(上升沿写入)
038     {
039         if (x & i) WR_IO_1(); else WR_IO_0(); SCLK_0();SCLK_1();
040     }
041 }
042
043 //-----
044 // 从 DS1302 读取 1 字节
045 //-----
046 INT8U Get_Byte_FROM_DS1302()
047 {
048     INT8U i,dat = 0x00;
049     DDR_IO_RD();                                //读 DS1302 I/O 口
050     for(i = 0; i < 8 ; i++)                     //串行读取 1 字节(下降沿读取)
051     {
052         SCLK_1(); SCLK_0();if (RD_IO()) dat |= _BV(i);
053     }
054     return dat / 16 * 10 + dat % 16;             //将 BCD 码转换为十进制数并返回
055 }
056
057 //-----
058 // 从 DS1302 指定位置读数据
059 //-----
060 INT8U Read_Data(INT8U addr)
061 {
062     INT8U dat;
063     RST_1();                                    //将 RST 拉高
064     Write_Byte_TO_DS1302(addr);                 //向 DS1302 写地址
065     dat = Get_Byte_FROM_DS1302();                //从指定地址读字节
066     RST_0();                                    //将 RST 拉低
067     return dat;
068 }
069
070 //-----
```

```

071 // 向 DS1302 指定地址写数据
072 //-----
073 void Write_DS1302(INT8U addr, INT8U dat)
074 {
075     RST_1();
076     Write_Bit_TO_DS1302(addr);           //写地址
077     Write_Bit_TO_DS1302(dat);           //写数据
078     RST_0();
079 }
080
081 //-----
082 // 读取当前日期时间
083 //-----
084 void GetDateTime()
085 {
086     INT8U i,addr = 0x81;                //从读秒地址 0x81 开始
087     for (i = 0; i < 7; i++)            //依次读取 7 字节(秒、分、时、日、月、周、年)
088     {
089         DateTime[i] = Read_Data(addr);
090         addr += 2;                    //读日期时间地址依次为 0x81、0x83、0x85...
091     }
092 }
093
094 //-----
095 // 设置时间(本例仅设置时/分/秒)
096 //-----
097 void SetDateTime()
098 {
099     INT8U i;
100     Write_DS1302(0x8E,0x00);          //写控制字节,取消写保护
101     for(i = 0; i < 3 ; i++)          //因本例仅写入秒分时,故只需 3 次循环
102     {
103         //秒的起始地址 10000000(0x80),
104         //后续依次是分,时,日,月,周,年,写入地址每次递增 2
105         //两位的十进制日期时间数据要转换为 BCD 码后再写入
106         Write_DS1302(0x80 + 2 * i, (DateTime[i]/10<<4) | (DateTime[i] % 10));
107     }
108     Write_DS1302(0x8E,0x80);          //加保护
109 }

```

## 5.9 高仿真数码管电子钟

本例运行时,数码管将高仿真显示当前时间,案例中时钟调节部分添加了 12/24 小时制调

节,出现了 AM/PM 显示等。本例电路及部分运行效果如图 5-9 所示,电路中仅添加了部分作显示驱动的 74LS244 芯片,剩余部分可自行添加。

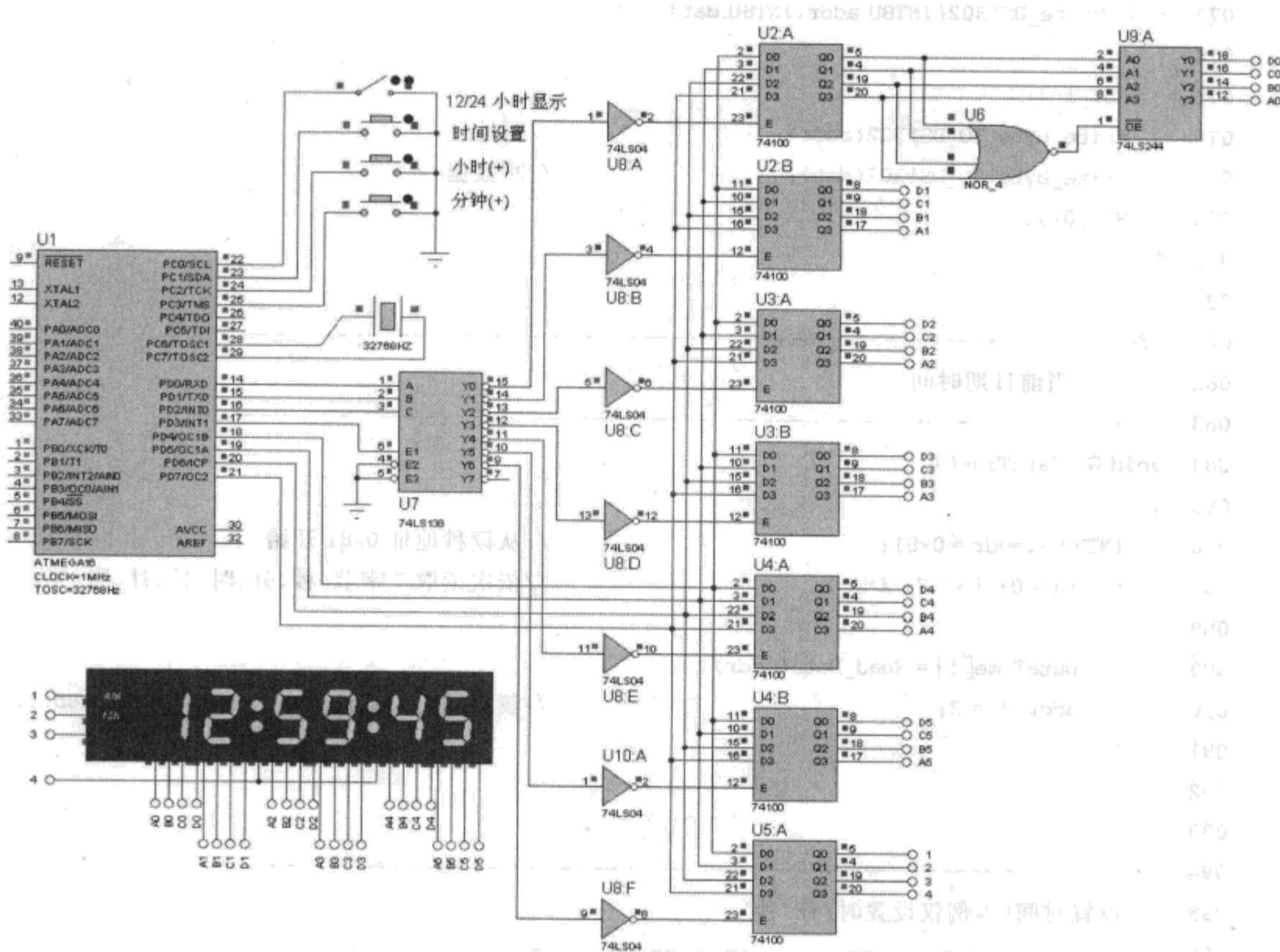


图 5-9 高仿真数码管电子钟

## 1. 程序设计与调试

本例用 Proteus 提供的电子钟组件实现了高度仿真的显示效果,电子钟显示屏由矩形外框与 3 类时钟仿真组件组成,它们分别是:

① 6 只七段 BCD 码数码管组件(7SEG),向数码管 4 只引脚输入 0000~1001 时可分别显示 0~9,程序中不需要编写段码表。

② 时分秒数字之间的分隔冒号(Colon)组件(“:”,CLKCOL),该组件仅有一只引脚,高电平时“:”点亮,反之则关闭。

③ 一块时钟指示(Indicator)组件(CLKIND),该组件的显示信息分为 3 行,由上到下分别是(AM/PM)、(12/24)、(SET)。对于(12/24),在低电平时(12)点亮,反之则(24)点亮;在(12)点亮时,如果(AM/PM)为低电平则(AM)点亮,反之则(PM)点亮;如果(12/24)为高电平,(24)被点亮,这时对(AM/PM)的控制将被禁止,无论(AM/PM)引脚为高电平还是低电平,(AM/PM)都会被关闭显示。当(SET)引脚为高电平时(SET)被点亮,它指示系统进行人设置状态,反之则(SET)被关闭显示,时钟重新处于正常显示状态。该组件的 3 只引脚都不允许悬空。

本例时钟由工作于异步模式的 T/C2 溢出中断控制运行,每隔 0.5 s 时“:”关显示,每隔 1 s 时秒数增加,“:”开显示。时钟的完整显示由函数 Display\_Time() 控制完成,函数中的主要语句如下:

```
for (i = 0; i < 7; i++)
{
    PORTD = (disp_Buffer[i] << 4) | i | _BV(E1_74LS138); _delay_ms(2);
    PORTD &= ~_BV(E1_74LS138); _delay_ms(2);
}
```

该函数每次发送 7 字节数据,前 6 字节为时/分/秒的 BCD 码(各 2 位),第 7 字节是指示屏及“:”号的控制码,disp\_Buffer 数组中各字节的高 4 位无用,BCD 码或指示屏及“:”控制码都在字节的低 4 位中。每一字节的发送与显示分两步完成:

① PD 端口高 4 位发送某位数码管 BCD 码或指示屏控制码,对 disp\_Buffer[i] 要左移 4 位,这 4 位将直接到达所有 7 片 74100 的 D 端,PD 端口低 3 位则向 3-8 译码器输入编码 i,取值为 000~110(本例未使用 111),在发送高 4 位与低 3 位的同时,该行还通过或\_BV(E1\_74LS138) 将 3-8 译器的 E1 置高电平,开译码器,7 片 74100 中的第 i 片被选通,待显示数据被正常输出。

② 2 ms 后关 3-8 译码器,之所以要关闭译码器是因为发送下一数据时,由于高 4 位先到达 74100 的 D 端,而译码器对当前第 x 片 74100 的选通是滞后的,这时本来是由第 x 片 74100 输出的编码会先由上次仍处于选通状态的第 x-1 片 74100 输出,然后又很快转而由第 x 片 74100 输出,这样显然会导致数码管的显示异常。

设计 Display\_Time() 函数时,颠倒上述 for 循环中两行语句的先后位置,程序同样能得到正常的运行效果。

## 2. 实训要求

① 将电路中的“SET”开关改成按键,重新设计本例,在按下设置键后开始调整 12/24 及时/分/秒数据,再次按下“SET”键时确认设置。如果调节后超过 20 s 仍没有再按下设置键确认,则系统能取消当前设置,时钟恢复原来的状态继续运行。

② 本例用工作于异步模式的 T/C2 控制时钟运行,完成本例调试后,将 32768 Hz 晶振改接到 DS1302 时钟芯片,仍完成上面的实训要求。

## 3. 源程序代码

```
001 //-----
002 // 名称: 高度仿真的可调式数码管电子钟
003 //-----
004 // 说明: 本例在 Proteus 中选用了高度仿真的电子钟元器件,并添加了
005 //          时分调整功能,冒号闪烁显示,AM/PM 切换,12/24 小时制选择等
006 //
007 //-----
008 #define F_CPU 4000000UL
009 #include <avr/io.h>
```



```
010 #include <avr/interrupt.h>
011 #include <util/delay.h>
012 #define INT8U unsigned char
013 #define INT16U unsigned int
014
015 //时钟设置开关及按键
016 #define S1_ON() ((PINC & _BV(PC0)) == 0x00) //设置
017 #define K2_DOWN() ((PINC & _BV(PC1)) == 0x00) //12/24 小时
018 #define K3_DOWN() ((PINC & _BV(PC2)) == 0x00) //小时加
019 #define K4_DOWN() ((PINC & _BV(PC3)) == 0x00) //分钟加
020
021 //时钟指示组件控制引脚定义(不要误定义为 1,2,3,4)
022 #define CLK_AM_PM 0 //AM/PM 切换
023 #define CLK_12_24 1 //12/24 小时制切换
024 #define CLK_SET 2 //SET 指示切换
025 #define CLK_COL 3 //冒号显示切换
026
027 //3-8 译码器使能控制引脚定义
028 #define E1_74LS138 PD3
029
030 //当前时间(时:分:秒)(本例设为 12:59:40,这样可便于快速观察到切换效果)
031 INT8U current_Time[] = {12,59,40};
032 //时分秒显示缓冲(各占 2 位,共 6 字节),
033 //第 7 个字节 0x00 控制(AM/PM),(12/24),(SET)及 ":" 显示.
034 //该字节低 4 位的对应关系是 XXXX-0(:)0(AM/PM)0(12/24)0(SET)
035 //这与上述的 4 个#define 对应,0x00 默认设置 AM,12,非 SET
036 INT8U disp_Buffer[] = {0,0,0,0,0,0,0x00};
037
038 //本例函数申明
039 void Add_Hour(); // + 小时
040 void Add_Minute(); // + 分钟
041 void Refresh_Disp_Buffer(); //刷新显示缓冲
042 void Display_Time(); //显示时钟(包括指示屏及 ":")
043 void Adjust_and_Set_Clock(); //调节与设置时钟
044 //-----
045 // 根据当前时间刷新时分秒显示缓冲
046 //-----
047 void Refresh_Disp_Buffer()
048 {
049     INT8U i;
050     //刷新显示缓冲,将 current_Time 数组中的时/分/秒 3 个数分解为 6 个数位
051     for (i = 0; i < 3; i++)
052     {
```

```

053     disp_Buffer[2 * i]      = current_Time[i] / 10;
054     disp_Buffer[2 * i + 1] = current_Time[i] % 10;
055 }
056 }
057
058 //-----
059 // 加时
060 //-----
061 void Add_Hour()
062 {
063     ++ current_Time[0];                                //小时数累加
064     //如果是 12 小时制且当前超过 12 小时,小时数归 1,AM/PM 标志通过异或(^)运算取反
065     if ( (disp_Buffer[6] & _BV(CLK_12_24)) == 0x00 && current_Time[0] > 12 )
066     {
067         current_Time[0] = 1;
068         disp_Buffer[6] ^= _BV(CLK_AM_PM);
069     }
070     else //如果是 24 小时制且到达 24 小时则小时数归 0,AM/PM 标志不作处理
071     if ( (disp_Buffer[6] & _BV(CLK_12_24)) != 0x00 && current_Time[0] == 24)
072     {
073         current_Time[0] = 0;
074     }
075 }
076
077 //-----
078 // 加分
079 //-----
080 void Add_Minute()
081 {
082     ++ current_Time[1];                                //分钟数累加
083     if (current_Time[1] == 60)                         //满 60 时归 0,小时递增
084     {
085         current_Time[1] = 0; Add_Hour();
086     }
087 }
088
089 //-----
090 // 显示时间
091 //-----
092 void Display_Time()
093 {
094     INT8U i;
095     //循环显示时钟显示屏的 7 位数据,前 6 个是时/分/秒,各 2 位

```



```
096     //最后一字节用于控制指示屏及": "开关
097     for ( i = 0; i < 7 ; i ++ )
098     {
099         //PD 端口高 4 位发送显示屏 BCD 码或指示屏控制码: disp_Buffer[i] ,
100         //这 4 位将直接到达所有 7 片 74100 的 D 端.
101         //低 3 位则向 3-8 译码器输入编码 i , 取值为 000~110(本例未使用 111) ,
102         //在组合发送高 4 位与低 3 位的同时开译码器: _BV(E1_74LS138)
103         //选通 7 片 74100 中的一片
104         PORTD = ( disp_Buffer[i] << 4 ) | i | _BV(E1_74LS138); _delay_ms(2);
105         //2ms 后关 3-8 译码器, 禁止所有 74100
106         PORTD &= ~_BV(E1_74LS138); _delay_ms(2);
107     }
108 }
109
110 //-----
111 // 处理 12/24 小时制按键切换后的数据变更及 AM/PM 显示开关
112 //-----
113 void Handle_12_24_and_AM_PM_Switch()
114 {
115     //处理 24 小时制下的数据变更等问题-----
116     if ( disp_Buffer[6] & _BV(CLK_12_24) )
117     {
118         //如果切换到 24 小时模式时 PM 是点亮的, 这里时应对 12.PM 以内的数加 12
119         if ( disp_Buffer[6] & _BV(CLK_AM_PM) )
120         {
121             if ( current_Time[0] != 12 ) current_Time[0] += 12;
122         }
123         else if ( current_Time[0] == 12 ) //如果是 12.AM 则将其转换为 0 点
124             current_Time[0] = 0;
125     }
126     else //处理 12 小时制下的数据变更及 AM/PM 开关问题-----
127     {
128         //如果遇到 24 小时模式下的值则转换为 12 小时模式下的值
129         //1. 0~11.....
130         if ( current_Time[0] > = 0 && current_Time[0] < = 11 )
131         {
132             if ( current_Time[0] == 0 ) current_Time[0] = 12; //0 点转换为 12.AM
133             disp_Buffer[6] &= ~_BV(CLK_AM_PM); //点亮 AM(0~12)
134         }
135         //2. 12~23.....
136         else
137             if ( current_Time[0] > = 12 && current_Time[0] < 24 )
138             {
```

```

139 //对 PM 范围内的时间减 12
140     if (current_Time[0] > 12) current_Time[0] -= 12;
141     disp_Buffer[6] |= _BV(CLK_AM_PM);           //点亮 PM
142 }
143 }
144 }
145
146 //-----
147 // 时钟调整与设置
148 //-----
149 void Adjust_and_Set_Clock()
150 {
151     if (!S1_ON()) return; //如果 K1 按下则进入设置状态(SET),否则返回
152
153     TIMSK = 0x00;          //禁止定时器中断,时钟停止运行,进入设置状态
154     disp_Buffer[6] |= _BV(CLK_SET); //点亮 SET
155
156     while (S1_ON()) //K1 未释放时保持在设置状态
157     {
158         if (K2_DOWN()) //设置 12/24 小时制-----
159         {
160             disp_Buffer[6] ^= _BV(CLK_12_24); //切换 12/24 显示标志
161             Handle_12_24_and_AM_PM_Switch(); //处理切换后的小时变更及 AM/PM 开关
162             _delay_ms(50);
163         }
164
165         if (K3_DOWN()) //加小时-----
166         {
167             //Add_Hour 函数在递增小时后还要处理 12/24 越界问题及切换 AM/PM 开关
168             _delay_ms(50); Add_Hour();
169         }
170
171         if (K4_DOWN()) //加分钟-----
172         {
173             //因为加分钟的函数不影响小时进位,故单独增加,不调用函数 Add_Minute()
174             _delay_ms(50); if (++current_Time[1] == 60) current_Time[1] = 0;
175         }
176         //刷新显示缓冲并显示当前时钟-----
177         Refresh_Disp_Buffer(); Display_Time();
178     }
179
180     //允许定时器中断,退出设置状态,时钟继续正常运行
181     TIMSK = _BV(TOIE2);

```



```
182     disp_Buffer[6] &= ~_BV(CLK_SET);           //关闭 SET
183 }
184
185 //-----
186 // 主程序
187 //-----
188 int main()
189 {
190     DDRC = 0x00; PORTC = 0xFF;                 //配置端口
191     DDRD = 0xFF;
192     ASSR = 0x08;                             //异步时钟使能
193     TCCR2 = 0x04;                            //预设分频:64,32768 Hz/64 = 512 Hz
194     TCNT2 = 0;                                //T2 计时初值
195     TIMSK = _BV(TOIE2);                      //允许 T2 定时器中断
196     sei();                                    //开中断
197     while (1)
198     {
199         Adjust_and_Set_Clock();                //检测按键,调整与设置时钟
200     }
201 }
202
203 //-----
204 // T/C2 溢出中断控制时钟运行
205 //-----
206 ISR (TIMER2_OVF_vect )
207 {
208     static INT8U tCount = 0;
209     //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
210     //TCNT0 = 0;
211     //T2 时钟被分频为 512 Hz,TCNT0 由 0 计数到 256 时溢出,故每 0.5 s 中断一次
212     //每 0.5 s:"关闭显示"
213     disp_Buffer[6] &= ~_BV(CLK_COL);
214     //每 0.5 s × 2 = 1 s 刷新显示缓冲,并显示":"
215     if ( ++ tCount == 2)
216     {
217         tCount = 0;
218         //每 1 s 闪烁 LED(:)打开
219         disp_Buffer[6] |= _BV(CLK_COL);
220         //秒递增,满 60 时归 0,并调用分钟递函数
221         if ( ++ current_Time[2] == 60)
222         {
223             current_Time[2] = 0; Add_Minute();
224         }

```

```

225 //刷新时分秒显示缓冲
226 Refresh_Disp_Buffer();
227 }
228 //显示时间(含":等信息的显示)
229 Display_Time();
230 }

```

## 5.10 1602 LCD 显示的秒表

本例运行时,利用 K1 按键可实现两段计时功能(1→2,3→4),计时精度为 1/100 s,K2 按键用于清零。本例电路及部分运行效果如图 5-10 所示。

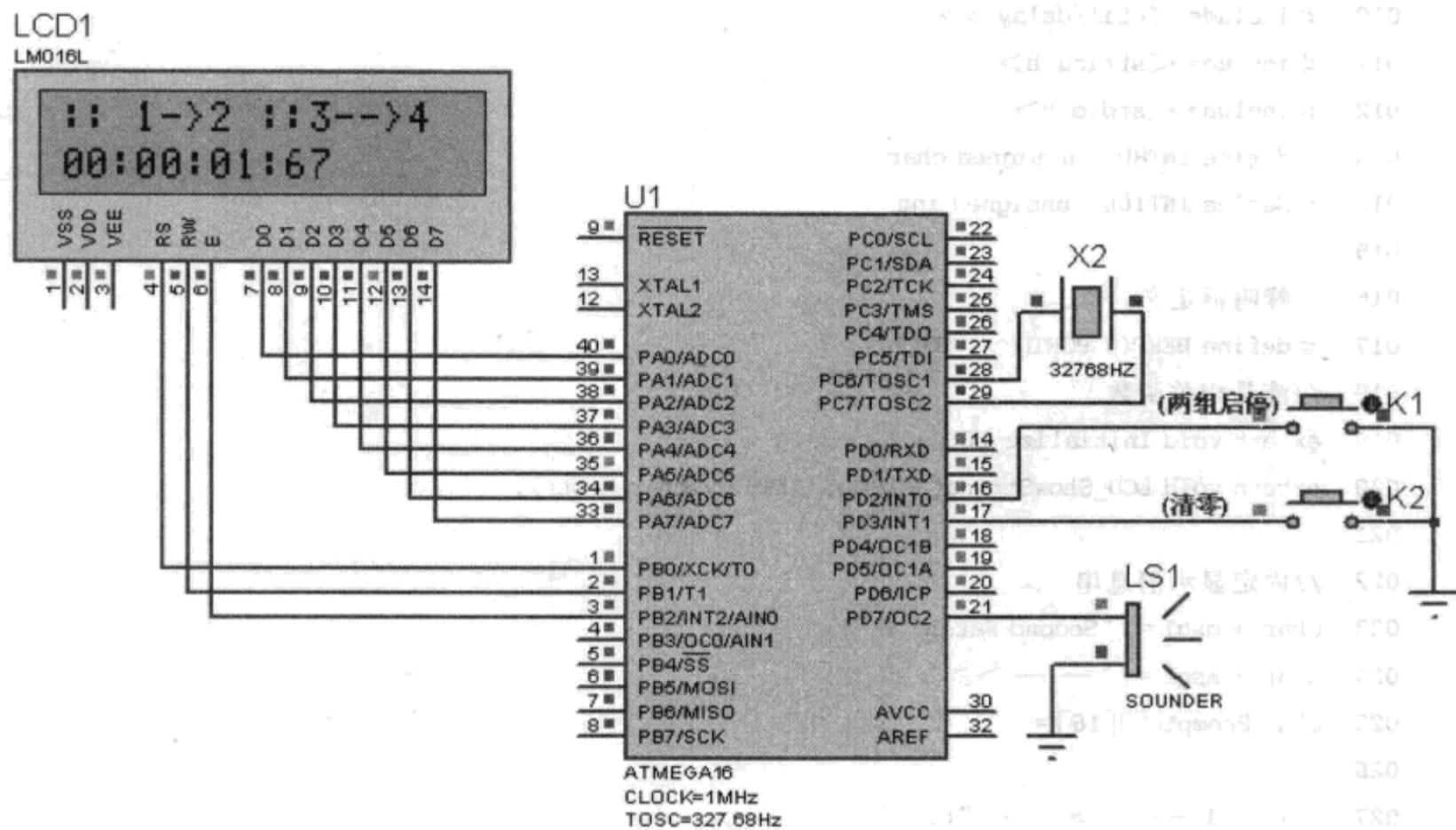


图 5-10 1602 LCD 显示的秒表

### 1. 程序设计与调试

本例驱动计时的 T/C2 与上一案例一样,仍工作于异步时钟模式。学习调试本例时,重点在于掌握在 K1 按键上集中 4 项不同功能的程序设计方法及 1/100 秒/秒/分/时的计时控制及显示控制。有关本例的程序设计技术已经在其他案例中讨论过了,相关细节可自行阅读分析,这里不再赘述。

### 2. 实训要求

- ① 重新设计本例,使液晶屏工作于 4 位模式,仍实现本例设定的功能。
- ② 删除 32768 Hz 晶振,T/C2 使用 1 分频的系统时钟,仍实现本例设定的功能。
- ③ 在本例电路中添加 24C04,系统能将计时值逐次保存到 24C04 中,能翻页查看历史记

录,还能将所有历史记录清除。

### 3. 源程序代码

```

001 //-----
002 // 名称: 用 1602 LCD 设计的秒表
003 //-----
004 // 功能: 首次按下 K1 时开始计时,再次按下时暂停,第 3 次按下时继续
005 // 累加计时,再按下时停止计时。K2 用来清零秒表
006 //
007 //-----
008 #include <avr/io.h>
009 #include <avr/interrupt.h>
010 #include <util/delay.h>
011 #include <string.h>
012 #include <stdio.h>
013 #define INT8U unsigned char
014 #define INT16U unsigned int
015
016 //蜂鸣器定义
017 #define BEEP() PORTD ^= _BV(PD7)
018 //液晶相关函数
019 extern void Initialize_LCD();
020 extern void LCD_ShowString(INT8U x, INT8U y, char * str);
021
022 //固定显示消息串
023 char * msg1 = "Second Watch 0 ";
024 char * msg2 = "---->>>      ";
025 char Prompts[][][16] =
026 {
027     {":: 1 -->      "},
028     {":: 1 --> ::2  "},
029     {":: 1->2 ::3->  "},
030     {":: 1->2 ::3->4 "}
031 };
032 //时、分、秒、百分秒计时缓冲与显示缓冲
033 INT8U Time_Buffer[] = {0,0,0,0};
034 char LCD_Display_Buffer[] = {"00:00:00:00"};
035 //Key_func_NO 用于在一个按钮上区分 4 种不同操作(取值限于 0,1,2,3)
036 volatile INT8U Key_func_NO = 0xFF;
037 //-----
038 // 蜂鸣器声音输出
039 //-----
040 void Sounder()

```

```

041 {
042     INT8U i;
043     for (i = 0; i < 150; i++)
044     {
045         BEEP(); _delay_us(300);
046     }
047 }
048
049 //-----
050 // T2 中断控制计时
051 //-----
052 ISR (TIMER2_OVF_vect )
053 {
054     TCNT2 = - (INT8U)(4096 * 0.01 + 0.5); //T2 计时初值:0.01 s
055     Time_Buffer[0]++;
056     if(Time_Buffer[0] == 100)           //1/100 s
057     {
058         Time_Buffer[0] = 0;
059         Time_Buffer[1]++;
060     }
061     if(Time_Buffer[1] == 60)           //秒
062     {
063         Time_Buffer[1] = 0;
064         Time_Buffer[2]++;
065     }
066     if(Time_Buffer[2] == 60)           //分
067     {
068         Time_Buffer[2] = 0;
069         Time_Buffer[3]++;
070     }
071     if(Time_Buffer[3] == 24)           //时
072         Time_Buffer[3] = 0;
073     //按指定格式生成显示字符串
074     sprintf(LCD_Display_Buffer, "%02d: %02d: %02d: %02d",
075             Time_Buffer[3], Time_Buffer[2], Time_Buffer[1], Time_Buffer[0]);
076     //显示时/分/秒/0.01 s
077     LCD_ShowString(0,1,LCD_Display_Buffer);
078 }
079
080 //-----
081 // 主函数
082 //-----
083 int main()

```



```
084  {
085      DDRA = 0xFF;                      //配置端口
086      DDRB = 0xFF;
087      DDRD = 0x00; PORTD = 0xFF;        //((外部中断输入,内部上拉)
088      ASSR = 0x08;                     //异步时钟使能
089      TCCR2 = 0x02;                   //预设分频:8,32768 Hz/8 = 4096 Hz
090      TCNT2 = - (INT8U)(4096 * 0.01 + 0.5); //T2 计时初值 0.01 s(+ 0.5 可将 40.96 进位为 41)
091      MCUCR = 0x0A;                  //INT0,INT1 中断下降沿触发
092      GICR = 0xC0;                   //INT0,INT1 中断许可
093      sei();                        //开中断
094      Initialize_LCD();            //初始化 LCD
095      LCD_ShowString(0,0,msg1);     //显示两行提示信息
096      LCD_ShowString(0,1,msg2);
097      while(1);
098  }
099
100 //-----
101 // INT0 中断服务程序(区分 4 档按键:0、2 为启动或继续,1、3 为暂停或停止)
102 //-----
103 ISR (INT0_vect)
104 {
105     //按键功能号变量 Key_func_NO 取值限制于 0、1、2、3
106     if (Key_func_NO == 3) return;
107     switch ( ++ Key_func_NO )
108     {
109         case 0:
110         case 2: TIMSK = _BV(TOIE2);          //0、2 启动/继续
111             LCD_ShowString(0,0,Prompts[Key_func_NO]);
112             break;
113         case 1:
114         case 3: TIMSK = 0x00;                //1、3 暂停/停止
115             LCD_ShowString(0,0,Prompts[Key_func_NO]);
116             break;
117     }
118     Sounder();                         //输出提示音
119 }
120
121 //-----
122 // INT1 中断服务程序
123 //-----
124 ISR (INT1_vect)
125 {
126     INT8U i;
```

```

127     TIMSK = 0x00;           //禁止定时器溢出中断,计时停止
128     Key_func_NO = 0xFF;    //清除按键功能号
129     for(i = 0; i < 4; i++) Time_Buffer[i] = 0; //清零计时缓冲
130     LCD_ShowString(0,0,msg1);          //显示固定提示信息串 msg1、msg2
131     LCD_ShowString(0,1,msg2);
132     Sounder();                  //输出提示音
133 }

```

## 5.11 用 DS18B20 与 MAX6951 驱动数码管设计的温度报警器

本例的温度数据用 MAX6951 驱动 6 位分立式数码管显示，在所显示的温度超过报警范围时，系统将输出报警声音，对应的报警指示灯将持续闪烁。本例电路及部分运行效果如图 5-11 所示。

启动时间为 1 秒，本程序将报警温度设为高：70 低：-20

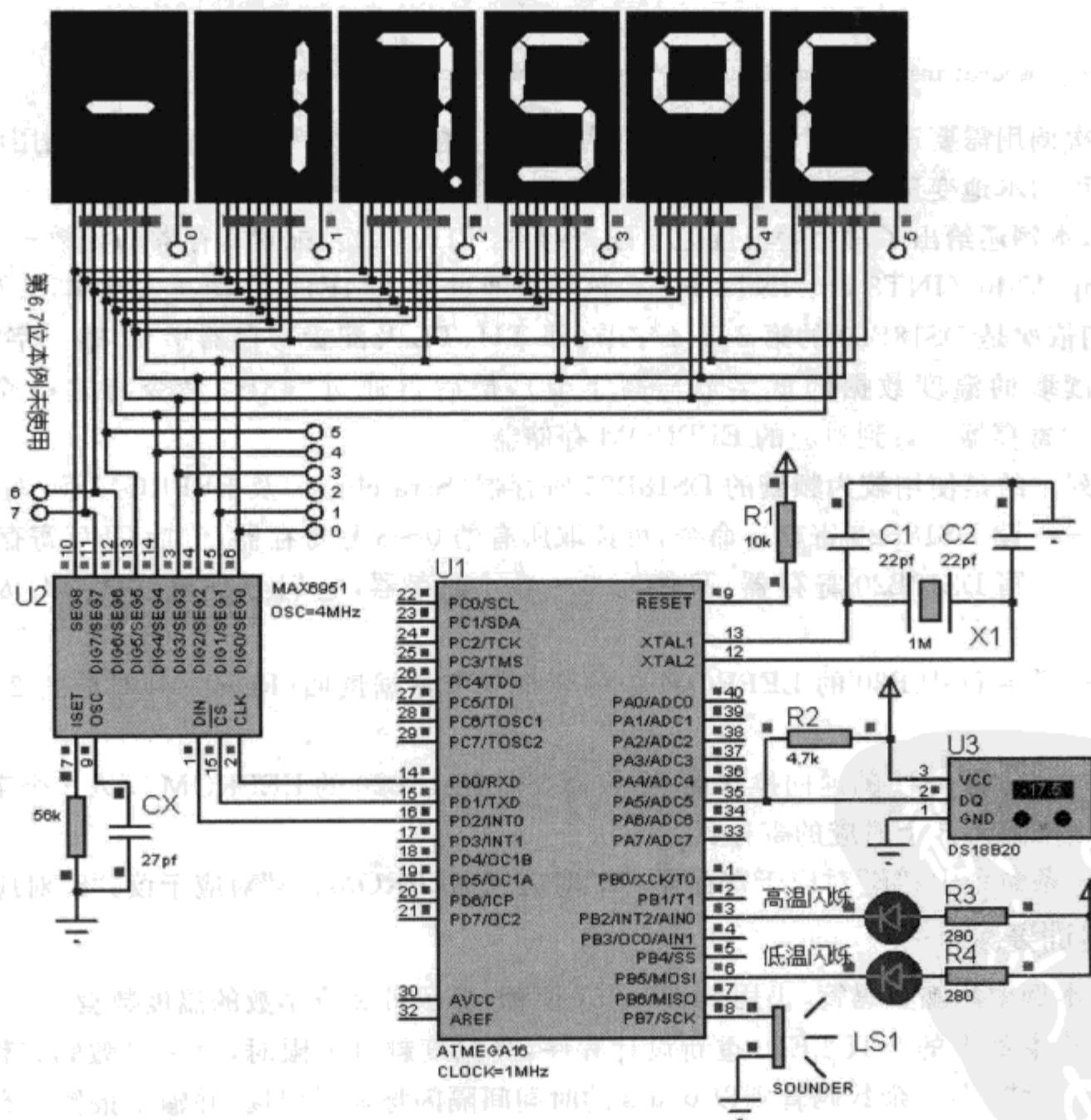


图 5-11 用 DS18B20 与 MAX6951 驱动数码管设计的温度报警器



## 1. 程序设计与调试

通过本例设计与调试,可进一步提高应用系统开发中多项功能的整合设计能力。

第 4 章中有关案例已经讨论过数码管驱动器 MAX6951 的应用,本例中该驱动器工作于全部不解码模式,程序中编写了各数字的段码表,要注意 MAX6951 使用的段码表不同于直接驱动时所使用的段码表。

另外,本例为了从 DS18B20 读取温度后,将温度符号、温度整数部分及小数部分分别独立返回,程序中提供了函数:

```
void Convert_Temperature(INT8 * sign, INT8U * iTemp, INT8U * fTemp)
```

该函数的 3 个参数分别是 1 个 INT8 \* 和 2 个 INT8U \* 类型,其中符号参数 sign 返回值为 1 或 -1,分别表示正温度或负温度,后 2 个参数 iTemp 与 fTemp 返回温度整数部分和小数部分。

本例程序中的温度显示与报警启停控制函数 Temp\_display\_and\_alarm() 在调用该函数时,对于参数传递的 2 种方式:传值与传址,该函数给出的 3 个参数全部为传址,调用语句如下:

```
Convert_Temperature(&i_sign, &i_curr_Temp, &f_curr_Temp);
```

当某次调用需要返回多个值时,可通过使用传址的方法设计与调用函数,被调用函数通过指针变量可向本地变量中“写入”数据,从而实现了多值“返回”。

另外,本例还给出了写报警温度上下限寄存器(TH、TL)及配置寄存器的函数——Set\_Alarm\_Temp\_Value(INT8 ha, INT8 la)。该函数通过命令“4EH”启动写入操作,共写入 3 个字节,它们依次是 DS18B20 的第 2、3、4 字节(即 TH、TL 及配置寄存器字节,第 0 字节与第 1 字节是待读取的温度数据的低字节与高字节),最后再通过“48H”命令将这 3 个字节由 DS18B20 的寄存器拷贝到对应的 EEPROM 存储器。

下面列举的是使用较为频繁的 DS18B20 寄存器(Scratchpad)及 EEPROM 读/写命令:

BEH——读 DS18B20 寄存器命令,可读取所有的 0~8 号寄存器,包括 CRC 寄存器。

4EH——写 DS18B20 寄存器,只能写 2~4 号寄存器,它们分别是 TH、TL 及配置寄存器。

B8H——将 DS18B20 的 EEPROM 中的 3 个字节数据读回(Recall)到对应的 2~4 号寄存器。

48H——它是 B8H 的逆向操作命令,该命令向 DS18B20 的 EEPROM 写入 3 个字节,这 3 个字节数据来自于 3 个对应的寄存器。

上述 4 条命令中:“E”对应于寄存器,“8”对应于 EEPROM;“4”对应于读,“8”对应于写。

## 2. 实训要求

- ① 在本例中添加数码管,仍用 MAX6951 驱动,显示带 2 位小数的温度数据。
- ② 删除本例中的 2 只 LED,重新设计程序:当温度越过上限时,第一只数码管稳定显示“H”,否则显示“L”,其余数码管则以 0.5 s 的时间间隔闪烁显示温度,并输出报警声音。

## 3. 源程序代码

```
001 //----- main.c -----
```

```

002 // 名称：用数码管与 DS18B20 设计温度报警器
003 //-
004 // 说明：本例将报警温度设为高：70，低：-20，当 DS18B20 感知到温度达到此
005 // 临界值时相应的 LED 闪烁，同时系统发出报警声
006 //
007 //-
008 #define F_CPU 1000000UL
009 #include <avr/io.h>
010 #include <avr/interrupt.h>
011 #include <util/delay.h>
012 #define INT8     signed   char
013 #define INT8U    unsigned char
014 #define INT16U   unsigned int
015
016 //双色自闪烁 LED 及蜂鸣器定义
017 #define HI_BI_LED_ON()      PORTB &= ~_BV(PB2)
018 #define HI_BI_LED_OFF()     PORTB |= _BV(PB2)
019 #define LO_BI_LED_ON()      PORTB &= ~_BV(PB5)
020 #define LO_BI_LED_OFF()     PORTB |= _BV(PB5)
021 #define BEEP_0()             PORTB &= ~_BV(PB7)
022 #define BEEP_1()             PORTB |= _BV(PB7)
023
024 //DS18B20 相关函数与变量
025 extern void Read_Temperature();
026 extern void Set_Alarm_Temp_Value();
027 extern void Convert_Temperature(INT8 * sign, INT8U * iTemp, INT8U * fTemp);
028 extern INT8U DS18B20_ERROR;
029 extern INT8  Alarm_Temp_HL[2];
030
031 //MAX695X 引脚操作定义
032 #define CLK_1() PORTD |= _BV(PDO)
033 #define CLK_0() PORTD &= ~_BV(PDO)
034 #define CS_1()  PORTD |= _BV(PD1)
035 #define CS_0()  PORTD &= ~_BV(PD1)
036 #define DIN_1() PORTD |= _BV(PD2)
037 #define DIN_0() PORTD &= ~_BV(PD2)
038
039 //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
040 //原来的各段顺序是： DP、G、F、E、D、C、B、A
041 //用 MAX6950/1 驱动顺序：DP、A、B、C、D、E、F、G
042 //除小数点位未改变外，其他位是逆向排列的
043 //下表中最后一位为黑屏
044 const INT8U SEG_CODE_695X[] =

```

```

045     {0x7E,0x30,0x6D,0x79,0x33,0x5B,0x5F,0x70,0x7F,0x7B,0x00},  

046 //报警标志  

047 INT8U HI_Alarm = 0, LO_Alarm = 0;  

048 //待显示的各温度数位,显示格式: - XX.X°C / XXX.X°C (-55.0 ~ 125.0)  

049 INT8U Temp_Display_Buffer[] = {0x00,0x00,0x00,0x00,0x63,0x4E};  

050 //-----  

051 // 向MAX695X写数据  

052 //-----  

053 void Write(INT8U Addr, INT8U Dat)  

054 {  

055     INT8U i;  

056     CS_0();  

057     for(i = 0; i < 8; i++)          //串行写入8位地址Addr  

058     {  

059         CLK_0(); if (Addr & 0x80) DIN_1(); else DIN_0();  

060         CLK_1(); _delay_us(20);  

061         Addr <<= 1;  

062     }  

063     for(i = 0; i < 8; i++)          //串行写入8位数据Dat  

064     {  

065         CLK_0(); if (Dat & 0x80) DIN_1(); else DIN_0();  

066         CLK_1(); _delay_us(20);  

067         Dat <<= 1;  

068     }  

069     CS_1();  

070 }  

071  

072 //-----  

073 // MAX695X初始化  

074 //-----  

075 void Init_MAX695X()  

076 {  

077     Write(0x02,0x07);           //设置亮度:中等亮度  

078     Write(0x03,0x05);           //扫描所有数码管  

079     Write(0x04,0x01);           //非关断0x01;关断0x00  

080 }  

081  

082 //-----  

083 // 温度显示,报警启停控制  

084 //-----  

085 void Temp_display_and_alarm()  

086 {  

087     INT8 i_sign;                //温度符号(1为正,-1为负),注意类型为INT8

```

```

088     INT8U i_curr_Temp;           //无符号的整数部分
089     INT8U f_curr_Temp;           //无符号的小数部分
090     INT8U i;
091     //将2字节的温度数据转换为有符号的整数部分和无符号的小数部分
092     Convert_Temperature(&i_sign, &i_curr_Temp, &f_curr_Temp);
093     Temp_Display_Buffer[3] = SEG_CODE_695X[f_curr_Temp];//小数段码
094
095     //将整数部分分解为3位待显示数字
096     Temp_Display_Buffer[0] = i_curr_Temp / 100;           //百位
097     Temp_Display_Buffer[1] = i_curr_Temp % 100 / 10;        //十位
098     Temp_Display_Buffer[2] = i_curr_Temp % 10;            //个位
099     if(Temp_Display_Buffer[0] == 0)                         //高位为0则不显示
100     {
101         Temp_Display_Buffer[0] = 10;
102         if(Temp_Display_Buffer[1] == 0)                     //高位为0且次高也为0时
103             Temp_Display_Buffer[1] = 10;                    //该位同样不显示
104     }
105     //得到整数部分的段码(最后一位整数加小数点)
106     Temp_Display_Buffer[0] = SEG_CODE_695X[Temp_Display_Buffer[0]];
107     Temp_Display_Buffer[1] = SEG_CODE_695X[Temp_Display_Buffer[1]];
108     Temp_Display_Buffer[2] = SEG_CODE_695X[Temp_Display_Buffer[2]] | 0x80;
109     //负符号显示
110     if (i_sign == -1)
111     {
112         if (i_curr_Temp >= 10) Temp_Display_Buffer[0] = 0x01;//两位前加"-
113         else                  Temp_Display_Buffer[1] = 0x01;//一位前加"-
114     }
115     //写MAX6951,全部不解码,通过发送段码显示
116     Write(0x01,0B00000000);
117     for(i=0; i<6; i++) Write( 0x60 | i, Temp_Display_Buffer[i]);
118
119     //高低温报警标志设置
120     //(与定义为INT8的有符号字节类型Alarm_Temp_HL比较,这样可区分正负比较)
121     HI_Alarm = i_sign * i_curr_Temp >= Alarm_Temp_HL[0] ? 1 : 0;
122     LO_Alarm = i_sign * i_curr_Temp <= Alarm_Temp_HL[1] ? 1 : 0;
123 }
124
125 //-----
126 // 定时器T0溢出中断持续读取温度管数据
127 // 并控制报警输出及数码管显示
128 //-----
129 ISR (TMR0_OVF_vect)
130 {

```

```

131     static INT8U Bx = 0;
132     TCNT0 = 256 - F_CPU / 1024.0 * 0.2;           //0.2 s 定时初值
133     Read_Temperature();                          //读取温度
134     if (!DS18B20_ERROR) Temp_display_and_alarm(); //温度显示并设置报警标识
135     if (HI_Alarm) HI_BI_LED_ON(); else HI_BI_LED_OFF(); //高温 LED 闪烁
136     if (LO_Alarm) LO_BI_LED_ON(); else LO_BI_LED_OFF(); //低温 LED 闪烁
137     if (HI_Alarm || LO_Alarm)                  //报警声音输出
138     {
139         if (Bx) BEEP_1(); else BEEP_0();
140         Bx = !Bx;
141     }
142 }
143
144 //-----
145 // 主程序
146 //-----
147 int main()
148 {
149     DDRA = 0xFF; DDRB = 0xFF;
150     DDRC = 0xFF; DDRD = 0xFF;
151     Init_MAX695X();                      //695X 初始化
152     HI_BI_LED_OFF();
153     LO_BI_LED_OFF();
154     Set_Alarm_Temp_Value(70, -20);       //设置报警温度上下限
155     Read_Temperature();                 //读取温度
156     _delay_ms(1000);                   //1 s 延时
157     TCCR0 = 0x05;                     //T0 预设分频:1024
158     TCNT0 = 256 - F_CPU / 1024.0 * 0.2; //晶振 1 MHz, 0.2 s 定时初值
159     TIMSK = _BV(TOIE0);               //允许定时器 0 溢出中断读取温度
160     sei();                           //开中断
161     while(1);
162 }

001 //----- DS18B20.c -----
002 // 名称: DS18B20 温度传感器程序
003 //-----
004 #include <avr/io.h>
005 #include <util/delay.h>
006 #define INT8    signed char
007 #define INT8U   unsigned char
008 #define INT16U  unsigned int
009 //DS18B20 引脚定义
010 #define DQ PA5

```

```

011 //设置数据方向
012 #define DQ_DDR_0() DDRA &= ~_BV(DQ)
013 #define DQ_DDR_1() DDRA |= _BV(DQ)
014 //温度管引脚操作定义
015 #define DQ_1() PORTA |= _BV(DQ)
016 #define DQ_0() PORTA &= ~_BV(DQ)
017 #define RD_DQ_VAL() (PINB & _BV(DQ)) //注意保留这一行的括号
018
019 //温度小数对照表(4 位的温度值 0000~1111 对应 16 个小数位)
020 const INT8U df_Table[] = {0,1,1,2,3,3,4,4,5,6,6,7,8,8,9,9};
021 //从 DS18B20 读取的温度值
022 INT8U Temp_Value[] = {0x00,0x00};
023 //传感器状态标志
024 INT8U DS18B20_ERROR = 0;
025 //-----
026 //读取报警温度上下限,为进行正负数比较,此处注意设为 INT8 类型(不是 INT8U)
027 //取值范围为 -128 ~ +127,DS18B20 支持范围为 -50 ~ +125
028 INT8 Alarm_Temp_HL[2];
029 //-----
030 // 初始化 DS18B20
031 //-----
032 INT8U Init_DS18B20()
033 {
034     INT8U status;
035     DQ_DDR_1(); DQ_0(); _delay_us(500); //主机拉低 DQ,占领总线
036     DQ_DDR_0(); _delay_us(50); //DQ 设为输入
037     status = RD_DQ_VAL(); _delay_us(500); //读总线,为 0 时器件在线
038     DQ_1(); //释放总线
039     return status; //返回器件状态(0 为正常)
040 }
041
042 //-----
043 // 读 1 字节
044 //-----
045 INT8U ReadOneByte()
046 {
047     INT8U i, dat = 0;
048     for (i = 0; i < 8; i++) //串行读取 8 位
049     {
050         DQ_DDR_1(); DQ_0(); //写 0 拉低 DQ 占领总线
051         DQ_DDR_0(); //读 DQ 引脚
052         if(RD_DQ_VAL()) dat |= _BV(i); //读取的第 i 位放入 dat 内对应位置
053         _delay_us(80); //延时

```

```

054      }
055      return dat;                                //返回读取的 1 字节数据
056  }
057
058 //-----
059 // 写 1 字节
060 //-----
061 void WriteOneByte(INT8U dat)
062 {
063     INT8U i ;
064     for (i = 0x01; i != 0x00; i <<= 1)          //串行写入 8 位
065     {
066         DQ_DDR_1(); DQ_0();                      //写 0 拉低 DQ 占领总线
067         if (dat & i) DQ_1(); else DQ_0();          //向 DQ 数据线写 0/1
068         _delay_us(80);                           //延时
069         DQ_1();                                 //释放总线
070     }
071 }
072
073 //-----
074 // 读取温度值
075 //-----
076 void Read_Temperature()
077 {
078     if( Init_DS18B20() != 0x00 )                //DS18B20 故障
079         DS18B20_ERROR = 1;
080     else
081     {
082         WriteOneByte(0xCC);                     //跳过序列号
083         WriteOneByte(0x44);                     //启动温度转换
084         Init_DS18B20();
085         WriteOneByte(0xCC);                     //跳过序列号
086         WriteOneByte(0xBE);                     //读取温度寄存器
087         Temp_Value[0] = ReadOneByte();          //读取当前温度低 8 位
088         Temp_Value[1] = ReadOneByte();          //读取当前温度高 8 位
089         Alarm_Temp_HL[0] = ReadOneByte();        //读取高温报警值
090         Alarm_Temp_HL[1] = ReadOneByte();        //读取低温报警值
091         DS18B20_ERROR = 0;
092     }
093 }
094
095 //-----
096 // 设置 DS18B20 温度报警值(含配置寄存器)(注意两字节的温度数据为有符号数)

```

```

097 //-----
098 void Set_Alarm_Temp_Value(INT8 ha, INT8 la)
099 {
100     Init_DS18B20();
101     WriteOneByte(0xCC);           //跳过序列号
102     WriteOneByte(0x4E);          //发送写 DS18B20 寄存器命令
103     WriteOneByte(ha);            //写 TH 寄存器
104     WriteOneByte(la);            //写 TL 寄存器
105     WriteOneByte(0x7F);          //写配置寄存器(设为 12 位精度)
106     Init_DS18B20();
107     WriteOneByte(0xCC);          //跳过序列号
108     WriteOneByte(0x48);          //将寄存器数据写入 EEPROM
109 }
110
111 //-----
112 // 转换温度数据,返回温度符号(INT8),整数部分和小数部分(INT8U)
113 //-----
114 void Convert_Temperature(INT8 * sign, INT8U * iTemp, INT8U * fTemp)
115 {
116     * sign = 1; //温度符号符号为正
117     //如果为负数则取反加 1,并设置负号标识
118     if (Temp_Value[1] & 0xF8) == 0xF8)
119     {
120         Temp_Value[1] = ~Temp_Value[1];
121         Temp_Value[0] = ~Temp_Value[0] + 1;
122         if (Temp_Value[0] == 0x00) Temp_Value[1]++;
123         * sign = -1;
124     }
125     //查表得到温度小数部分
126     * fTemp = df_Table[Temp_Value[0] & 0x0F];
127     //获取温度整数部分
128     * iTemp = (Temp_Value[0] >> 4)|(Temp_Value[1] << 4);
129 }

```

## 5.12 用 1602 LCD 与 DS18B20 设计的温度报警器

本例设置了 DS18B20 报警温度上下限,温度超出-20~70 °C 的范围时将触发报警信号,另外还添加了 DS18B20 的 ROMCODE 及报警温度下限显示功能。本例电路及部分运行效果如图 5-12 所示。

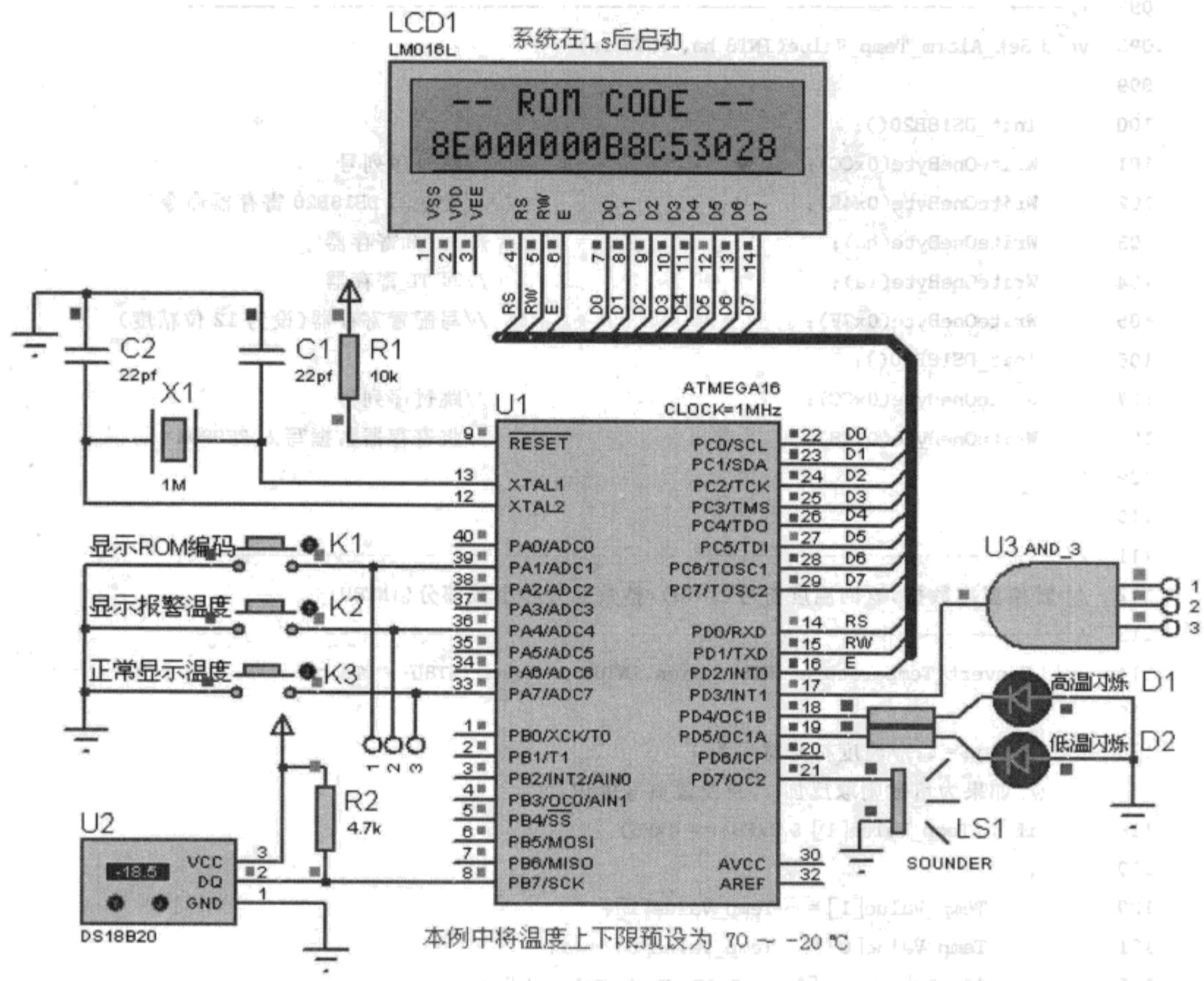


图 5-12 用 1602 LCD 与 DS18B20 设计的温度报警器

## 1. 程序设计与调试

本例与上一案例有较多相似之处。对于本例，下面重点讨论 DS18B20 的光刻 ROMCODE 的作用，ROMCODE 读取与显示以及如何在 1-Wire 方式下利用 ROMCODE 实现多点温度监测。

DS18B20 的 8 字节 64 位唯一光刻 ROMCODE 在出厂时即被设定，这 64 位由高到低分别是：

- ① 8 位的循环冗余校验码 CRC (cyclic redundancy check)；
- ② 48 位的序列号 (Serial Number)；
- ③ 8 位的 DS18B20 1-Wire 家族代码 (family code) 28H。

其中最低的 8 位家族代码 (family code) 28H 是固定的，最高的 8 位 CRC 是其后 56 位 (48+8) 编码的循环冗余校验码，总共 64 位 ROMCODE 可以看成是各 DS18B20 的唯一“地址码”。正是由于 DS18B20 具有唯一“地址码”，在 1-Wire 单总线上才可能同时挂装多个 DS18B20 而不会产生混淆。

要分别读取不同的 DS18B20 的数字温度，首先要分别获取各温度传感器的 ROMCODE，

本例函数 `Display_RomCode` 即可用于读取并显示单只 DS18B20 的 ROMCODE。

由技术手册可知,读取 ROMCODE 的命令是“33H”,该函数在初始化传感器后随即发送读 ROMCODE 命令,然后用 8 次循环读取 8 个字节(64 位)的 ROMCODE,所读取的各字节按十六进制形式转换为 2 位字符,由于读取顺序是先低字节后高字节,语句 `sprintf(t,"%02X", ReadOneByte())` 先将当前字节转换为 2 位大写字符(十六进制形式,a~f 转换为大写),内存复制函数 `memcpy` 则将每次读取并转换得到的 2 个字符按由后向前的顺序(14,15)(12,13)…(2,3)(0,1)逐一存入字符串 `RomCodeString`。

在读取所有 8 字节 64 位 ROMCODE 后,`Display_RomCode` 再调用 `LCD_show_string` 函数在液晶屏上显示 ROMCODE。

如果希望 Proteus 仿真环境下的多只 DS18B20 具有不同的 ROMCODE,可在其属性对话框中设置 ROM Serial Number。现假定有 4 只 DS18B20 的 ROMCODE 已经被分别提前读取,接下来它们被同时挂装到单片机 PB7 引脚上,为分别读取 4 个传感器的温度数据,代码可按下述流程编写:

- ① 主机发送复位脉冲,传感器以存在脉冲(presence pulse)响应,完成初始化过程,这可用本例提供的函数 `Init_DS18B20()` 完成。
- ② 主机发送 ROM 匹配命令“55H”,对应语句: `WriteOneByte(55H)`。
- ③ 主机发送 8 字节(64 位)ROMCODE,使用 for 循环连续 8 次调用函数 `WriteOneByte`。
- ④ 主机发送启动温度转换命令“44H”,对应语句: `WriteOneByte(44H)`。
- ⑤ 如果 DS18B20 使用寄生电源(Parasite Power,由数据线供电),在转换期间主机要对 DQ 引脚应用强上拉,如果使用独立供电可不应用强上拉。
- ⑥ 主机发送复位脉冲,传感器以存在脉冲响应,再次完成初始化。
- ⑦ 主机再次发送 ROM 匹配命令“55H”,语句略。
- ⑧ 主机再次发送 8 字节(64 位)ROMCODE,语句略。
- ⑨ 主机发送读 DS18B20 寄存器命令“BEH”,语句略。
- ⑩ 读取 0~8 号寄存器数据,共 9 字节数据。如果只需要读取温度数据,可只读取 9 个寄存器中的前 2 个。读取时根据需要 2 次或更多次调用函数 `ReadOneByte()` 即可。

通过上述步骤可完成对其中一只传感器数字温度的读取操作,再次执行上述步骤时发送另一传感器的 ROMCODE,这样即可完成对另一只温度传感器的读取操作。如此反复循环,所有 DS18B20 的数字温度即可被逐一读取。

## 2. 实训要求

① 本例首先向 DS18B20 的寄存器及 EEPROM 写入报警温度上限与下限值,后面读取的报警温度值则来自于刚写入的 2、3 号寄存器 TH 与 TL。如果下次上电时不再重新写入报警温度上下限数据,此时 2、3 号寄存器中将不存在报警温度上下限设置。完成本例调试后进一步改进程序,解决这个问题。

② 在 PB7 引脚上同时挂装 2~3 个 DS18B20,重新编程实现多点温度检测,各点温度可在液晶屏上翻页查看,在所读取温度的平均值超出设定范围时触发报警器。

③ 改用温度传感器 DS1621 或 LM35 重新设计本例。

## 3. 源程序代码



```
002 // 名称: DS18B20 温度传感器程序(单只传感器,采用非寄生供电方式)
003 //-----
004 #include <avr/io.h>
005 #include <avr/interrupt.h>
006 #include <util/delay.h>
007 #include <stdio.h>
008 #include <string.h>
009 #define INT8     signed char           //有符号字节整数
010 #define INT8U    unsigned char
011 #define INT16U   unsigned int
012
013 //DS18B20 引脚定义
014 #define DQ PB7
015 //设置数据方向
016 #define DQ_DDR_0()    DDRB &= ~_BV(DQ)
017 #define DQ_DDR_1()    DDRB |= _BV(DQ)
018 //温度管引脚操作定义
019 #define DQ_1()        PORTB |= _BV(DQ)
020 #define DQ_0()        PORTB &= ~_BV(DQ)
021 #define RD_DQ_VAL()  (PINB & _BV(DQ))//注意保留这一行的括号
022
023 //温度小数对照表(仅保存一位小数,已四舍五入)
024 const INT8U df_Table[] = {0,1,1,2,3,3,4,4,5,6,6,7,8,8,9,9};
025 //传感器状态标志
026 INT8U DS18B20_ERROR = 0;
027 //当前温度显示缓冲
028 char Curr_Temp_DispBuffer[] = {" TEMP:           "};
029 //ROM 光刻编码提示信息及 64 位 ROMCODE
030 char RomCodePrompt[] = {" --- ROM CODE --- "};
031 char RomCodeString[] = {"0000000000000000"};
032 //报警温度提示信息及报警温度上下限值
033 char Alarm_Temp[]      = {" --- ALARM TEMP --- "};
034 char Alarm_HI_LO_STR[] = {"Hi:      Lo:      "};
035 //从 DS18B20 读取的 2 字节当前温度数据(需要转换才能得到当前有符号温度值)
036 INT8U Temp_Value[] = {0x00,0x00};
037 //-----
038 //报警温度上下限,DS18B20 温度范围可在: -55~ + 125 °C
039 //数组中前一位为高温值,后一位为低温值
040 //因为后面要进行有符号数的比较,注意这里设为有符号字节整数类型
041 INT8 Alarm_Temp_HL[2];
042 //-----
043 //高、低温报警标志
044 volatile INT8U HI_Alarm = 0, LO_Alarm = 0;
```

```

045 //液晶相关函数
046 extern void Set_LCD_POS(INT8U x, INT8U y);
047 extern void Write_LCD_Data(INT8U dat);
048 extern void Write_LCD_Command(INT8U cmd);
049 extern void LCD_ShowString(INT8U x, INT8U y, char * str);
050 //-----
051 // 初始化 DS18B20
052 //-----
053 INT8U Init_DS18B20()
054 {
055     INT8U status;
056     DQ_DDR_1(); DQ_0(); _delay_us(500); //主机拉低 DQ, 占领总线
057     DQ_DDR_0(); _delay_us(50); //DQ 设为输入
058     status = RD_DQ_VAL(); _delay_us(500); //读总线, 为 0 时器件在线
059     DQ_1(); //释放总线
060     return status; //返回器件状态(0 为正常)
061 }
062
063 //-----
064 // 读 1 字节
065 //-----
066 INT8U ReadOneByte()
067 {
068     INT8U i, dat = 0;
069     for (i = 0; i < 8; i++) //串行读取 8 位
070     {
071         DQ_DDR_1(); DQ_0(); //写 0 拉低 DQ 占领总线
072         DQ_DDR_0(); //读 DQ 引脚
073         if(RD_DQ_VAL()) dat |= _BV(i); //读取的第 i 位放入 dat 内对应位置
074         _delay_us(80); //延时
075     }
076     return dat; //返回读取的 1 字节数据
077 }
078
079 //-----
080 // 写 1 字节
081 //-----
082 void WriteOneByte(INT8U dat)
083 {
084     INT8U i ;
085     for (i = 0x01; i != 0x00; i <<= 1) //串行写入 8 位
086     {
087         DQ_DDR_1(); DQ_0(); //写 0 拉低 DQ 占领总线

```



```
088         if (dat & i) DQ_1(); else DQ_0();      //向 DQ 数据线写 0/1
089         _delay_us(80);                      //延时
090         DQ_1();                           //释放总线
091     }
092 }
093
094 //-----
095 // 读取温度值
096 //-----
097 void Read_Temperature()
098 {
099     if( Init_DS18B20() != 0x00 )           //DS18B20 故障
100         DS18B20_ERROR = 1;
101     else
102     {
103         WriteOneByte(0xCC);               //跳过序列号
104         WriteOneByte(0x44);               //启动温度转换
105         Init_DS18B20();
106         WriteOneByte(0xCC);               //跳过序列号
107         WriteOneByte(0xBE);               //读取 DS18B20 寄存器(Scratchpad)
108         Temp_Value[0] = ReadOneByte();    //温度寄存器低 8 位
109         Temp_Value[1] = ReadOneByte();    //温度寄存器高 8 位
110         Alarm_Temp_HL[0] = ReadOneByte(); //读高温报警值 TH
111         Alarm_Temp_HL[1] = ReadOneByte(); //读低温报警值 TL
112         DS18B20_ERROR = 0;
113     }
114 }
115
116 //-----
117 // 温度转换与显示(同时刷新报警标志)
118 //-----
119 void Convert_and_Show_Temp()
120 {
121     INT8U ng = 0; //负数标识
122     //当前读取的温度整数部分(有符号)
123     INT8  Curr_int_temp = 0;
124     //小数部分(仅用于附在整数后面显示,不需要再设为有符号)
125     INT8U Curr_df_temp = 0;
126     //如果为负数则取反加 1,并设置负数标识
127     //按技术手册说明,高 5 位为符号位,与上 0xF8 进行 + / - 判断
128     if ( (Temp_Value[1] & 0xF8) == 0xF8)
129     {
130         Temp_Value[1] = ~Temp_Value[1];
```

```

131     Temp_Value[0] = ~Temp_Value[0] + 1;
132     if (Temp_Value[0] == 0x00) Temp_Value[1]++;
133     //负数标识置为 1
134     ng = 1;
135 }
136 //温度整数部分
137 Curr_int_temp = ((Temp_Value[0] & 0xF0) >> 4) | ((Temp_Value[1] & 0x07) << 4);
138 //上面这一行可以改写成:
139 //Curr_int_temp = (Temp_Value[0] >> 4) | (Temp_Value[1] << 4);
140 //温度小数部分
141 Curr_df_temp = df_Table[Temp_Value[0] & 0x0F];
142
143 //如果为负温度则在整数部分前面加"-"
144 if (ng) Curr_int_temp = -Curr_int_temp;
145 //显示当前温度提示文字"-- CURRENT TEMP --"
146 LCD_ShowString(0,0,"-- CURRENT TEMP --");
147 //生成 LCD 显示输出字符串(因 GCC 不支持 sprintf 中使用 %f, 这里是分开显示的)
148 sprintf(Curr_Temp_DispBuffer, " TEMP: %3d. %1d", Curr_int_temp, Curr_df_temp);
149 LCD_ShowString(0,1,Curr_Temp_DispBuffer);
150
151 //在最后面补充显示温度符号°C(根据本例 1602 液晶技术手册, 其中"°"的编码为 0xDF)
152 //在度的符号后面再输出 C('C'的编码为 0x43)
153 LCD_ShowString(12,1,"\\xDF\\x43");           //注意"\x"不要写成"\0x"
154 //上面这一行还可以用下面的代码代替
155 //Set_LCD_POS(12,1); Write_LCD_Data(0xDF); Set_LCD_POS(13,1); Write_LCD_Data('C');
156
157 //刷新报警标志(报警温度仅为 1 字节整数, 因此不进行小数部分比较)
158 HI_Alarm = (Curr_int_temp >= Alarm_Temp_HL[0]) ? 1:0;
159 LO_Alarm = (Curr_int_temp <= Alarm_Temp_HL[1]) ? 1:0;
160 }
161
162 //-----
163 // 设置 DS18B20 温度报警值(含配置)
164 //-----
165 void Set_Alarm_Temp_Value(int ht,int lt)
166 {
167     Init_DS18B20();                      //初始化 DS18B20
168     WriteOneByte(0xCC);                  //跳过序列号
169     WriteOneByte(0x4E);                  //发送写 DS18B20 寄存器命令
170     WriteOneByte(ht);                   //写 TH
171     WriteOneByte(lt);                   //写 TL
172     WriteOneByte(0x7F);                  //写配置寄存器, 12 位精度(最高精度)
173     Init_DS18B20();                    //重新初始化

```

```

174     WriteOneByte(0xCC);           //跳过序列号
175     WriteOneByte(0x48);           //将 TH、TL 及 Config 寄存器写入对应的 EEPROM
176 }
177
178 //-----
179 // 显示 RomCode
180 //-----
181 void Display_RomCode()
182 {
183     INT8U i;
184     char t[3];
185     LCD_ShowString(0,0,RomCodePrompt); //第 1 行显示提示信息串
186     Init_DS18B20();                //初始化 DS18B20
187     WriteOneByte(0x33);            //发送读 RomCode 命令
188     for (i = 0; i < 8; i++)       //读取 8 字节(64 位)RomCode,从低字节开始读取
189     {
190         sprintf(t, "%02X", ReadOneByte()); //将当前字节转换为十六进制字符串(2 字符)
191         //将各字节转换后的 2 个十六进制字符由后向前存入 RomCodeString
192         //各字节的 2 字符存入顺序依次是:(14,15)(12,13)(10,11).....(2,3)(0,1)
193         memcpy(RomCodeString + 14 - 2 * i, t, 2);
194         //上面这一行还可以改成以下两行
195         //RomCodeString[15 - 2 * i - 1] = t[0];
196         //RomCodeString[15 - 2 * i ]      = t[1];
197     }
198     LCD_ShowString(0,1,RomCodeString); //第 2 行显示 64 位 RomCode(共 16 个十六进制字符)
199 }
200
201 //-----
202 // 显示报警温度
203 //-----
204 void Disp_Alarm_Temperature()
205 {
206     sprintf(Alarm_HI_LO_STR,"Hi: %4d Lo: %4d ",Alarm_Temp_HL[0],Alarm_Temp_HL[1]);
207     LCD_ShowString(0,0,Alarm_Temp);           //显示标题文字
208     LCD_ShowString(0,1,Alarm_HI_LO_STR);    //显示高低报警温度
209 }
210
211 //----- main.c -----
212 // 名称: 用 1602 LCD 与 DS18B20 设计的温度报警器
213 //-----
214 // 说明: 本例运行时,如果按下 K1,K2,K3 可分别显示 ROMCODE, 报警温度上下限,
215 //        以及实时显示当前温度,在当前温度在 70~ - 20 ℃ 范围之外时报警指
216 //        示灯闪烁,并同时输出报警声音

```

```

007 //
008 //-----
009 #include <avr/io.h>
010 #include <avr/interrupt.h>
011 #include <util/delay.h>
012 #include <string.h>
013 #define INT8U unsigned char
014 #define INT16U unsigned int
015
016 //液晶相关函数
017 extern void Initialize_LCD();
018 extern void Write_LCD_Command(INT8U cmd);
019 extern void LCD_ShowString(INT8U x, INT8U y, char * str);
020
021 //温度传感器相关函数与相关变量
022 extern void Read_Temperature();
023 extern void Convert_and_Show_Temp();
024 extern void Set_Alarm_Temp_Value(int ha, int la);
025 extern void Display_RomCode();
026 extern void Disp_Alarm_Temperature();
027 extern char Current_Temp_Display_Buffer[];
028 extern INT8U DS18B20_ERROR;
029 extern volatile INT8U HI_Alarm, LO_Alarm;
030
031 //按键定义
032 #define K1_DOWN() (PIN_A & _BV(PA1)) == 0x00 //查看 ROMCODE
033 #define K2_DOWN() (PIN_A & _BV(PA4)) == 0x00 //显示报警温度
034 #define K3_DOWN() (PIN_A & _BV(PA7)) == 0x00 //正常显示温度,越界时报警
035
036 //报警指示灯操作定义
037 #define H_LED_Blink() PORTD ^= _BV(PD4) //高温报警闪烁
038 #define L_LED_Blink() PORTD ^= _BV(PD5) //低温报警闪烁
039 #define H_LED_OFF() PORTD &= ~_BV(PD4) //高温指示灯灭
040 #define L_LED_OFF() PORTD &= ~_BV(PD5) //低温指示灯灭
041
042 //蜂鸣器定义
043 #define BEEP() PORTD ^= _BV(PD7)
044 //当前操作码,初始设 3,默认进行温度显示与报警,主程序与中断函数
045 //共享此变量,注意添加 volatile.
046 volatile INT8U curr_op = 3;
047 //-----
048 // 主函数
049 //-----

```



```
050 int main()
051 {
052     DDRA = 0x00; PORTA = 0xFF; //配置端口
053     DDRC = 0xFF;
054     DDRD = ~_BV(PD3); PORTD |= _BV(PD3);
055
056     Initialize_LCD(); //液晶初始化并显示提示信息
057     LCD_ShowString(0,0,"DS18B20 DEMO PRG");
058     LCD_ShowString(0,1,"    waiting...    ");
059
060     TCCR0 = 0x01; //预分频:1
061     TCNT0 = 256 - F_CPU / 1 * 0.0002; //晶振 1 MHz, 200 μs 定时
062     MCUCR = 0x08; //INT1 为下降沿触发
063     GICR = _BV(INT1); //INT1 中断使能
064     SREG = 0x80; //开中断
065
066     Set_Alarm_Temp_Value(70, -20); //设置报警温度上下限为 70 °C, -20°C
067     Read_Temperature(); //读取当前温度
068     _delay_ms(1000); //延时 1 s
069     while(1)
070     {
071         switch (curr_op) //根据当前操作代号 curr_op 完成不同操作
072         {
073             case 1: Display_RomCode(); //显示 DS18B20 RomCode
074                 break;
075             case 2: Disp_Alarm_Temperature(); //显示报警温度上下限
076                 break;
077             case 3: Read_Temperature(); //读取当前温度
078                 if ( !DS18B20_ERROR )
079                 {
080                     Convert_and_Show_Temp(); //转换并显示温度
081                     if (HI_Alarm == 1 || LO_Alarm == 1) //越界时报警
082                     {
083                         TIMSK |= _BV(TOIE0); _delay_ms(400);
084                         TIMSK &= ~_BV(TOIE0); _delay_ms(400);
085                     }
086                     else { H_LED_OFF(); L_LED_OFF(); } //否则关闭 LED
087                 }
088                 break;
089         }
090         _delay_ms(100);
091     }
092 }
```

```

093
094 //-----
095 // INT1 中断根据不同按键选择不同操作代号
096 //-----
097 ISR (INT1_vect)
098 {
099     if      (K1_DOWN()) curr_op = 1; //根据不同按键设置不同的操作代号
100     else if (K2_DOWN()) curr_op = 2;
101     else if (K3_DOWN()) curr_op = 3;
102     Write_LCD_Command(0x01); //清除屏幕(0x01 为清屏命令)
103     _delay_ms(50);
104 }
105
106 //-----
107 // 定时器中断,控制警报声音输出及对应指示灯闪烁
108 //-----
109 ISR (TIMER0_OVF_vect)
110 {
111     static INT16U tCount = 0; //计时累加变量(注意设为静态存储类型)
112     TCNT0 = 256 - F_CPU / 1 * 0.0002; //重设定时初值
113     BEEP(); //报警声音输出
114     if ( ++ tCount == 1200) //报警时控制对应指示灯闪烁
115     {
116         tCount = 0;
117         if (HI_Alarm) H_LED_Blink(); //高温闪烁
118         if (LO_Alarm) L_LED_Blink(); //低温闪烁
119     }
120 }

```

## 5.13 温控电机在 L298 驱动下改变速度与方向运行

L298 芯片是一种高电压、大电流双 H 桥式单片集成驱动器,封装形式有 Multiwatt15(V/H)和 PowerSO20,其中 Multiwatt15 封装有垂直式(Vertical)的 L298N 和水平式的 L298HN (Horizontal),PowerSO20 封装有 L298P,L298 可输入标准的 TTL 逻辑电平,可驱动感性负载,如继电器、螺线管、直流电机与步进电机等。本例使用的是 Multiwatt15V 封装的 L298N,它可以驱动两路直流电机,本例仅用它驱动一路电机,当外界温度在 45 ℃以上时电机加速正转,当温度达到 75 ℃及以上时电机全速正转;外界温度小于 10 ℃时电机加速反转,温度在 0 ℃及以下时达到全速反转;温度回到 10~45 ℃之间时电机逐渐停止转动。案例电路及部分运行效果如图 5-13 所示。

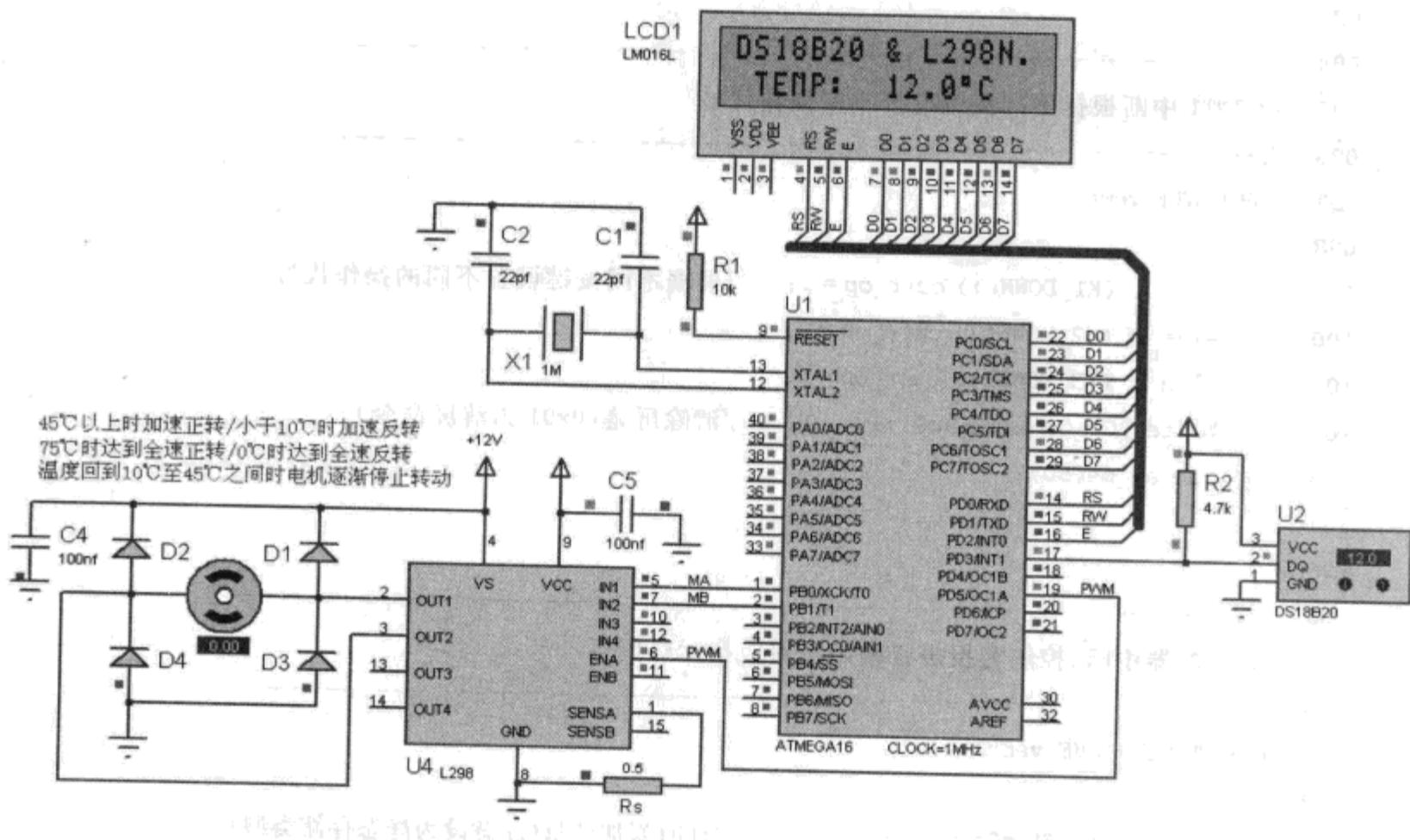


图 5-13 温控电机在 L298 驱动下改变速度与方向运行

## 1. 程序设计与调试

对于本例所使用的用于控制直流电机的 L298N 驱动器,下面简要说明其引脚功能:

- ① IN1/IN2 引脚是 TTL 兼容的 H 桥 A 控制输入端,OUT1/OUT2 是 H 桥 A 输出;
- ② IN3/IN4 与 OUT3/OUT4 则对应 H 桥 B 的输入与输出;
- ③ SENSA 与 SENDB 引脚分别与地之间串接  $R_s$  电阻,分别控制 H 桥 A/B 的负载电流;
- ④ VS 为功率输出部分提供电压,VCC 为逻辑控制部分提供电压;
- ⑤ ENA 与 ENB 分别使能或禁止 H 桥 A/B。

第 3 章中已经讨论过直流电机正反转控制的 H 桥驱动电路,在本例中:

- ① ENA 为高电平时,IN1/IN2=(1,0)则电机正转,IN1/IN2=(0,1)则电机反转;
- ② ENA 为高电平时,IN1=IN2,即 IN0/IN1=(0,0)或(1,1)时,电机快速过渡到停止状态;
- ③ ENA 为低电平时,电机不受 IN1 与 IN2 控制,由自由运行状态逐渐过渡到停止状态。

根据上述 L298N 驱动器的简要说明,本例的程序编写就比较容易了。下面再讨论一下电机的转速控制问题,对于使能 H 桥 A 的 ENA 引脚,本例通过向其输入 PWM 信号来调节电机正转或反转转速。

为通过输出 PWM 信号控制电机转速,主程序通过下面两行语句使 T/C1 工作于 10 位正向 PWM 方式:

```
TCCR1A = 0x83; //10 位 PWM(1023),正向 PWM
TCCR1B = 0x02; //时钟 8 分频,PWM 频率: F_CPU/8/2046
```

PWM 波形由 OC1A 引脚输出,程序通过调整 OCR1A 寄存器的值即可改变输出波形的

占空比，在正向 PWM 模式下，OCR1A 取值越大则占空比越大，电机转速越快，反之则越慢，但两个端点值 0 和 1023 例外，OCR1A 取 0 时占空比为 100%，取值 1023 时则为 0%。正是因为这个原因，当 OCR1A 取值越来越大时，如果到达极值 1023 则将其重新赋值为 0，反之，在递减 OCR1A 的过程中，如果 OCR1A 到达 0 时则将其重新赋值为 1023。

本例程序通过占空比控制变量 PWMx 调节 OCR1A，重新改变输出占空比，实现对电机的调速控制，源程序给出了对相关语句的详细说明，这里不再赘述。

## 2. 实训要求

① 在案例电路中再添加一路直流电机和一只温度传感器，使系统可同时显示两个位置的温度并控制电机运行。

② 进一步编程加强本例功能，用 3 个自定义液晶字符循环显示来模拟显示电机正转，另 3 个自定义字符则用于模拟显示电机反转。

## 3. 源程序代码

```

001 //----- main.c -----
002 // 名称：温控电机在 L298N 驱动下改变速度与方向运行
003 //-----
004 // 说明：本例运行过程中：
005 //      1. 外界温度在 45 ℃ 以上时电机加速正转/小于 10 ℃ 时加速反转
006 //      2. 温度达到 75 ℃ 及以上时电机全速正转/温度在 0 ℃ 及以下时达到全速反转
007 //      3. 温度回到 10~45 ℃ 之间时电机快速过渡到停止状态
008 //      4. 通过虚拟示波器可观察 PWM 波形
009 //
010 //-----
011 #include <avr/io.h>
012 #include <avr/interrupt.h>
013 #include <util/delay.h>
014 #include <string.h>
015 #include <stdio.h>
016 #define INT8     signed char
017 #define INT8U    unsigned char
018 #define INT16U   unsigned int
019
020 //温度传感器相关函数及相关变量
021 extern void Read_Temperature();
022 extern void Convert_Temp_Data();
023 extern volatile INT8U DS18B20_ERROR;
024 extern volatile INT8U Temp_Value[];
025 extern volatile INT8  Curr_int_temp ;
026 extern volatile INT8U Curr_df_temp;
027
028 //液晶相关函数

```



```
029 extern void Initialize_LCD();
030 extern void Set_LCD_POS(INT8U x, INT8U y);
031 extern void Write_LCD_Data(INT8U dat);
032 extern void LCD_ShowString(INT8U x, INT8U y, char * str);
033
034 //L298N 控制引脚操作定义
035 #define MA_1() PORTB |= _BV(PB0)
036 #define MA_0() PORTB &= ~_BV(PB0)
037 #define MB_1() PORTB |= _BV(PB1)
038 #define MB_0() PORTB &= ~_BV(PB1)
039 //-----
040 // 主函数
041 //-----
042 int main()
043 {
044     DDRB = 0xFF;                                //端口定义
045     DDRC = 0xFF;
046     DDRD = 0xFF & ~_BV(PD3); PORTD |= _BV(PD3);
047
048     Initialize_LCD();                          //初始化液晶,然后输出两行提示信息
049     LCD_ShowString(0,0,"DS18B20 & L298N.");
050     LCD_ShowString(0,1,"Control Motor... ");
051     Read_Temperature();                      //读取当前温度(含报警温度)
052     _delay_ms(1000);                         //延时 1 s 后在第二行输出 16 个空格,清除第二行
053     LCD_ShowString(0,1, " ");
054
055     TCCR1A = 0x83;                            //10 位 PWM(1023),正向 PWM
056     TCCR1B = 0x02;                            //时钟 8 分频,PWM 频率:F_CPU/8/2046
057
058     //设置 T0 中断
059     TCCR0 = 0x05;                            //预分频:1024
060     TCNT0 = 256 - F_CPU / 1024 * 0.2;      //晶振 1 MHz,0.2 s 定时
061     TIMSK = _BV(TOIE0);                     //允许 T0 定时器中断
062     sei();                                  //开中断
063     while(1);
064
065 //-----
066 // 定时器中断,持续读取当前温度,刷新 LCD 显示并控制 L298N 变速变向运行
067 //-----
068 ISR (TIMER0_OVF_vect)
069 {
070     //上次读取的温度数据备份
071     static INT8U Back_Temp_Value[] = {0x00,0x00};
```

```

072 //当前温度显示缓冲
073 char Curr_Temp_DispBuffer[17];
074 //PWMx 通过改变比较寄存器 OCR1A 来改变占空比
075 INT16U PWMx = 0;
076
077 Read_Temperature();           //读取温度
078 if ( DS18B20_ERROR ) return; //读错时返回
079 //读取正常且温度发生变化则刷新显示,否则返回
080 if ( Temp_Value[0] != Back_Temp_Value[0] ||
081     Temp_Value[1] != Back_Temp_Value[1] )
082 {
083     //备份本次读取的温度数据
084     Back_Temp_Value[0] = Temp_Value[0];
085     Back_Temp_Value[1] = Temp_Value[1];
086     //转换温度数据,得到温度的整数与小数部分
087     Convert_Temp_Data();
088
089     //生成 LCD 显示输出字符串 Curr_Temp_DispBuffer
090     //((因 GCC 不支持 sprintf 使用 %f, 这里将整数与小数部分分开构造)
091     //格式串中\xDF\x43 是:""与"C"的编码
092     sprintf(Curr_Temp_DispBuffer, " TEMP: %3d.%1d\xDF\x43",
093             Curr_int_temp,Curr_df_temp);
094     //按格式:" TEMP: XXX.XC"显示当前温度值及温度符号"C"
095     LCD_ShowString(0,1,Curr_Temp_DispBuffer);
096 }
097 else return;
098
099 //温度到达 75 度或 0 度时,电机全速转动,占空比为 100 %
100 if (Curr_int_temp > 75) Curr_int_temp = 75;
101 if (Curr_int_temp < 0) Curr_int_temp = 0;
102 //大于或等于高温 45 度时加速正转,75 度时全速运行
103 if ( Curr_int_temp >= 45 )
104 {
105     MA_1(); MB_0();           //正转
106     PWMx = (Curr_int_temp - 45) / 30.0 * 1023;
107 }
108 //小于或等于低温 10 度时加速反转,0 度时全速运行
109 else if ( Curr_int_temp <= 10 )
110 {
111     MA_0(); MB_1();           //反转
112     PWMx = (10 - Curr_int_temp) / 10.0 * 1023;
113 }
114 //否则快速过渡到停止

```



```
115     else
116     {
117         MA_0(); MB_0(); //或 MA_1(); MB_1(); //电机停止
118         PWMx = 1023;    //设 1023 时可使 PWMx 异或为 0,EA 将呈现高电平,电机快速停止
119         //PWMx = 0;      //如果设为 0,则 EA 将呈现低电平,电机由自由转动过渡逐渐停止
120     }
121     //PWMx 遇到极值时用异或(^)交换 0 ->1023,1023 ->0 (1023 即 0x03FF)
122     if (PWMx == 0 || PWMx == 1023) PWMx ^= 0x03FF;
123     OCR1A = PWMx;                      //调整输出比较寄存器
124     TCNT0 = 256 - F_CPU / 1024 * 0.2; //重设 T/C0 定时初值
125 }
```

```
001 //----- DS18B20.c -----
002 // 名称: DS18B20 温度传感器程序
003 //-----
004 #include <avr/io.h>
005 #include <util/delay.h>
006 #define INT8     signed char           //有符号字节整数
007 #define INT8U    unsigned char
008 #define INT16U   unsigned int
009
010 //DS18B20 引脚定义
011 #define DQ PD3
012 //设置数据方向
013 #define DQ_DDR_0()    DDRD &= ~_BV(DQ)
014 #define DQ_DDR_1()    DDRD |= _BV(DQ)
015 //温度管引脚操作定义
016 #define DQ_1()        PORTD |= _BV(DQ)
017 #define DQ_0()        PORTD &= ~_BV(DQ)
018 #define RD_DQ_VAL()   (PIND & _BV(DQ)) //注意保留这一行的括号
019
020 //温度小数表(低字节中的低四位对应 16 个小数位)
021 const INT8U df_Table[] = {0,1,1,2,3,3,4,4,5,6,6,7,8,8,9,9};
022 //传感器状态标志
023 volatile INT8U DS18B20_ERROR = 0;
024 //从 DS18B20 读取的 2 字节当前温度数据(需要转换才能得到当前有符号温度值)
025 volatile INT8U Temp_Value[] = {0x00,0x00};
026 //以下两变量的值由 Temp_Value 中的 2 字节转换而来
027 //当前读取的温度整数部分(有符号字节整数)
028 volatile INT8  Curr_int_temp = 0;
029 //当前读取的温度小数部分(仅用于附在整数后面显示,不需要再设为有符号)
030 volatile INT8U Curr_df_temp = 0;
031 //-----
```

```

032 // 初始化 DS18B20
033 //-----
034 INT8U Init_DS18B20()
035 {
036     INT8U status;
037     DQ_DDR_1(); DQ_0(); _delay_us(500); //主机拉低 DQ, 占领总线
038     DQ_DDR_0(); _delay_us(50); //DQ 设为输入
039     status = RD_DQ_VAL(); _delay_us(500); //读总线, 为 0 时器件在线
040     DQ_1(); //释放总线
041     return status; //返回器件状态(0 为正常)
042 }
043
044 //-----
045 // 读 1 字节
046 //-----
047 INT8U ReadOneByte()
048 {
049     INT8U i, dat = 0;
050     for (i = 0; i < 8; i++) //串行读取 8 位
051     {
052         DQ_DDR_1(); DQ_0(); //写 0 拉低 DQ 占领总线
053         DQ_DDR_0(); //读 DQ 引脚
054         if(RD_DQ_VAL()) dat |= _BV(i); //读取的第 i 位放入 dat 内对应位置
055         _delay_us(80); //延时
056     }
057     return dat; //返回读取的 1 字节数据
058 }
059
060 //-----
061 // 写 1 字节
062 //-----
063 void WriteOneByte(INT8U dat)
064 {
065     INT8U i ;
066     for (i = 0x01; i != 0x00; i <<= 1) //串行写入 8 位
067     {
068         DQ_DDR_1(); DQ_0(); //写 0 拉低 DQ 占领总线
069         if (dat & i) DQ_1(); else DQ_0(); //向 DQ 数据线写 0/1
070         _delay_us(80); //延时
071         DQ_1(); //释放总线
072     }
073 }
074

```

```
075 //-----  
076 // 读取温度值  
077 //-----  
078 void Read_Temperature()  
079 {  
080     if( Init_DS18B20() != 0x00 )           //DS18B20 故障  
081         DS18B20_ERROR = 1;  
082     else  
083     {  
084         WriteOneByte(0xCC);                //跳过序列号匹配  
085         WriteOneByte(0x44);                //启动温度转换  
086         Init_DS18B20();  
087         WriteOneByte(0xCC);                //跳过序列号  
088         WriteOneByte(0xBE);                //读取温度寄存器  
089         Temp_Value[0] = ReadOneByte();      //温度低 8 位  
090         Temp_Value[1] = ReadOneByte();      //温度高 8 位  
091         DS18B20_ERROR = 0;  
092     }  
093 }  
094  
095 //-----  
096 // 温度数据转换  
097 //-----  
098 void Convert_Temp_Data()  
099 {  
100     INT8U ng = 0; //负数标识  
101     //如果为负数则取反加 1,并设置负数标识  
102     //按技术手册说明,高 5 位为符号位,与上 0xF8 进行 + / - 判断  
103     if ( (Temp_Value[1] & 0xF8) == 0xF8)  
104     {  
105         Temp_Value[1] = ~Temp_Value[1];  
106         Temp_Value[0] = ~Temp_Value[0] + 1;  
107         if (Temp_Value[0] == 0x00) Temp_Value[1] ++ ;  
108         //负数标识置为 1  
109         ng = 1;  
110     }  
111     //温度整数部分  
112     Curr_int_temp = (Temp_Value[0] >> 4) | (Temp_Value[1] << 4);  
113     //温度小数部分  
114     Curr_df_temp = df_Table[ Temp_Value[0] & 0x0F ];  
115     //如果为负温度则在整数部分前面加" - "  
116     if (ng) Curr_int_temp = - Curr_int_temp;  
117 }
```

## 5.14 PG160128 中文显示日期时间及带刻度显示当前温度

本例运行时,液晶屏除以中文方式显示当前日期时间及环境温度信息以外,还以图形方式显示当前温度信息。本例电路及部分运行效果如图 5-14 所示。

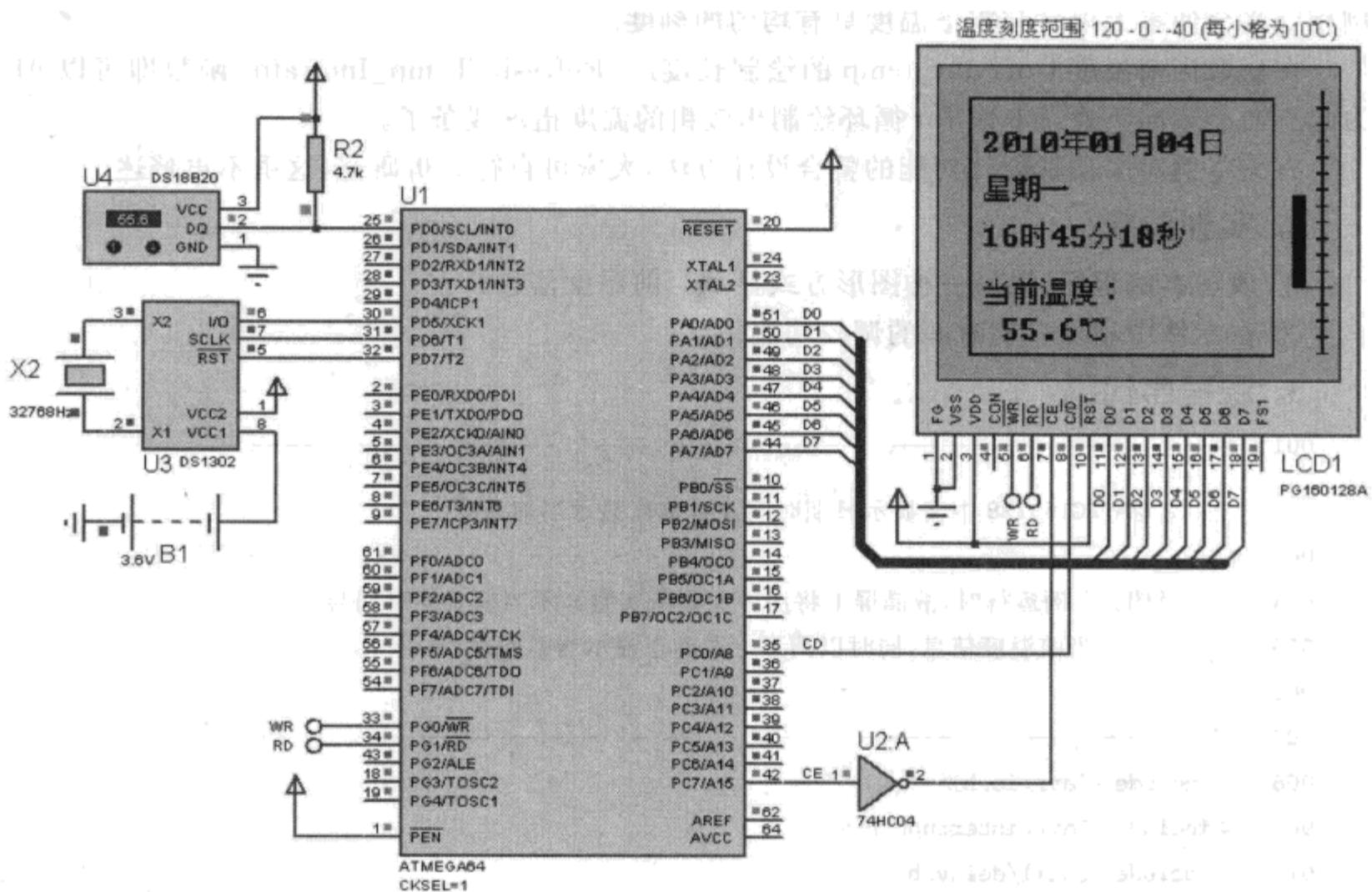


图 5-14 PG160128 中文显示日期时间及带刻度显示当前温度

### 1. 程序设计与调试

本例综合应用了 DS1302、DS18B20 及 PG12864 液晶,对液晶屏的显示控制通过接口扩展方式完成。通过对此前有关案例的学习与调试,大家已经掌握了这些芯片及器件的技术要点与程序设计方法,在完成本例的设计调试后,要进一步提高液晶屏特殊显示功能的设计能力。

本例源程序中提供了刷新显示温度指示器的函数 Refresh\_Temp\_Indicator(), 它以 0℃ 刻度为起点,在向上至 120℃、向下至 -40℃ 的范围内绘制指示线条,超出此范围的则限制在此范围之内。

每当温度变化时(仅针对温度的整数部分),Refresh\_Temp\_Indicator 函数首先擦除先绘制的指示线条,由于较粗的指示线条由 6 条细线构成,因而擦除操作需要 6 次循环。

在擦除线条之后,为从 0~Curr\_int\_temp 绘制线条,程序首先将 Curr\_int\_temp 的值限制于 -40~120 范围之内。根据当前温度 Curr\_int\_temp,以下两行语句可分别计算出从 0℃ 位置(纵坐标为 91)向上或向下绘制线条的长度,其中 ceil 是取最高限函数(ceiling, 或称为取天花板函数),它用于获取大于/等于当前参数的最小整数:

```
len=ceil( Curr_int_temp / 120.0 * (91-7))
len=ceil(-Curr_int_temp / 40.0 * (119-91))
```

其中第一行为顶端留下 7 像素空间,实际可用的绘制范围为:7~91(共 84 像素);第二行为底端预留 9 个像素( $128 - 119 = 9$ ),实际可用的绘制范围为:91~119 (共 28 像素)。

由以上绘制范围可以看出,上下像素比与上下温度比是相等的,即  $84:28 = 120:40$ ,该比例相等将会使零上温度与零下温度具有均匀的刻度。

在获取当前温度 Curr\_int\_temp 的绘制长度后,Refresh\_Temp\_Indicator 函数即可以 91 为起点向上或向下通过 6 次 for 循环绘制出较粗的温度指示线条了。

有关本例对多项软硬件功能的整合设计方法,大家可自行分析研究,这里不再赘述。

## 2. 实训要求

- ① 改写本例程序,以另一种图形方式显示当前温度信息。
- ② 在本例中添加日期时间的调校功能。

## 3. 源程序代码

```
001 //----- main.c -----
002 // 名称: PG160128 中文显示日期时间及带刻度显示当前温度
003 //-
004 // 说明: 本例运行时,液晶屏上将用中文刷新实时显示当前日期时间与
005 //       当前温度信息,同时以图形方式动态显示当前温度
006 //
007 //-
008 #include <avr/io.h>
009 #include <avr/interrupt.h>
010 #include <util/delay.h>
011 #include <stdio.h>
012 #include <math.h>
013 #define INT8     signed   char
014 #define INT8U    unsigned char
015 #define INT16U   unsigned int
016 #define INT32U   unsigned long
017
018 //PG160128D 相关函数
019 extern void LCD_Initialise();
020 extern void Display_Str_at_xy(INT8U x, INT8U y, char * Buffer, INT8U wb);
021 extern void Line(INT8U x1, INT8U y1, INT8U x2, INT8U y2, INT8U Mode);
022
023 //DS1302 实时时钟程序相关函数与变量
024 extern void GetDateTime();
025 extern void Read_Temperature();
026 extern volatile INT8U DateTime[7];
027 extern char * WeeksTable[];
```

```

028
029 //DS18B20 温度传感器相关函数与变量
030 extern INT8U DS18B20_ERROR;
031 extern void Read_Temperature();
032 extern void Convert_Temp_Data();
033 extern volatile INT8 Curr_int_temp;
034 extern volatile INT8U Curr_df_temp;
035
036 //LCD 字符串显示缓冲
037 char Disp_Str_Buffer[14];
038 //-----
039 // 主程序
040 //-----
041 int main()
042 {
043     INT8U i;
044     MCUCR |= _BV(SRE);           //SRE 位置 1, 允许访问外部 SRAM/XMEM
045     XMCRA = 0x00;
046     DDRA = 0xFF;                //配置端口
047     DDRC = 0xFF;
048     DDRD = 0xFF;
049
050     LCD_Initialise();          //初始化 LCD
051     //绘制方框
052     Line(10,10,135,10,1);
053     Line(10,10,10, 120,1);
054     Line(10,120,135,120,1);
055     Line(135,120,135,10,1);
056     //绘制垂直线
057     Line(155,7,155,119,1);
058     //绘制刻度
059     for (i = 2; i < 18; i++) Line(153,i * 7 ,157,i * 7,1);
060     //加长绘制 0 度线
061     Line(149,91,159,91,1);
062     Display_Str_at_xy(16,88,"当前温度:",0);
063     Display_Str_at_xy(60,104,"℃",0);
064
065     TCCR0 = 0x05;                //预分频:1024
066     TCNT0 = 256 - F_CPU / 1024.0 * 0.05;    //晶振 4 MHz, 0.05 s 定时
067     TIMSK = 0x01;                //使能 T0 中断
068     sei();                      //开中断
069     Read_Temperature();         //预读取温度
070     _delay_ms(3000);

```



```
071     while(1);  
072 }  
073  
074 //-----  
075 // 刷新温度指示线条  
076 //-----  
077 void Refresh_Temp_Indicator()  
078 {  
079     INT8U i,len;  
080     //擦除原有的指示线条  
081     for (i = 0; i < 6; i++)  
082         Line(143 + i,0,143 + i,127,0);  
083     //在绘制刻度指示线条时将值限制在 -40~120 之内。  
084     if (Curr_int_temp > 120) Curr_int_temp = 120;  
085     if (Curr_int_temp < -50) Curr_int_temp = -40;  
086     //根据正负温度整数部分计算线条长度  
087     if (Curr_int_temp > 0)           //零上温度  
088     {  
089         len = ceil(Curr_int_temp / 120.0 * (91 - 7));  
090         for (i = 0; i < 6 ; i++)  
091             Line(143 + i ,91,143 + i ,91 - len,1);  
092     }  
093     else if (Curr_int_temp < 0)      //零下温度  
094     {  
095         len = ceil(- Curr_int_temp / 40.0 * (119 - 91));  
096         for (i = 0; i < 6 ; i++)  
097             Line(143 + i ,91,143 + i ,91 + len,1);  
098     }  
099 }  
100  
101 //-----  
102 // 定时器 0 刷新 LCD 显示(每次均备份上次读取的时间与温度,显示前进行比较,  
103 // 如果变化则刷新显示,这样可减少液晶屏的显示抖动)  
104 //-----  
105 ISR (TIMER0_OVF_vect)  
106 {  
107     static INT8U DateTimeBack[7];           //备份上次读取的时间  
108     static INT8 TempBack = 85;              //温度整数部分备份(有符号数)  
109     INT8U i;  
110     TCNT0 = 256 - F_CPU / 1024.0 * 0.05;    //晶振 4 MHz, 0.05 s 定时  
111  
112     GetDateTime();                      //读取当前日期时间  
113     //显示年月日
```

```

114     if ( DateTime[6] != DateTimeBack[6] || DateTime[4] != DateTimeBack[4] ||
115         DateTime[3] != DateTimeBack[3] )
116     {
117         sprintf(Disp_Str_Buffer,"20 % 02d 年 % 02d 月 % 02d 日",
118                 DateTime[6],DateTime[4],DateTime[3]);
119         Display_Str_at_xy(16,24,Disp_Str_Buffer,0);
120     }
121     //显示星期
122     if ( DateTime[5] != DateTimeBack[5] )
123     {
124         sprintf(Disp_Str_Buffer,"星期 % s",WeeksTable[DateTime[5]-1]);
125         Display_Str_at_xy(16,44,Disp_Str_Buffer,0);
126     }
127     //显示时分
128     if ( DateTime[2] != DateTimeBack[2] || DateTime[1] != DateTimeBack[1] )
129     {
130         sprintf(Disp_Str_Buffer,"% 02d 时 % 02d 分",DateTime[2],DateTime[1]);
131         Display_Str_at_xy(16,64,Disp_Str_Buffer,0);
132     }
133     //显示秒(为减少"时分"显示在秒频繁更新时的抖动,这里将其单独显示)
134     if ( DateTime[0] != DateTimeBack[0] )
135     {
136         sprintf(Disp_Str_Buffer,"% 02d 秒",DateTime[0]);
137         Display_Str_at_xy(72,64,Disp_Str_Buffer,0);
138     }
139     //备份本次读取的时钟信息
140     for ( i = 0; i < 7; i++ ) DateTimeBack[i] = DateTime[i];
141     //读取并显示当前温度
142     Read_Temperature();
143     //传感器错误时返回
144     if ( DS18B20_ERROR ) return;
145     //转换温度数据
146     Convert_Temp_Data();
147     //温度未变化时返回
148     if ( TempBack == Curr_int_temp ) return;
149     //备份本次读取的温度整数部分
150     TempBack = Curr_int_temp;
151     //生成待输出温度字符串
152     sprintf(Disp_Str_Buffer,"% 3d. % d°C ",Curr_int_temp,Curr_df_temp);
153     //显示温度
154     Display_Str_at_xy(16,104,Disp_Str_Buffer,0);
155     //刷新刻度指示线条
156     Refresh_Temp_Indicator();

```

157 }

```

001 //----- LCD_160128.c -----
002 // 名称:PG160128LCD 显示控制程序
003 //-----
004 #include <avr/io.h>
005 #include <avr/pgmspace.h>
006 #include <util/delay.h>
007 #include <string.h>
008 #include <stdio.h>
009 #include <math.h>
010 #include "LCD_160128.h"
011 #define INT8U unsigned char
012 #define INT16U unsigned int
.....限于篇幅,这里省略了此前案例中出现过的类似代码
122 struct typFNT_GB16 // 汉字字模显示数据结构
123 {
124     char Index[2];
125     INT8U Msk[24];
126 };
127
128 //本例汉字点阵库
129 const struct typFNT_GB16 GB_16[] = { //12x12 点阵,宋体小五号,用 Zimo 软件取得点阵
130     {"年"}, {0x20,0x00,0x3F,0xE0,0x42,0x00,0x82,0x00,0x3F,0xC0,0x22,0x00,
131         0x22,0x00,0xFF,0xE0,0x02,0x00,0x02,0x00,0x02,0x00,0x00,0x00} },
132     {"月"}, {0x1F,0x80,0x10,0x80,0x10,0x80,0x1F,0x80,0x10,0x80,0x10,0x80,
133         0x1F,0x80,0x10,0x80,0x10,0x80,0x20,0x80,0x43,0x80,0x00,0x00} },
134     {"日"}, {0x3F,0xC0,0x20,0x40,0x20,0x40,0x20,0x40,0x3F,0xC0,0x20,0x40,
135         0x20,0x40,0x20,0x40,0x20,0x40,0x3F,0xC0,0x20,0x40,0x00,0x00} },
136     {"时"}, {0x00,0x80,0xF0,0x80,0x9F,0xE0,0x90,0x80,0x94,0x80,0xF2,0x80,
137         0x92,0x80,0x90,0x80,0xF0,0x80,0x90,0x80,0x03,0x80,0x00,0x00} },
138     {"分"}, {0x11,0x00,0x11,0x00,0x20,0x80,0x20,0x80,0x40,0x40,0xBF,0xA0,
139         0x08,0x80,0x08,0x80,0x10,0x80,0x20,0x80,0xC7,0x00,0x00,0x00} },
140     {"秒"}, {0x31,0x00,0xE1,0x00,0x25,0x40,0xFD,0x20,0x25,0x20,0x75,0x00,
141         0x69,0x40,0xA0,0x40,0xA0,0x80,0x23,0x00,0x3C,0x00,0x00,0x00} },
142     {"星"}, {0x3F,0xC0,0x20,0x40,0x3F,0xC0,0x20,0x40,0x3F,0xC0,0x24,0x00,
143         0x3F,0xC0,0x44,0x00,0xBF,0xC0,0x04,0x00,0xFF,0xE0,0x00,0x00} },
144     {"期"}, {0x49,0xE0,0xFD,0x20,0x49,0x20,0x79,0xE0,0x49,0x20,0x79,0x20,
145         0x49,0xE0,0xFD,0x20,0x29,0x20,0x45,0x20,0x82,0x60,0x00,0x00} },
146
147     {"一"}, {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40,0xFF,0xE0,
148         0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00} },
149     {"二"}, {0x00,0x00,0x00,0x80,0x7F,0xC0,0x00,0x00,0x00,0x00,0x00,0x00,0x00} ,

```

```

150         0x00,0x00,0x00,0x00,0x00,0xFF,0xE0,0x00,0x00,0x00,0x00} },
151     {"三"},{0x00,0x80,0x7F,0xC0,0x00,0x00,0x00,0x00,0x3F,0x80,
152         0x00,0x00,0x00,0x00,0x00,0x00,0x40,0xFF,0xE0,0x00,0x00} },
153     {"四"},{0x7F,0xE0,0x49,0x20,0x49,0x20,0x49,0x20,0x49,0x20,
154         0x51,0x20,0x61,0xE0,0x40,0x20,0x7F,0xE0,0x40,0x20,0x00,0x00} },
155     {"五"},{0x7F,0xC0,0x08,0x00,0x08,0x00,0x08,0x00,0x7F,0x80,0x08,0x80,
156         0x08,0x80,0x10,0x80,0x10,0x80,0x10,0x80,0xFF,0xE0,0x00,0x00} },
157     {"六"},{0x08,0x00,0x04,0x00,0x04,0x00,0xFF,0xE0,0x00,0x00,0x12,0x00,
158         0x11,0x00,0x20,0x80,0x20,0x80,0x40,0x40,0x80,0x40,0x00,0x00} },
159     {"日"},{0x3F,0xC0,0x20,0x40,0x20,0x40,0x20,0x40,0x3F,0xC0,0x20,0x40,
160         0x20,0x40,0x20,0x40,0x20,0x40,0x3F,0xC0,0x20,0x40,0x00,0x00} },
161
162     {"当"},{0x04,0x00,0x44,0x40,0x24,0x80,0x05,0x00,0xFF,0xC0,0x00,0x40,
163         0x00,0x40,0x7F,0xC0,0x00,0x40,0x00,0x40,0xFF,0xC0,0x00,0x00} },
164     {"前"},{0x11,0x00,0x0A,0x40,0xFF,0xE0,0x00,0x00,0x79,0x40,0x49,0x40,
165         0x79,0x40,0x49,0x40,0x79,0x40,0x48,0x40,0x59,0xC0,0x00,0x00} },
166     {"温"},{0x8F,0xC0,0x48,0x40,0x0F,0xC0,0x88,0x40,0x4F,0xC0,0x40,0x00,
167         0x5F,0xC0,0x95,0x40,0x95,0x40,0x95,0x40,0xBF,0xE0,0x00,0x00} },
168     {"度"},{0x02,0x00,0x7F,0xE0,0x48,0x80,0x7F,0xE0,0x48,0x80,0x4F,0x80,
169         0x40,0x00,0x5F,0x80,0x45,0x00,0x87,0x00,0xB8,0xE0,0x00,0x00} },
170     {":"}, {0x00,0x00,0x00,0x00,0x0C,0x00,0x0C,0x00,0x00,0x00,0x00,0x00,0x00,
171         0x00,0x00,0x0C,0x00,0x0C,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00} },
172     {"℃"}, {0x00,0x00,0xE7,0x40,0xA8,0xC0,0xF0,0x40,0x10,0x00,0x10,0x00,
173         0x10,0x00,0x10,0x40,0x08,0x40,0x07,0x80,0x00,0x00,0x00,0x00} }
174 };

```

.....限于篇幅,这里省略了此前案例中出现过的类似代码

```

191 //-----
192 // 状态位 STA1,STA0 判断(读写指令和读/写数据)
193 //-----
194 INT8U Status_BIT_01()
195 {
196     INT8U i;
197     for(i = 10;i > 0;i--)
198     {
199         if(( * LCMCW & 0x03) == 0x03) break;
200     }
201     return i;                                //错误时返回 0
202 }
203
204 //-----
205 // 状态位 ST3 判断(数据自动写状态)
206 //-----
207 INT8U Status_BIT_3()

```



```
208  {
209      INT8U i;
210      for(i = 10; i > 0; i--)
211      {
212          if(( * LCMCW & 0x08) == 0x08) break;
213      }
214      return i;                                //错误时返回 0
215  }
216
217 //-----
218 // 写双参数的指令
219 //-----
220 INT8U LCD_Write_Command_P2(INT8U cmd, INT8U para1, INT8U para2)
221 {
222     if(Status_BIT_01() == 0) return 1;
223     * LCMDW = para1;
224     if(Status_BIT_01() == 0) return 2;
225     * LCMDW = para2;
226     if(Status_BIT_01() == 0) return 3;
227     * LCMCW = cmd;
228     return 0;                                //成功时返回 0
229 }
230
231 //-----
232 // 写单参数的指令
233 //-----
234 INT8U LCD_Write_Command_P1(INT8U cmd, INT8U para1)
235 {
236     if(Status_BIT_01() == 0) return 1;
237     * LCMDW = para1;
238     if(Status_BIT_01() == 0) return 2;
239     * LCMCW = cmd;
240     return 0;                                //成功时返回 0
241 }
242
243 //-----
244 // 写无参数的指令
245 //-----
246 INT8U LCD_Write_Command(INT8U cmd)
247 {
248     if(Status_BIT_01() == 0) return 1;
249     * LCMCW = cmd;
250     return 0;                                //成功时返回 0
```

```

251 }
252
253 //-----
254 // 写数据
255 //-----
256 INT8U LCD_Write_Data(INT8U dat)
257 {
258     if(Status_BIT_3() == 0) return 1;
259     *LCMDW = dat;
260     return 0;           //成功时返回 0
261 }
262
263 //-----
264 // 读数据
265 //-----
266 INT8U LCD_Read_Data()
267 {
268     if(Status_BIT_01() == 0) return 1;
269     return *LCMDW;
270 }

```

……限于篇幅,这里省略了此前案例中出现过的类似代码

```

01 //-----LCD_160128.h-----
02 // 名称:160128LCD 显示控制程序头文件
03 //-----
……限于篇幅,这里省略了此前案例中出现过的类似代码
29 //T6963C 端口定义
30 #define LCMDW (INT8U *)0x8000      //数据口
31 #define LCMCW (INT8U *)0x8100      //命令口
……限于篇幅,这里省略了此前案例中出现过的类似代码

```

## 5.15 液晶屏曲线显示两路 A/D 转换结果

本例用 PG160128 液晶屏曲线显示两路模/数转换结果,调节两个可变电阻时,液晶屏上除仍以数字方式显示当前电压外,还将电压变化轨迹以曲线方式绘制在液晶屏上。本例电路及部分运行效果如图 5-15 所示。

### 1. 程序设计与调试

本例学习与调试要点部分仍在于液晶屏特殊显示功能的设计与实现。

本例程序中,只有当前电压发生连续变化时才会在液晶屏连续绘制曲线。对于 A/D 通道 0,变量 Pre\_CH0\_Result 用于保存其上一次的 A/D 转换结果,以便判断当前转换结果是否发



生变化。

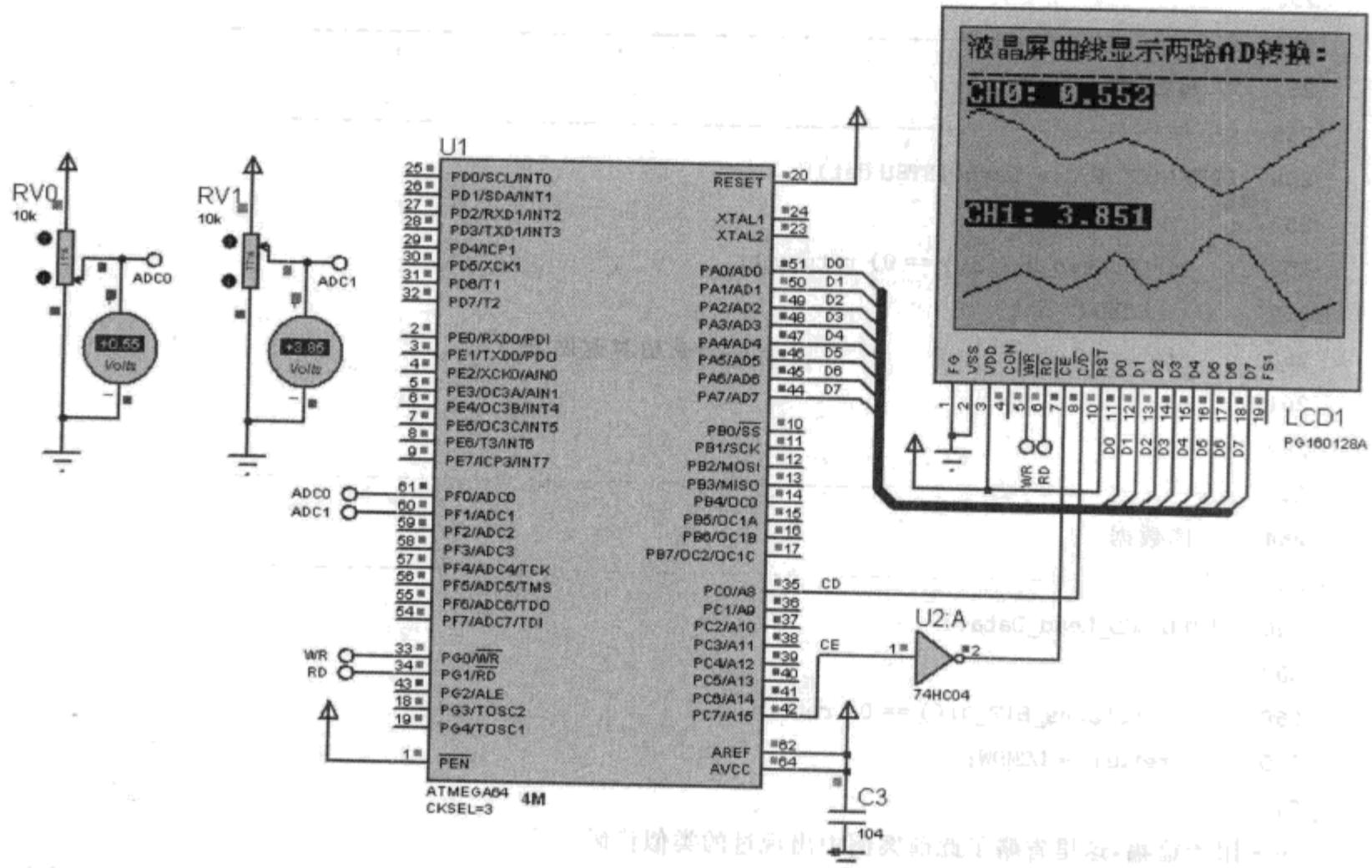


图 5-15 液晶屏曲线显示两路模/数转换结果

为绘制出较为平滑的曲线, 主程序使用了在上一坐标点与当前坐标点之间绘制直线的方法。程序中的变量  $y_0$  是根据当前转换结果  $AD\_Result$  计算得到的当前点的纵坐标, 计算语句如下:

$$y_0 = \text{fabs}(40 - AD\_Result * 40 / 1023.0)$$

语句中的 40 是  $y_0$  的最大取值范围, 由于液晶屏纵坐标是由上向下增长的, 这与平面几何中的坐标系统第一象限是相反的, 因而还要用 40 减去  $AD\_Result * 40 / 1023.0$ , 为避免该语句中因乘除运算的精度问题而导致可能出现负数, 语句中还需要使用绝对值函数  $\text{fabs}$ 。

横坐标  $x_0$  总是 +1 递增。如果当前的  $x_0$  为 0, 则上一像素的纵坐标  $py_0$  与当前点纵坐标  $y_0$  相等, 即  $py_0 = y_0$ 。

经过以上分析与计算, 程序中已经得到了上一坐标点  $(x_0, py_0)$  和当前坐标点  $(x_0 + 1, y_0)$ , 在这两个点之间用 Line 绘制直线, 如此绘制下去, 最后得到的就是平滑的曲线了。

另一通道 CH1 的程序设计方法与上面讨论的通道 CH0 类似, 其代码设计细节留给大家自行阅读分析。另外, 本例使用 Line 函数在相邻两点之间绘制直线之前, 总是先调用 Line 函数完成了一次擦除操作, 其设计目的留给大家自行分析, 这里不再继续讨论了。

## 2. 实训要求

- ① 重新设计程序, 以上、下两个  $180^\circ$  扇形的方式显示两路 A/D 转换结果。
- ② 在本例电路中使用温度传感器 DS18B20, 以自定义图形方式(譬如三角形斜坡方式)显

示外界温度变化。

### 3. 源程序代码

```

01 //----- main.c -----
02 // 名称：液晶屏曲线显示两路模/数转换结果
03 //-----
04 // 说明：本例运行时，AD 转换端口的两路模/数转换结果除以数字方式显示
05 // 在液晶屏上以外，变化过程还会以曲线方式呈现出来
06 //
07 //-----
08 #include <avr/io.h>
09 #include <util/delay.h>
10 #include <stdio.h>
11 #include <math.h>
12 #define INT8U unsigned char
13 #define INT16U unsigned int
14
15 //PG160128 相关函数
16 extern void LCD_Initialise();
17 extern void Display_Str_at_xy(INT8U x, INT8U y, char * Buffer, INT8U wb);
18 extern void Line(INT8U x1, INT8U y1, INT8U x2, INT8U y2, INT8U Mode);
19 //LCD 字符串显示缓冲
20 char Disp_Str_Buffer[14];
21 //-----
22 // 对通道 CH 进行模/数转换
23 //-----
24 INT16U ADC_Convert(INT8U CH)
25 {
26     //ADC 通道选择
27     ADMUX = CH;
28     //ADC 转换置位，启动转换，64 分频
29     ADCSRA = 0xE6; _delay_ms(10);
30     //读取并返回转换结果
31     return (INT16U)(ADCL + (ADCH << 8));
32     //或使用：return ADC;
33 }
34
35 //-----
36 // 主程序
37 //-----
38 int main()
39 {
40     INT16U AD_Result, Pre_CH0_Result = 0, Pre_CH1_Result = 0;

```

```

41     INT8U x0 = 0, y0 = 0, x1 = 0, y1 = 0;
42     INT8U py0 = 0, py1 = 0;
43     float f_result;
44
45     MCUCR |= _BV(SRE);           //SRE 位置 1, 允许访问外部 SRAM/XMEM
46     ADCSRA = 0xE6;              //ADC 转换置位, 启动转换, 64 分频
47     _delay_ms(1000);            //延时等待系统稳定
48     LCD_Initialise();          //初始化 LCD
49     Display_Str_at_xy(0,0,"液晶屏曲线显示两路 AD 转换:",0);
50     Display_Str_at_xy(0,12,"-----",0);
51     ADC_Convert(0); _delay_ms(100); ADC_Convert(1);
52     while(1)
53     {
54         //通道 0 转换与显示-----
55         AD_Result = ADC_Convert(0); //获取通道 0 转换结果
56         //由 10 位精度数字转换为模拟电压
57         f_result = (float)(AD_Result * 5.0 / 1023.0);
58         //按指定格式生成输出字符串
59         sprintf(Disp_Str_Buffer,"CH0: %d.%d - 3d",
60                 (int)f_result,(int)((f_result-(int)f_result)*1000));
61         //液晶显示字符串
62         Display_Str_at_xy(0,20,Disp_Str_Buffer,1);
63         //如果本次转换与上次结果比较后有变化则刷新曲线
64         if (Pre_CH0_Result != AD_Result)
65         {
66             //备份本次转换结果
67             Pre_CH0_Result = AD_Result;
68             //显示曲线 0 (显示区域 0,33 — 159,73)
69             y0 = fabs(40 - AD_Result * 40 / 1023.0);
70             if (x0 == 0)      py0 = y0;
71             if (++x0 == 160) x0 = 0;
72             Line(x0,33,x0 + 1,73,0);
73             Line(x0,33 + py0,x0 + 1,33 + y0,1);
74             py0 = y0;
75         }
76         //通道 1 转换与显示-----
77         //以下注释可参考上面通道 0 的转换说明
78         AD_Result = ADC_Convert(1);
79         f_result = (float)(AD_Result * 5.0 / 1023.0);
80         sprintf(Disp_Str_Buffer,"CH1: %d.%d - 3d",
81                 (int)f_result,(int)((f_result-(int)f_result)*1000));
82         Display_Str_at_xy(0,74,Disp_Str_Buffer,1);
83         if (Pre_CH1_Result != AD_Result)

```

```

84     {
85         Pre_CH1_Result = AD_Result;
86         //显示曲线1(显示区域0,87—159,127)
87         y1 = fabs(40 - AD_Result * 40 / 1023.0);
88         if (x1 == 0)    py1 = y1;
89         if (++x1 == 160) x1 = 0;
90         Line(x1,87,x1+1,159,0);
91         Line(x1,87 + py1,x1+1,87 + y1,1);
92         py1 = y1;
93     }
94 }
95 }

001 //-----LCD_160128.c-----
002 // 名称:PG160128LCD 显示控制程序
003 //-----
.....限于篇幅,本例省略了其他案例中出现过的类似代码
128 //本例汉字点阵库
129 const struct typFNT_GBT16 GB_16[] = { //12 * 12 点阵,宋体小五号,用 Zimo 软件取得点阵
130 {{"液"},{0x81,0x00,0x5F,0xE0,0x25,0x00,0x85,0xC0,0xAA,0x40,0x4E,0xC0,
131           0x5A,0x40,0x49,0x80,0xC8,0x80,0x49,0x40,0x4E,0x20,0x00,0x00}},
132 {{"晶"},{0x1F,0x80,0x10,0x80,0x1F,0x80,0x10,0x80,0x1F,0x80,0x00,0x00,
133           0x7B,0xE0,0x4A,0x20,0x7B,0xE0,0x4A,0x20,0x7B,0xE0,0x00,0x00}},
134 {{"屏"},{0x7F,0xC0,0x40,0x40,0x7F,0xC0,0x50,0x80,0x49,0x00,0x5F,0xC0,
135           0x49,0x00,0x7F,0xE0,0x49,0x00,0x91,0x00,0xA1,0x00,0x00,0x00}},
136 {{"转"},{0x21,0x00,0xF9,0x00,0x47,0xE0,0x61,0x00,0xA7,0xE0,0xFA,0x00,
137           0x23,0xC0,0x38,0x40,0xE0,0x80,0x23,0x00,0x20,0x80,0x00,0x00}},
138 {{"换"},{0x22,0x00,0x27,0x80,0xF8,0x80,0x2F,0xC0,0x29,0x40,0x39,0x40,
139           0xE9,0x40,0x3F,0xE0,0x22,0x80,0x24,0x40,0xF8,0x20,0x00,0x00}},
140 {{"显"},{0x3F,0x80,0x20,0x80,0x3F,0x80,0x20,0x80,0x3F,0x80,0x00,0x00,
141           0x4A,0x40,0x2A,0x40,0x2A,0x80,0x0B,0x00,0xFF,0xE0,0x00,0x00}},
142 {{"示"},{0x00,0x80,0x7F,0xC0,0x00,0x00,0x00,0x00,0xFF,0xE0,0x04,0x00,
143           0x14,0x80,0x24,0x40,0x44,0x20,0x84,0x20,0x1C,0x00,0x00,0x00}},
144 {{"两"},{0xFF,0xE0,0x12,0x00,0x12,0x00,0x7F,0xC0,0x52,0x40,0x52,0x40,
145           0x5B,0x40,0x64,0xC0,0x48,0x40,0x40,0x41,0xC0,0x00,0x00}},
146 {{"路"},{0xF2,0x00,0x93,0xC0,0x94,0x40,0xFA,0x80,0xA1,0x00,0x22,0x80,
147           0xB7,0xE0,0xAA,0x40,0xA2,0x40,0xFA,0x40,0x83,0xC0,0x00,0x00}},
148 {{"结"},{0x21,0x00,0x21,0x00,0x4F,0xE0,0x91,0x00,0xE1,0x00,0x2F,0xE0,
149           0x50,0x00,0xE7,0xC0,0x04,0x40,0x34,0x40,0xC7,0xC0,0x00,0x00}},
150 {{"果"},{0x3F,0xC0,0x24,0x40,0x3F,0xC0,0x24,0x40,0x3F,0xC0,0x04,0x00,
151           0xFF,0xE0,0x0E,0x00,0x15,0x00,0x24,0x80,0xC4,0x60,0x00,0x00}},
152 {{"曲"},{0x09,0x00,0x09,0x00,0x7F,0xE0,0x49,0x20,0x49,0x20,0x49,0x20,
153           0x7F,0xE0,0x49,0x20,0x49,0x20,0x49,0x20,0x7F,0xE0,0x00,0x00}};
```



```
154  {"线"}, {0x22,0x80,0x22,0x40,0x52,0x60,0x97,0x80,0xE2,0x60,0x4F,0x80,
155      0xB2,0x40,0xC2,0x80,0x31,0x20,0xC6,0xA0,0x18,0x60,0x00,0x00}
156  };
```

……限于篇幅,本例省略了其他案例中出现过的类似代码

## 5.16 用 74LS595 与 74LS154 设计的 16×16 点阵屏

本例用串入并出芯片 74LS595 和 4-16 译码器 74LS154 控制 16×16 点阵屏汉字滚动显示。案例电路及部分运行效果如图 5-16 所示。

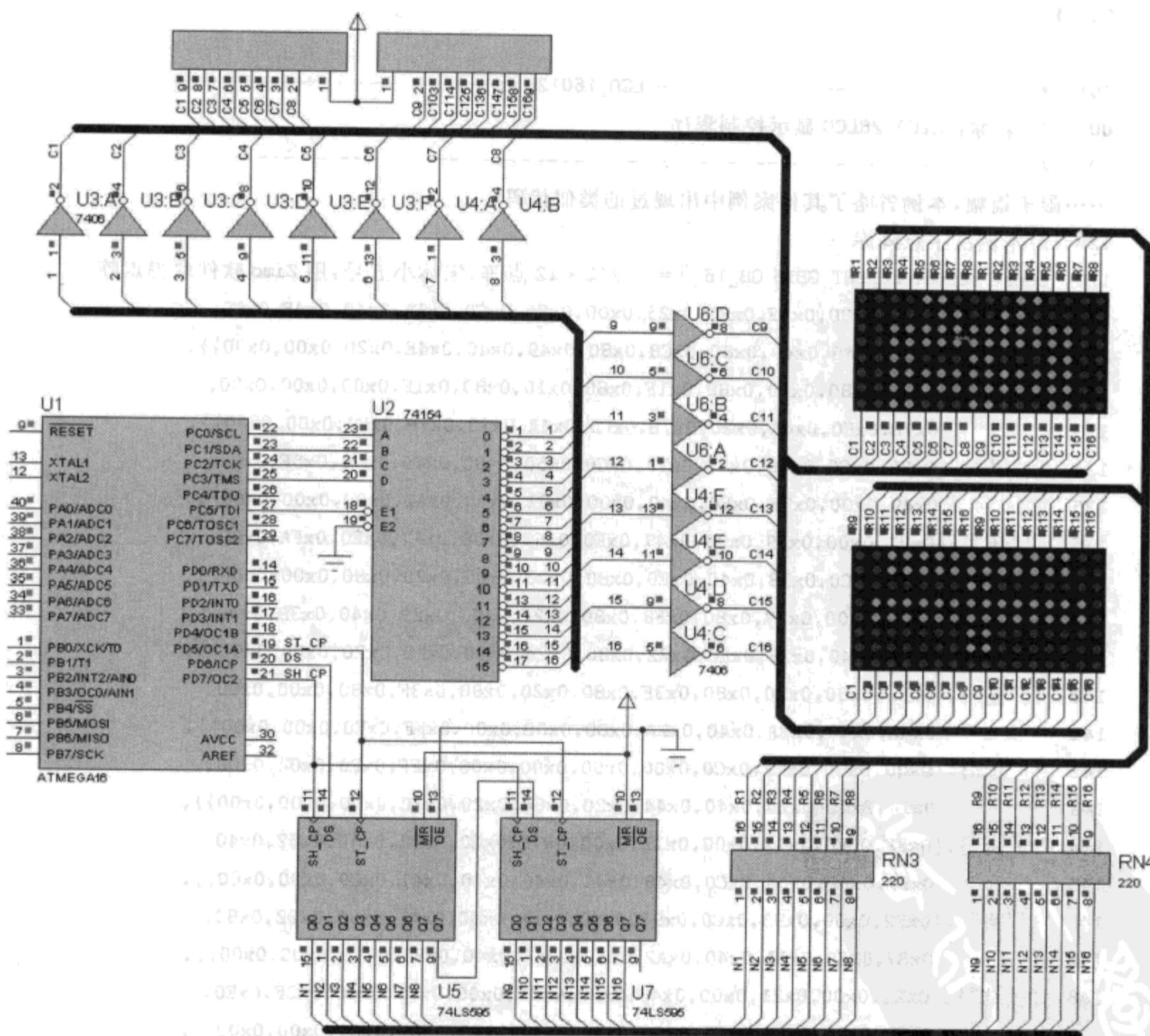


图 5-16 用 74LS595 与 74LS154 设计的 16×16 点阵屏

## 1. 程序设计与调试

本例点阵屏共有 16 行，两字节的行码由两片 595 的并行输出端提供，16 列由 4-16 译码器控制扫描选通。由于在本例电路布局下，点阵屏的列是共阳的，而译码器的译码输出位是低电平的，因此译码器的输出端选用了反向放大器 7406。

本例点阵屏的扫描显示由 T/C0 定时器溢出中断程序控制实现，显示过程如下：

① 中断函数内的静态变量 Scan\_Column 首先获取当前待扫描列号，其取值在 0~15 列范围内反复循环，Scan\_Column 初值设为 15(0x0F)，这样可使得首次调用时通过 +1 而从第 0 列开始扫描显示。实际上 Scan\_Column 的初值可设为 0~15 中的任何一列，因为下一步中的函数 Serial\_Input\_Pin 在取行码并发送时，所取的行码总是与 Scan\_Column 对应，因而不会出现显示偏移或错位现象。

② 程序两次调用函数 Serial\_Input\_Pin，向 74LS595 串行输入当前列的上下各 8 行点阵数据。

③ 在译码器选通当前列之前，函数 Parallel\_Output\_595() 将等待垂直显示的 16 位数据由 595 的并行输出端输出到点阵屏当前列的 16 行，完成 16 位行码发送。

④ PC 端口低 4 位向译码器输入列号 Scan\_Column，要注意的是每次都需要先清零低 4 位，然后再输入新的列号。

⑤ 开译码器译码，通过 7406 反向后选通当前的共阳列，该列点阵显示完毕。

⑥ 重设定时初值为 2 ms。

在定时器溢出中断的驱动下，上述过程将每隔 2 ms 被执行一遍，每次约 32 ms 可完成一个汉字全部 16 列的刷新显示。

由于中断函数内还使用了汉字索引变量 wIndex，而该变量的控制由主程序完成，在主程序内该变量每隔 300 ms 递增一次，由此即可形成每个汉字在显示屏上保持刷新显示 300 ms，然后继续循环显示下一汉字的滚动显示效果。

## 2. 实训要求

① 分析本例汉字的取模方式，用 Zimo 软件重新取得另一组汉字点阵，将新的点阵数据替换本例中的点阵数据，仍实现本例运行效果。

② 在本例基础上进一步设计出  $32 \times 16$  的 LED 点阵屏，实现多个汉字的滚动显示。

③ 在系统中整合串口通信功能，将用 Zimo 等软件获取的汉字点阵数据通过串口调试助手软件发送给单片机，单片机能根据接收到的点阵数据在 LED 显示屏上显示汉字信息。

④ 用自己熟悉的 Windows 平台可视化软件开发工具设计上位机程序，用户在上位机程序中输入任意汉字信息，在单击发送按钮后上位机程序能自动获取所输入汉字的点阵信息并发送给下位单片机显示。

## 3. 源程序代码

```

001 //-----
002 // 名称：用 74LS595 与 74LS154 设计的 16 * 16 点阵屏
003 //-----
004 // 说明：本例综合使用了串入并出芯片 74LS595,4-16 译码器 74LS154
005 //          在 16 * 16 点阵屏上实现多个汉字交替显示效果

```



```
006 //  
007 //-----  
008 #define F_CPU 4000000UL  
009 #include <avr/io.h>  
010 #include <avr/pgmspace.h>  
011 #include <avr/interrupt.h>  
012 #include <util/delay.h>  
013 #define INT8U unsigned char  
014 #define INT16U unsigned int  
015  
016 //74595 及 74154 相关引脚定义  
017 #define ST_CP PD5 //输出锁存器控制脉冲  
018 #define DS PD6 //串行数据输入  
019 #define SH_CP PD7 //移位时钟脉冲  
020 #define E1_74LS154 PC5 //74LS154 译码器使能端  
021  
022 //74595 及 74154 相关引脚操作  
023 #define ST_CP_1() PORTD |= _BV(ST_CP)  
024 #define ST_CP_0() PORTD &= ~_BV(ST_CP)  
025 #define DS_1() PORTD |= _BV(DS)  
026 #define DS_0() PORTD &= ~_BV(DS)  
027 #define SH_CP_1() PORTD |= _BV(SH_CP)  
028 #define SH_CP_0() PORTD &= ~_BV(SH_CP)  
029  
030 //74154 译码器使能与禁止  
031 #define EN_74LS154() PORTC &= ~_BV(E1_74LS154)  
032 #define DI_74LS154() PORTC |= _BV(E1_74LS154)  
033  
034 //存放于 Flash 空间的待显示文字点阵  
035 prog_uchar Word_Set_OF_16x16[][][32] =  
036 {  
037     /* -----单----- */  
038     { 0xFF, 0xFF, 0xFF, 0xE7, 0x03, 0xE4, 0x03, 0xE4,  
039         0x92, 0xE4, 0x90, 0xE4, 0x91, 0xE4, 0x03, 0x80,  
040         0x03, 0x80, 0x91, 0xE4, 0x90, 0xE4, 0x92, 0xE4,  
041         0x03, 0xE4, 0x03, 0xE4, 0xFF, 0xE7, 0xFF, 0xFF },  
042     /* -----片----- */  
043     { 0xFF, 0xFF, 0xFF, 0x9F, 0xFF, 0xC7, 0x01, 0xE0,  
044         0x01, 0xF8, 0xCF, 0xFC, 0xCF, 0xCF, 0xFC,  
045         0xCF, 0xFC, 0xC0, 0xFC, 0xC0, 0x80, 0xCF, 0x80,  
046         0xCF, 0xFF, 0xCF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF },  
047     /* -----机----- */  
048     { 0xE7, 0xF9, 0x67, 0xFC, 0x00, 0x80, 0x00, 0x80,
```

```

049     0x67,0xFE,0xE7,0xDC,0xFF,0x8F,0x01,0xC0,
050     0x01,0xF0,0xF9,0xFF,0xF9,0xFF,0x01,0xC0,
051     0x01,0x80,0xFF,0x9F,0xFF,0x8F,0xFF,0xFF },
052     /* ----- C ----- */
053     { 0xFF,0xFF,0xFF,0xFF,0xFF,0x1F,0xFC,
054       0xEF,0xFB,0xF7,0xF7,0xFB,0xEF,0xFB,0xEF,
055       0xFB,0xEF,0xFB,0xEF,0xF7,0xF7,0xE3,0xFB,
056       0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF },
057     /* ----- 语 ----- */
058     { 0x9F,0xFF,0x9D,0xFF,0x11,0x80,0x13,0xC0,
059       0xFF,0xE7,0x39,0xFF,0x29,0x81,0x09,0x81,
060       0x01,0xCD,0x21,0xCD,0x29,0xCD,0x09,0xCD,
061       0x09,0x81,0x39,0x81,0x3F,0xFF,0xFF,0xFF },
062     /* ----- 言 ----- */
063     { 0xF3,0xFF,0xF3,0xFF,0x53,0x81,0x53,0x81,
064       0x53,0xC9,0x53,0xC9,0x50,0xC9,0x50,0xC9,
065       0x53,0xC9,0x53,0xC9,0x53,0x81,0x53,0x81,
066       0xF3,0xFF,0xF3,0xFF,0xFF,0xFF,0xFF,0xFF },
067     /* ----- 程 ----- */
068     { 0x9B,0xF3,0x9B,0xF9,0x03,0x80,0x01,0x80,
069       0x99,0xFC,0x99,0xF9,0xFF,0x9F,0x41,0x92,
070       0x41,0x92,0x49,0x80,0x49,0x80,0x49,0x92,
071       0x41,0x92,0x41,0x92,0xFF,0x9F,0xFF,0xFF },
072     /* ----- 序 ----- */
073     { 0xFF,0x9F,0x01,0xC0,0x01,0xE0,0xF9,0xFF,
074       0xF9,0xFC,0xC9,0xFC,0xC9,0x9C,0x88,0x9C,
075       0x08,0x80,0x49,0xC0,0x09,0xFC,0x89,0xFC,
076       0xC9,0xF8,0xF9,0xF8,0xFF,0xFF,0xFF,0xFF },
077     /* ----- 设 ----- */
078     { 0x9F,0xFF,0x9F,0xFF,0x1C,0xC0,0x11,0xC0,
079       0xFB,0xE7,0x0F,0x97,0x00,0x9C,0x20,0xC8,
080       0x3C,0xC3,0x3C,0xE7,0x20,0xE1,0x00,0xC8,
081       0x0F,0x9E,0xCF,0x9F,0xCF,0xDF,0xFF,0xFF },
082     /* ----- 计 ----- */
083     { 0x9F,0xFF,0x9F,0xFF,0x19,0x80,0x13,0x80,
084       0xF7,0xCF,0x9F,0xE7,0x9F,0xFF,0x9F,0xFF,
085       0x9F,0xFF,0x01,0x80,0x01,0x80,0x9F,0xFF,
086       0x9F,0xFF,0x9F,0xFF,0x9F,0xFF,0xFF,0xFF }
087   };
088
089 //待显示汉字索引,注意添加 volatile
090 volatile INT8U wIndex = 0;
091 //-----

```

```

092 // 595 串行输入子程序
093 //-
094 void Serial_Input_Pin(INT8U dat)
095 {
096     INT8U i;
097     for(i = 0x80; i != 0x00; i >>= 1)          //从高位开始向 595 串行输入 8 位
098     {
099         if (dat & i) DS_1(); else DS_0();
100         SH_CP_0(); _delay_us(2);
101         SH_CP_1(); _delay_us(2);                //移位时钟脉冲上升沿移位
102     }
103 }
104
105 //-
106 // 595 并行输出子程序
107 //-
108 void Parallel_Output_595()
109 {
110     ST_CP_0(); _delay_us(2);
111     ST_CP_1(); _delay_us(2);                  //上升沿将数据送到输出锁存器
112 }
113
114 //-
115 // T/C0 溢出中断,在主程序中的延时期间以 2 ms 的间隔动态显示每列数据
116 // 所显示的每列数据由两片 595 并行输出
117 //-
118 ISR (TIMER0_OVF_vect)
119 {
120     //当前扫描列
121     static INT8U Scan_Column = 0x0F;
122     //当前扫描列号加 1,屏蔽高 4 位,Scan_Column = 0 ~ 15 (0x00~0x0F)
123     Scan_Column = (Scan_Column + 1) & 0x0F;
124     //从 Flash 内存读取汉字点阵串行发送给两片 595 芯片
125     Serial_Input_Pin(pgm_read_byte(&Word_Set_OF_16x16[wIndex][Scan_Column * 2 + 1]));
126     Serial_Input_Pin(pgm_read_byte(&Word_Set_OF_16x16[wIndex][Scan_Column * 2]));
127
128     DI_74LS154(); _delay_us(1);                //禁止译码
129     Parallel_Output_595();                     //2 片 595 并行输出 16 位行码
130     PORTC = (PORTC & 0xF0) | Scan_Column;      //PC 低 4 位清 0,放入新的列号(0~15),等待译码
131     EN_74LS154(); _delay_us(1);                //使能译码
132     TCNT0 = 256 - F_CPU / 64.0 * 0.002;       //重置延时初值(2 ms)
133 }
134

```

```

135 //-----
136 // 主程序
137 //-----
138 int main()
139 {
140     INT8U Total_Word = sizeof(Word_Set_OF_16x16) / 32; //全角字符个数
141     DDRC = 0xFF; DDRD = 0xFF; //配置端口
142     TCCR0 = 0x03; //预设分频:64
143     TCNT0 = 256 - F_CPU / 64.0 * 0.002; //晶振 4 MHz,2 ms 定时初值
144     TIMSK = 0x01; //允许 T0 定时器溢出中断
145     sei(); //使能全局中断
146     while(1)
147     {
148         //下一个待显示的文字索引
149         for (wIndex = 0; wIndex < Total_Word; wIndex++)
150         {
151             _delay_ms(300); //每个汉字保持显示 300ms
152         }
153     }
154 }

```

## 5.17 用 8255 与 74LS154 设计的 16×16 点阵屏

本例与上一案例实现的功能相同,唯一的区别在于本例中的点阵屏行码改由接口扩展芯片 8255 提供,而上一案例中使用是两片串入并出芯片 74LS595。本例电路及部分运行效果如图 5-17 所示。

### 1. 程序设计与调试

前面已提到,本例与上一案例实现的功能相同,LED 点阵屏的刷新显示过程也相同,两者的程序非常相似,唯一的差别是本例 16 位的行码改用 8255 的 PA 与 PB 端口提供。对于 T/C0 定时器溢出中断控制的显示屏刷新显示程序,上一案例通过两次调用 Serial\_Input\_Pin 串行写入 16 位,再调用 Parallel\_Output\_595() 并行输出 16 位来提供行码,而本例则用 8255 的 PA 与 PB 端口输出语句替换了上一案例中的串入并出函数:

```

* PA = ~pgm_read_byte(Word_Set_OF_16x16 + wIndex * 32 + Col_Index);
* PB = ~pgm_read_byte(Word_Set_OF_16x16 + wIndex * 32 + Col_Index + 16);

```

第 4 章已经讨论过 8255 的程序设计方法,大家可参考此前 8255 的有关案例,进一步熟练掌握 8255 的端口定义,端口配置及端口读/写程序设计方法。

### 2. 实训要求

- ① 在本例基础上进一步设计出更大点阵的 LED 显示屏,实现每屏多个汉字的滚动显示。

② 编写程序使点阵屏具备多种可选的汉字显示特效,例如水平逐列滚动显示、垂直逐行滚动显示、百页窗式显示等。

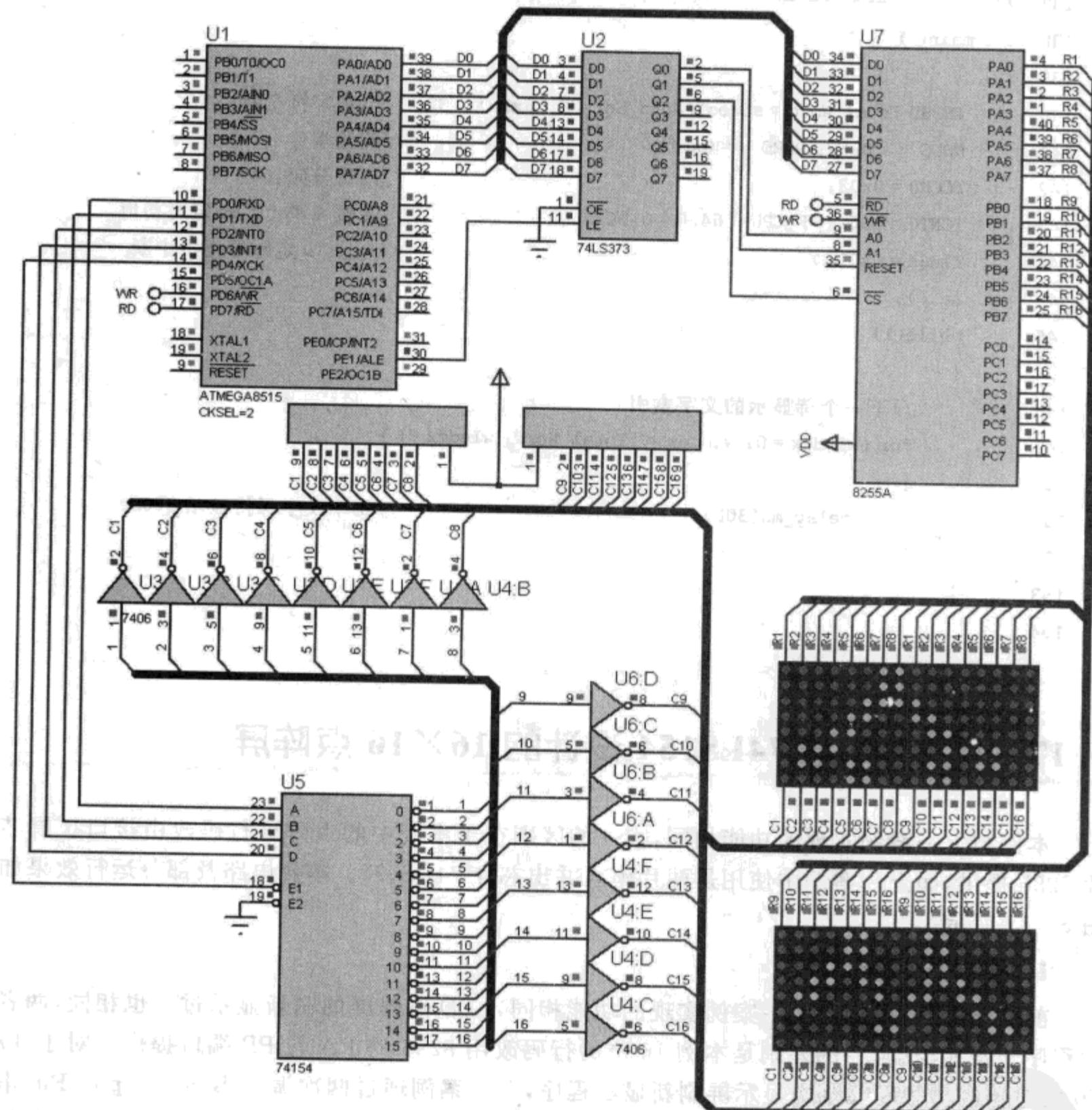


图 5-17 用 8255 与 74LS154 设计的 16×16 点阵屏

### 3. 源程序代码

```

001 //-
002 // 名称: 用 8255 与 74LS154 设计的 16 * 16 点阵屏
003 //-
004 // 说明: 本例用 8255 扩展接口,发送 4 片 8 * 8 点阵屏的行编码,列码由 4-16
005 //      译码器控制,实现了 16 * 16 点阵文字的显示
006 //-

```

```

007 //-----
008 #define F_CPU 2000000UL
009 #include <avr/io.h>
010 #include <avr/pgmspace.h>
011 #include <avr/interrupt.h>
012 #include <util/delay.h>
013 #define INT8U unsigned char
014 #define INT16U unsigned int
015
016 //PA、PB、PC 端口及命令端口地址定义
017 #define PA (INT8U *)0xFF00
018 #define PB (INT8U *)0xFF01
019 #define PC (INT8U *)0xFF02
020 #define COM (INT8U *)0xFF03
021
022 //74LS154 译码器开关
023 #define EN_74LS154() PORTD &= ~_BV(PD4)
024 #define DI_74LS154() PORTD |= _BV(PD4)
025 //存放在 Flash 内存中的汉字点阵数据
026 prog_uchar Word_Set_OF_16x16[] =
027 {
028 /* -- 上 -- */
029 0x00,0x00,0x00,0x00,0x00,0x00,0xFF,0xFF,0x60,0x60,0x60,0x60,0x60,0x60,0x00,0x00,
030 0x30,0x30,0x30,0x30,0x30,0x30,0x3F,0x3F,0x30,0x30,0x30,0x30,0x30,0x30,0x30,0x00,
031 /* -- 海 -- */
032 0x30,0x63,0x66,0x04,0x30,0xFC,0xFF,0x37,0x76,0xF6,0x36,0xF6,0x06,0x00,0x00,
033 0x30,0x7E,0x0E,0x00,0x03,0x1F,0x1F,0x1B,0x1F,0x5F,0x7B,0x7F,0x3F,0x1B,0x03,0x00,
034 /* -- 大 -- */
035 0x00,0x30,0x30,0x30,0x30,0x30,0xFF,0xFF,0x30,0x30,0x30,0x30,0x30,0x30,0x00,0x00,
036 0x00,0x40,0x60,0x30,0x18,0x0E,0x07,0x03,0x06,0x0C,0x18,0x30,0x60,0x40,0x00,0x00,
037 /* -- 众 -- */
038 0x00,0x60,0x60,0x30,0xD8,0xCC,0x07,0x07,0x0C,0xD8,0xF0,0x20,0x60,0x60,0x00,0x00,
039 0x00,0x30,0x18,0x0C,0x07,0x0F,0x58,0x60,0x30,0x1F,0x0F,0x18,0x30,0x60,0x40,0x00,
040 /* -- 汽 -- */
041 0x00,0x22,0x66,0xCC,0x20,0xB8,0x9F,0xAF,0xAC,0xAC,0xAC,0xAC,0x0C,0x00,0x00,
042 0x20,0x70,0x3C,0x06,0x00,0x01,0x01,0x01,0x01,0x01,0x3F,0x7F,0x60,0x30,0x00,
043 /* -- 车 -- */
044 0x00,0x0C,0x8C,0xEC,0xFC,0xBC,0x8F,0xEF,0xEC,0x8C,0x8C,0x8C,0x0C,0x00,0x00,
045 0x00,0x18,0x19,0x19,0x19,0x19,0x19,0x7F,0x7F,0x19,0x19,0x19,0x19,0x18,0x00,0x00,
046 /* -- 有 -- */
047 0x00,0x0C,0x8C,0xCC,0xEC,0xFC,0x7F,0x6F,0x6C,0x6C,0xEC,0x0C,0x0C,0x00,
048 0x02,0x03,0x01,0x00,0x7F,0x7F,0x09,0x09,0x09,0x09,0x69,0x7F,0x3F,0x00,0x00,0x00,
049 /* -- 限 -- */

```



```
050 0x00,0xFE,0xFE,0x66,0xFE,0x9E,0x00,0xFE,0x0E,0xD6,0xD6,0xD6,0xFE,0x00,0x00,
051 0x00,0x7F,0x7F,0x0C,0x0F,0x07,0x00,0x7F,0x7F,0x30,0x0F,0x1C,0x36,0x62,0x20,0x00,
052 /* -- 公 -- */
053 0x00,0x80,0xC0,0x70,0x1E,0x0E,0xC0,0xC0,0x00,0x0E,0x1E,0x30,0x60,0xC0,0x80,0x00,
054 0x00,0x01,0x30,0x38,0x3C,0x37,0x33,0x30,0x10,0x16,0x1E,0x38,0x30,0x00,0x00,0x00,
055 /* -- 司 -- */
056 0x00,0x30,0xB6,0xB6,0xB6,0xB6,0xB6,0xB6,0xB6,0x36,0x06,0xFE,0xFE,0x00,0x00,
057 0x00,0x00,0x1F,0x1F,0x19,0x19,0x19,0x19,0x1F,0x1F,0x60,0x60,0x7F,0x3F,0x00,0x00
058 };
059
060 //当前待显示的汉字索引
061 //T0 中断程序要使用主程序中不断变化的 wIndex,因此前面必须添加 volatile
062 volatile INT8U wIndex = 0xFF;
063 //待显示汉字总个数
064 INT8U Total_Words = sizeof(Word_Set_OF_16x16) / 32;
065 //-----
066 // 定时器 0 中断,以 2 ms 的间隔动态显示每列数据
067 // 所显示的每列数据由 8255 并行输出
068 //-----
069 ISR (TIMERO_OVF_vect)
070 {
071     static INT8U Col_Index = 0xFF;
072     TCNT0 = 256 - F_CPU / 64.0 * 0.002;           //重置延时初值
073     DI_74LS154();                                //关闭译码器
074     Col_Index = (Col_Index + 1) & 0x0F;            //列号递增
075
076     //从 Flash 内存中读取并通过 8255 的 PA 与 PB 端口发送当前列上下各 8 行汉字点阵
077     *PA = ~pgm_read_byte(Word_Set_OF_16x16 + wIndex * 32 + Col_Index);
078     *PB = ~pgm_read_byte(Word_Set_OF_16x16 + wIndex * 32 + Col_Index + 16);
079
080     PORTD = (PORTD & 0xF0) | Col_Index;          //向译码器输入列码 0~15
081     EN_74LS154();                                //使能译码,选通 Col_Index 列
082 }
083
084 //-----
085 // 主程序
086 //-----
087 int main()
088 {
089     DDRA = 0xFF; DDRD = 0xFF;                    //配置端口
090     MCUCR |= 0x80;                             //允许访问外部存储器/接口等
091     TCCR0 = 0x03;                            //预设分频:64
092     TCNT0 = 256 - F_CPU / 64.0 * 0.002;        //晶振 4 MHz,2 ms 定时初值
```

```

093 //8255 工作方式选择:
094     * COM = 0x80;
095     TIMSK |= _BV(TOIE0);
096     sei();
097     while(1) //循环显示汉字
098     {
099         //待显示汉字索引在 0~Total_Words - 1 之间循环
100         for (wIndex = 0; wIndex < Total_Words; wIndex++)
101         {
102             _delay_ms(500); //每个汉字保持显示 500 ms
103         }
104     }
105 }

```

## 5.18 8×8 LED 点阵屏仿电梯数字滚动显示

本例运行时,按下对应楼层按键,点阵屏数字将从当前位置向上或向下平滑滚动显示到指定楼层位置,在到达终点后蜂鸣器输出提示音。案例电路及部分运行效果如图 5-18 所示。

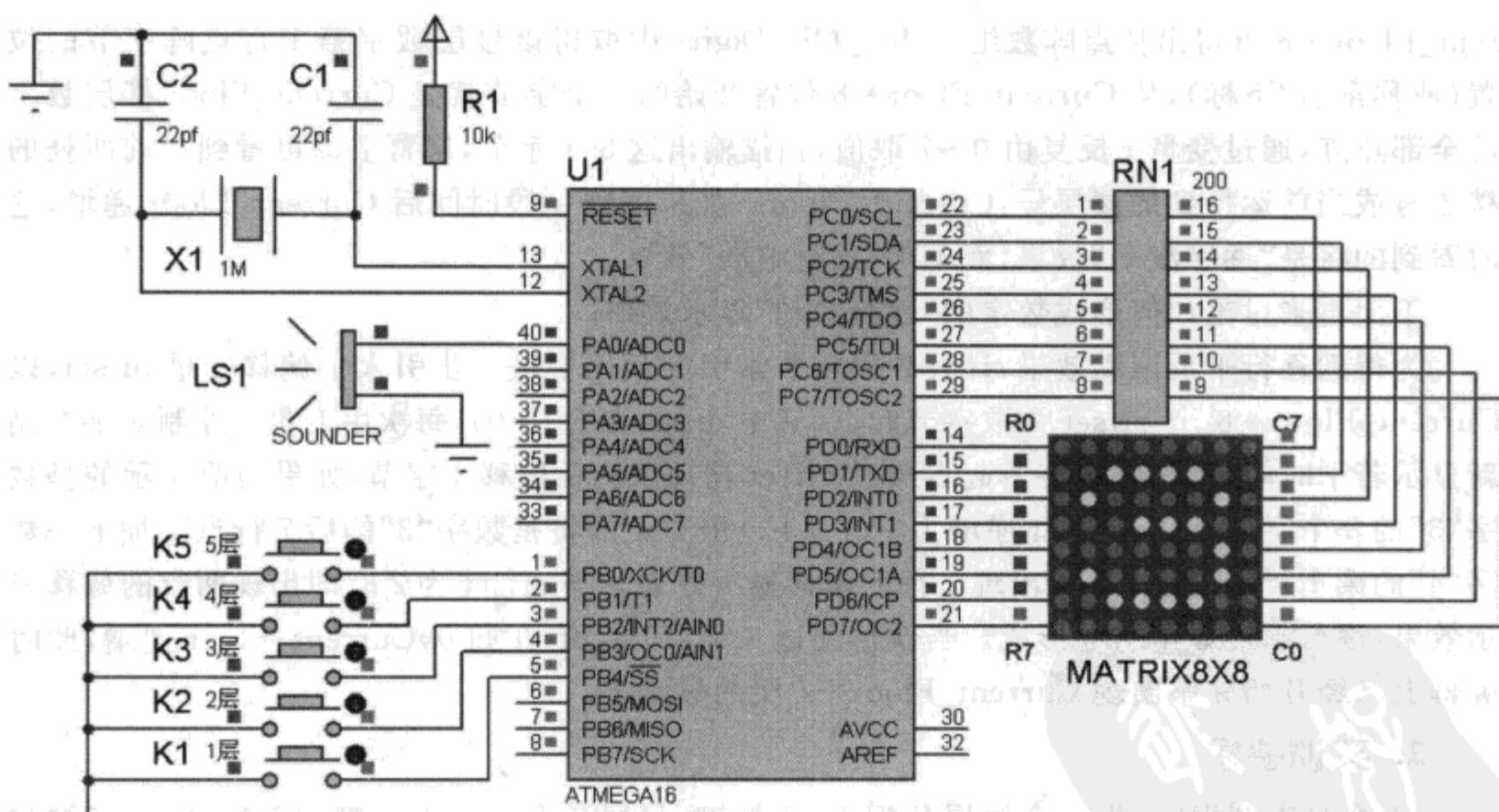


图 5-18 8×8 LED 点阵屏仿电梯数字滚动显示

### 1. 程序设计与调试

本例设计与调试要点在于 LED 点阵屏的滚动显示控制部分。电路中的 8×8 点阵屏左边 8 根引脚连接行码,右边 8 根引脚连接列码。在 LED 屏上显示数字时,R0~R7 负责行扫描,



而 C0~C7 则是当前选通行中的 8 个点阵数据位,也称列码字节。在本例电路中,显示屏同一行上的各位 LED 是共阳的,C0~C7 对应输出的列码字节中,0 对应的 LED 将被点亮,而 1 对应的 LED 则熄灭。

为本例电路中的 LED 点阵屏取字模时,须选择“横向取模,字节不倒序”,以 3 的点阵数据为例:0x00,0x3C,0x42,0x1C,0x02,0x42,0x3C,0x00//3。这 8 个字节与 LED 显示的 8 行对应,前 3 个与后 3 个是对称的,观察图 5-18 也可以看出,“3”的上 3 行与下 3 行点阵是完全相同的,由此也可以明显看出其取模方式为横向取模,至于取模时的字节是否倒序,这要看电路中 PC0~PC7 是与 C0~C7 还是 C7~C0 对连。

本例 LED 屏扫描显示由 T/C0 定时器溢出中断控制实现,对于下面的 3 行代码:

```
PORTD = _BV(r);           //发送行扫描码(共阳行的扫描码)  
i = Current_Floor * 8 + offset + r; //计算列码字节索引  
PORTC = ~Table_OF_Digits[i]; //发送列码(用~转换共阴共阳编码)
```

第 1 行代码选通 LED 屏的第 r 行(r 取值范围为 0~7,电路中的 R0~R7,由\_BV(r)也可以看出该行是共阳的),第 2 行取得待输出的列码字节索引 i,第 3 行输出列码字节,也就是第 r 行中的 8 个点阵数据。

楼层数字滚动显示的设计要点在于上面 3 行代码中的第 2 行。

该行语句中的变量 Current\_Floor 为当前所处的楼层号或已经运行到的楼层号,由 Current\_Floor \* 8 可得出从点阵数组 Table\_OF\_Digits 中取得该楼层数字第 1 行点阵字节的位置(或称索引/下标),从 Current\_Floor \* 8 位置开始的 8 个字节就是 Current\_Floor 楼层数字的全部点阵,通过变量 r 反复由 0~7 取值,扫描输出这 8 个字节,屏幕上即可看到当前所处的楼层号或当前运行到的楼层号:Current\_Floor。如果刷新一段时间后 Current\_Floor 递增,这时看到的将是“逐字滚动”效果,而不是“逐行滚动”效果。

下面再来讨论如何形成数字的“逐行滚动”显示效果。

为得到逐行平滑滚动效果,计算列码字节索引的代码中进一步引入了偏移变量 offset,以 Current\_Floor \* 8 + offset 为取字节起点(其中 offset 初值为 0),每次由变量 r 控制 8 字节刷新显示若干时间后,再将取字节起点通过 offset 向前或向后偏移 1 字节,如果当前显示的是数字“3”的 8 个字节,则当 offset 递增为 1 时,LED 屏显示的将是数字“3”的后 7 行点阵加下一数字“4”的第 1 行点阵,LED 屏出现 1 行的偏移显示效果。当 offset 为 2 时即出现两行的偏移显示效果,逐行滚动显示由此形成。当偏移到达 8 字节时,offset 归 0,Current\_Floor 递增,此时屏幕上已经开始完整出现 Current\_Floor+1 层的数字了。

## 2. 实训要求

① 修改本例程序,为 5 个楼层分配 10 个按键,每层各有一个上行键与下行键,上行键与下行键旁有对应的 LED 指示灯,再用 5 个按键仿真电梯轿箱内的楼层数字按键,轿箱内各数字按键旁边有 LED 指示灯。编程仿真电梯在 5 个楼层中的运行过程,电梯上行时程序控制电机正转,否则反转;电梯到达各设定的楼层时停顿 2 s,然后继续运行。系统完成各楼层人员的上下行要求及电梯轿箱内所指定到达的楼层要求后停止运行,此时电梯内外各 LED 指示灯将处于熄灭状态,LED 点阵显示屏保持显示当前停留的楼层。

② 用接口扩展芯片设计更高楼层的电梯运行仿真系统,系统可显示两位楼层号,能仿真

开门与关门电机、上行与下行电机,能仿真楼层到达提示音等,电梯内的楼层显示屏可改用液晶显示屏。

### 3. 源程序代码

```

001 //-
002 // 名称:8 * 8 LED 点阵屏仿电梯数字滚动显示
003 //-
004 // 说明:本例模拟了电梯显示屏上下滚动显示楼层的效果,当目标楼层大于
005 // 当前楼层时将向上滚动显示,反之则向下滚动显示,到达目标楼层时
006 // 将发出蜂鸣声
007 //
008 //-
009 #define F_CPU 4000000UL
010 #include <avr/io.h>
011 #include <avr/interrupt.h>
012 #include <util/delay.h>
013 #define INT8U unsigned char
014 #define INT16U unsigned int
015
016 #define BEEP() PORTA ^= _BV(PA0) //蜂鸣器定义
017 const INT8U Table_OF_Digits[] = //0~9 的数字点阵
018 {
019     0x00,0x3C,0x66,0x42,0x42,0x66,0x3C,0x00,//0
020     0x00,0x08,0x38,0x08,0x08,0x08,0x3E,0x00,//1
021     0x00,0x3C,0x42,0x04,0x08,0x32,0x7E,0x00,//2
022     0x00,0x3C,0x42,0x1C,0x02,0x42,0x3C,0x00,//3
023     0x00,0x0C,0x14,0x24,0x44,0x3C,0x0C,0x00,//4
024     0x00,0x7E,0x40,0x7C,0x02,0x42,0x3C,0x00,//5
025     0x00,0x3C,0x40,0x7C,0x42,0x42,0x3C,0x00,//6
026     0x00,0x7E,0x44,0x08,0x10,0x10,0x10,0x00,//7
027     0x00,0x3C,0x42,0x24,0x5C,0x42,0x3C,0x00,//8
028     0x00,0x38,0x46,0x42,0x3E,0x06,0x3C,0x00 //9
029 };
030
031 INT8U Current_Floor = 1, Dest_Floor = 1; //当前楼层,目标楼层
032 //-
033 // 主程序
034 //-
035 int main()
036 {
037     DDRA = 0xFF; //配置端口
038     DDRB = 0x00; PORTB = 0xFF;
039     DDRC = 0xFF;

```



```
040     DDRD = 0xFF;
041     TCCR0 = 0x03;                      //T0 预设分频:64
042     TCCR1B = 0x01;                     //T1 预设分频:1
043     TCNT0 = 256 - F_CPU / 64.0 * 0.004; //T0,4 ms 定时初值
044     TCNT1 = 65536 - F_CPU / 1 * 0.0005; //T1,500 μs 定时初值
045     TIMSK = _BV(TOIE0);                //允许 T0 定时器溢出中断
046     sei();                            //开中断
047     while (1);
048 }
049
050 //-----
051 // T1 定时器控制声音输出
052 //-----
053 ISR (TIMER1_OVF_vect)
054 {
055     static INT8U tCount = 0;           //计时累加变量
056     TCNT1 = 65536 - F_CPU / 1 * 0.0005; //重设初值 500 μs
057     BEEP();                          //蜂鸣器输出(频率为 1 kHz)
058     if (++tCount == 150)             //150 * 500 μs 后停止蜂鸣器输出
059     {
060         TIMSK &= ~_BV(TOIE1);
061         tCount = 0;
062     }
063 }
064
065 //-----
066 // T0 定时器控制楼层数字滚动及刷新显示
067 //-----
068 ISR (TIMERO_OVF_vect)
069 {
070     //楼层到达提示音输出控制
071     static INT8U NoSound = 0;
072     //每屏点阵的刷新次数控制变量,用于避免一屏点阵的过快滚动
073     static INT8U x = 0;
074     //每屏数字有 8 行字节,r 为当前行号
075     static INT8U r = 0;
076     //取字节偏移量,因为可能出现负数,注意定义为
077     //signed char/char 或 int 类型
078     //不要定义为 unsigned char(INT8U)或 unsigned int(INT16U)
079     static signed char offset = 0;
080
081     INT8U i;
```

```

082     TCNT0 = 256 - F_CPU / 64.0 * 0.004;
083     //在停止滚动时,如果有键按下则判断目标楼层
084     if (PINB != 0xFF && Current_Floor == Dest_Floor)
085     {
086         if (PINB == 0xFE) Dest_Floor = 5; else
087             if (PINB == 0xFD) Dest_Floor = 4; else
088                 if (PINB == 0xFB) Dest_Floor = 3; else
089                     if (PINB == 0xF7) Dest_Floor = 2; else
090                         if (PINB == 0xEF) Dest_Floor = 1;
091                         NoSound = 1;
092     }
093
094     //以下程序由 PORTD 发送行扫描码,PORTC 发送列码字节(对应于一行中的 8 列,或 8 位)
095     //PORTD 每次选通一行,PORTC 随即发送这一行上的 8 位
096     PORTD = _BV(r);                                //发送行扫描码
097     i = Current_Floor * 8 + offset + r;            //计算列码字节索引
098     PORTC = ~Table_OF_Digits[i];                   //发送列码(用~转换共阴共阳编码)
099
100    //上升显示-----
101    if (Current_Floor < Dest_Floor)
102    {
103        if( ++r == 8)      //每屏需要完成 8 行刷新,输出 8 个字节(对应于 8 行 LED)
104        {
105            r = 0;
106            if ( ++x == 4)    //每屏被整体刷新 4 次后 offset 后偏,以便形成上滚效果
107            {
108                x = 0;          //偏移为 8 时归 0,Current_Floor 进入下一层
109                if ( ++offset == 8) { offset = 0; Current_Floor++ ; }
110            }
111        }
112    }
113    //下降显示-----
114    else if (Current_Floor > Dest_Floor)
115    {
116        if( ++r == 8)  //每屏需要完成 8 行刷新,输出 8 个字节(对应于 8 行 LED)
117        {
118            r = 0;
119            if ( ++x == 4)//每屏被整体刷新 4 次后 offset 前偏,以便形成下滚效果
120            {
121                x = 0;      //偏移为 -8 时归 0,Current_Floor 进入上一层
122                if ( --offset == -8) { offset = 0; Current_Floor-- ; }
123            }
124        }

```

```

125      }
126      //停止滚动,保持稳定的刷新显示,并且输出声音-----
127      else
128      {
129          if( ++r == 8) r = 0;
130          if (NoSound) { NoSound = 0; TIMSK |= _BV(TOIE1); }
131      }
132  }

```

## 5.19 用内置 EEPROM 与 1602 液晶设计的带 MD5 加密的电子密码锁

本例用 AVR 单片机 EEPROM 保存密码,输入正确密码时开锁灯亮,液晶屏显示开锁成功;开锁成功后用户可按下重设密码键设置新密码,在输入 10 位以内的新密码后按下存入键可将新密码用 MD5 算法加密并写入 EEPROM,下次开锁时用新密码才能打开。案例电路及部分运行效果如图 5-19 所示。

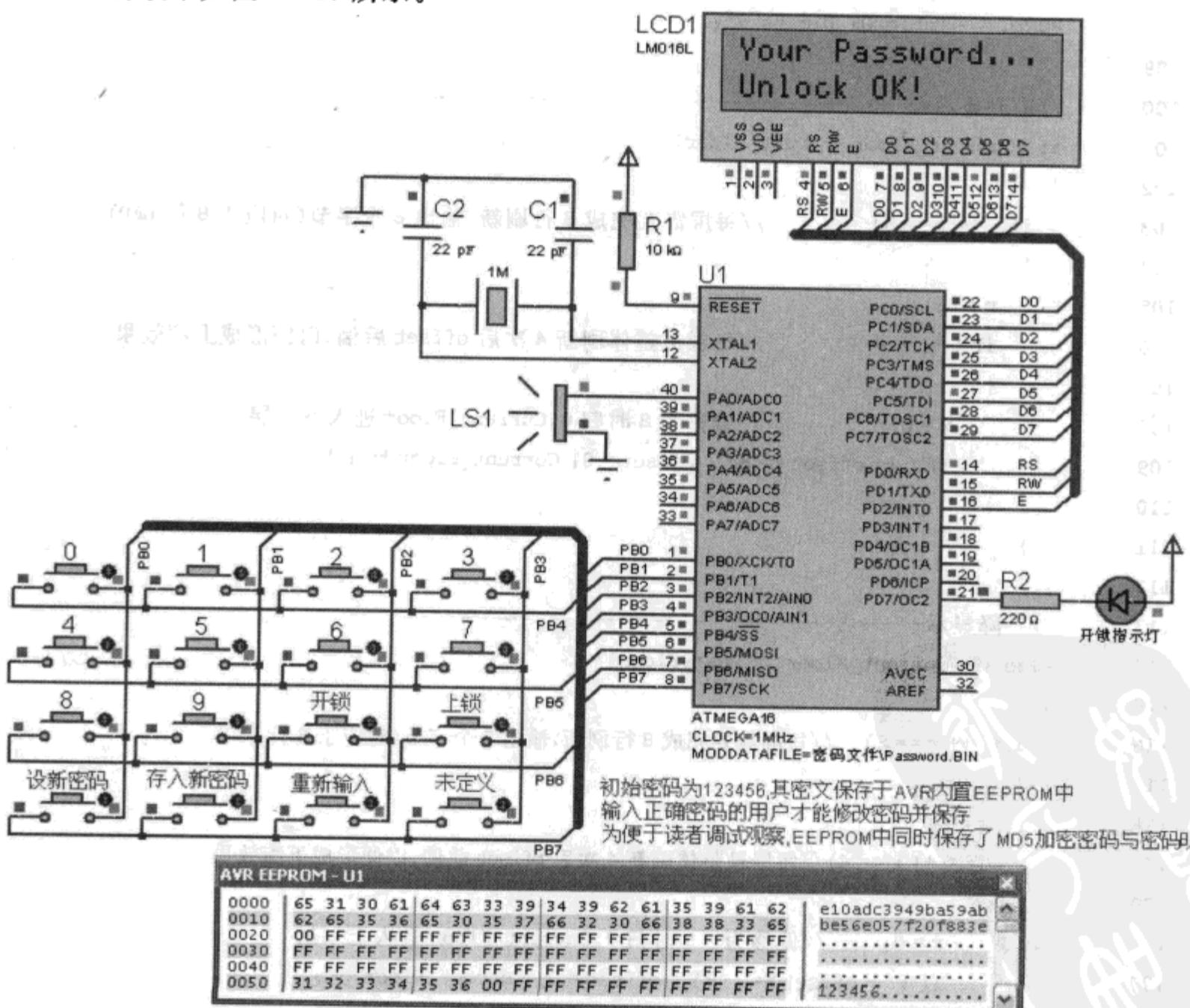


图 5-19 用内置 EEPROM 与 1602 液晶设计的带 MD5 加密的电子密码锁

## 1. 程序设计与调试

MD5 算法即信息摘要算法(Message Digest Algorithm 5),在 20 世纪 90 年代初由 MIT 的计算机科学实验室和 RSA Data Security Inc 发明,经历的版本有 MD2/MD3/MD4。MD5 广泛用于加密技术,很多系统用户密码都以 MD5 加密方式保存,用户登录时,系统将用户输入的密码转换成 MD5 值,然后再与系统中保存的 MD5 值比较,以此来验证用户的合法性,这样比保存密码明文要安全得多。密码明文容易被窃取和使用,而经 MD5 加密后的密码,由于其不可逆运算的特征,即使 MD5 加密后的密码被窃取,系统安全也不会受到威胁。

下面简单介绍 MD5 加密算法:

MD5 以 512 位(bit)分组来处理输入的信息,每一分组又被划分为若干子分组,经过了一系列的处理后,算法的输出由 4 个 32 位分组组成,将这 4 个 32 位分组组合后将生成一个 128 位散列值,这 128 位二进制数相当于 16 个字节,由这 16 个字节转换的“字节串”就是最后输出的 32 个字符(每字节转换为 2 个字符)。

本例 MD5 算法中,上下文结构变量 context 的 buffer 成员是 512 位的信息处理缓冲, state 成员保存 MD5 算法的 4 个 32 位初始幻数及最终的组合输出数位, count 成员保存信息位长。

在初始化上下文结构变量 context 以后,算法首先调用 MD5Update 函数对原始信息串进行变换。核心变换函数为 MD5Update,对于长串会进行尽可能多次的 MD5 四轮变换,每次进行的四轮变换由函数 MD5Transform 完成,变换后的结果存入于 context 的状态成员 state。

MD5 算法接着再进行信息串填充,使其位长度对 512 求余的结果等于 448(即 512-64),信息的位长度被扩展至  $n \times 512 + 448$  位(bit),即  $n \times 64 + 56$  字节( $n$  为一个正整数)。

MD5 算法填充信息串时,在原始信息的后面填充一个 1 和若干个 0,直到满足上面的条件为止。MD5.c 中使用的数组 PADDING[64]={0x80,0,0,...} 用于填充处理,该数组所有字节展开为二进制数时就是 10000000……,它以 1 个 1 开头,后面是 511 个 0,共 512 位。实际填充时会使用其第 0 个字节开始的若干个连续字节。

在进行填充处理后再调用核心函数 MD5Update 继续进行变换,完成填充及变换后再使 64 位二进制表示的填充前信息长度参与变换,此时是第 3 次调用 MD5Update 函数,本例中附加的填充前信息长度由 8 字节的 bits 数组给出。

MD5 加密算法中,4 个加密幻数为:

$a=0x01234567, b=0x89abcdef, c=0xfedcba98, d=0x76543210$

它们按 Little Endian 方式初始存放于 context->state 中。

MD5 加密的四轮变换函数 MD5Transform 使用了以下 4 个非线性函数(每轮 1 个):

$$F(x, y, z) = (x \& y) | ((\sim x) \& z)$$

$$G(x, y, z) = (x \& z) | (y \& (\sim z))$$

$$H(x, y, z) = x ^ y ^ z$$

$$I(x, y, z) = y ^ (x | (\sim z))$$

在经过 MD5Final 的最后变换以后,context->state 中所保存的 16 字节数据就是待输出的加密数据,这 16 个字节被复制到摘要字节数组 digest,程序最后将这 16 个字节转换为十六进制字符,每字节转换为 2 个字符,得到最后的加密输出字符串。

为利于进一步理解本例的 MD5 加密算法,下面给出对原始密码“123456”进行 MD5 加密

的跟踪过程：

① 调用 MD5Init 函数初始化 context, 初始化以后的 context 各成员初值如下：

位长成员 count[0]~count[1]:0x00000000 00000000;

状态成员 state[0]~state[3]:0x10325476 98BADC9E EFCDAB89 67452301;

缓冲成员 buffer:64 个 0x00。

② 首次调用 MD5 核心计算函数 MD5Update: MD5Update(&context, (INT8U \*)str, len)。

本次调用 MD5Update 对原始信息串进行变换处理, 处理后的 context 内容如下：

count[0]~count[1]:0x00000030 00000000, 即 count=48(这里的 48 表示原始串长为 6 字节, 共 48 位);

state[0]~state[3]:0x10325476 98BADC9E EFCDAB89 6745230;

缓冲成员 buffer 前 6 个字节为 0x31~36, buffer 中被填入“123456”的 ASCII 码, 其余全部为 0x00。

③ MD5Final 函数 2 次调用 MD5Update 函数：

第 1 次调用 MD5Update 函数: MD5Update(context, (INT8U \*)PADDING, padLen)

对信息位进行填充处理, 填充时使用了字节数组 PADDING[64]。

count[0]~count[1]:0x000001C0 00000000, 即 count=448, 由于原始串长 6 字节, 故需要填充 50 字节达到 56 字节, 即 448 位。

state[0]~state[3]:0x10325476 98BADC9E EFCDAB89 6745230;

缓冲成员 buffer:31 32 33 34 35 36 80 00 ..... 00, 其中 80 开始的 50 个字节为填充字节, 以 1 个二进制 1 开头, 后面为 447 个 0, buffer 的最后 8 个字节仍为 0x00。

再次调用 MD5Update 函数: MD5Update(context, bits, 8), 附加用 64 位(8 字节)表示的填充前串长进行变换:

count[0]~count[1]:0x00000200 00000000, 即 count=512; (512=448 + 64)

state[0]~state[3]:0x39DC0AE1 0xAB59BA49 0x57E056BE 0x3E880FF2;

buffer 的内容不变。

最后输出的 MD5 加密密码为: e10adc3949ba59abbe56e057f20f883e

该字符串是将 state[0]~state[3] 中 4 个十六进制长整数逆转并转换为小写以后的结果。

为将初始密码“123456”进行 MD5 加密以后的密码存入单片机的 EEPROM 存储器, 需要先将“e10adc3949ba59abbe56e057f20f883e”保存到 Password. bin 文件。要将这一密文字符串写入 Password. bin, 可使用 Turbo C 编写程序或直接使用 UltraEdit 完成。

创建 Password. bin 文件以后, 打开单片机属性窗口找到“Advanced Properties”下拉框, 选择“Initial contents of EEPROM”项, 在其后面的文本框中选择 Password. bin 文件即可完成对 EEPROM 的初始数据绑定。

本例其他功能设计留给大家自行分析, 这里不再进行更多分析与说明。

## 2. 实训要求

① 跟踪本例中 MD5 加密函数, 观察串长超过 64 的字符串的加密过程, 特别要注意的是加密程序 MD5.c 中的第 109~121 行。

② 用 4×4 键盘解码芯片 74C922 重新设计本例, 仍实现本例功能。

③ 选用另一种加密算法重新设计本例。

### 3. 源程序代码

```

001 //----- main.c -----
002 // 名称：用内置 EEPROM 与 1602 液晶设计的带 MD5 加密的电子密码锁
003 //-
004 // 说明：初始密码由 Passwrod.BIN 设定为：
005 // "e10adc3949ba59abbe56e057f20f883e"，它由明文密码"123456"进行
006 // MD5 加密后得到
007 //
008 // 数字键 0~9 中用于输入密码，密码不超过 10 位，输入完成后按下
009 // "开锁键"开锁，密码正确时 LED 点亮，液晶屏显示开锁成功
010 // 另外，本例还具备：上锁，重新输入密码，保存新密码，清除等功能
011 // 重设密码时要求先输入正确的密码并成功开锁
012 //
013 //-
014 #include <avr/io.h>
015 #include <avr/eeprom.h>
016 #include <util/delay.h>
017 #include <string.h>
018 #define INT8U unsigned char
019 #define INT16U unsigned int
020
021 //电子锁指示灯开关定义
022 #define LED_ON() PORTD &= ~_BV(PD7)
023 #define LED_OFF() PORTD |= _BV(PD7)
024 //蜂鸣器
025 #define BEEP() PORTA ^= _BV(PA0)
026
027 //液晶相关函数
028 extern void Initialize_LCD();
029 extern void LCD_ShowString(INT8U x, INT8U y, char * str);
030 //MD5 加密函数
031 extern char * MD5String(char * str);
032 //键盘扫描相关函数即按键键值
033 extern INT8U Keys_Scan();
034 extern INT8U KeyMatrix_Down();
035 extern INT8U KeyNo;
036
037 //LCD 提示字符串
038 const char * Title_Text = "Your Password... ";
039 //显示缓冲
040 char DSY_BUFFER[10] = "";

```



```
041 //保存在 EEPROM 中的密码(MD5 加密密码,其长度为 32 位)
042 char EEPROM_Password[33];
043 //用户输入的密码(密码不超过 10 位)
044 char UserInputPassword[11];
045 //-----
046 // 蜂鸣器子程序
047 //-----
048 void Sounder()
049 {
050     INT8U i;
051     for (i = 0; i < 100; i++)
052     {
053         _delay_ms(1); BEEP();
054     }
055 }
056
057 //-----
058 // 清除密码
059 //-----
060 void Clear_Password()
061 {
062     UserInputPassword[0] = '\0';
063     DSY_BUFFER[0] = '\0';
064 }
065
066 //-----
067 // 读取 EEPROM 中的密码(以'\0'或 0xFF 为结果标志)
068 //-----
069 void Read_EEPROM_Password()
070 {
071     INT8U i, *addr = 0x0000;
072     for (i = 0; i < 40; i++, addr++)
073     {
074         eeprom_busy_wait();
075         EEPROM_Password[i] = (char)eeprom_read_byte(addr);
076         if (EEPROM_Password[i] == '\0' ||
077             EEPROM_Password[i] == 0xFF) break;
078     }
079     if (EEPROM_Password[i] != '\0') EEPROM_Password[i] = '\0';
080 }
081
082 //-----
083 // 将新密码保存到 EEPROM,为便于读者调试观察
```

```

084 // 本函数同时保存了新密码的明文和密文,保存位置分别为:0x0050,0x0000)
085 //-----
086 void Save_Password_TO_EEPROM()
087 {
088     INT8U i, * addr = 0x0000;
089     //对新输入的密码用 MD5 加密
090     char * ps = MD5String(UserInputPassword);
091     //保存所输入新密码的密文,注意循环终止条件中的"<=",
092     //这是为了保证将字符串末尾的'\0'也写入 EEPROM
093     for (i = 0; i <= strlen(ps) ; i ++ , addr ++ )
094     {
095         eeprom_busy_wait();
096         eeprom_write_byte(addr,(INT8U)ps[i]);
097     }
098     //同时在 0x0050 地址保存明文,以便于调试观察,实际应用时应该删除
099     addr = (INT8U *)0x0050;
100     for (i = 0; i <= strlen(UserInputPassword); i ++ , addr ++ )
101     {
102         eeprom_busy_wait();
103         eeprom_write_byte(addr,(INT8U)UserInputPassword[i]);
104     }
105 }
106
107 //-----
108 // 主程序
109 //-----
110 int main()
111 {
112     INT8U i = 0;
113     INT8U IS_Valid_User = 0;
114     DDRA = 0xFF; PORTA = 0xFF;           //配置端口
115     DDRC = 0xFF;
116     DDRD = 0xFF;
117     LED_OFF();                         //初始时关闭 LED 指示灯
118     Initialize_LCD();                  //LCD 初始化
119     LCD_ShowString(0,0,(char *)Title_Text); //在第 0 行显示提示信息
120
121     //AVR EEPROM 的密码已由初始化 Password.BIN 文件导入
122     //下面将 EEPROM 中预设的密码读入 EEPROM_Password
123     Read_EEPROM_Password();
124     while(1)
125     {
126         //如果键盘矩阵有键按下则扫描键值,否则提前进入下一趟循环

```



```
127     if (KeyMatrix_Down()) Keys_Scan(); else continue;
128     //如果键值有效则发出按键声音,否则提前进入下一趟循环
129     if (KeyNo != 0xFF) Sounder(); else continue;
130     switch ( KeyNo )
131     {
132         //处理数字密码按键 0~9 -----
133         case 0;case 1; case 2; case 3; case 4;
134         case 5;case 6; case 7; case 8; case 9;
135
136         if ( i <= 9 ) //密码限制在 10 位以内
137         {
138             //如果 i 为 0 则执行一次清屏(输出 16 个空格符)
139             if ( i == 0 ) LCD_ShowString(0,1,"");
140
141             UserInputPassword[i] = KeyNo + '0';
142             UserInputPassword[i + 1] = '\0';
143
144             DSY_BUFFER[i] = '*';
145             DSY_BUFFER[ ++ i ] = '\0';
146
147             LCD_ShowString(0,1,DSY_BUFFER);
148         }
149         break;
150     case 10: //按 A 键开锁-----
151         //将读取的密码串与用户输入的密码进行 MD5 加密后的字符串比较
152         if (strcmp(MD5String(UserInputPassword),EEPROM_Password) == 0)
153         {
154             Clear_Password();
155             LCD_ShowString(0,1,"Unlock OK!");
156             IS_Valid_User = 1;
157             LED_ON();
158         }
159         else
160         {
161             Clear_Password();
162             LCD_ShowString(0,1,"ERROR!");
163             IS_Valid_User = 0;
164             LED_OFF();
165         }
166         i = 0;
167         break;
168
169     case 11: //按 B 键上锁-----
```

```

170         LED_OFF();
171         Clear_Password();
172         LCD_ShowString(0,0,(char *)Title_Text);
173         LCD_ShowString(0,1,"");
174         i = 0;
175         IS_Valid_User = 0;
176         break;
177
178     case 12://按 C 键设置新密码-----
179         //如果是合法用户则提示输入新密码
180         if ( !IS_Valid_User ) LCD_ShowString(0,1,"No rights! ");
181         else
182         {
183             i = 0;
184             LCD_ShowString(0,0,"New Password: ");
185             LCD_ShowString(0,1,"");
186         }
187         break;
188
189     case 13://按 D 键保存新密码-----
190         if ( !IS_Valid_User ) LCD_ShowString(0,1,"No rights! ");
191         else
192         {
193             //如果密码有效则保存新输入的密码
194             if ( strlen(UserInputPassword) != 0 )
195             {
196                 //为便于查看 EEPROM 内的密码数据,本例同时保存了两套密码
197                 //即:加密密码与未加密密码
198                 Save_Password_TO_EEPROM();
199                 LCD_ShowString(0,1,"Password Saved!");
200                 _delay_ms(1000); //1 s 后重新提示输入密码开锁
201                 LCD_ShowString(0,0,(char *)Title_Text);
202                 LCD_ShowString(0,1,"");
203                 //重新读取 EEPROM 中的新密码,保存到 EEPROM_Password 字符串中
204                 //或者用 strcpy 将 UserInputPassword 直接拷贝到 EEPROM_Password
205                 //以便后续开锁时与新密码比对.
206                 Read_EEPROM_Password();
207             }
208             else LCD_ShowString(0,1,"Password Empty!");
209             i = 0;
210         }
211         break;
212

```



```
213         case 14: //按 E 键消除所有输入
214             i = 0;
215             Clear_Password();
216             LCD_ShowString(0,1,"");
217         }
218         while (KeyMatrix_Down()); //如果按键未释放则等待
219     }
220 }

001 //----- MD5.c -----
002 // 名称: MD5 加密程序
003 //-----
004 #include <stdio.h>
005 #include <string.h>
006 #define INT8U unsigned char
007 #define INT16U unsigned int
008 #define INT32U unsigned long
009 #define PINT8U unsigned char *
010

011 //MD5 变换程序常量
012 #define S11 7
013 #define S12 12
014 #define S13 17
015 #define S14 22
016 #define S21 5
017 #define S22 9
018 #define S23 14
019 #define S24 20
020 #define S31 4
021 #define S32 11
022 #define S33 16
023 #define S34 23
024 #define S41 6
025 #define S42 10
026 #define S43 15
027 #define S44 21
028

029 typedef struct      //MD5 加密处理上下文结构
030 {
031     INT32U count[2]; //信息位长(bits length)
032     INT32U state[4]; //MD5 加密初始幻数及 MD5 摘要计算数据(128 位,16 字节,32 个十六进制字符)
033     INT8U buffer[64]; //处理缓冲(512 位)
034 } MD5_CTX;
```

```

035
036 static INT8U PADDING[64] = //512个填充位,第1位为1,其他位为0
037 {
038     0x80, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
039     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
040     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
041 };
042
043 //MD5 加密的基本位操作函数 F、G、H、I,其中 x、y、z 全部为 32-bit 的长整型数据
044 #define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
045 #define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
046 #define H(x, y, z) ((x) ^ (y) ^ (z))
047 #define I(x, y, z) ((y) ^ ((x) | (~z)))
048
049 //将 x 循环左移 n 位
050 #define ROTATE_LEFT(x, n) (((x) << (n)) | ((x) >> (32 - (n))))
051
052 //FF、GG、HH 与 II 分别用于第 1、2、3、4 轮转换
053 #define FF(a, b, c, d, x, s, ac) \
054 { (a) += F((b),(c),(d)) + (x) + (INT32U)(ac); (a) = ROTATE_LEFT((a),(s)); (a) += (b); }
055 #define GG(a, b, c, d, x, s, ac) \
056 { (a) += G((b),(c),(d)) + (x) + (INT32U)(ac); (a) = ROTATE_LEFT((a),(s)); (a) += (b); }
057 #define HH(a, b, c, d, x, s, ac) \
058 { (a) += H((b),(c),(d)) + (x) + (INT32U)(ac); (a) = ROTATE_LEFT((a),(s)); (a) += (b); }
059 #define II(a, b, c, d, x, s, ac) \
060 { (a) += I((b),(c),(d)) + (x) + (INT32U)(ac); (a) = ROTATE_LEFT((a),(s)); (a) += (b); }
061
062 //MD5 相关函数申明
063 void MD5Init (MD5_CTX * context);
064 void MD5Update(MD5_CTX * context, INT8U * input, INT16U inputLen);
065 void MD5Final (INT8U digest[16], MD5_CTX * context);
066 static void MD5Transform(INT32U [4], INT8U [64]);
067 static void Encode(INT8U *, INT32U *, INT16U);
068 static void Decode(INT32U *, INT8U *, INT16U);
069 //-----
070 // MD5 初始化
071 //-----
072 void MD5Init(MD5_CTX * context)
073 {
074     //初始位长(bits length)为 0
075     context->count[0] = 0;
076     context->count[1] = 0;
077     //4 个用于计算摘要的长整型幻数为:

```



```
078     //A: 01 23 45 67
079     //B: 89 AB CD EF
080     //C: FE DC DA 98
081     //D: 76 54 32 10
082     //按 Little Endian 方式存放于 state
083     context ->state[0] = 0x67452301;
084     context ->state[1] = 0xEFCDAB89;
085     context ->state[2] = 0x98BADCFC;
086     context ->state[3] = 0x10325476;
087 }
088
089 //-----
090 // MD5 核心计算更新过程
091 //-----
092 void MD5Update(MD5_CTX * context, INT8U * input, INT16U inputLen)
093 {
094     INT16U i, index, partLen;
095     //将位长 >> 3 然后 & 0x3F, 即 / 8 % 64, 得到字节数对 64 的余数, 存放于变量 index
096     //在本例 3 次对 MD5Update 的调用中, 首次调用该函数处理原始串时, index 将为 0
097     index = (INT16U)((context ->count[0] >> 3) & 0x3F);
098     //将字节长度 inputLen 转换为位长(<<3 即 * 8), 累加到 count[0], 如果溢出则向 count[1] 进位
099     if ((context ->count[0] += ((INT32U)inputLen << 3)) < ((INT32U)inputLen << 3))
100         context ->count[1]++;
101     //count[1]累加 intputlen 扩展为 32 位并 * 8 以后的高 32 位
102     //即 intput * 8>>32 位, 也就是 intputlen>>29
103     //本例中由于 inputlen 仅定义为 16 位, 扩展为 32 位并 * 8 以后, 实际最大有效位为 19 位.
104     //故本例(INT32U)inputLen >> 29 实际恒为 0.
105     context ->count[1] += ((INT32U)inputLen >> 29);
106     //摘取部分的字节长度(64 - 余数 index)
107     partLen = 64 - index;
108     //进行尽可能多次的变换
109     if (inputLen >= partLen)
110     {
111         //将 input 中的前 partlen 个字节拷贝到 context ->buffer 中从 index 开始的位置
112         //初始调用 MD5Update 处理原始串时, index = 0, partLen = 64
113         memcpy((PINT8U)&context ->buffer[index], (PINT8U)input, partLen);
114         //对 context ->state 及 context ->buffer 进行四轮变换
115         MD5Transform(context ->state, context ->buffer);
116         //对 input 中从从 partlen 开始的每 64 个字节与 state 进行变换, 直到超出 inputlen
117         for (i = partLen; i + 63 < inputLen; i += 64)
118             MD5Transform(context ->state, &input[i]);
119
120         index = 0;
```

```

121     }
122     else i = 0;
123     //将 input 中长度不足 64 字节未参与变换的部分复制到 context->buffer 中 index 开始的位置
124     //初始调用 MD5Update 时, index = 0
125     memcpy((PINT8U)&context->buffer[index],(PINT8U)&input[i],inputLen - i);
126 }
127
128 //-----
129 // MD5 进行最后变换处理,并将加密结果写入摘要数组 digest
130 //-----
131 void MD5Final(INT8U digest[16], MD5_CTX * context)
132 {
133     INT8U bits[8];
134     INT16U index, padLen;
135     //将保存于 count 中用 2 个长整数表示的位长转换为用 8 个字节数组 bits 表示
136     Encode(bits, context->count, 8);
137     //求字节长度对 64 的余数,以便求取填充字节
138     index = (INT16U)((context->count[0] >> 3) & 0x3F);
139     //计算待填充字节的个数,计算表达式为:(n * 64 + 56) - 原始串总字节数
140     //其中 n 为整数, index<56 时取 56 - index
141     //否则取 64 + 56 - index, 即 120 - index
142     padLen = (index < 56) ? (56 - index) : (120 - index);
143     //用补位填充字节再进行 MD5 更新计算,调用变换函数
144     //补位填充字节数组 PADDING 中仅第 1 字节为 0x80,以二进制的 1 开头,其他为全 0
145     MD5Update(context,(INT8U *)PADDING, padLen);
146     //最后用 8 字节表示的原始串长进行 MD5 更新计算,调用变换函数
147     //至此,信息串长为:n * 64 + 56 + 8
148     MD5Update(context, bits, 8);
149     //将 context->state 中的 4 个长整数转换为 16 个字节,存入摘要数组 digest
150     //digest 所保存的就是最后将输出的 MD5 加密结果
151     Encode(digest, context->state, 16);
152     //将 context 全部清零
153     memset((PINT8U)context, 0, sizeof (*context));
154 }
155
156 //-----
157 // MD5 四轮转换程序
158 //-----
159 static void MD5Transform(INT32U state[4], INT8U block[64])
160 {
161     //变换之前 a,b,c,d 首先分别获取上次变换后的 state[0]~state[3]的值
162     INT32U a = state[0], b = state[1], c = state[2], d = state[3], x[16];
163     //将 512 位的 block 解码为长整型(INT32U)数组 x(16 个元素)

```



```
164     Decode (x, block, 64);  
165     //第一轮变换-----  
166     FF (a, b, c, d, x[ 0], S11, 0xd76aaa478); //1  
167     FF (d, a, b, c, x[ 1], S12, 0xe8c7b756); //2  
168     FF (c, d, a, b, x[ 2], S13, 0x242070db); //3  
169     FF (b, c, d, a, x[ 3], S14, 0xc1bdceee); //4  
170     FF (a, b, c, d, x[ 4], S11, 0xf57c0faf); //5  
171     FF (d, a, b, c, x[ 5], S12, 0x4787c62a); //6  
172     FF (c, d, a, b, x[ 6], S13, 0xa8304613); //7  
173     FF (b, c, d, a, x[ 7], S14, 0xfd469501); //8  
174     FF (a, b, c, d, x[ 8], S11, 0x698098d8); //9  
175     FF (d, a, b, c, x[ 9], S12, 0x8b44f7af); //10  
176     FF (c, d, a, b, x[10], S13, 0xfffff5bb1); //11  
177     FF (b, c, d, a, x[11], S14, 0x895cd7be); //12  
178     FF (a, b, c, d, x[12], S11, 0x6b901122); //13  
179     FF (d, a, b, c, x[13], S12, 0xfd987193); //14  
180     FF (c, d, a, b, x[14], S13, 0xa679438e); //15  
181     FF (b, c, d, a, x[15], S14, 0x49b40821); //16  
182     //第二轮变换-----  
183     GG (a, b, c, d, x[ 1], S21, 0xf61e2562); //17  
184     GG (d, a, b, c, x[ 6], S22, 0xc040b340); //18  
185     GG (c, d, a, b, x[11], S23, 0x265e5a51); //19  
186     GG (b, c, d, a, x[ 0], S24, 0xe9b6c7aa); //20  
187     GG (a, b, c, d, x[ 5], S21, 0xd62f105d); //21  
188     GG (d, a, b, c, x[10], S22, 0x02441453); //22  
189     GG (c, d, a, b, x[15], S23, 0xd8a1e681); //23  
190     GG (b, c, d, a, x[ 4], S24, 0xe7d3fb8); //24  
191     GG (a, b, c, d, x[ 9], S21, 0x21e1cde6); //25  
192     GG (d, a, b, c, x[14], S22, 0xc33707d6); //26  
193     GG (c, d, a, b, x[ 3], S23, 0xf4d50d87); //27  
194     GG (b, c, d, a, x[ 8], S24, 0x455a14ed); //28  
195     GG (a, b, c, d, x[13], S21, 0xa9e3e905); //29  
196     GG (d, a, b, c, x[ 2], S22, 0xfcfa3f8); //30  
197     GG (c, d, a, b, x[ 7], S23, 0x676f02d9); //31  
198     GG (b, c, d, a, x[12], S24, 0x8d2a4c8a); //32  
199     //第三轮变换-----  
200     HH (a, b, c, d, x[ 5], S31, 0xfffffa3942); //33  
201     HH (d, a, b, c, x[ 8], S32, 0x8771f681); //34  
202     HH (c, d, a, b, x[11], S33, 0x6d9d6122); //35  
203     HH (b, c, d, a, x[14], S34, 0xfde5380c); //36  
204     HH (a, b, c, d, x[ 1], S31, 0xa4beeaa4); //37  
205     HH (d, a, b, c, x[ 4], S32, 0x4bdecfa9); //38  
206     HH (c, d, a, b, x[ 7], S33, 0xf6bb4b60); //39
```

```

207     HH (b, c, d, a, x[10], S34, 0xbefbc70); //40
208     HH (a, b, c, d, x[13], S31, 0x289b7ec6); //41
209     HH (d, a, b, c, x[ 0], S32, 0xeaa127fa); //42
210     HH (c, d, a, b, x[ 3], S33, 0xd4ef3085); //43
211     HH (b, c, d, a, x[ 6], S34, 0x04881d05); //44
212     HH (a, b, c, d, x[ 9], S31, 0xd9d4d039); //45
213     HH (d, a, b, c, x[12], S32, 0xe6db99e5); //46
214     HH (c, d, a, b, x[15], S33, 0x1fa27cf8); //47
215     HH (b, c, d, a, x[ 2], S34, 0xc4ac5665); //48
216     //第四轮变换-----
217     II (a, b, c, d, x[ 0], S41, 0xf4292244); //49
218     II (d, a, b, c, x[ 7], S42, 0x432aff97); //50
219     II (c, d, a, b, x[14], S43, 0xab9423a7); //51
220     II (b, c, d, a, x[ 5], S44, 0xfc93a039); //52
221     II (a, b, c, d, x[12], S41, 0x655b59c3); //53
222     II (d, a, b, c, x[ 3], S42, 0x8f0ccc92); //54
223     II (c, d, a, b, x[10], S43, 0xffeff47d); //55
224     II (b, c, d, a, x[ 1], S44, 0x85845dd1); //56
225     II (a, b, c, d, x[ 8], S41, 0x6fa87e4f); //57
226     II (d, a, b, c, x[15], S42, 0xfe2ce6e0); //58
227     II (c, d, a, b, x[ 6], S43, 0xa3014314); //59
228     II (b, c, d, a, x[13], S44, 0x4e0811a1); //60
229     II (a, b, c, d, x[ 4], S41, 0xf7537e82); //61
230     II (d, a, b, c, x[11], S42, 0xbd3af235); //62
231     II (c, d, a, b, x[ 2], S43, 0x2ad7d2bb); //63
232     II (b, c, d, a, x[ 9], S44, 0xeb86d391); //64
233
234     //将变换后的 a,c,b,d 分别累加到 state[0]~state[3]
235     state[0] += a; state[1] += b; state[2] += c; state[3] += d;
236     //将数组 x 清零
237     memset ((PINT8U)x, 0, sizeof (x));
238 }
239
240 //-----
241 // 将 32 位的长整型数组 input 转换为 8 位的字节数组 output,(INT32U) -->(INT8U)
242 //-----
243 static void Encode(INT8U * output, INT32U * input, INT16U len)
244 {
245     INT16U i, j;
246     for (i = 0, j = 0; j < len; i++, j += 4)
247     {
248         output[j] = (INT8U)(input[i] & 0xFF);
249         output[j + 1] = (INT8U)((input[i] >> 8) & 0xFF);

```

```

250         output[j + 2] = (INT8U)((input[i] >> 16) & 0xFF);
251         output[j + 3] = (INT8U)((input[i] >> 24) & 0xFF);
252     }
253 }
254
255 //-----
256 // 将 8 位的字节数组 input 转换为 32 位的长整型数组 output,(INT8U) —>(INT32U)
257 //-----
258 static void Decode(INT32U * output, INT8U * input, INT16U len)
259 {
260     INT16U i, j;
261     for (i = 0, j = 0; j < len; i++, j += 4) output[i] =
262         (((INT32U)input[j]) |
263          (((INT32U)input[j + 1]) << 8) |
264          (((INT32U)input[j + 2]) << 16) |
265          (((INT32U)input[j + 3]) << 24));
266 }
267
268 //-----
269 // 对原始字符串 str 进行 MD5 加密并返回密文字符串
270 //-----
271 char * MD5String(char * str)
272 {
273     INT8U i;
274     MD5_CTX context;           //MD5 加密上下文结构变量
275     INT8U digest[16];          //摘要字节数组
276     INT16U len = strlen(str); //原始字符串长
277
278     static char MD5Str[33];      //最后输出的 MD5 加密字符串
279     MD5Init(&context);        //初始化 context
280     MD5Update(&context, (INT8U *)str, len); //首次更新 context
281     MD5Final(digest, &context); //MD5 最终变换处理
282
283     //将 16 字节的摘要数组转换为 32 个十六进制字符,存入 MD5Str
284     for (i = 0; i < 16; i++)
285         sprintf(MD5Str + 2 * i, "%02x", digest[i]);
286
287     return MD5Str;
288 }

```

## 5.20 12864LCD 显示 24C08 保存的开机画面

本例 AT24C08 中预先保存有一幅画面,运行本系统时单片机程序从 AT24C08 中读取该

画面并显示在 LGM12641BS1R(128x64)液晶屏上。本例电路及运行效果如图 5-20 所示。

## 1. 程序设计与调试

本例程序设计与调试要点在于如何读/写 24C08 存储器,24C08 存储空间为 1K 字节,访问 24C08 时需要 10 位内存地址。该存储器硬地址引脚仅有 A2,在同一 I<sup>2</sup>C 总线上可并联两片 24C08 存储器。

由 24C08 技术手册可知,其控制字节为 1010—A2—P1—P0—R/W。其中 A2 为硬地址位,P1 与 P0 为 10 位内存地址的高 2 位。对于接收到的 10 位地址 addr,为将其中的最高位 2 位分离出来,可使用语句:

page=(INT8U)(addr>>8<<1) 或

page=(INT8U)((addr>>7)&0x06)

第 1 行语句将最高 2 位移到最低 2 位后再左移 1 位,与控制字节中的 P1~P0 位匹配,注意这种写法不能简写为直接右移 7 位,因为这样不能确保首先将最低的 R/W 位设为 0。

第 2 行语句先右移 7 位,使 10 位地址的最高 2 位与 P1~P0 匹配,同时通过 &0x06 将除这 2 位以外的其他位清零。对于第一种写法,要避免函数所接收到的地址可能非法(例如可能超过 10 位),也应将最后结果 & 0x06。

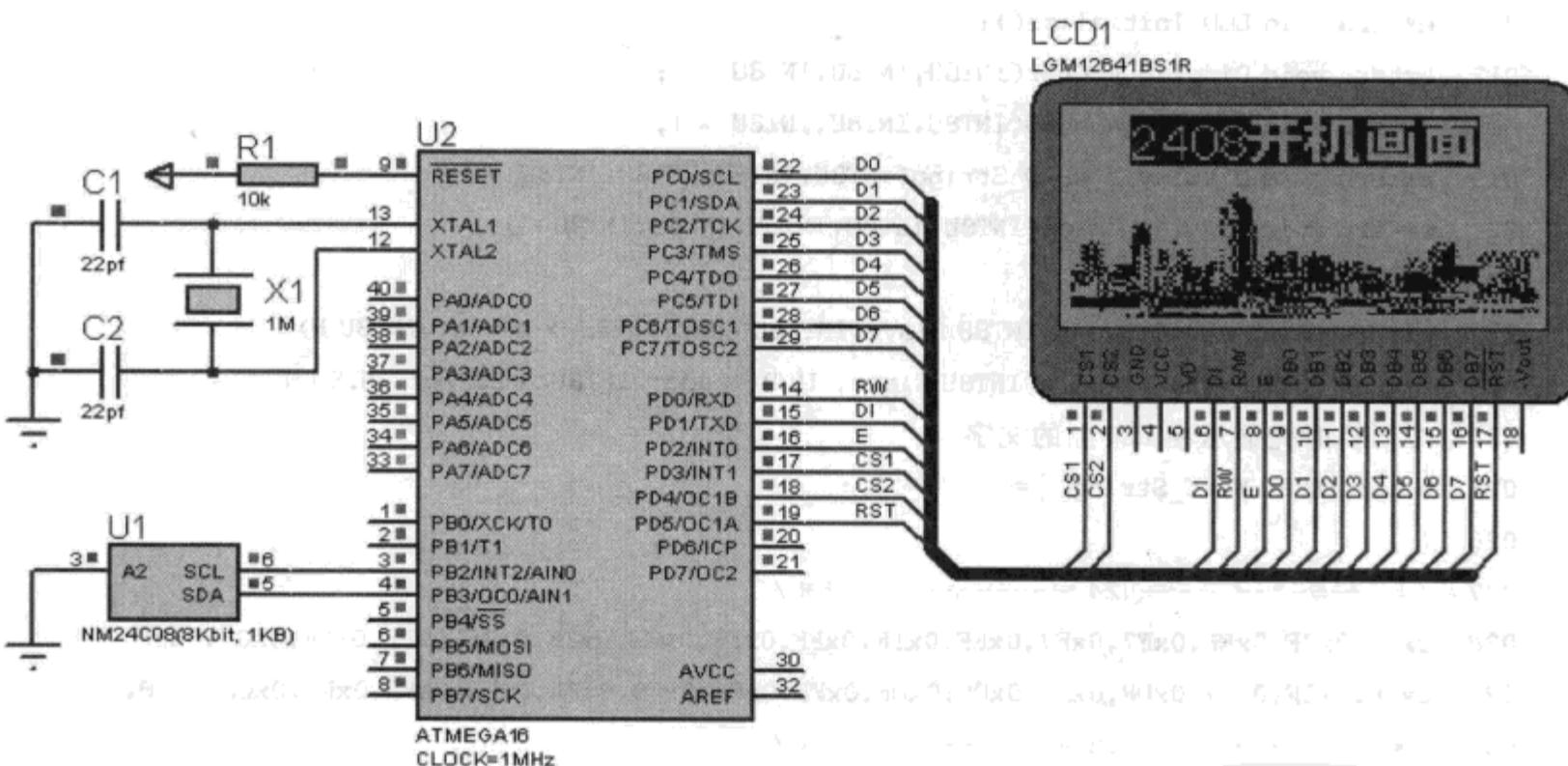


图 5-20 12864LCD 显示 24C08 保存的开机画面

有关本例其他代码的设计与调试问题,大家可进一步参阅 AT24C08 的技术手册文件。

## 2. 实训要求

① 重新改用 TWI 接口程序读/写 AT24C08 存储器。

② 将某网络 IP 地址、子网掩码、网关、2 个 DNS 地址存入 24C08,系统运行时各项地址信息能显示在 LCD 屏上。完成该设计后进一步在电路中添加矩阵键盘,实现对各项地址信息的设置与保存。

### 3. 源程序代码

```

001 //----- main.c -----
002 // 名称:开机显示 24C08 中的画面
003 //-----
004 // 说明:开机时系统从 24C08 中读取画面并显示到 12864LCD
005 // 如果需要先将数据写入 I2C,然后再读取并显示,可将本例中
006 // 用/* */注释掉的代码重新允许执行
007 //
008 //-----
009 # include <avr/io.h>
010 # include <avr/pgmspace.h>
011 # include <util/delay.h>
012 # define INT8U unsigned char
013 # define INT16U unsigned int
014
015 //12864LCD 相关函数
016 extern void LCD_Initialize();
017 extern void Display_A_Char(INT8U, INT8U, INT8U * );
018 extern void Display_A_WORD(INT8U, INT8U, INT8U * );
019 extern void Display_A_WORD_String(INT8U, INT8U, INT8U, INT8U * );
020 extern void Display_Image(INT8U, INT8U, INT8U, INT8U, INT8U * );
021 //I2C 相关函数
022 extern INT8U AT24CxxRead(INT8U Slave, INT16U addr, INT8U * Buffer, INT8U N);
023 extern INT8U AT24CxxWrite(INT8U Slave, INT16U addr, INT8U * Buffer, INT8U N);
024 //开机时先显示在 LCD 上的文字
025 const INT8U Word_String[] =
026 {
027 /* ----- 24 ----- */
028 0xFF, 0x9F, 0xEF, 0xF7, 0xF7, 0xEF, 0x1F, 0xFF, 0xFF, 0xFF, 0x3F, 0xDF, 0xEF, 0x07, 0xFF,
029 0xFF, 0xCF, 0xD7, 0xDB, 0xDD, 0xDE, 0xDF, 0xFF, 0xFF, 0xF9, 0xFA, 0xFB, 0xFB, 0xFB, 0xC0, 0xFB,
030 /* ----- 08 ----- */
031 0xFF, 0x1F, 0xEF, 0xF7, 0xF7, 0xEF, 0x1F, 0xFF, 0x9F, 0x6F, 0xF7, 0xF7, 0x6F, 0x9F,
032 0xFF, 0xF0, 0xEF, 0xDF, 0xDF, 0xEF, 0xF0, 0xFF, 0xF3, 0xED, 0xDE, 0xDE, 0xED, 0xF3,
033 /* ----- 开 ----- */
034 0x3F, 0x39, 0x39, 0x39, 0x01, 0x01, 0x39, 0x39, 0x39, 0x01, 0x01, 0x39, 0x39, 0x39, 0x3F, 0xFF,
035 0xFF, 0xDF, 0x9F, 0xC7, 0xE0, 0xF8, 0xFF, 0xFF, 0x80, 0x80, 0xFF, 0xFF, 0xFF, 0xFF,
036 /* ----- 机 ----- */
037 0xE7, 0x67, 0x00, 0x00, 0x67, 0xE7, 0xFF, 0x01, 0x01, 0xF9, 0xF9, 0x01, 0x01, 0xFF, 0xFF, 0xFF,
038 0xF9, 0xFC, 0x80, 0x80, 0xFE, 0xDC, 0x8F, 0xC0, 0xF0, 0xFF, 0xFF, 0xC0, 0x80, 0x9F, 0x8F, 0xFF,
039 /* ----- 画 ----- */
040 0xFF, 0x19, 0x19, 0xF9, 0x09, 0x09, 0x69, 0x09, 0x69, 0x09, 0x09, 0x09, 0xF9, 0x19, 0x19, 0x19, 0xFF, 0xFF,
041 0xFF, 0xC0, 0xC0, 0xCF, 0xC8, 0xC8, 0xCB, 0xC8, 0xCB, 0xC8, 0xC8, 0xCF, 0x80, 0x80, 0xFF, 0xFF,
```

```

042 /* ----- 面 ----- */
043 0xF9,0x09,0x09,0xC9,0x09,0x09,0x41,0x41,0x09,0x09,0xC9,0xC9,0x09,0x09,0xF9,0xFF,
044 0xFF,0x80,0x80,0xCF,0xC0,0xC0,0xCB,0xCB,0xC0,0xC0,0xCF,0xCF,0x80,0x80,0xFF,0xFF
045 };
046
047 //-----
048 // 保存到 24C08 的图片:某城市图片,宽度 * 高度 = 128 * 40 (共 128 * 40/8 = 640 字节)
049 // 这些数据已经存入了 24C08 芯片,故下面不需要重新调用写入 24C08 的代码
050 // (如果不在本例中做 24C08 写入实验,下面的点阵数组可省略)
051 //-----
052 /* prog_uchar Start_Screen_Image[] = {
053 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
054 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
.....限于篇幅,这里省略了待写入 24C08 存储器的图像点阵数据
091 0x6F,0x37,0x67,0x7C,0x25,0x7C,0x24,0x54,0x2C,0x42,0x24,0x24,0x21,0x30,0x23,0x14,
092 0x03,0x24,0x01,0x24,0x05,0x20,0x07,0x20,0x27,0x21,0x23,0x27,0x17,0x27,0x15,0x23
093 }; */
094
095 //显示缓冲,因 RAM 限制,仅定义为 64 字节
096 INT8U DisplayBuffer[64];
097 //-----
098 // 主程序
099 //-----
100 int main()
101 {
102     INT8U p; //INT8U i,
103     DDRC = 0xFF; PORTC = 0xFF;
104     DDRD = 0xFF; PORTD = 0xFF;
105     DDRB = 0xFF; PORTB = 0xFF;
106     //初始化 LCD
107     LCD_Initialize();
108     //首先显示“2408 开机画面”
109     //从第 0 页开始,左边距 16,共显示 6 个 16x16 的汉字与数字信息,
110     //每 2 个数字合为一个汉字.
111     Display_A_WORD_String(0,16,6,(INT8U *)Word_String);
112     //-----
113     //将屏幕图像写入 24C08,本例中可省略,因为数据已经绑定到了 I2C
114     //每页写入时需要保证适当的延时,否则会出现间隔性写入失败
115     //-----
116     /* for(i = 0 ; i < 40; i++)
117     {
118         AT24CxWrite(0xa0,i * 16,(prog_uchar *) (Start_Screen_Image + i * 16),16);
119         _delay_ms(10);

```

```

120     } */
121
122 //-----
123 //从 1K 字节的 24C08 中读取 640 个字节的屏幕图像
124 //因 RAM 空间有限,单次仅读取 64 字节放在显示缓冲 DisplayBuffer 中
125 //每次循环显示 128 字节,每 128 字节分别显示在第 3、4、5、6、7 页
126 //每页左边距为 0
127 //-----
128 for(p = 0; p < 5 ;p++)
129 {
130     AT24CxxRead(0xa0,p * 2 * 64,DisplayBuffer,64);
131     Display_Image( 3 + p, 0, 64, 1, DisplayBuffer);
132     AT24CxxRead(0xa0,(p * 2 + 1) * 64,DisplayBuffer,64);
133     Display_Image( 3 + p, 64, 64, 1, DisplayBuffer);
134 }
135 while (1);
136 }

```

```

001 //----- 24C08.c -----
002 // 名称:AT24C08 读/写程序
003 // (本例未使用 TWI 接口程序,改用模拟 I2C 时序读/写 AT24C08 存储器)
004 //-----
005 #include <avr/io.h>
006 #include <avr/pgmspace.h>
007 #include <util/delay.h>
008 #define INT8U unsigned char
009 #define INT16U unsigned int
010 #define INT32U unsigned long
011
012 //AT24XXXX 引脚定义
013 #define SCL PB2
014 #define SDA PB3
015 //AT24XXXX 引脚操作定义
016 #define SDA_1() PORTB |= _BV(SDA)
017 #define SDA_0() PORTB &= ~_BV(SDA)
018 #define SCL_1() PORTB |= _BV(SCL)
019 #define SCL_0() PORTB &= ~_BV(SCL)
020 #define SDA_DDR_1() DDRB |= _BV(SDA)
021 #define SDA_DDR_0() DDRB &= ~_BV(SDA)
022 #define R_SDA() PINB & _BV(SDA)
023 //-----
024 // 起始位
025 //-----

```

```

026 void Start()
027 {
028     SDA_1(); _delay_us(4);
029     SCL_1(); _delay_us(4);
030     SDA_0(); _delay_us(4);
031     SCL_0();
032 }
033
034 //-----
035 // 停止位
036 //-----
037 void Stop()
038 {
039     SDA_0(); _delay_us(4);
040     SCL_1(); _delay_us(4);
041     SDA_1(); _delay_us(4);
042     SDA_0();
043 }
044
045 //-----
046 // 输出 ACK(ACK/NACK)
047 //-----
048 void W_ACK(INT8U a)
049 {
050     SCL_0(); _delay_us(4);
051     SDA_0(); if(a) SDA_1(); _delay_us(4);
052     SCL_1(); _delay_us(4);
053     SCL_0();
054 }
055
056 //-----
057 // 读 ACK
058 //-----
059 INT8U R_ACK()
060 {
061     INT8U n = 10;
062     SCL_0(); _delay_us(4);
063     SDA_DDR_0(); //SDA 设为输入
064     SDA_1(); //内部上拉
065     SCL_1();
066     _delay_us(4);
067     while(R_SDA() && n) n--;
068     SCL_0();

```



```
069     SDA_DDR_1();                                //SDA 设为输出
070     return n ? 0 : 1;
071 }
072
073 //-----
074 // 写 1 字节
075 //-----
076 INT8U WriteByte(INT8U dat)
077 {
078     INT8U i;
079     SCL_0();
080     for(i = 0x80; i != 0x00; i >>= 1)
081     {
082         if(dat & i) SDA_1(); else SDA_0(); _delay_us(4);
083         SCL_1(); _delay_us(4);
084         SCL_0(); _delay_us(4);
085     }
086     return R_ACK();
087 }
088
089 //-----
090 // 读 1 字节
091 //-----
092 INT8U ReadByte()
093 {
094     INT8U dat = 0, i;
095     SCL_0();
096     SDA_DDR_0();                                //SDA 设为输入
097     SDA_1();                                    //内部上拉
098     for(i = 0; i < 8; i++)
099     {
100         dat <<= 1;
101         SCL_1(); _delay_us(4); if(R_SDA()) dat |= 0x01;
102         SCL_0(); _delay_us(4);
103     }
104     SDA_DDR_1();
105     return dat;
106 }
107
108 //-----
109 // 从 AT24C08 连续读取数据
110 //-----
111 INT8U AT24CxxRead(INT8U Slave, INT16U addr, INT8U * Buffer, INT8U N)
```

```

112 {
113     //将 AT24C08 共 10 位地址的高 2 位移到技术手册中控制字节内 A2,P1,P0 中的
114     //P1,P0 对应位置,针对格式 1010 - A2 - P1 - P0 - R/W. 根据本例电路,其中 A2 固定为 0
115     // page = 0000 - 0 - P1 - P0 - 0
116     //(注意不要将下面的语句简化为直接右移 7 位)
117     INT8U page = (INT8U)(addr >> 8 << 1);
118     //要防止接收到超过 10 位的异常地址 addr,上面语句可改成
119     //INT8U page = (INT8U)(addr >> 7) & 0x06;
120
121     Start();                                //I2C 启动
122     if (WriteByte( Slave | page)) return 0;    //器件地址(及页地址)
123     if (WriteByte(addr))      return 0;        //器件数据地址
124     Start();                                //重新启动
125     //器件数据地址,读操作
126     if (WriteByte( Slave | page | 0x01)) return 0;
127     while(N--)                            //读取 N 位
128     {
129         * (Buffer++) = ReadByte();
130         if(N) W_ACK(0); else W_ACK(1);        //前 N-1 位发送 ACK,最后一位发送 NACK
131     }
132     Stop();                                //I2C 停止
133     return 1;
134 }
135
136 //-----
137 // 将 Flash 存储器中的多个字节连续写入 AT24C08
138 //-----
139 INT8U AT24CxxWrite(INT8U Slave, INT16U addr,prog_uchar * Buffer, INT8U N)
140 {
141     INT8U i, page = (INT8U)(addr >> 8 << 1);
142     Start();                                //I2C 启动
143     if (WriteByte(Slave | page)) return 0;    //器件地址(及页地址)
144     if (WriteByte(addr))      return 0;        //器件数据地址
145     for(i = 0; i < N; i++)                  //写入 N 个字节数据
146     {
147         if (WriteByte(pgm_read_byte(&Buffer[i]))) return 0; //发送数据
148         _delay_ms(2);
149     }
150     Stop();                                //I2C 停止
151     return 1;
152 }

```

## 5.21 12864LCD 显示 EPROM27C256 保存的开机画面

本例在 EPROM 存储器 27C256 中预先保存了一幅图像数据,案例运行时单片机程序从 27C256 中读取画面数据并显示在 12864 液晶显示屏上。本例电路及运行效果如图 5-21 所示。

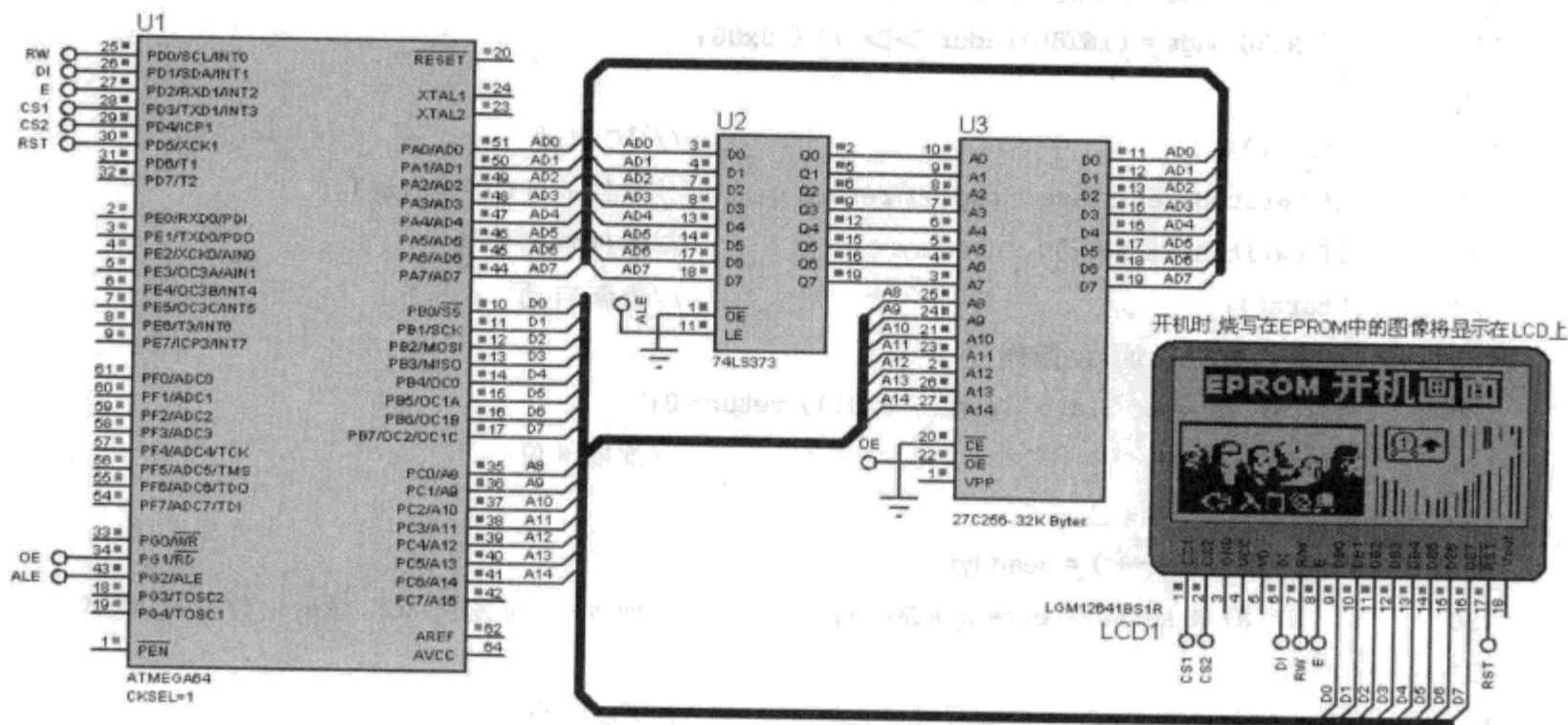


图 5-21 12864LCD 显示 EPROM27C256 保存的开机画面

### 1. 程序设计与调试

27C256 的 A0~A14 地址引脚可寻址  $2^{15} = 32K$  字节内存空间,  $\overline{CE}$  为片选引脚,  $\overline{OE}$  为输出使能引脚, VCC 与 GND 引脚在 Proteus 中被隐藏, VPP 引脚用于提供编程电压(本例中未连接)。

根据本例电路可定义外部内存起始地址:

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

根据该定义可知外部第 i 字节空间地址为:

```
EXTMEM_START_ADDR + i
```

访问该字节空间数据时可用表达式:

```
EXTMEM_START_ADDR[i] 或 *(EXTMEM_START_ADDR + i)
```

对于 27C256 所存储的图像数据,可先通过工具软件或自编程序生成 bin 文件,然后进行烧写,在 Proteus 仿真环境中则只需要通过设置 27C256 的 Image File 属性为该 bin 文件即可。

在 Image File 属性的下面还有:File Base Address(hex)与 File Address Shift(bits)属性,它用于重新映射 bin 文件,其中 Base 设置从 bin 文件中读入数据的起始地址,Shift 指明跳过的字节数。下面是 2 个设置示例:

- ① 将本例 Base Address 设为 0x0020, Shift 设置为 0, 读入 27C256 中的数据将从 bin 文件的第 0x0020 地址位置开始顺序读入字节。

② 假设某电路中有 2 片 27C256 共保存 64K( $32\text{K} \times 2 = 64\text{K}$ )字节数据, 地址范围为 0x0000~0xFFFF, 现要求一片 27C256 中保存偶地址字节数据, 另一片中保存奇地址字节数据, 则 2 片 27C256 的设置分别如下:

第 1 片: BASE=0x0000 SHIFT=1;

第 2 片: BASE=0x0001 SHIFT=1。

最后讨论一下如何生成待绑定到 27C256 的初始图像文件的 Turbo C 程序: 为将数据写入 bin 文件, 以便映射到 27C256, 首先需要用 Zimo 软件按本例液晶取模方式获取某图形文件 (BPM、JPG、GIF、PNG 等) 点阵, 然后用 Windows 平台编程工具将所获取的点阵字节写入 bin 文件。下面用 Turbo C 生成 bin 文件的源程序可供大家参考:

```
# include <stdio.h>
unsigned char bitmap[] = //128x40
{
    0xFF, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03,
    .....限于篇幅, 这里省略了本例所使用的图像点阵数据
    0xFF, 0x00, 0x00, 0x00, 0xFF, 0x00, 0x00, 0x00, 0xFF, 0x00, 0x00, 0xFF
};

void main()
{
    FILE * fp;
    fp = fopen("c:\\27C256.bin", "wb");
    fwrite(bitmap, 1, 128 * 64, fp);
    fclose(fp);
}
```

## 2. 实训要求

① 本例 EPROM 芯片使用了扩展接口 AD0~AD7、A8~A15, 而液晶屏则未使用扩展接口, 完成本例调试后, 重新修改电路, 使液晶屏与 EPROM 共用扩展接口, 完成 EPROM 数据读取与液晶屏显示。

② 在本例中再添加 1 片 27C256 存储器, 存储多幅图像点阵数据, 编程实现本例液晶的多幅图像滚动显示效果。

## 3. 源程序代码

```
001 //----- main.c -----
002 // 名称: 开机显示 EPROM27C256 中的画面
003 //-----
004 // 说明: 开机时系统从 EPROM27C256 中读取画面并显示到 12864LCD
005 //
006 //-----
007 # include <avr/io.h>
008 # include <util/delay.h>
009 # define INT8U unsigned char
```



```

053 //0x03,0x03,0x03,0x03,0x03,0x03,0x03,0x03,0x03,0x03,0x03,0x03,0x03,0x03,
.....限于篇幅,这里省略了大部分图像数据
090 //0x00,0xFF,0x00,0x00,0xFF,0x00,0x00,0xFF,0x00,0x00,0xFF,0x00,0x00,0xFF,0x00,
091 //0xFF,0x00,0x00,0x00,0xFF,0x00,0x00,0x00,0xFF,0x00,0x00,0x00,0xFF,0x00,0x00,0xFF
092 //}
093
094 //-----
095 // 主程序
096 //-----
097 int main()
098 {
099     INT8U Page;
100     DDRA = 0xFF; PORTA = 0xFF;
101     DDRB = 0xFF; PORTB = 0xFF;
102     DDRC = 0xFF; PORTC = 0xFF;
103     DDRD = 0xFF; PORTD = 0xFF;
104     //允许访问外部存储器
105     MCUCR |= 0x80;
106     //初始化 LCD
107     LCD_Initialize();
108     //首先显示"EPROM 开机画面"
109     //从第 0 页开始,左边距 9,共显示 7 个 16 * 16 的汉字与数字信息,
110     //每 2 个字母合为一个汉字,M 单独占一个汉字宽度
111     Display_A_WORD_String(0, 9, 7, (INT8U *)Word_String);
112     //-----
113     //接着从外部 EPROM 中读取 640 个字节的屏幕图像
114     //外部内存起始地址为:EXTMEM_START_ADDR
115     //每次循环读取并显示 128 字节
116     //每 128 字节分别显示在第 3、4、5、6、7 页,每页左边距为 0
117     //-----
118     for(Page = 0; Page < 5; Page++)
119     {
120         Display_Image( Page + 3, 0, 128, 1,
121                         (INT8U * )(EXTMEM_START_ADDR + Page * 128));
122     }
123     while (1);
124 }

```

## 5.22 I<sup>2</sup>C – AT24C1024×2 硬字库应用

本例在两块 AT24C1024 芯片中内置了 16×16 点阵汉字库文件,汉字库文件 HZK16 共 262 KB,两块芯片各保存 128K,多出的部分(262K–128×2=6 KB)被截除。本例运行时,程



序根据汉字内码得到区位码，再根据区位码从硬件字库中提取汉字点阵，所提取的字库点阵再进一步转换为本例液晶屏汉字显示所需要的格式，最后显示在液晶屏上。本例显示任何汉字时，不再需要使用要用专门的取字模软件（例如 Zimo 软件）提取固定汉字字模。本例电路及程序运行效果如图 5-22 所示。

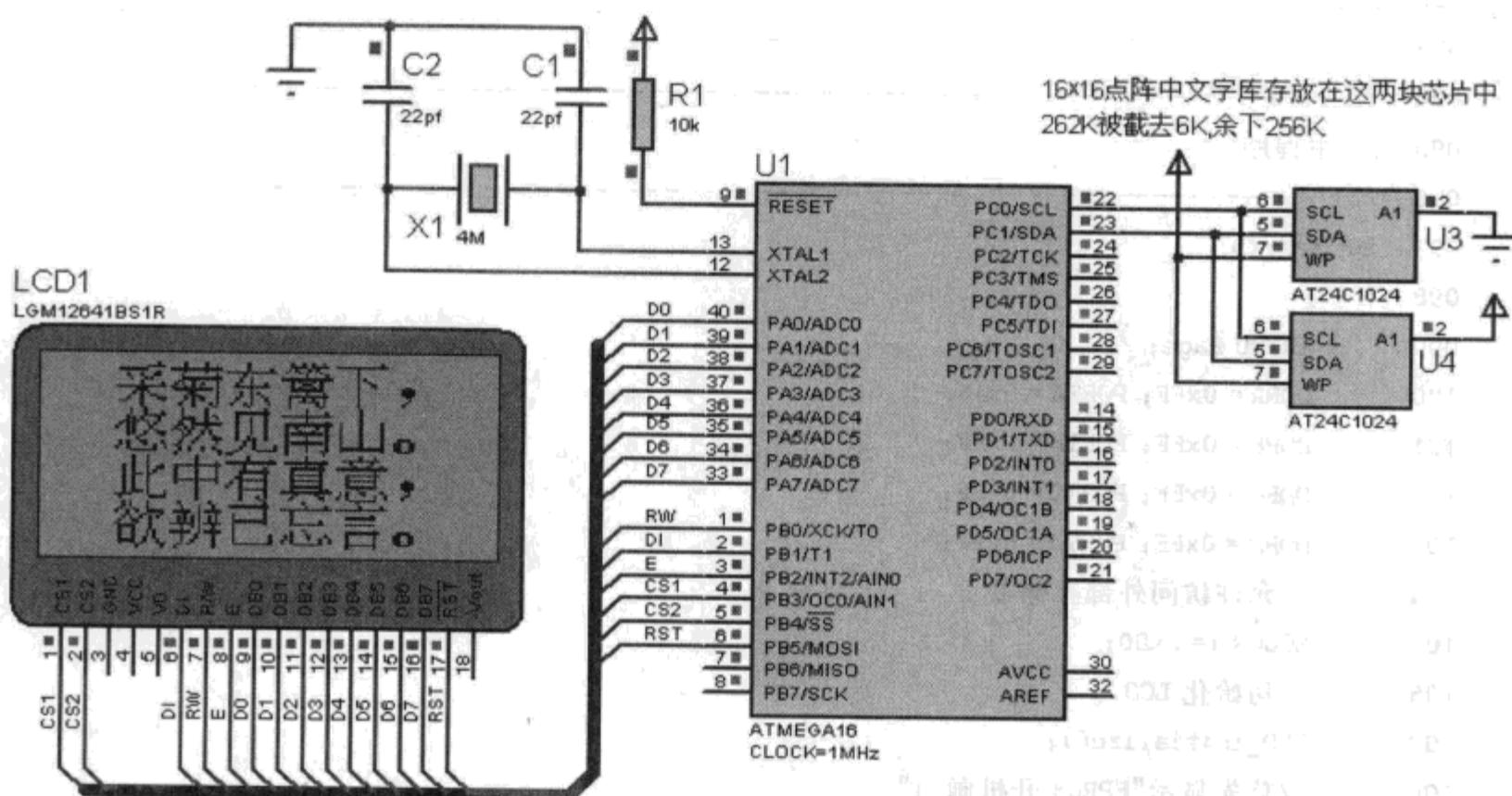


图 5-22 I<sup>2</sup>C-AT24C1024×2 硬字库应用

## 1. 程序设计与调试

由于当前版本的 Proteus 中没有兼容 I<sup>2</sup>C 接口的大容量 EEPROM，本例使用了 2 片具有 128K 字节空间的 AT24C1024 分别保存汉字库的前、后两部分。拆分子库文件时可以自己编写 TC 程序完成，也可以直接使用本书案例压缩包中提供的文件拆分软件。

本例程序读取各汉字内码后，将两字节汉字内码分别减去 0xA0 得到区位码，再根据区位码求出汉字点阵在字库中的位置。由于汉字被存放于 94 行 94 列的区域中，每个汉字点阵占 32 字节 ( $16 \times 16 / 8 = 32$ )，根据上述字库结构，由汉字区位码（即汉字在字库表中的行/列位置）可得出汉字点阵在字库中起点位置（或称偏移位置）计算公式：

$$\text{Offset} = (94 \times (\text{SectionCode} - 1)) + (\text{PlaceCode} - 1) \times 32L$$

其中 SectionCode 与 PlaceCode 分别为区位与位码。

汉字库文件 HZK 中各汉字的 32 字节点阵是逐行取模的，每行 16 个像素，由 2 个字节保存，从上到下 16 行共 32 字节，其取模格式如图 5-23（左）所示。

本例液晶显示汉字时，需要的汉字点阵取模顺序是从汉字上半部分开始，从左到右垂直取得 16 字节，且各字节是高位在下、低位在上，然后再从左到右取得汉字下半部分的 16 字节。取模格式如图 5-22（右）所示。

由于两者取模方式不同，本例还需要将 HZK 点阵格式转换为 LCD 点阵格式。

本例中函数 Read\_HZ\_dot\_Matrix\_AND\_Convert\_TO\_LCD\_Fmt 首先根据汉字内码得到区位码，根据区位码计算偏移量 Offset，再从两片 24C1024 读取汉字点阵，然后将这 32 字节

的 HZK 点阵格式转换为 LCD 点阵格式,由图 5-23 左边的 HZK 取模格式转换为右边的 LCD 取模格式,转换程序算法如下:

① 外层循环控制 4 个独立的块(block)分别进行转换,上两块序号为 0、1,下两块序号为 2、3,循环控制变量  $block=0 \sim 3$ 。

② 内层循环扫描 HZK 格式下当前块内的 8 列,生成 LCD 格式下的 8 个字节,LCD 格式下当前块内的第  $i$  个字节即 HZK 格式下当前块内的第  $i$  列,循环控制变量  $i=0 \sim 7$ 。

③ 最内层循环获取 HZK 格式下当前块内第  $i$  列中的 8 位,循环获取各位的控制变量  $j=0 \sim 7$ ,每次遇到位 1 时即写入 LCD 格式下当前块内第  $i$  字节的第  $j$  位。

案例源程序中对各语句给出了很详细的说明,阅读时可参考注释语句仔细分析。

本例的 256K 字节汉字库数据被拆分为两部分,分别存放于 2 片 AT24C1024 存储器中,变量 Offset 为 18 位数据,可寻址整个汉字库空间。由 18 位数据的最高位可判断当前汉字的 32 字节点阵数据处于哪一片存储器,Offset 的低 17 位可寻址一片 AT24C1024 内的所有存储空间。从该器件中读取字节时,需要发送一个控制字节与两个地址字节。其控制字节格式是:

1010—\*—A1—P0—R/W

“\*”号对应的 A2 位不使用,A1 位是硬地址位,2 片 EEPROM 的 A1 引脚分别连接低电平与高电平,18 位 Offset 的最高位对应于 A1 位,用于选择 2 片存储器之一,控制字节之后接着发送的是 2 字节地址(16 位),17 位存储器地址的最高位先于这 2 字节,由最前面的控制字节中的 P0 位携带发送。

掌握上述技术要点后,AT24C1024 的读字节程序就很容易编写了。

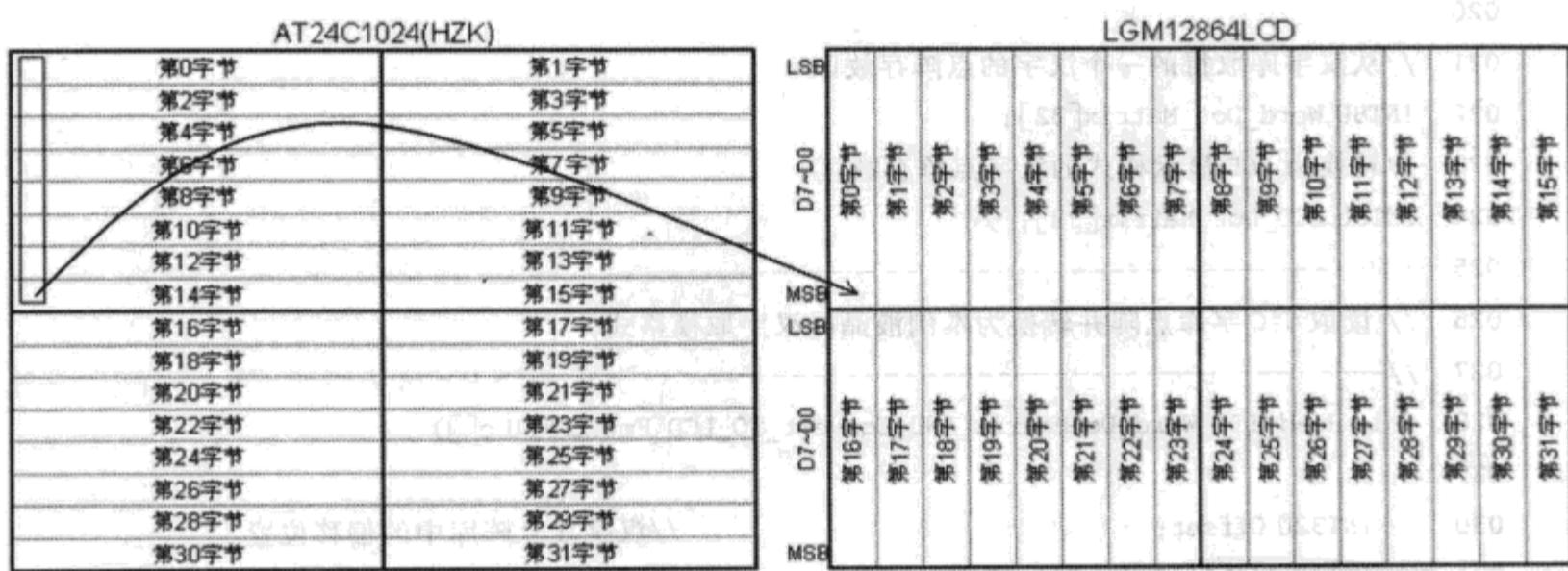


图 5-23 16×16 点阵汉字在 HZK(左)及本例 LCD(右)中的取模格式

## 2. 实训要求

① 本例各汉字的 32 字节点阵数据是通过逐个读取单字节方式获取的,完成本例调试后,进一步阅读 AT24C1024 技术手册中顺序读取多字节的操作时序,改用从指定地址连续读取多字节的方法重新编写本例程序。

② 本例仅实现了全角汉字和全角英文及数字等字符的显示,完成本例调试后进一步改编本例,使之能实现各类中英文全角或半角字符的混合显示。

③ 尝试将案例中的液晶屏改成 32×16 LED 点阵屏,实现任意设定字符串的滚动显示。

### 3. 源程序代码

```

001 //----- main.c -----
002 // 名称: IIC-AT24C1024×2 硬字库应用
003 //-----
004 // 说明: 本例运行时, 液晶屏将显示几行文字, 这些文字的点阵由本例
005 // 程序自动从 2 片 AT24C1024 中读取
006 //
007 //-----
008 #include <avr/io.h>
009 #include <util/delay.h>
010 #include <string.h>
011 #define INT8U unsigned char
012 #define INT16U unsigned int
013 #define INT32U unsigned long
014
015 //12864LCD 相关函数
016 extern void LCD_Initialize();
017 extern void Display_A_WORD_String(INT8U P, INT8U L, INT8U C, INT8U * M);
018 //I2C 相关函数
019 extern void Rec_AT24C1024_Bytes(INT8U Slave, INT32U ROM_Addr, INT8U * Buf, INT8U N);
020
021 //从汉字库取得的一个汉字的点阵存放区
022 INT8U Word_Dot_Matrix[32];
023 //转换为 LCD 显示格式的汉字点阵存放区
024 INT8U LCD_Dot_Matrix[32];
025 //-----
026 // 读取 I2C 字库点阵并转换为本例液晶屏汉字取模格式
027 //-----
028 void Read_IIC_Word_DotMatrix_AND_Convert_TO_LCD_Fmt(INT8U c[])
029 {
030     INT32U Offset; //汉字在点阵库中的偏移位置
031     INT8U SectionCode, PlaceCode; //汉字区码与位码
032     INT8U AT24C1024_A1; //标识 24C1024 芯片编号 0、1
033     INT8U i, j, block; //格式转换变量
034     INT8U Idx[4] = {0, 1, 16, 17}; //4 个板块转换的起始字节索引
035     SectionCode = c[0] - 0xA0; //取得汉字区位码
036     PlaceCode = c[1] - 0xA0;
037
038     //根据区位码计算该汉字点阵字节在字库中的偏移位置
039     //((18 位的 offset 可寻址 256K 点阵字库空间)
040     Offset = (94L * (SectionCode - 1) + (PlaceCode - 1)) * 32L;
041

```

```

042 //根据 Offset 的第 18 位可判断该汉字点阵处在字库前半段(第一片 24C1024)
043 //还是后半段(第二片 24C1024),变量 AT24C1024_A1 为 0/1,分别用于选择
044 //第 1 片或第 2 片 AT24C1024
045 AT24C1024_A1 = Offset >> 17 & 0x00000001;
046 //余下的 17 位 Offset 是第 1 片或第 2 片 AT24C1024 内的偏移量(可寻址 128K 字节空间)
047 Offset &= 0x0001FFFF;
048
049 //从 Offset 开始读取该汉字 32 个字节的点阵数据
050 //AT24C1024 的控制字节格式为:1010 - * - A1 - P0 - R/W(其中 * 是无用的,取值可固定为 0)
051 //变量 AT24C1024_A1 影响控制字节中的 A1 位,即 A1 位 = AT24C1024_A1
052 //它用于选择第 1 片或第 2 片 I2C 存储器
053 //传给函数的参数 Offset 现在是 17 位的,控制字节后的 2 个地址字节携带 16 位后还余下最高位
054 //Rec_AT24C1024_Bytes 函数将根据 offset 的最高位(第 17 位)决定 P0 位的值
055 Rec_AT24C1024_Bytes(0B10100000 | (AT24C1024_A1<<2), Offset, Word_Dot_Matrix, 32);
056
057 //将 16 * 16 点阵分为 4 个 8 * 8 点阵区域进行转换(汉字上半部分与下半部各占两个区域)
058 //下面循环对 4 个块分别进行转换(每块点阵为 8 * 8)
059 for (block = 0; block < 4; block++)
060 {
061     //由 HZK 格式下当前块中的 8 个字节生成 LCD 格式下当前块中的 8 个字节
062     //LCD 格式下的 8 字节来自于 HZK 格式下的 8 列
063     //LCD 格式下的第 i 字节即 HZK 格式下的第 i 列,i = 0~7
064     for (i = 0; i < 8; i++)
065     {
066         //生成 LCD 格式下的第 block 块中第 i 字节时先将其清零
067         LCD_Dot_Matrix[block * 8 + i] = 0x00;
068
069         //转换 HZK 格式下的当前块内第 i 列的 8 位,位扫描变量 j = 0~7
070         //i = 0 时,循环获取 HZK 格式下该块中 8 个字节(j = 0~7)各自的最高位(共 8 位)
071         //i = 1 时,循环获取 HZK 格式下该块中 8 个字节(j = 0~7)各自的次高位(共 8 位)
072         //依此类推...
073         for (j = 0; j < 8; j++)
074         {
075             //HZK 格式下当前块中第 i 列第 j 位为 1 则将其写入
076             //LCD 格式下当前块内第 i 字节第 j 位,依次类推...
077             if ((Word_Dot_Matrix[Idx[block] + 2 * j] & (0x80>>i)) != 0x00)
078                 LCD_Dot_Matrix[block * 8 + i] |= _BV(j);
079         }
080     }
081 }
082 }
083
084 //-----

```

```

085 // 主程序
086 //-----
087 int main()
088 {
089     INT8U i, j;
090     //Poem 中可输入任意文字,注意输入标点符号或英文数字时必须采用全角方式
091     //关于中英文及全/半角混合支持可参考"多汉字点阵屏"案例中的相关代码
092     const char Poem[][][15] =
093     {
094         "采菊东篱下,",
095         "悠然见南山。",
096         "此中有真意,",
097         "欲辨已忘言。"
098     };
099     //配置端口
100     DDRA = 0xFF; PORTA = 0xFF;
101     DDRB = 0xFF; PORTB = 0xFF;
102     //初始化 LCD
103     LCD_Initialize();
104     //共显示 4 行,分别显示在 0,2,4,6 页,每行占 2 页
105     for (i = 0; i < 4; i++)
106     {
107         //显示每行文字
108         for (j = 0; j < strlen(Poem[i]); j += 2)
109         {
110             //从每行第 j 个字节,每次跨度为 2 字节(1 个汉字),
111             //取得汉字点阵并转换为本例液晶格式
112             Read_IIC_Word_DotMatrix_AND_Convert_TO_LCD_Fmt((INT8U * )(Poem[i] + j));
113
114             //从第 i 页开始,左边距 19,每次显示一个汉字
115             Display_A_WORD_String(i * 2 ,j/2 * 16 + 19 , 1, LCD_Dot_Matrix);
116         }
117     }
118     while (1);
119 }

01 //----- AT24C1024.c -----
02 // 名称:用 TWI 接口读/写 AT24C1024 子程序
03 //-----
04 #include <avr/io.h>
05 #include <util/delay.h>
06 #include <util/TWI.h>
07 #define INT8U unsigned char

```

```

08 #define INT16U unsigned int
09 #define INT32U unsigned long
10
11 //TWI 通用操作
12 #define Wait() while ((TWCR & _BV(TWINT)) == 0)
13 #define START() {TWCR = _BV(TWINT) | _BV(TWSTA) | _BV(TWEN); Wait();}
14 #define STOP() {TWCR = _BV(TWINT) | _BV(TWSTO) | _BV(TWEN)}
15 #define TWI() {TWCR = _BV(TWINT) | _BV(TWEN); Wait();}
16 #define WriteByte(x) {TWDR = (x); TWCR = _BV(TWINT) | _BV(TWEN); Wait();}
17 #define ACK() {TWCR |= _BV(TWEA)}
18 #define NACK() {TWCR &= ~_BV(TWEA)}
19 //-----
20 // 从 AT24C1024 读 1 字节
21 //-----
22 INT8U Read_A_Byt(INT8U Slave, INT32U ROM_Addr)
23 {
24     INT8U page, dat; INT16U addr16;
25     //AT24C1024 控制字节格式:1010 - * - A1 - P0 - R/W
26     //一片 AT24C1024 的存储空间为 128K 字节,需要 17 位地址进行寻址
27     //地址的最高位对应于控制字节(或称器件地址字节)中的 P0 位
28     page = (INT8U)((ROM_Addr>>16) & 0x00000001) << 1;
29     //在控制字节后是 16 位的字地址
30     addr16 = (INT16U)(ROM_Addr & 0x0000FFFF);
31     START();
32     if (TW_STATUS != TW_START) return 0;
33     //发送器件地址及页地址
34     WriteByte(Slave | page);
35     if (TW_STATUS != TW_MT_SLA_ACK) return 0;
36     //下面再发送余下的 16 位地址
37     //其中 17 位地址的最高位已由控制字节中的 P0 位携带
38     WriteByte((INT8U)(addr16 >> 8)); //先发高 8 位
39     if (TW_STATUS != TW_MT_DATA_ACK) return 0;
40     WriteByte((INT8U)(addr16 & 0x00FF)); //再发低 8 位
41     if (TW_STATUS != TW_MT_DATA_ACK) return 0;
42     START();
43     if (TW_STATUS != TW REP_START) return 0;
44     //器件地址(读)
45     WriteByte(Slave | 0x01 | page);
46     if (TW_STATUS != TW_MR_SLA_ACK) return 0;
47     //启动主 I2C 读方式
48     TWI();
49     if (TW_STATUS != TW_MR_DATA_NACK) return 0;
50     dat = TWDR;

```

```

51     STOP();
52     return dat;
53 }
54
55 //-----
56 // 从 AT24C1024 接收多字节
57 //-----
58 void Rec_AT24C1024_Bytes(INT8U Slave, INT32U ROM_Addr, INT8U * Buf, INT8U N)
59 {
60     INT8U i;
61     for ( i = 0; i < N; i ++ )
62     {
63         Buf[ i ] = Read_A_Byte(Slave, ROM_Addr + i );
64     }
65 }

```

## 5.23 SPI—AT25F2048 硬件字库应用

兼容 SPI 接口的 AT25F2048 具有 256K 字节内存,本例将  $16 \times 16$  点阵汉字库文件 HZK 存放于该 EEPROM 中, HZK 文件共有 262K 字节,本例截除了多出的 6K 数据。案例电路如图 5-24 所示,所实现的运行效果与上一案例相同。

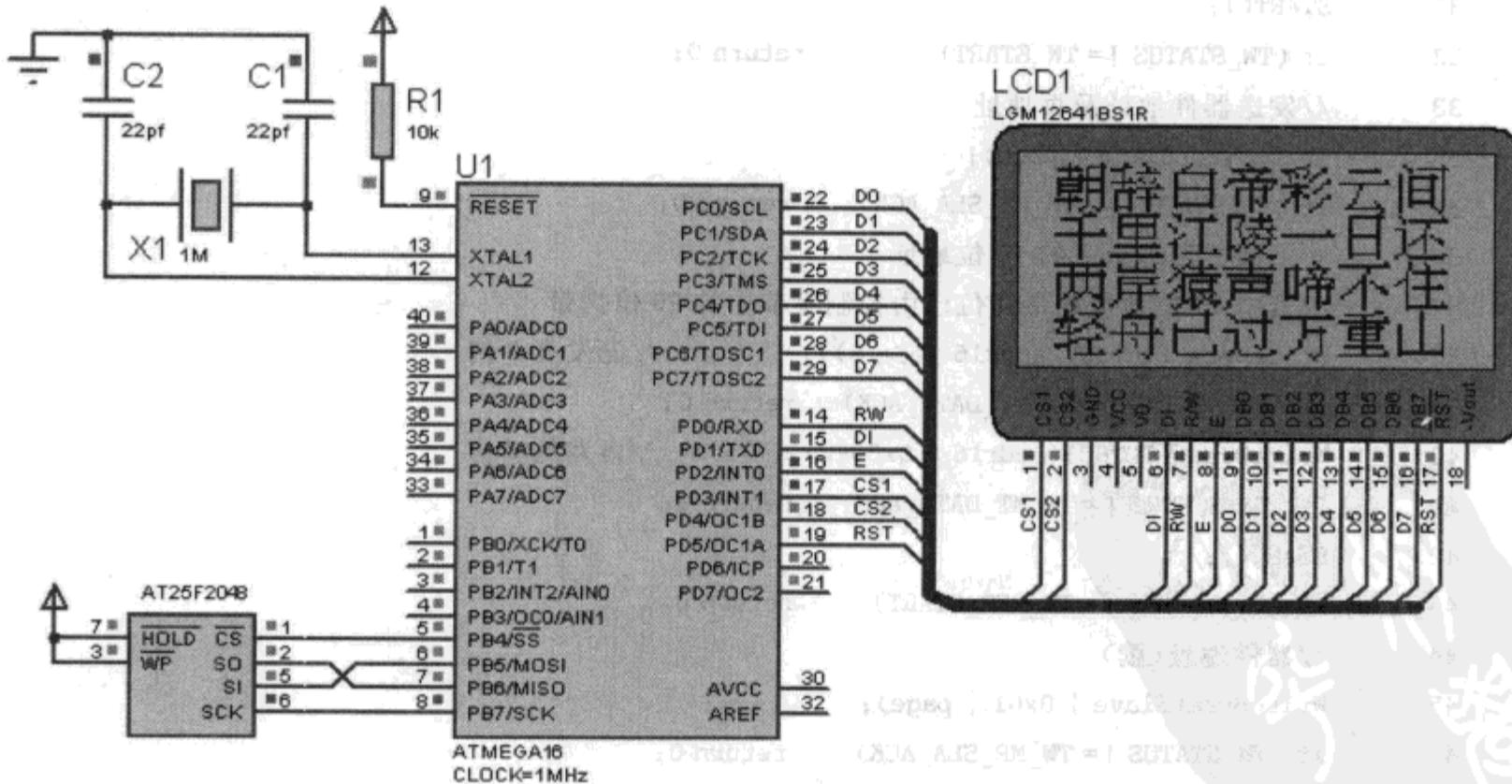


图 5-24 SPI—AT25F2048 硬件字库应用

## 1. 程序设计与调试

本例运行效果与上一案例完全相同,不同之处仅在于使用了容量较大的兼容 SPI 接口的 AT25F2048 存储器(256K 字节),由于空间扩展了一倍,本例不需要再将字库分割为两部分,分别由两片 EEPROM 保存。

根据 AT25F2048 技术手册文件可知,在读取字节之前首先需要发送读字节命令 READ(0x03),然后发送 3 字节地址 addr(INT32U 类型),32 位的长整型地址变量 addr 实际有效位为 18 位,可寻址 256K EEPROM 内存空间。发送 3 字节地址的语句如下:

```
SPI_Transmit((INT8U)(addr >> 16));
SPI_Transmit((INT8U)(addr >> 8));
SPI_Transmit((INT8U)(addr));
```

在发送 3 个字节地址 addr 以后即可通过连续发送 0xFF 来连续接收 32 个字节的点阵数据:

```
for( i = 0; i < len; i++ ) p[i] = SPI_Transmit(0xFF);
```

与上一案例一样,本例中的偏移变量 Offset 也是 18 位的,通过区位码获取汉字点阵在字库中的起始位置 Offset,从 SPI 接口存储器中的 Offset 位置即可读取 32 字节汉字点阵数据:

```
Read_Some_Bytes_FROM_AT25F1024A(Offset,Word_Dot_Matrix,32)
```

由于本例 LCD 与上一案例相同,因而本例的汉字点阵数据格式转换算法也与上一案例相同,转换函数中余下的代码与上一案例中的对应代码完全相同。

## 2. 实训要求

- ① 改用取模方式不同于本例的其他液晶屏,重新设计程序,仍实现相同的运行效果。
- ② 改用 AT25F4096 存储器(512K 字节),同时保存中英文字库文件,实现全角与半角字符的混合显示。
- ③ 在字库中最末尾的部分空白区域设计保存若干特殊字符点阵数据,例如扬声器点阵、时钟面板点阵、各种气象标志点阵等,编程实现对这些特殊字符点阵的读取、转换与液晶显示。

## 3. 源程序代码

```
01 //----- main.c -----
02 // 名称: SPI - AT25F2048 硬件字库应用
03 //-----
04 // 说明: 本例与上一案例的差别在于字库存放在兼容 SPI 接口的 AT25F2048
05 //       存储器中
06 //
07 //-----
08 #include <avr/io.h>
09 #include <string.h>
10 #define INT8U unsigned char
11 #define INT16U unsigned int
12 #define INT32U unsigned long
```

```

013
014 //12864LCD 相关函数
015 extern void LCD_Initialize();
016 extern void Display_A_WORD_String(INT8U P, INT8U L, INT8U C, INT8U * M);
017 //SPI 相关函数
018 extern void SPI_MasterInit();
019 extern void Read_Some_Bytes_FROM_AT25F1024A(INT32U addr, INT8U * p, INT16U len);
020
021 //从汉字库取得的一个汉字的点阵存放区
022 INT8U Word_Dot_Matrix[32];
023 //转换为 LCD 显示格式的汉字点阵存放区
024 INT8U LCD_Dot_Matrix[32];
025 //-----
026 // 读取 SPI 字库汉字点阵并将字库点阵格式转换为本例液晶屏汉字取模格式
027 //-----
028 void READ_SPI_Word_DotMatrix_AND_Convert_TO_LCD_Fmt(INT8U c[])
029 {
030     INT32U Offset;                                //汉字在点阵库中的偏移位置
031     INT8U SectionCode, PlaceCode;                //汉字区码与位码
032     INT8U i,j,block;                            //格式转换变量
033     INT8U Idx[4] = {0,1,16,17};                  //4 个板块转换的起始字节索引
034     SectionCode = c[0] - 0xA0;                   //取得汉字区位码
035     PlaceCode = c[1] - 0xA0;
036
037     //根据区位码计算该汉字在字库中的偏移位置
038     Offset = (94L * (SectionCode - 1) + (PlaceCode - 1)) * 32L;
039     //从 SPI 接口存储器 AT25F2048 中 Offset 位置读取该汉字点阵
040     Read_Some_Bytes_FROM_AT25F1024A(Offset,Word_Dot_Matrix,32);
041
042     //将 16 * 16 点阵分为 4 个 8 * 8 点阵区域进行转换(汉字上半部分与下半部各占两个区域)
043     //下面循环对 4 个块分别进行转换(每块点阵为 8x8)
044     for (block = 0; block < 4; block++)
045     {
046         //由 HZK 格式下当前块中的 8 个字节生成 LCD 格式下当前块中的 8 个字节
047         //LCD 格式下的 8 字节来自于 HZK 格式下的 8 列
048         //LCD 格式下的第 i 字节即 HZK 格式下的第 i 列,i=0~7
049         for (i = 0; i < 8; i++)
050         {
051             //生成 LCD 格式下的第 block 块中第 i 字节时先清将其 0
052             LCD_Dot_Matrix[block * 8 + i] = 0x00;
053
054             //转换 HZK 格式下的当前块内第 i 列的 8 位,位扫描变量 j = 0~7
055             //i = 0 时,循环获取 HZK 格式下该块中 8 个字节(j = 0~7)各自的最高位(共 8 位)

```

```

056         //i = 1 时,循环获取 HZK 格式下该块中 8 个字节(j = 0~7)各自的次高位(共 8 位)
057         //依此类推...
058         for( j = 0; j < 8; j++ )
059         {
060             //HZK 格式下当前块中第 i 列第 j 位为 1 则将其写入
061             //LCD 格式下当前块内第 i 字节第 j 位,以此类推
062             if ((Word_Dot_Matrix[Idx[block] + 2 * j] & (0x80>>i)) != 0x00)
063                 LCD_Dot_Matrix[block * 8 + i] |= _BV(j);
064         }
065     }
066 }
067 }
068
069 //-----
070 // 主程序
071 //-----
072 int main()
073 {
074     INT8U i,j;
075     //Poem 中可输入任意文字,注意输入标点符号或英文数字时必须采用全角方式
076     //关于中英文及全/半角混合支持可参考“多汉字点阵屏”案例中的相关代码
077     const char Poem[][][16] =
078     {
079         "朝辞白帝彩云间",
080         "千里江陵一日还",
081         "两岸猿声啼不住",
082         "轻舟已过万重山"
083     };
084     //配置端口
085     DDRC = 0xFF; PORTC = 0xFF;
086     DDRD = 0xFF; PORTD = 0xFF;
087     //SPI 主机初始化
088     SPI_MasterInit();
089     //初始化 LCD
090     LCD_Initialize();
091     //共显示 4 行,分别显示在 0、2、4、6 页,每行占 2 页
092     for (i = 0; i < 4; i++)
093     {
094         //显示每行文字
095         for (j = 0; j < strlen(Poem[i]); j += 2)
096         {
097             //从每行第 j 个字节,每次跨度为 2 字节(1 个汉字)
098             //取得汉字点阵并转换为本例液晶格式

```



```
099         READ_SPI_Word_DotMatrix_AND_Convert_TO_LCD_Fmt((INT8U*)(Poem[i] + j));
100
101     //从第 i 页开始,左边距 7,每次显示一个汉字
102     Display_A_WORD_String(i * 2, j / 2 * 16 + 7, 1, LCD_Dot_Matrix);
103 }
104 }
105 while (1);
106 }

001 //----- AT25F2048.c -----
002 // 名称:用 SPI 接口读/写 AT25F2048 子程序
003 //-----
004 #include <avr/io.h>
005 #include <util/delay.h>
006 #define INT8U unsigned char
007 #define INT16U unsigned int
008 #define INT32U unsigned long
009
010 // AT25F2048 指令集
011 #define WREN 0x06 //使能写
012 #define WRDI 0x04 //禁止写
013 #define RDSR 0x05 //读状态
014 #define WRSR 0x01 //写状态
015 #define READ 0x03 //读字节
016 #define PROGRAM 0x02 //写字节
017 #define SECTOR_ERASE 0x52 //删除区域数据
018 #define CHIP_ERASE 0x62 //删除芯片数据
019 #define RDID 0x15 //读厂商与产品 ID
020 //SPI 使能与禁用
021 #define SPI_EN() (PORTB &= 0xEF)
022 #define SPI_DI() (PORTB |= 0x10)
023 //-----
024 // SPI 主机初始化
025 //-----
026 void SPI_MasterInit()
027 {
028     //设置 SS、MOSI、SCK 为输出,MISO 为输入
029     DDRB = 0B10110000; PORTB = 0xFF;
030     //SPI 使能,主机模式,16 分频
031     SPCR |= _BV(SPE) | _BV(MSTR) | _BV(SPR0);
032 }
033
034 //-----
```

```

035 // SPI 数据传输
036 //-----
037 INT8U SPI_Transmit(INT8U dat)
038 {
039     SPDR = dat;                                //启动数据传输
040     while(!(SPSR & _BV(SPIF)));             //等待结束
041     SPSR |= _BV(SPIF);                      //清中断标志
042     return SPDR;
043 }
044
045 //-----
046 // 读 AT25F2048 芯片状态
047 //-----
048 INT8U Read_SPI_Status()
049 {
050     INT8U status;
051     SPI_EN();
052     SPI_Transmit(RDSR);                      //发送读状态指令
053     status = SPI_Transmit(0xFF);
054     SPI_DI();
055     return status;
056 }
057
058 //-----
059 // AT25F2048 忙等待
060 //-----
061 void Busy_Wait()
062 {
063     while(Read_SPI_Status() & 0x01);          //忙等待
064 }
065
066 /*
067 //-----
068 // 删除 AT25F2048 芯片未加保护的所有区域数据
069 //-----
070 void ChipErase()
071 {
072     SPI_EN();
073     SPI_Transmit(WREN);                      //使能写
074     SPI_DI();
075     Busy_Wait();
076     SPI_EN();
077     SPI_Transmit(CHIP_ERASE);                //清除芯片数据指令

```



```
078     SPI_DI();
079     Busy_Wait();
080 }
081 */
082
083 //-----
084 // 向 AT25F2048 写入 3 个字节的地址 0x000000 ~ 0x01FFFF
085 //-----
086 void Write_3_Bytes_AT25F2048_Address(INT32U addr)
087 {
088     SPI_Transmit((INT8U)(addr >> 16 & 0xFF));
089     SPI_Transmit((INT8U)(addr >> 8  & 0xFF));
090     SPI_Transmit((INT8U)(addr & 0xFF));
091 }
092
093 //-----
094 // 从指定地址读单字节
095 //-----
096 INT8U Read_BytE_FROM_AT25F2048(INT32U addr)
097 {
098     INT8U dat;
099     SPI_EN();
100    SPI_Transmit(READ);           //发送读指令
101    Write_3_Bytes_AT25F2048_Address(addr); //发送 3 字节地址
102    dat = SPI_Transmit(0xFF);        //读取字节数据
103    SPI_DI();
104    return dat;
105 }
106
107 //-----
108 // 从指定地址读多字节到缓冲
109 //-----
110 void Read_Some_Bytes_FROM_AT25F2048(INT32U addr, INT8U * p, INT16U len)
111 {
112     INT16U i;
113     SPI_EN();
114     SPI_Transmit(READ);           //发送读指令
115     Write_3_Bytes_AT25F2048_Address(addr); //发送 3 字节地址
116     for( i = 0; i < len; i ++ )      //读数据序列
117         p[i] = SPI_Transmit(0xFF);
118     SPI_DI();
119 }
120
```

```

121 //-----
122 // 向 AT25F2048 指定地址写入单字节数据
123 //-----
124 void Write_Byte_TO_AT25F2048(INT32U addr, INT8U dat)
125 {
126     SPI_EN();
127     SPI_Transmit(WREN);           //使能写
128     SPI_DI();
129     Busy_Wait();
130     SPI_EN();
131     SPI_Transmit(PROGRAM);      //写指令
132     Write_3_Bytes_AT25F2048_Address(addr); //发送 3 字节地址
133     SPI_Transmit(dat);          //写字节数据
134     SPI_DI();
135     Busy_Wait();
136 }
137 /*
138 //-----
139 // 向 AT25F2048 指定地址开始写入多字节数据
140 //-----
141 void Write_Some_Bytes_TO_AT25F2048(INT32U addr, INT8U * p, INT16U len)
142 {
143     INT16U i;
144     SPI_EN();
145     SPI_Transmit(WREN);           //使能写
146     SPI_DI();
147     Busy_Wait();
148     SPI_EN();
149     SPI_Transmit(PROGRAM);      //写指令
150     Write_3_Bytes_AT25F2048_Address(addr); //发送 3 字节地址
151     for (i = 0; i < len; i++)        //写数据序列
152         SPI_Transmit(p[i]);
153     SPI_DI();
154     Busy_Wait();
155 }
156 */

```

## 5.24 带液晶显示的红外遥控调速仿真

本例通过红外遥控器调节受控端的电机转速,PG160128 液晶屏用于显示当前相对转速。本例电路及程序运行效果如图 5-25 所示。

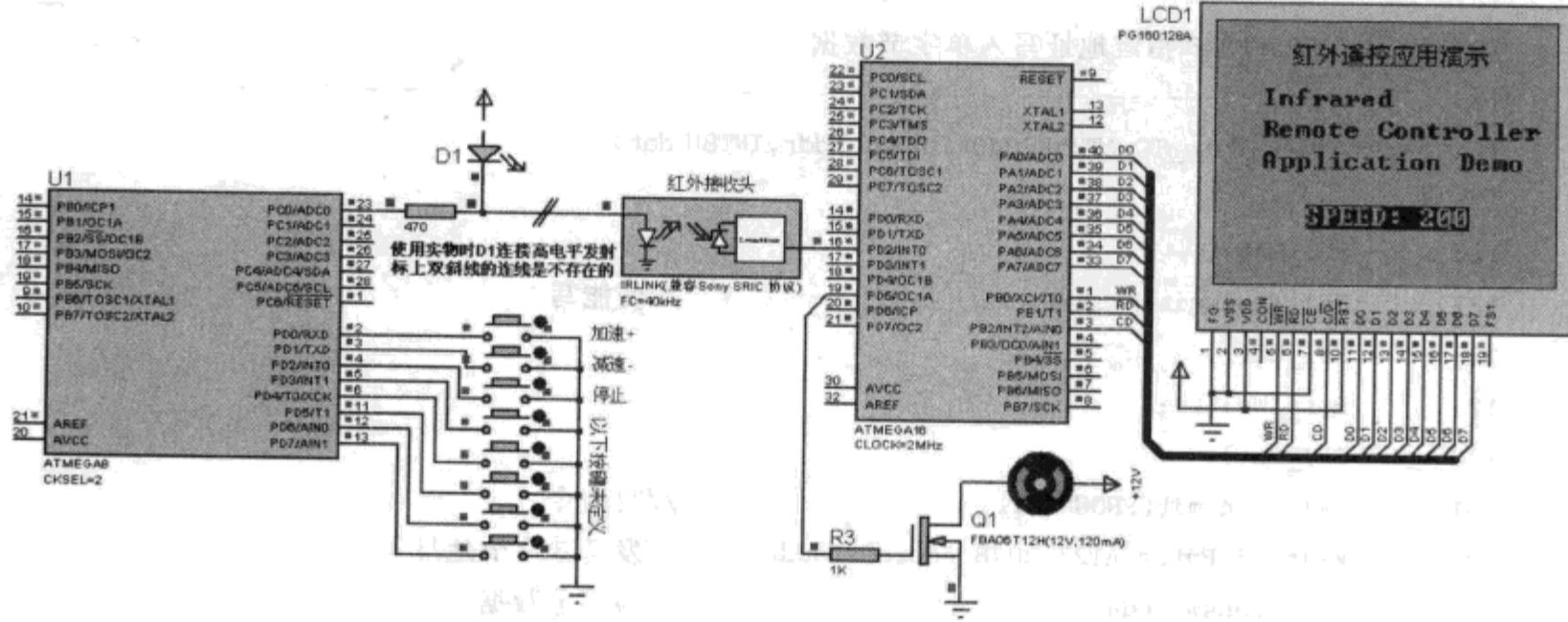


图 5-25 带液晶显示的红外遥控调速仿真

## 1. 程序设计与调试

本例整合了红外遥控收发、液晶显示、PWM 电机调速 3 项功能，受控端程序根据遥控器的“+”/“-”按键分别增加或减少输出比较寄存器 OCR1A 的值，从而改变 PD5(OC1A)引脚输出信号的占空比，实现对电机转速的控制，液晶屏同时显示出当前所调转速的相对值。

阅读调试本例时，可参考第 3 章与第 4 章中的相关案例进行分析。有此前章节的相关案例作基础，本例的设计与调试就显得比较容易了。

## 2. 实训要求

① 修改受控端程序，仿照遥控调节 TV 音量的显示效果，使液晶屏能以“进程条”方式显示当前转速大小。

② 进一步修改受控端程序，以遥控菜单的方式上下选择某项功能，按下 OK 键时则执行相应功能。

③ 改用 Nokia N17 红外遥控协议重新设计本例的发送与接收程序，实现所设定的遥控功能。

## 3. 源程序代码

```

001 //----- 红外遥控器受控端程序.c -----
002 // 名称：红外遥控器受控端程序
003 //-
004 // 说明：程序运行时，根据 SONY 红外协议接收数据并解码，然后根据接收到
005 //          的不同编码完成不同的操作
006 //
007 //-
008 #define F_CPU 2000000UL
009 #include <avr/io.h>
010 #include <avr/interrupt.h>
011 #include <util/delay.h>

```

```

012 #include <stdio.h>
013 #define INT8U unsigned char
014 #define INT16U unsigned int
015
016 //端口设备操作定义
017 #define LED1_ONOFF() PORTB ^= _BV(PB7)           //LED1
018 #define LED2_ONOFF() PORTB ^= _BV(PB4)           //LED2
019 #define MOTOR_SP()  PORTA ^= _BV(PA0)            //电机
020 //读取红外输入信号
021 #define Read_IR()   (PIND & _BV(PD2))
022 //当前速度值(初始值设为 200)
023 volatile INT16U Current_Speed = 200;
024 //当前接收到的 12 位红外编码
025 volatile INT16U IR_D12 = 0x0000;
026
027 //液晶屏相关函数
028 extern void Clear_Screen();
029 extern char LCD_Initialise();
030 extern void Display_Str_at_xy(INT8U x, INT8U y, char * Buffer, INT8U wb);
031 //当前速度显示缓冲
032 char Speed_Displ_Buff[10];
033 //-----
034 // PWM 调速并显示
035 //-----
036 void PWM_speed_and_show()
037 {
038     TCCR1A = 0x83; OCR1A = Current_Speed;
039     sprintf(Speed_Displ_Buff, "SPEED: % 4d", Current_Speed);
040     Display_Str_at_xy(41, 90, Speed_Displ_Buff, 1);
041 }
042
043 //-----
044 // 主程序
045 //-----
046 int main()
047 {
048     DDRD = 0xFF & ~_BV(PD2);                  //配置端口
049     DDRA = 0xFF;
050     DDRB = 0xFF;
051     LCD_Initialise();                      //LCD 初始化
052     Clear_Screen();                        //清屏
053     Display_Str_at_xy(34, 8, "红外遥控应用演示", 0);
054     Display_Str_at_xy(20, 30, "Infrared", 0);

```



```
055     Display_Str_at_xy(20,46, "Remote Controller",0);
056     Display_Str_at_xy(20,62, "Application Demo",0);
057
058     TCCR1A = 0x83;                      //10 位 PWM(1023), 正向 PWM
059     TCCR1B = 0x02;                      //时钟 8 分频, PWM 频率: F_CPU/8/2046
060     PWM_speed_and_show();             //PWM 调整并显示
061     MCUCR = 0x02;                      //INT0 为下降沿触发
062     GICR |= _BV(INT0);                //INT0 中断使能
063     sei();                            //使能总中断
064     while(1);
065 }
066
067 //-----
068 // INT0 中断函数 (通过实测, 以 122,242 为两个时长的上限)
069 //-----
070 ISR (INT0_vect)
071 {
072     INT8U i;
073     INT16U IR_us = 0;                  //红外载波时长
074     GICR &= ~_BV(INT0);              //禁止外部中断
075     _delay_ms(2);                   //红外信号引导部分共长 2.4 ms
076     if (Read_IR() != 0x00) goto end; //如果 2 ms 后已经变为高则退出
077
078     while (Read_IR() == 0x00)
079     {
080         _delay_us(1);
081         if (++IR_us > 2400) goto end; //异常时退出
082     }
083     //收集 12 位数据
084     for (i = 0; i < 12; i++)
085     {
086         //等待 IR 变为低电平, 跳过 600 μs 空白区
087         while (Read_IR() != 0x00)
088         {
089             _delay_us(1);
090             if (++IR_us > 600) goto end; //异常时退出
091         }
092         //计算低电平时长
093         IR_us = 0;
094         while (Read_IR() == 0x00)
095         {
096             _delay_us(1);
097             if (++IR_us > 300) goto end; //超过该值时异常退出

```

```

098     }
099     //12位红外数据的高位默认补0
100     IR_D12 >>= 1;
101     //如果时长为1200则在高位补1
102     //通过对本代码检测,两者计时上限分别为:122,242,
103     //故这里选择150为0/1的分界值
104     if (IR_us > 150) IR_D12 |= 0x0800;
105 }
106
107     //根据12位的红外遥控信号完成不同操作
108     switch (IR_D12)
109     {
110         case 0x0771: if (Current_Speed <= 920) //加速
111             Current_Speed += 100;
112             PWM_speed_and_show();
113             break;
114         case 0x0334: if (Current_Speed >= 120) //减速
115             Current_Speed -= 100;
116             PWM_speed_and_show();
117             break;
118         case 0x0556: Current_Speed = 200;           //停止时还原为200
119             TCCR1A = 0x00;                         //电机停止
120             break;
121         case 0x0778: break;                      //以下操作未定义
122         case 0x09AA: break;
123         case 0x0BCC: break;
124         case 0x0DEE: break;
125         case 0x0F00: break;
126     }
127     //重新允许INT0中断
128     end: GICR |= _BV(INT0);
129 }

```

```

01 //----- 红外遥控仿真发射器.c -----
02 // 名称: 红外遥控仿真发射器
03 //
04 // 说明: 本例运行时,按键键值以40 kHz红外线载波发射出去,所模拟的载波
05 // 数据格式符合索尼红外遥控编码格式
06 //
07 //
08 #define F_CPU 2000000UL
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 //按键定义
16 # define K1_DOWN() (PIND & _BV(PD0)) == 0x00
17 # define K2_DOWN() (PIND & _BV(PD1)) == 0x00
18 # define K3_DOWN() (PIND & _BV(PD2)) == 0x00
19 # define K4_DOWN() (PIND & _BV(PD3)) == 0x00
20 # define K5_DOWN() (PIND & _BV(PD4)) == 0x00
21 # define K6_DOWN() (PIND & _BV(PD5)) == 0x00
22 # define K7_DOWN() (PIND & _BV(PD6)) == 0x00
23 # define K8_DOWN() (PIND & _BV(PD7)) == 0x00
24
25 //红外发射管操作定义
26 # define IRLED_BLINK() PORTC ^= _BV(PC0)
27 # define IRLED_1() PORTC |= _BV(PC0)
28 # define IRLED_0() PORTC &= ~_BV(PC0)
29 //-----
30 // 发送 N 倍的 600 μs 载波(1/40K/2≈12 μs)
31 //-----
32 void Emit_IR_Wave_Nx600us(INT8U N)
33 {
34     INT8U i;
35     for (i = 0; i < N * 50; i++)
36     {
37         _delay_us(12); IRLED_BLINK();
38     }
39 }
40
41 //-----
42 // 发送 12 位数据
43 //-----
44 void Emit_D12(INT16U D12)
45 {
46     INT16U i;
47     //首先发送引导部分 2.4 ms 的 40 kHz 载波
48     Emit_IR_Wave_Nx600us(4);
49     IRLED_0(); _delay_us(600);
50     //共发送 12 位的命令与数据码(7 + 5)
51     for (i = 0x0001; i < 0x1000; i <<= 1)
52     {
53         //从低位开始,每遇到 1/0 时分别输出 1.2 ms/600 μs 载波
```

```

54     if ( D12 & i)
55         Emit_IR_Wave_Nx600us(2);
56     else
57         Emit_IR_Wave_Nx600us(1);
58     //输出 600 μs 的低电平
59     IRLED_0(); _delay_us(600);
60 }
61 }
62
63 //-----
64 // 主程序
65 //-----
66 int main()
67 {
68     DDRC = 0xFF;                                //配置端口
69     DDRD = 0x00; PORTD = 0xFF;
70     while(1)
71     {
72         if      (K1_DOWN()) Emit_D12(0x0771);
73         else if (K2_DOWN()) Emit_D12(0x0334);
74         else if (K3_DOWN()) Emit_D12(0x0556);
75         else if (K4_DOWN()) Emit_D12(0x0778);
76         else if (K5_DOWN()) Emit_D12(0x09AA);
77         else if (K6_DOWN()) Emit_D12(0x0BCC);
78         else if (K7_DOWN()) Emit_D12(0x0DEE);
79         else if (K8_DOWN()) Emit_D12(0x0F00);
80         _delay_ms(10);
81     }
82 }

```

## 5.25 能接收串口信息的带中英文硬字库的 80×16 点阵显示屏

本例整合了 74HC595+74HC154 设计的 LED 点阵屏、SPI 接口硬字库及串口接收 3 项功能。运行时 LED 屏首先滚动显示“★点阵演示 V1.0★...”，该字符串同时包含有全角与半角字符，所显示的点阵数据来自于含有中英文字库的 SPI 接口存储器 AT25F4096。另外，在案例运行过程中，按规定格式在串口助手软件中输入的汉字或半角英文字符可以直接发送到 LED 点阵屏滚动显示。本例电路及程序运行效果如图 5-26 所示。

### 1. 程序设计与调试

本例 EEPROM 保存了 16×16 点阵的中文字库及 8×16 点阵的英文字库，该合并字库文件由中文字库文件 HZK 与英文字库文件 ASC 构成。下面介绍 2 种合并方法。

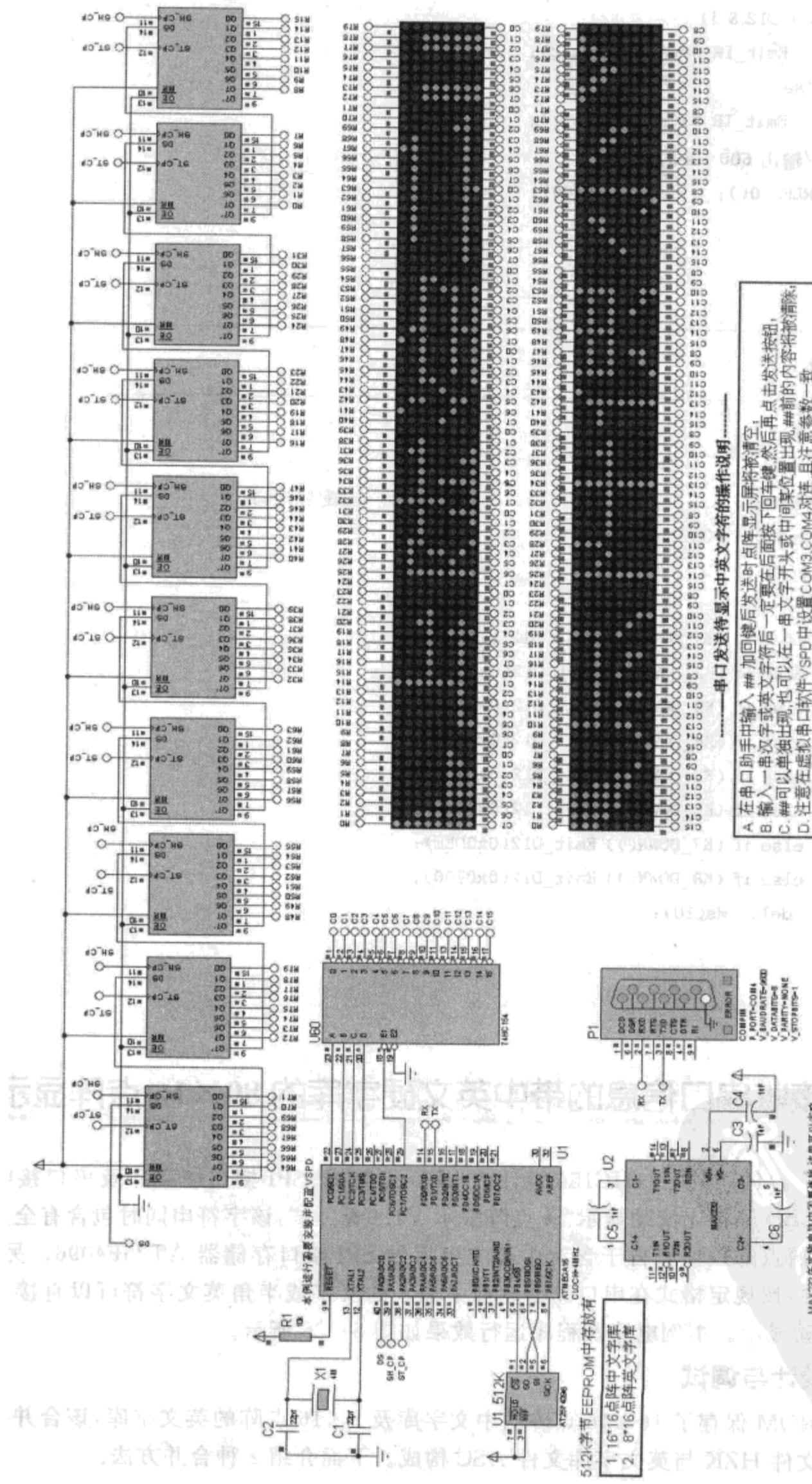


图5-26 能接收串口信息的带中英文硬字库的80×16点阵显示屏

### (1) 使用案例压缩包中提供的“文件拆分与合并器.exe”

假设要将两个文件合并为 HZK\_ASC.bin 文件,在运行该软件之前要先将 HZK 改名为 HZK\_ASC.3h0,再将 ASC 改名为 HZK\_ASC.3h1,然后运行该软件,单击“Join”或“合并”选项卡,在“File to join”或“打开要合并的文件”框中选择 HZK\_ASC.3h0,这时生成的文件将自动命名为 HZK\_ASC,该文件名可手动添加后缀“.bin”,单击“开始合并”按钮后,后缀为 3h0 与 3h1 的文件将自动被合并到 HZK\_ASC.bin 中(例如还有同名,但后缀为 3h2 的文件,那么该文件也会被合并)。

### (2) 使用 DOS 命令“copy”合并 2 个文件

为避免在 DOS 命令行状态下出现过长的路径名,可首先将原始文件拷贝到 C 盘根下某一名称简短的文件夹中(例如 C:\my\_HA),然后单击 Windows 菜单中的“开始”→“运行”,在命令框中输入“cmd”进入 DOS 命令窗口,在该窗口中依次输入如下命令:

```
C:  
CD\my_HA  
copy /b HZK + ASC HZK_ASC.bin
```

第 1 行命令将当前盘符设为 C 盘,如果当前已经处于 C 盘某文件夹中,则该行命令可以省略,第 2 行命令用于进入 my\_HA 文件夹,第 3 行命令将二进制文件 HZK 与 ASC 合并复制到 HZK\_ASC.bin 文件中,其中的/b 参数不可省略。

合后的 HZK\_ASC.bin 文件中,0~267615 字节(0x00000000~0x0004155F)是汉字及全角字符点阵字节,每个字符占 32 字节,从 267616(0x00041560)开始是半角英文字符点阵字节,每个字符占 16 字节,图 5-27 中从 0x00041560 开始的灰色部分就是半角字符点阵数据。

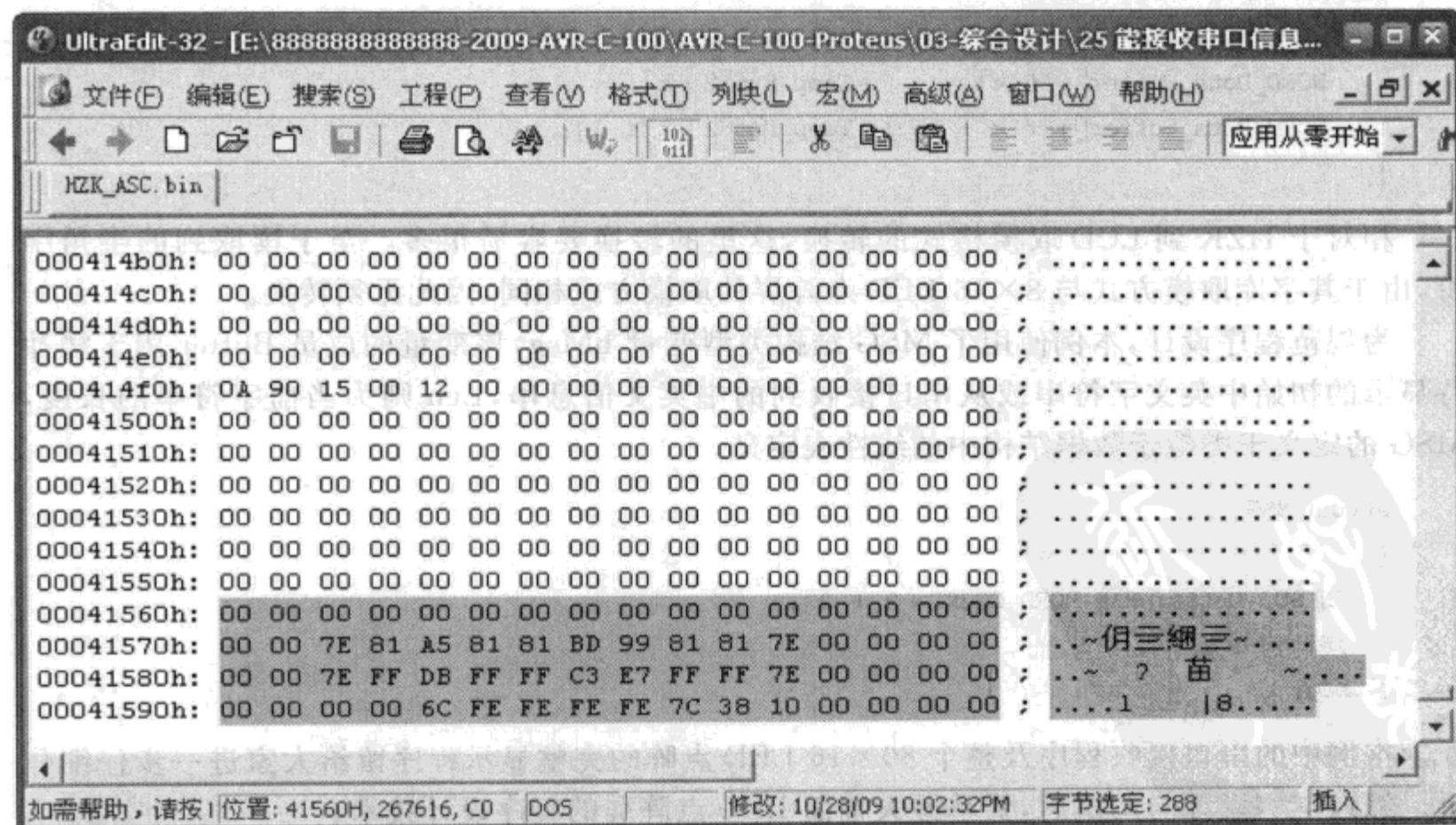


图 5-27 由 HZK 与 ASC 合并成的中英文点阵字库



对于合并后的点阵字库,全角与半角字符点阵在字库的偏移位置计算公式分别如下:

全角:Offset = (94L \* (SectionCode - 1) + (PlaceCode - 1)) \* 32L

半角:Offset = 267616L + bMsg.Buffer[i] \* 16

前者通过区位码可计算出偏移位置,后者则通过 ASCII 编码乘以 16 再加上 267616L 计算偏移位置,公式中的 bMsg.Buffer[i] 是第 i 个半角字符的 ASCII 码。

此前相关案例中进行过 HZK 点阵格式到 LCD 点阵格式的转换。类似地,在本例电路中,对于含有 32 个字节点阵的全角字符,在发送 LED 屏显示时也需要将 HZK\_ASC 点阵格式转换为 LED 屏点阵格式。

以第一块  $16 \times 16$  点阵 LED 屏左边的两片  $8 \times 8$ LED 屏为例,R0~R7 连接两块屏的上端引脚,C0~C7 连接第一块屏下端的引脚,C8~C15 连接第二块屏下端的引脚,扫描显示这块  $8 \times 16$  的点阵区域时,16 行点阵字节(每行 8 个点)逐一发送给 R0~R7,每发送一个点阵字节的同时,4-16 译码器选通 C0~C15 其中之一所对应的一个共阴行(注意不是一列,虽然这里使用了符号 C),可见这块  $8 \times 16$  的点阵区域是逐行显示的,而且行的扫描是由上到下,每 16 次扫描完成一次  $8 \times 16$  点阵的刷新,即一块  $16 \times 16$  点阵左半部分的刷新。

再观察该  $16 \times 16$  点阵区域的右半边,它同样是 2 片  $8 \times 8$ LED 屏构成的  $8 \times 16$  点阵区域,由上至下 16 行的选通仍由 C0~C15 完成,每行的点阵则由 R8~R15 输入。

综合观察以上扫描刷新过程可知,一块  $16 \times 16$  的点阵区域是分成左右 2 块  $8 \times 16$  的区域同时扫描显示完成的,为适应本例 LED 屏以  $8 \times 16$  的点阵区域为最小刷新显示单位的设计布局,转换函数将 HZK 中点阵格式为 16 行依次左→右、左→右取模的方式转换为先取左半边 16 行,再右半边 16 行,下面的代码片段完成了这项转换:

```
for (k = 0; k < 16; k++)
{
    WORD_Dots_Buffer[j + k]      = Temp_Buf[2 * k];
    WORD_Dots_Buffer[j + k + 16] = Temp_Buf[2 * k + 1];
}
```

相对于 HZK 到 LCD 取模格式的转换,这里的转换要容易很多。至于读取到的半角字符,由于其字库取模方式与  $8 \times 16$  LED 点阵屏的取模方式相同,因此无须转换。

为规范程序设计,本例使用了 MSG 结构类型变量 bMsg,该变量的成员 Buffer 用于缓冲待显示的初始中英文字符串或从串口接收到的中英文信息串,Len 则为当前字符串的长度,MSG 的定义于类似于数据结构中的线性表定义。

```
struct MSG
{
    INT8U Buffer[MAX_WORD_COUNT * 2 + 2];
    INT16U Len;
} bMsg;
```

本例中的串口接收程序及整个  $80 \times 16$  LED 点阵的完整显示程序留给大家进一步仔细分析。经过本例设计与调试后,要掌握大幅面 LED 点阵屏的程序设计技术。

## 2. 实训要求

- ① 本例未考虑显示屏的功率驱动问题,完成本例调试后在电路中添加显示驱动器,仍实

现本例运行效果。

- ② 为本例添加多种显示特效,例如由下向上滚动显示、逐字飞入显示、百叶窗式显示。
- ③ 将本例 LED 显示屏改成 80×32 点阵,编程实现更大幅面 LED 屏的显示功能。
- ④ 用自己所熟悉的 Windows 平台软件开发工具设计上位机软件,通过 PC 机串口将待显示中英文信息发送给下位机 LED 点阵屏滚动显示。

### 3. 源程序代码

```

001 //----- AT25F4096.c -----
002 // 名称:用 SPI 接口读/写 AT25F4096 子程序
003 //-----
004 #define F_CPU 4000000UL
005 #include <avr/io.h>
006 #include <util/delay.h>
007 #define INT8U unsigned char
008 #define INT16U unsigned int
009 #define INT32U unsigned long
010
011 //AT25F4096 指令集
012 #define WREN 0x06 //使能写
013 #define WRDI 0x04 //禁止写
014 #define RDSR 0x05 //读状态
015 #define WRSR 0x01 //写状态
016 #define READ 0x03 //读字节
017 #define PROGRAM 0x02 //写字节
018 #define SECTOR_ERASE 0x52 //删除区域数据
019 #define CHIP_ERASE 0x62 //删除芯片数据
020 #define RDID 0x15 //读厂商与产品 ID
021 //SPI 使能与禁用
022 #define SPI_EN() (PORTB &= 0xEF)
023 #define SPI_DI() (PORTB |= 0x10)
024 //-----
025 // SPI 主机初始化
026 //-----
027 void SPI_MasterInit()
028 {
029     //SPI 接口配置
030     DDRB = 0b10110000; PORTB = 0xFF;
031     //SPI 使能,主机模式,16 分频
032     SPCR |= _BV(SPE) | _BV(MSTR) | _BV(SPR0);
033 }
034
035 //-----

```

```
036 // SPI 数据传输
037 //-----
038 INT8U SPI_Transmit(INT8U dat)
039 {
040     SPDR = dat;                      //启动数据传输
041     while(!(SPSR & _BV(SPIF)));      //等待结束
042     SPSR |= _BV(SPIF);              //清中断标志
043     return SPDR;
044 }
045
046 //-----
047 // 读 AT25F4096 芯片状态
048 //-----
049 INT8U Read_SPI_Status()
050 {
051     INT8U status;
052     SPI_EN();
053     SPI_Transmit(RDSR);            //发送读状态指令
054     status = SPI_Transmit(0xFF);
055     SPI_DI();
056     return status;
057 }
058
059 //-----
060 // AT25F4096 忙等待
061 //-----
062 void Busy_Wait()
063 {
064     while(Read_SPI_Status() & 0x01); //忙等待
065 }
066
067 //-----
068 // 向 AT25F4096 写入 3 个字节的地址 0x000000~0x01FFFF (实际有效位为 19 位)
069 //-----
070 void Write_3_Bytes_AT25F4096_Address(INT32U addr)
071 {
072     SPI_Transmit((INT8U)(addr >> 16));
073     SPI_Transmit((INT8U)(addr >> 8));
074     SPI_Transmit((INT8U)(addr));
075 }
076
077 //-----
078 // 从指定地址读单字节
```

```

079 //-----
080 INT8U Read_Byte_FROM_AT25F4096(INT32U addr)
081 {
082     INT8U dat;
083     SPI_EN();
084     SPI_Transmit(READ);           //发送读指令
085     Write_3_Bytes_AT25F4096_Address(addr); //发送3字节地址
086     dat = SPI_Transmit(0xFF);      //读取字节数据
087     SPI_DI();
088     return dat;
089 }
090
091 //-----
092 // 从指定地址读多字节到缓冲
093 //-----
094 void Read_Some_Bytes_FROM_AT25F4096(INT32U addr, INT8U * p, INT16U len)
095 {
096     INT16U i;
097     SPI_EN();
098     SPI_Transmit(READ);           //发送读指令
099     Write_3_Bytes_AT25F4096_Address(addr); //发送3字节地址
100    for( i = 0; i < len; i++)        //读数据序列
101        p[i] = SPI_Transmit(0xFF);
102    SPI_DI();
103 }
104
105 //-----
106 // 向 AT25F4096 指定地址写入单字节数据
107 //-----
108 void Write_Byte_TO_AT25F4096(INT32U addr, INT8U dat)
109 {
110     SPI_EN();
111     SPI_Transmit(WREN);          //使能写
112     SPI_DI();
113     Busy_Wait();
114     SPI_EN();
115     SPI_Transmit(PROGRAM);      //写指令
116     Write_3_Bytes_AT25F4096_Address(addr); //发送3字节地址
117     SPI_Transmit(dat);          //写字节数据
118     SPI_DI();
119     Busy_Wait();
120 }

```



```
001 //----- main.c -----  
002 // 名称：能接收串口信息的带中英文硬字库的 80 * 16 点阵显示屏  
003 //-----  
004 // 说明：本例运行时，点阵屏将滚动显示一组固定信息  
005 // 当接收到串口发送来的中英文/全角/半角字符时，点阵屏将开始  
006 // 滚动显示新接收到的信息  
007 //  
008 //-----  
009 #define F_CPU 4000000UL  
010 #include <avr/io.h>  
011 #include <avr/interrupt.h>  
012 #include <string.h>  
013 #include <stdio.h>  
014 #include <util/delay.h>  
015 #define INT8     signed   char  
016 #define INT8U    unsigned char  
017 #define INT16U   unsigned int  
018 #define INT32U   unsigned long  
019  
020 //74595 及 74154 相关引脚定义  
021 #define DS          PA0           //串行数据输入  
022 #define SH_CP       PA1           //移位时钟脉冲  
023 #define ST_CP       PA2           //输出锁存器控制脉冲  
024 #define E1_74HC154  PC7           //74HC154 译码器使能  
025  
026 //74595 及 74154 相关引脚操作  
027 #define DS_1()      PORTA |= _BV(DS)  
028 #define DS_0()      PORTA &= ~_BV(DS)  
029 #define SH_CP_1()   PORTA |= _BV(SH_CP)  
030 #define SH_CP_0()   PORTA &= ~_BV(SH_CP)  
031 #define ST_CP_1()   PORTA |= _BV(ST_CP)  
032 #define ST_CP_0()   PORTA &= ~_BV(ST_CP)  
033  
034 //74154 译码器使能与禁止  
035 #define EN_74HC154() PORTC &= ~_BV(E1_74HC154)  
036 #define DI_74HC154() PORTC |= _BV(E1_74HC154)  
037  
038 //SPI 相关函数  
039 extern void SPI_MasterInit();  
040 extern void Read_Some_Bytess_FROM_AT25F4096(INT32U addr, INT8U * p, INT16U len);  
041  
042 //最多可接收的汉字个数  
043 #define MAX_WORD_COUNT 50
```

```

044 //开始时待显示的中英文字符串
045 //及从串口接收的中英文数字等字符信息都将覆盖保存到 bMsg 中
046 struct MSG
047 {
048     INT8U Buffer[MAX_WORD_COUNT * 2 + 2];
049     INT16U Len;
050 } bMsg;
051
052 //缓冲可保存汉字点阵数据的最大汉字个数(如果为半角字符则 * 2)
053 #define MAX_DOT_WORD_COUNT 20
054 //待显示汉字点阵数据缓冲
055 INT8U WORD_Dots_Buffer[MAX_DOT_WORD_COUNT * 32];
056 //-----
057 // USART 初始化
058 //-----
059 void Init_USART()
060 {
061     UCSRB = _BV(RXEN) | _BV(RXCIE);           //允许接收,接收中断使能
062     UCSRC = _BV(URSEL) | _BV(UCSZ1) | _BV(UCSZ0); //8 位数据位,1 位停止位
063     UBRRRL = (F_CPU / 9600 / 16 - 1) % 256;    //波特率:9600
064     UBRRRH = (F_CPU / 9600 / 16 - 1) / 256;
065 }
066
067 //-----
068 // 串行输入子程序
069 //-----
070 void Serial_Input_595(INT8U dat)
071 {
072     INT8U i;
073     for(i = 0x80; i != 0x00; i >>= 1)          //由高位到低位,串行输入 8 位
074     {
075         if (dat & i) DS_1(); else DS_0();
076         SH_CP_0(); _delay_us(2);
077         SH_CP_1(); _delay_us(2);                //移位时钟脉冲上升沿移位
078     }
079 }
080
081 //-----
082 // 并行输出子程序
083 //-----
084 void Parallel_Output_595()
085 {
086     ST_CP_0(); _delay_us(1);

```

```

087     ST_CP_1(); _delay_us(1);                                //上升沿将数据送到输出锁存器
088 }
089
090 //-----
091 // 根据 bMsg.Buffer,从硬字库读取全角或半角字符点阵数据并完成必要转换
092 //-----
093 void Read_SPI_Word_Dot_Matrix_AND_Convert()
094 {
095     INT16U i,j = 0,k;
096     INT32U Offset;                                         //汉字在点阵库中的偏移位置
097     INT8U SectionCode, PlaceCode;                         //汉字区码与位码
098     INT8U Temp_Buf[32];                                    //转换用临时缓冲
099     for (i = 0; i < MAX_DOT_WORD_COUNT * 32; i++) //清空点阵缓冲
100         WORD_Dots_Buffer[i] = 0x00;
101
102     i = 0;
103     while ( i < bMsg.Len )
104     {
105         if ( bMsg.Buffer[i] >= 0xA0 )                      //处理汉字编码
106         {
107             //取得汉字区位码
108             SectionCode = bMsg.Buffer[ i ] - 0xA0;
109             PlaceCode = bMsg.Buffer[ i + 1 ] - 0xA0;
110
111             //根据当前汉字区位码计算其点阵在字库中的偏移位置
112             Offset = (94L * (SectionCode - 1) + (PlaceCode - 1)) * 32L;
113
114             //从 SPI 存储器 AT25F4096 读取 32 字节汉字点阵
115             Read_Some_Bytes_FROM_AT25F4096(Offset,Temp_Buf,32);
116
117             //汉字字库中点阵格式为 16 行依次左->右,左->右取字节,为适应点阵屏显示
118             //下面将其转换为先取左半边 16 行,再取右半边 16 行
119             for (k = 0; k < 16; k++)
120             {
121                 WORD_Dots_Buffer[j + k] = Temp_Buf[2 * k];
122                 WORD_Dots_Buffer[j + k + 16] = Temp_Buf[2 * k + 1];
123             }
124             //每个汉字点阵保存到 WORD_Dots_Buffer 后跳过 32 字节
125             //((每个汉字点阵占 32 字节)
126             //从 bMsg.Buffer 中取字符的索引递增 2(每个汉字编码占两 2 字节)
127             j += 32; i += 2;
128         }
129     else //处理半角字符编码

```

```

130     {
131         //ASCII 字符偏移地址 = ASCII 字库在合成字库中的起始地址 + ASCII 码 * 16
132         //半角的 ASCII 字符点阵在合成字库中汉字点阵字库的后面,汉字点阵共计 267616 字节
133         Offset = 267616L + bMsg.Buffer[i] * 16;
134         Read_Some_Bytes_FROM_AT25F4096(Offset,WORD_Dots_Buffer + j,16);
135         //每个半角 ASCII 字符点阵保存到 WORD_Dots_Buffer 以后跳过 16 字节
136         //((每个 ASCII 字符点阵占 16 字节)
137         //从 bMsg.Buffer 中取字符的索引递增 1(每个 ASCII 字符编码占 1 字节)
138         j += 16; i++;
139     }
140 }
141 }
142
143 //-----
144 // 主程序
145 //-----
146 int main()
147 {
148     INT8U i,j,z,d = 0;
149
150     DDRA = 0xFF; PORTA = 0xFF;           //配置端口
151     DDRC = 0xFF; PORTC = 0xFF;
152     DDRD = 0x02; PORTD = 0xFF;
153
154     //在显示缓冲中先预设初始时待显示的字符串
155     strcpy((char *)bMsg.Buffer,"★点阵演示 V1.0★... ");
156     bMsg.Len = strlen((char *)bMsg.Buffer);
157
158     SPI_MasterInit();                  //SPI 主机初始化
159     Init_USART();                    //串口初始化
160     sei();                          //接收中断许可
161
162     //根据 bMsg.Buffer 从 SPI 存储器读取全角或半角字符点阵数据并完成必要的转换
163     Read_SPI_Word_Dot_Matrix_AND_Convert();
164
165     while(1)
166     {
167         for (z = 0; z <= bMsg.Len - 10; z++)
168         {
169             for(d = 0; d < 10; d++)          //此循环用于控制显示滚动的速度
170             {
171                 for(i = 0; i < 16 ; i++)      //完成每个汉字的 16 列扫描
172                 {

```



```
173 //数据串行输入 595(5 块 16 * 16 点阵屏,共 10 片 595)
174     for (j = 0; j < 5; j++)
175     {
176         Serial_Input_595(WORD_Dots_Buffer[z * 16 + j * 32 + i + 16]);
177         Serial_Input_595(WORD_Dots_Buffer[z * 16 + j * 32 + i]);
178     }
179
180     DI_74HC154();           //先禁用译码器
181     Parallel_Output_595(); //595 数据并行输出
182
183     PORTC = (PORTC & 0xF0) | i; //写译码器
184     EN_74HC154();          //使能译码器,译码输出,选通第 i 列
185
186     _delay_ms(2);
187 }
188 }
189 }
190 }
191 }
192
193 //-----
194 // 串口接收中断函数
195 //-----
196 ISR (USART_RXC_vect)
197 {
198     //将当前接收到的字符存入 c
199     INT8U c = UDR;
200     //接收到'\r'时忽略
201     if (c == '\r') return;
202     //如果接收到'\n'表示本次接收完毕
203     if (c == '\n')
204     {
205         //重新从 SPI 存储器读取 bMsg.Buffer 的汉字点阵
206         Read_SPI_Word_Dot_Matrix_AND_Convert();
207         return;
208     }
209     //缓存新接收的字符
210     if (bMsg.Len < MAX_WORD_COUNT * 2) bMsg.Buffer[bMsg.Len++] = c;
211     //任何时候接收到"##"时清空缓冲
212     if (bMsg.Len >= 2 && bMsg.Buffer[bMsg.Len - 1] == '#'
213         && bMsg.Buffer[bMsg.Len - 2] == '#')
214     {
215         bMsg.Len = 0;
```

216 }

217 }

## 5.26 用 AVR 与 1601 LCD 设计的计算器

本例用单行字符液晶、简易计算器键盘、矩阵键盘解码芯片及 AVR 单片机设计了整数计数器，该计算器仿真案例可进行四则运算的单次或连续运算，案例暂不支持带优先级的表达式求值。例电路及部分运行效果如图 5-28 所示。

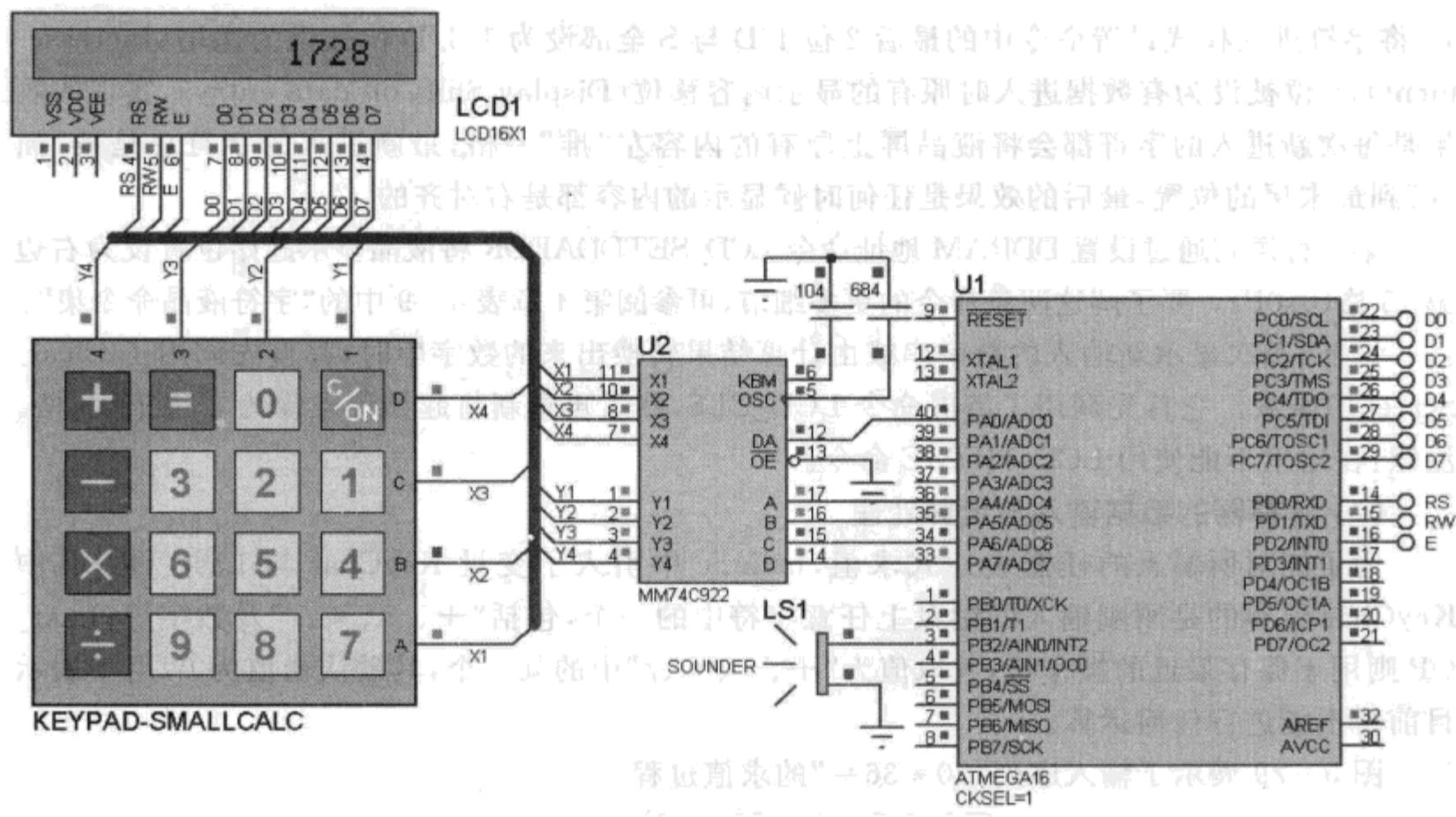


图 5-28 用 AVR 与 1601 LCD 设计的计算器

### 1. 程序设计与调试

本例程序设计要点在于单行液晶以右端为起点的显示设计与计算器的输入及数据运算程序设计，下面分别讨论这两项设计：

#### (1) 本例液晶的显示程序设计

与此前字符液晶程序设计不同的是，本例未通过调用宏定义 RS\_1()、RS\_0()、RW\_1()、RW\_0()来选择寄存器及设置读/写操作，本例给出了这些操作的 4 种组合所对应的操作地址定义：

```
#define LCD_CMD_WR      0x00      //RW,RS = 00
#define LCD_DATA_WR      0x01      //RW,RS = 01
#define LCD_BUSY_RD       0x02      //RW,RS = 10
#define LCD_DATA_RD       0x03      //RW,RS = 11(本例未用)
```

将上述地址定义直接发给液晶控制端口 LCD\_CONTROL(本例为 PORTD)即可设置写

命令寄存器、写数据寄存器、读取忙状态、读数据共 4 项操作,这样设计比使用宏调用的方法各少用了一条指令。

本例液晶显示设计的第 2 个差异是所有显示的内容总是从右端开始,且字符向左移位,这样可使得任何时候输入的操作数或运算结果总是右对齐显示。为实现该设计要求,液晶初始化函数 Initialize\_LCD() 中使用了语句:

```
Write_LCD_Command(LCD_SETMODE + 0x03); //自动递增,显示左移
Write_LCD_Command(LCD_SETDDADDR + 0x0F); //初始显示位置在右边
```

第 1 行的字符进入模式命令 LCD\_SETMODE 附带“参数”0x03(即 0B00000011),其中的 11 将字符进入模式设置命令中的最后 2 位 I/D 与 S 全部设为 1,I/D 位被设为递增 (Increase-ment),S 位被设为有数据进入时原有的显示内容移位 (Display Shift on data entry),实现的效果是每次新进入的字符都会将液晶屏上原有的内容左“推”一格,最新进入的字符总是被“插入”到最末尾的位置,最后的效果是任何时候显示的内容都是右对齐的。

第 2 行语句通过设置 DDRAM 地址命令 LCD\_SETDDADDR 将液晶显示起始位置设为右边第 15 格(0x0F)。要了解这两条命令的更多细节,可参阅第 4 章表 4-9 中的“字符液晶命令集”。

另外,每次显示新输入的数字串或由计算结果转换出来的数字串时,都首先调用了 ClearScreen() 函数。它首先调用了清屏命令 LCD\_CLS,然后再重新将起始位置设为右边第 15 格。注意:在这里不能使用 LCD\_HOME 命令。

## (2) 计算器的数据输入与运算处理

为了能对所输入的任意表达式求值,main.c 中引入了变量 KeyChar 与 Last\_OP,其中 KeyChar 保存的是刚刚输入的键盘上任意字符中的一个,包括“+、-、\*、/”及数字等;Last\_OP 则用于保存最近的操作符,其取值为“+、-、\*、/”中的某一个,其默认初值为 0,用于表示目前暂不能进行任何运算。

图 5-29 展示了输入序列“20 \* 36=”的求值过程。

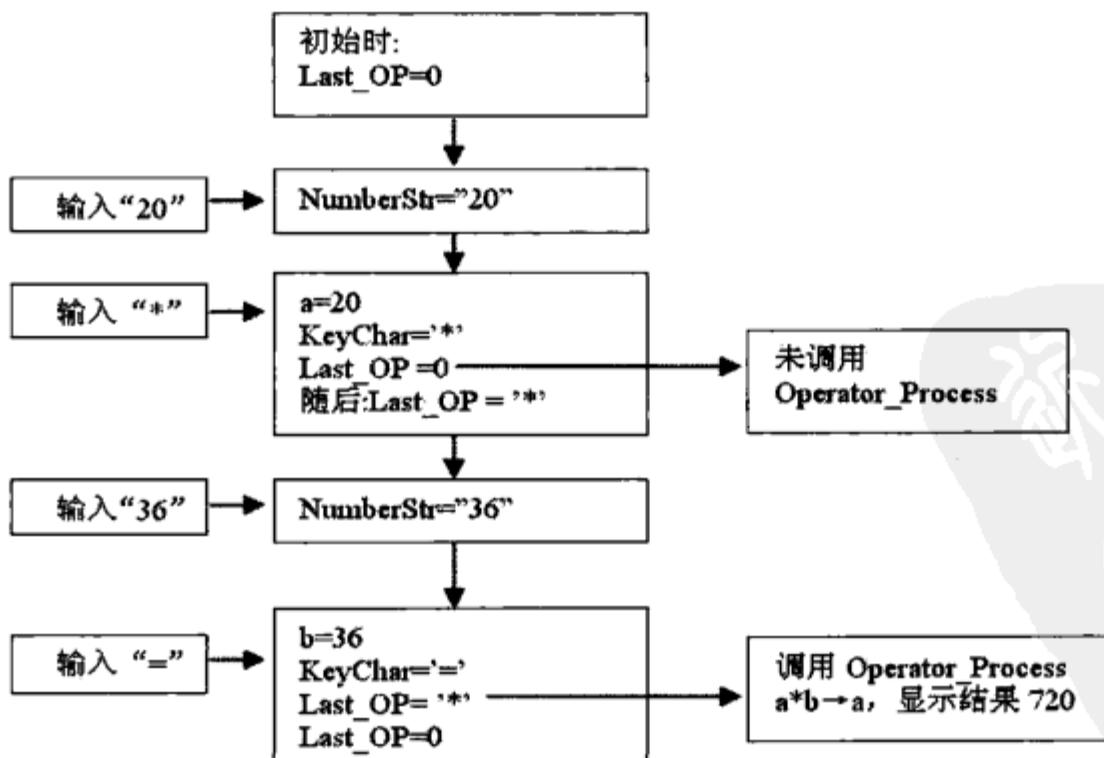


图 5-29 输入序列“20 \* 36=”的计算过程

当输入序列为“20 \*”时,由于此时  $a=20$ ,  $\text{Last\_OP}$  还不是“\*”号,因而不会调用 `Operator_Process` 进行运算处理,随后  $\text{Last\_OP}$  才取得“\*”;当继续输入“36”,随后再输入“=”时, $b=36$ , $\text{KeyChar}$  为“=”,此时的  $\text{Last\_OP}$  还是“\*”,它刚好是  $a$ 、 $b$  的运算符,调用 `Operator_Process` 即可得到运算结果,在结果求出并显示以后,因为  $\text{KeyChar}$  保存的是“=”, $\text{Last\_OP}$  没有遇到新的操作符,它被还原为 0。

如果用户输入的序列是“20 \* 36 + 9 =”,这时情况又如何呢?

显然,当  $\text{KeyChar}$  取得“+”时,  $\text{Last\_OP}$  仍为“\*”,前 2 个数的运算仍得以正常进行,结果保存在变量  $a$  中,此时  $a=720$ 。由于  $\text{KeyChar}$  新取得的符号是运算符,在“\*”的运算之后,  $\text{Last\_OP}$  重新取得该运算符,即  $\text{Last\_OP}='+'$ ,随后再输入 9,再接着  $\text{KeyChar}$  又取得了“=”,这时调用 `Operator_Process`,完成的运算是  $720 + 9$ ,结果仍存入  $a$  中,并且  $\text{Last\_OP}$  再次变为 0。

由以上分析可见,本例设计可以很好地处理单次运算或连续运算,实际上本例设计对于出现的异常输入序列也能很好处理。

参考上述两例求值过程,大家可进一步分析与跟踪下面 5 个输入序列的计算过程:

- ① 210 \* 36 \* =
- ② 23 \* \* 8 =
- ③ 23 + 9 \* 8 - 12 + \* / =
- ④ 59 \* = 32 + 9 / 8 =
- ⑤ 23 \* - 9 \* 80 = \* 12 / 9 =

其中第 5 行不会被解析为“ $23 * (-9) * 80 = * 12 / 9 =$ ”,而是会被解析为“ $23 - 9 * 80 = * 12 / 9 =$ ”,其原因留给大家自行分析。通过上述跟踪,大家会进一步理解  $\text{Last\_OP}$  及  $\text{KeyChar}$  在求解输入表达式序列过程中所扮演的角色。

通过分析上述输入序列中的异常序列时,大家还会明白为什么调用运算处理函数 `Operator_Process` 时,除了要求  $\text{Last\_OP}$  非 0(即只能是 4 个运算符中的一个),还添加了 `Number_Idx` 非 0 的条件,即:

```
if ( Last_OP && Number_Idx ) Operator_Process(Last_OP)
```

## 2. 实训要求

① 用数码管替换本例的显示器件,并用定时器控制输出按键提示音,重新编程实现本例的整数计算器功能。

② 从 Proteus 元件库中选择计算器键盘“KEYPAD—CALCULATOR”替换本例小键盘,并进一步改写本例程序,使计算器能实现浮点数运算。

③ 使用双堆栈技术实现带优先级的表达式运算,例如对表达式  $2+3\times 4$  求值时,其结果为 14 而不是 20。

## 3. 源程序代码

```
001 //----- LCD1601.c -----
002 // 名称: 1601 液晶显示驱动程序
003 //-----
004 #include <avr/io.h>
```

```

005 #include <util/delay.h>
006 #include "LCD1601.h"
007 #define INT8U unsigned char //参阅案例压缩包
008 #define INT16U unsigned int
009
010 //液晶端口定义
011 #define LCD_PORT PORTC //发送 LCD 数据/命令端口
012 #define LCD_PIN PINC //接收 LCD 数据/状态端口
013 #define LCD_DDR DDRC //LCD 端口数据方向
014 #define LCD_CONTROL PORTD //液晶控制端口
015 //液晶寄存器地址定义(写命令,写数据,读忙状态,读数据寄存器)
016 #define LCD_CMD_WR 0x00 //RW,RS = 00
017 #define LCD_DATA_WR 0x01 //RW,RS = 01
018 #define LCD_BUSY_RD 0x02 //RW,RS = 10
019 #define LCD_DATA_RD 0x03 //RW,RS = 11
020 //液晶命令集
021 #define LCD_CLS 0x01 //清屏
022 #define LCD_HOME 0x02 //光标归位
023 #define LCD_SETMODE 0x04 //进入模式设置
024 #define LCD_SETVISIBLE 0x08 //开显示
025 #define LCD_SHIFT 0x10 //移位方式
026 #define LCD_SETFUNCTION 0x20 //功能设置
027 #define LCD_SETCGADDR 0x40 //设置 CGRAM 地址
028 #define LCD_SETDDADDR 0x80 //设置 DDRAM 地址
029 //液晶使能引脚操作定义
030 #define EN_1() (LCD_CONTROL |= _BV(PD2))
031 #define EN_0() (LCD_CONTROL &= ~_BV(PD2))
032 //-----
033 // LCD 忙等待
034 //-----
035 void LCD_BUSY_WAIT()
036 {
037     INT8U LCD_Status;
038     LCD_DDR = 0x00; //方向设为输入
039     LCD_CONTROL = LCD_BUSY_RD; //读状态寄存器
040     do
041     {
042         EN_1(); asm("nop"); LCD_Status = LCD_PIN; //读取状态
043         EN_0();
044     } while (LCD_Status & 0x80); //液晶忙时继续
045 }
046
047 //-----

```

```

048 // 写 LCD 命令寄存器
049 //-----
050 void Write_LCD_Command(INT8U cmd)
051 {
052     LCD_DDR = 0xFF;                                //设为输出
053     LCD_PORT = cmd;                               //发送命令
054     LCD_CONTROL = LCD_CMD_WR;                   //写命令寄存器
055     EN_1(); asm("nop"); EN_0();                //写入
056     LCD_BUSY_WAIT();                            //忙等待
057 }
058
059 //-----
060 // 写 LCD 数据寄存器
061 //-----
062 void Write_LCD_Data(INT8U dat)
063 {
064     LCD_DDR = 0xFF;                                //设为输出
065     LCD_PORT = dat;                             //发送数据
066     LCD_CONTROL = LCD_DATA_WR;                 //写数据寄存器
067     EN_1(); asm("nop"); EN_0();                //写入
068     LCD_BUSY_WAIT();                            //忙等待
069 }
070
071 //-----
072 // LCD 初始化
073 //-----
074 void Initialize_LCD()
075 {
076     Write_LCD_Command(LCD_SETFUNCTION + 0x10);    //单行 8 位模式
077     Write_LCD_Command(LCD_SETVISIBLE + 0x04);      //开显示,关光标
078     Write_LCD_Command(LCD_SETMODE + 0x03);        //自动递增,显示左移
079     Write_LCD_Command(LCD_SETDDADDR + 0x0F);      //初始显示位置在右边第 15 格
080 }
081
082 //-----
083 // 清屏
084 //-----
085 void ClearScreen()
086 {
087     Write_LCD_Command(LCD_CLS);                  //清屏
088     Write_LCD_Command(LCD_SETDDADDR + 0x0F);      //从右边第 15 格开始显示
089 }
090

```



```
091 //-----  
092 // 显示字符串  
093 //-----  
094 void ShowString(char * str)  
095 {  
096     INT8U i = 0;  
097     ClearScreen();                                //刷新显示之前先清屏  
098     while (str[i] && i < MAX_DISPLAY_CHAR)        //输出字符串 str  
099     {  
100         Write_LCD_Data(str[i++]);  
101     }  
102 }  
  
001 //----- main.c -----  
002 // 名称：用 AVR 与 1601LCD 设计的计算器  
003 //-----  
004 // 说明：本例运行时，可完成整数的加、减、乘、除 4 种运算，该计算器  
005 // 不支持带优先级的表达式运算，但允许连续进行整数运算。  
006 // 如果运算结果超出有效范围则显示 *ERR*  
007 //  
008 //-----  
009 # include <avr/io.h>  
010 # include <avr/pgmspace.h>  
011 # include <util/delay.h>  
012 # include <stdio.h>  
013 # include <string.h>  
014 # include <stdlib.h>  
015 # include <ctype.h>  
016 # include "LCD1601.h"  
017 # define INT8U unsigned char  
018  
019 //蜂鸣器及键盘相关定义  
020 # define BEEP()      PORTB ^= _BV(PB2)          //蜂鸣器定义  
021 # define Key_Pressed (PIN_A & _BV(PA0))        //按键判断  
022 # define Key_NO    ((PIN_A & 0xF0) >> 4)        //按键键值  
023 //计算器相关变量，状态及字符表定义  
024 char Last_OP = 0;                            //最近的操作符  
025 long a = 0, b = 0;                            //操作数 a,b  
026 char LCD_DISP_BUFFER[17];                    //LCD 显示缓冲  
027 char NumberStr[17];                          //输入数字串缓冲  
028 INT8U Number_Idx = 0;                        //数字串缓冲索引  
029 const char KEY_CHAR_TABLE[] = "741C8520963=/*-+"; //键盘字符表  
030 //-----
```

```

031 // 根据操作符完成运算或清屏等操作
032 //-----
033 void Operator_Process(char OP)
034 {
035     //根据 OP 分别完成"+","_","*","/","C"操作
036     switch( OP )
037     {
038         case '+' : a += b;           break;
039         case '-' : a -= b;           break;
040         case '*' : a *= b;           break;
041         case '/' : if (b)          //除数非 0 时才进行运算
042             {
043                 a /= b; break;
044             }
045         else                      //否则提示出错,复位变量并返回
046         {
047             ShowString(" * ERR * ");
048             a = b = 0;
049             Last_OP = 0;
050             return;
051         }
052         case 'C' : a = b = 0;
053             Last_OP = 0; break;
054     }
055     //显示结果
056     sprintf(LCD_DISP_BUFFER, "%ld",a);
057     ShowString(LCD_DISP_BUFFER);
058 }
059
060 //-----
061 // 蜂鸣器输出提示音
062 //-----
063 void Sounder()
064 {
065     INT8U i;
066     for (i = 0; i < 20; i++)
067     {
068         BEEP(); _delay_us(350);
069     }
070 }
071
072 //-----
073 // 主程序

```

```

074 //-----
075 int main()
076 {
077     char KeyChar;
078     DDRA = 0x00; PORTA = 0xFF;                                //配置端口
079     DDRB = 0xFF;
080     DDRC = 0xFF;
081     DDRD = 0xFF;
082     //初始化LCD并在最右端显示"0"
083     Initialize_LCD(); ShowString("0");
084     for(;;)
085     {
086         //如果无按键则继续-----
087         if (!Key_Pressed) { _delay_ms(10); continue; }
088         //输出按键音
089         Sounder();
090         //根据键值获取按键字符
091         KeyChar = KEY_CHAR_TABLE[Key_NO];
092         //-----
093         if (isdigit(KeyChar)) //如果输入的是数字字符则存入NumberStr
094         {
095             if (Number_Idx != MAX_DISPLAY_CHAR - 2)
096             {
097                 NumberStr[Number_Idx] = KeyChar;
098                 NumberStr[ ++ Number_Idx] = '\0';
099                 ShowString(NumberStr);
100            }
101        }
102        //-----
103        else //如果输入的是"+,-,*,/,,C,="中的某一个则进行运算或清零等处理
104        {
105            //将NumberStr字符串转换为长整数a或b
106            if (Number_Idx != 0)
107            {
108                if (Last_OP == 0)
109                    a = strtol(NumberStr, '\0', 10);
110                else
111                    b = strtol(NumberStr, '\0', 10);
112            }
113            //如果为"C"则清0且将相关变量复位
114            if (KeyChar == 'C')          Operator_Process('C');
115            //如果为"=,+,-,*,/"且此前有数字字符输入则进行运算
116            else

```

```

117     if ( Last_OP && Number_Idx ) Operator_Process(Last_OP);
118
119         //NumberStr 数字缓冲索引归 0, 并清除数字串输入缓冲
120         Number_Idx = 0; NumberStr[0] = '\0';
121         //Last_OP 保存最近按下的操作符
122         if (KeyChar != 'C' && KeyChar != '=')
123             Last_OP = KeyChar;
124         else
125             Last_OP = 0;
126     }
127     //等待释放按键
128     while (Key_Pressed);
129 }
130 }

```

## 5.27 电子秤仿真设计

本例综合应用计算器(乘法)、压力传感器、键盘解码器及 1602 英文液晶，案例运行过程中，用户可设置当前商品单价，当压力变化时(本例未将压力单位换算为质量单位)，液晶屏将实时计算并刷新显示金额。本例电路及程序部分运行效果如图 5-30 所示。

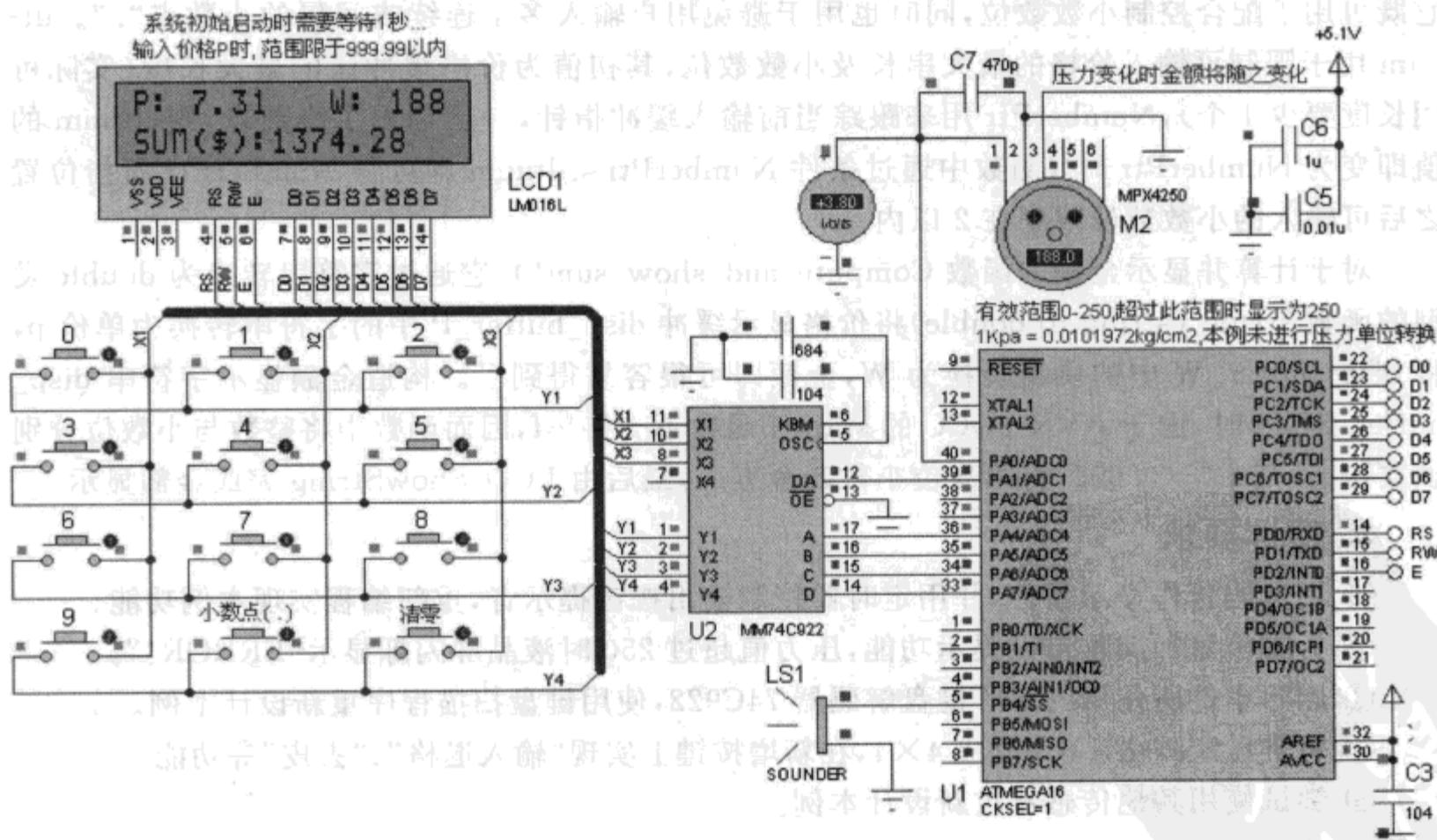


图 5-30 电子秤仿真设计

## 1. 程序设计与调试

压力数据转换的程序设计比较简单,本例忽略对这一部分的讨论。

下面重点讨论价格输入功能设计与金额计算功能设计,Calc.c 中的 KeyBoard\_Handle() 与 Compute\_and\_show\_sum() 分别完成了这两项功能。

为简化矩阵键盘输入扫描,案例使用了 74C922 解码芯片,这样可使得读取按键的代码变得非常简单,键盘操作处理函数 KeyBoard\_Handle() 首先读取按键,在进行金额计算之前,该函数需要完成的处理要求如下:

- ① 将输入价格的范围限制在 0~999.99;
- ② 输入价格的整数部分已达到 3 位时只允许输入小数部分;
- ③ 任何时候只要开始输入了小数点,程序即开始限制可输入的小数位,保证所输入的小数位不超过 2 位;
- ④ 不允许输入 2 个以上的连续或间隔小数点;
- ⑤ 价格中没有非 0 整数部分时,允许用户直接从小数点开始输入,例如输入“0.83”时可改成直接输入“.83”。
- ⑥ 价格清零处理。

KeyBoard\_Handle() 函数将所输入的价格数位存放于字符串 disp\_buffer\_P,在字符串长为 3 时,如果还未输入过小数点时则只允许输入小数点及小数位,否则返回,这样可限制只能输入 3 位整数。

为处理小数点问题,函数中引入变量 havedot,它用于标识当前是否已经输入了小数点,它既可用于配合控制小数数位,同时也用于避免用户输入多个连续或间隔的小数点“.”。dt\_num 用于限制可输入价格的最大串长及小数数位,其初值为价格缓冲区的最大长度(实际可用长度要少 1 个),NumberPtr 用来跟踪当前输入缓冲指针,一旦输入了小数点,则 dtbuf 的值即变为 NumberPtr+2,函数中通过条件 NumberPtr<dt\_num 即可将 NumberPtr 所指位置之后可输入的小数数位限制在 2 以内。

对于计算并显示金额的函数 Compute\_and\_show\_sum(),它通过字符串转换为 double 类型的函数 strtod (string to double) 将价格显示缓冲 disp\_buffer\_P 中的字符串转换为单价 p,将 disp\_buffer\_W 中的质量转换为 W,金额即可很容易得到了。构造金额显示字符串 disp\_buffer\_SUM 时,由于 AVR-GCC 的 sprintf 函数不支持 %f,因而函数中将整数与小数位分别独立构造,其中“+0.005”将第 3 位小数四舍五入,最后由 LCD\_ShowString 完成金额显示。

## 2. 实训要求

- ① 用数码管作显示器件,并用定时器控制输出按键提示音,重新编程实现本例功能。
- ② 为本例添加超重错误提示功能,压力值超过 250 时液晶屏闪烁显示“ERROR!”。
- ③ 删除本例所使用的矩阵键盘解码器 74C922,使用键盘扫描程序重新设计本例。
- ④ 将键盘矩阵由  $4 \times 3$  改成  $4 \times 4$ ,在新增按键上实现“输入退格”、“去皮”等功能。
- ⑤ 尝试使用其他传感器重新设计本例。

## 3. 源程序代码

```
001 //----- Calc.c -----
002 // 名称: 电子秤价格输入与金额计算程序
```

```

003 //-----
004 #include <avr/io.h>
005 #include <util/delay.h>
006 #include <stdio.h>
007 #include <string.h>
008 #include <stdlib.h>
009 #include <ctype.h>
010 #define INT8U unsigned char
011 #define INT16U unsigned int
012
013 //蜂鸣器定义
014 #define BEEP() PORTB ^= _BV(PB4)
015 //按键判断及按键键值
016 #define Key_Pressed (PIN_A & _BV(PA1)) //DA(PA1)为高电平时有键按下
017 #define Key_NO ((PIN_A & 0xF0) >> 4) //解码器输出线连接在 PA 高 4 位
018 //键盘字符表(其中注意 2、5、8 后各保留一个空格)
019 const char KEY_CHAR_TABLE[] = "012 345 678 9.C";
020 //液晶显示字符串函数
021 extern void LCD_ShowString(INT8U x, INT8U y, char * str);
022
023 //LCD 显示输出缓冲(价格/质量/金额)的最大长度
024 //因为要预留结束标志,实际串长比定义少 1 位
025 #define PLEN 7
026 #define WLEN 4
027 #define SUMLEN 10
028 //LCD 显示输出缓冲(价格/质量/金额)
029 char disp_buffer_P[PLEN];
030 char disp_buffer_W[WLEN];
031 char disp_buffer_SUM[SUMLEN];
032 //价格输入缓冲的指针
033 INT8U NumberPtr = 0;
034
035 //-----
036 // 蜂鸣器输出
037 //-----
038 void Sounder()
039 {
040     INT16U i;
041     for (i = 0; i < 350; i++)
042     {
043         BEEP(); _delay_us(220);
044     }
045 }

```



```
046
047 //-----
048 // 处理运算并显示金额
049 //-----
050 void Compute_and_show_sum()
051 {
052     double p,w;
053     p = strtod(disp_buffer_P,'0');           //将价格字符串转换为 double 类型
054     w = strtod(disp_buffer_W,'0');           //将质量字符串转换为 double 类型
055     p *= w;                                //计算金额
056     sprintf(disp_buffer_SUM,"%ld.%02ld", (long)p,
057               (long)((p - (long)p + 0.005) * 100));
058     LCD_ShowString(7,1,"");                 //清除金额(输出 9 个空格)
059     LCD_ShowString(7,1,disp_buffer_SUM);    //显示金额
060 }
061
062 //-----
063 // 处理键盘操作
064 //-----
065 void KeyBoard_Handle()
066 {
067     char KeyChar;
068     //是否已经输入了价格 P 的小数点
069     static INT8U havedot = 0;
070     //在还没有输入价格中的小数点时可继续输入字符的个数
071     static INT8U dtnum = PLEN;
072     //如果有键按下则获取按键字符(根据解码器 DA 引脚是否输出高电平来判断)
073     if (Key_Pressed)
074     {
075         //每次按键时输出按键提示音
076         Sounder();
077         //根据键值获取按键字符
078         KeyChar = KEY_CHAR_TABLE[Key_NO];
079         //如果输入的是数字字符或小数点(但此前未输入过小数点)
080         //-----
081         if (isdigit(KeyChar) || (KeyChar == '.' && !havedot))
082         {
083             //在目前还未输入小数点,且当前输入的不是小数点,而此时串长已为 3 时返回
084             //((由于输入范围为 0~999.99,程序不允许输入 3 位以上的整数)
085             if (strlen(disp_buffer_P) == 3 && (KeyChar != '.' && !havedot)) return;
086
087             //将所输入的字符存入缓冲
088             if (NumberPtr < dtnum)
```

```

089 {
090     //如果输入的第一个字符是'0'或"."则直接处理为"0."
091     //这样设计可允许用户在没有非 0 的价格整数位时直接从小数点开始输入
092     //例如要输入"0.86"时可直接输入".86"
093     if ( NumberPtr == 0 && (KeyChar == '0' || KeyChar == '.')) {
094     {
095         disp_buffer_P[NumberPtr ++ ] = '0'; KeyChar = '.';
096         disp_buffer_P[NumberPtr ++ ] = '.';
097     }
098     else
099     {
100         //否则正常存入新输入字符
101         disp_buffer_P[NumberPtr ++ ] = KeyChar;
102     }
103     disp_buffer_P[NumberPtr] = '\0';      //加串终点标志
104     LCD_ShowString(3,0,disp_buffer_P); //刷新显示价格
105 }
106
107 //遇到小数点且此前未输入过小数点则开始限定可输入的小数位
108 if (KeyChar == '.' &&!havedot)
109 {
110     dtnum = NumberPtr + 2;
111     havedot = 1;
112 }
113 }
114 //清除当前所输入的价格
115 //-----
116 else if (KeyChar == 'C')
117 {
118     NumberPtr = 0;                      //disp_buffer_P 数字缓冲指针归 0
119     havedot = 0;                        //小数点输入标志清零
120     dtnum = PLEN - 2;                  //复位小数点后可输入字符个数
121
122 //清除价格及金额输出缓冲
123 disp_buffer_P[0] = '\0';
124 disp_buffer_SUM[0] = '\0';
125
126 LCD_ShowString(3,0,"      ");        //输出 6 个空格,清除价格
127 LCD_ShowString(7,1,"      ");        //输出 9 个空格,清除金额
128 }
129 if (Key_Pressed) Compute_and_show_sum(); //计算并显示总金额
130 while (Key_Pressed);                  //等待按键释放
131 }

```



```
132 }

01 //----- main.c -----
02 // 名称：电子秤仿真设计
03 //-----
04 // 说明：本例运行时，LCD 显示当前压力（未转换为质量），所输入的价格将
05 // 直接与该值相乘，LCD 显示计算后的应付金额
06 //
07 //-----
08 #define F_CPU 1000000UL
09 #include <avr/io.h>
10 #include <avr/pgmspace.h>
11 #include <util/delay.h>
12 #include <stdio.h>
13 #include <string.h>
14 #include <stdlib.h>
15 #include <ctype.h>
16 #define INT8U unsigned char
17 #define INT16U unsigned int
18
19 //蜂鸣器输出
20 extern void Sounder();
21 //液晶相关函数
22 extern void Initialize_LCD();
23 extern void LCD_ShowString(INT8U x, INT8U y, char * str);
24 //键盘处理及金额计算与显示函数
25 extern void KeyBoard_Handle();
26 extern void Compute_and_show_sum();
27 //液晶显示缓冲(质量)
28 extern char disp_buffer_W[];
29 //模/数转换结果及压力换算结果
30 volatile INT16U AD_Result, Pre_Result = 0, Pressure_Value;
31 //-----
32 // 主程序
33 //-----
34 int main()
35 {
36     DDRA = 0x00; PORTA = 0xFF; //配置端口
37     DDRB = 0xFF;
38     DDRC = 0xFF;
39     DDRD = 0xFF;
40     Initialize_LCD(); //初始化 LCD
41     LCD_ShowString(0, 0, "P:"); //第一行显示 P:W:标志(价格/质量)
```

```

42 //其中"P:"后面空 8 格
43 LCD_ShowString(0,1,"SUM( $ ):");
44 ADMUX = 0x00;
45 ACSR = _BV(ACD);
46 ADCSRA = 0xC7;
47 _delay_ms(1000);
48
49 while(1)
50 {
51     //开始 A/D 转换(SC:Start Conversion)
52     ADCSRA |= _BV(ADSC);
53     //读取转换结果(10 位精度,获取的值为 0~1023)
54     AD_Result = ADC;
55     //根据 MPX4250 技术手册,经下面的公式换算出当前压力
56     Pressure_Value = (AD_Result * 5.0 / 1023.0 / 5.1 - 0.04) / 0.00369 - 3.45;
57     //-----
58     //... 在这里可根据压力到质量的转换公式再次进行转换
59     //-----
60
61     //转换显示结果
62     sprintf(disp_buffer_W, "%-d", Pressure_Value);
63     LCD_ShowString(13,0,"      ");           //输出 4 个空格,清除质量
64     LCD_ShowString(13,0,disp_buffer_W);
65     KeyBoard_Handle();                     //处理键盘操作
66     if (Pre_Result != AD_Result)          //压力变化则计算金额
67     {
68         Compute_and_show_sum();
69         Pre_Result = AD_Result;
70         Sounder();
71     }
72     _delay_ms(100);
73 }
74 }
```

## 5.28 模拟射击训练游戏

本例在 PG160128 液晶屏上模拟射击训练游戏程序,程序启动时液晶屏显示游戏封面,随后显示射击游戏区域。游戏默认提供弹药 20 发,K1 与 K2 键用于上下移动枪支位置,以便跟踪随机移动的被射击目标,按下 K3 时发射并输出逼真的模拟枪声,每次发射时如果击中则加 1 分,弹药用完后可按下 K4 重新开始。本例电路及程序运行效果如图 5-31 所示。

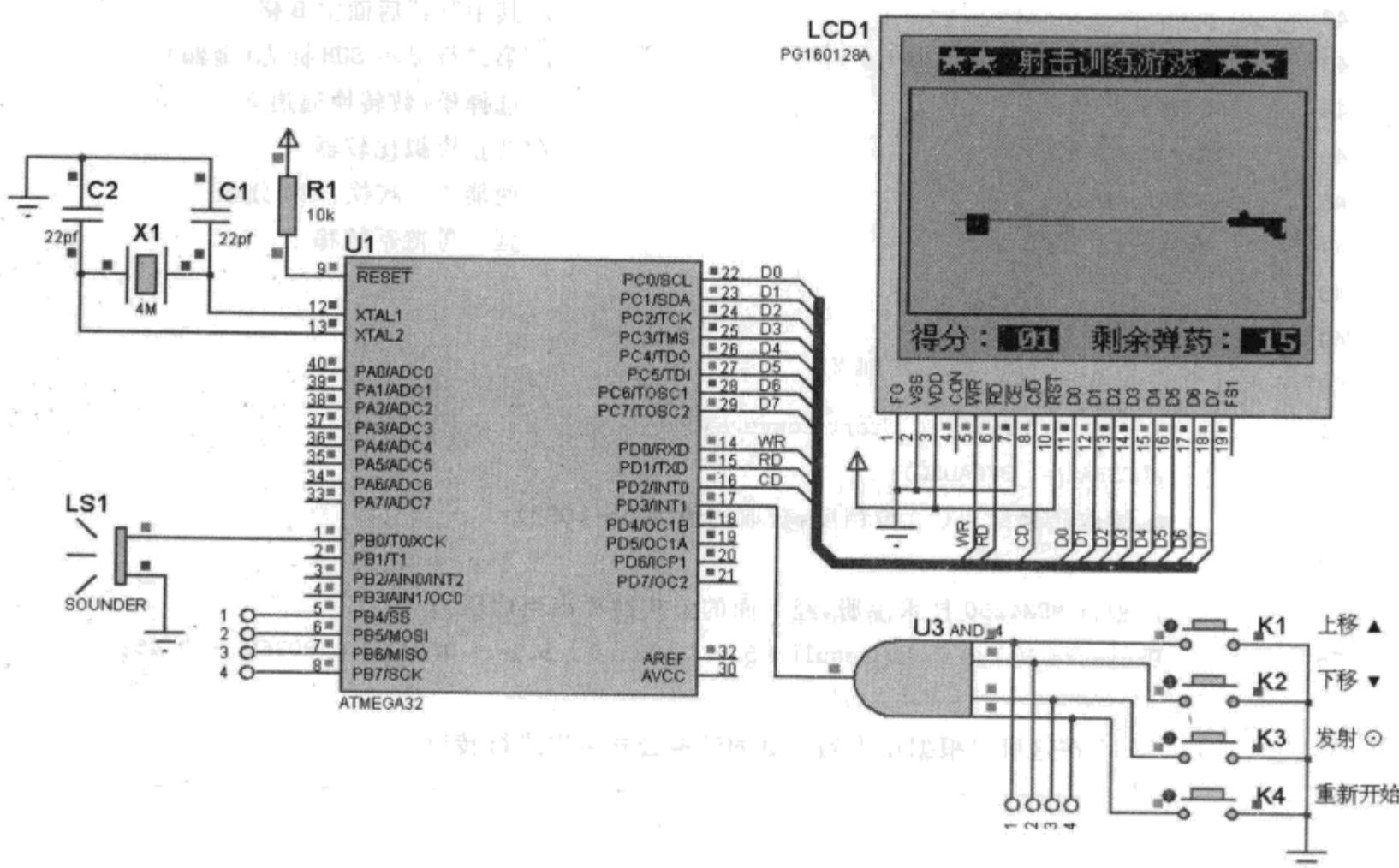


图 5-31 模拟射击训练游戏

## 1. 程序设计与调试

本例设计要点在于目标物体的随机移动、枪支在按键控制下的上下移动和击中判断以及枪声的模拟输出。下面逐一进行讨论：

### (1) 目标物体的随机移动

为控制目标物体随机移动，程序中启用了 T/C0 定时器溢出中断，定时器控制目标物体每隔 1.5 s 移动一次，移动之前先根据目标物体当前的横纵坐标 (Target\_x, Target\_y) 清除处于当前位置的目标物体，然后再用随机函数 random 生成新的坐标 (Target\_x, Target\_y)，并在新的位置绘制目标物体。目标物体上次所处纵坐标位置由 INT1 中断中的 Pre\_Target\_y 备份，为形成明显的随机移动感，当本次纵坐标值与上次纵坐标值之差小于 4 时，定时器中断函数内的 while 循环会反复获取新的随机纵坐标 Target\_y，直到满足条件为止。

### (2) 4 个操作按键的处理

本例 4 个按键通过 4 输入的与门触发 INT1 下降沿中断，在 INT1 中断函数内完成按键处理。其中前 2 个按键通过修改枪支的纵坐标 gun\_y(分别 -8/+8) 来上下移动枪支。

中断函数内最主要的部分在于按下 K3 时的发射功能设计，每次按下 K3 时，通过允许 T/C1 溢出中断来模拟输出逼真的枪声，然后再绘制和清除弹道，并递减弹药。对于 K3 按键，最重要的是判断是否击中目标，击中判断可通过比较枪支与目标物体的纵坐标变量 gun\_y 与 Target\_y 来完成，函数内通过检查 gun\_y+4 是否处于 (Target\_y, Target\_y+11) 这个区间，如果处于这个区间则即被认为击中，其中“+4”是因为 gun\_y 是枪支的纵坐标，而弹道线的纵坐标为 gun\_y+4，击中判断就是检查弹道线纵坐标是否处于 Target\_y~Target\_y+11 这个

范围以内。

为避免同一物体在同一位置被多次击中而反复得分,这里还进一步引入变量 Pre\_Target\_y 来解决这个问题,每当上述击中条件成立,而目标物体纵坐标未改变时,Score 不再累加得分。

### (3) 枪声的模拟输出

如同本书多个案例使用定时器控制输出声音一样,本例使用了 T/C1 定时器溢出中断控制输出模拟枪声,T/C1 定时器溢出中断中定义的静态变量 Tcnt1x 初值为 0,Tcnt1x 使 T/C1 定时/计数寄存器 TCNT1 在中断触发过程中由 0xFFFF~0xFE50 递减取值,由于 T/C1 时钟为 4M/8,故蜂鸣器的输出频率范围为  $4M/8/(65536 - 0xFFFF)/2 \sim 4M/8/(65536 - 0xFE50)/2$ ,即 4M/256~4M/6912。

计算后可得频率范围为 15.6 kHz~579 Hz,人耳可听到的范围为 20 kHz~20 Hz,T/C1 定时器溢出中断模拟了由接近人耳可听到的最高频率到较低频率的快速衰减过程,从而输出了逼真的枪声。图 5-32 给出了使用 MathCad2000 生成的模拟输出的枪声频率的衰减曲线。

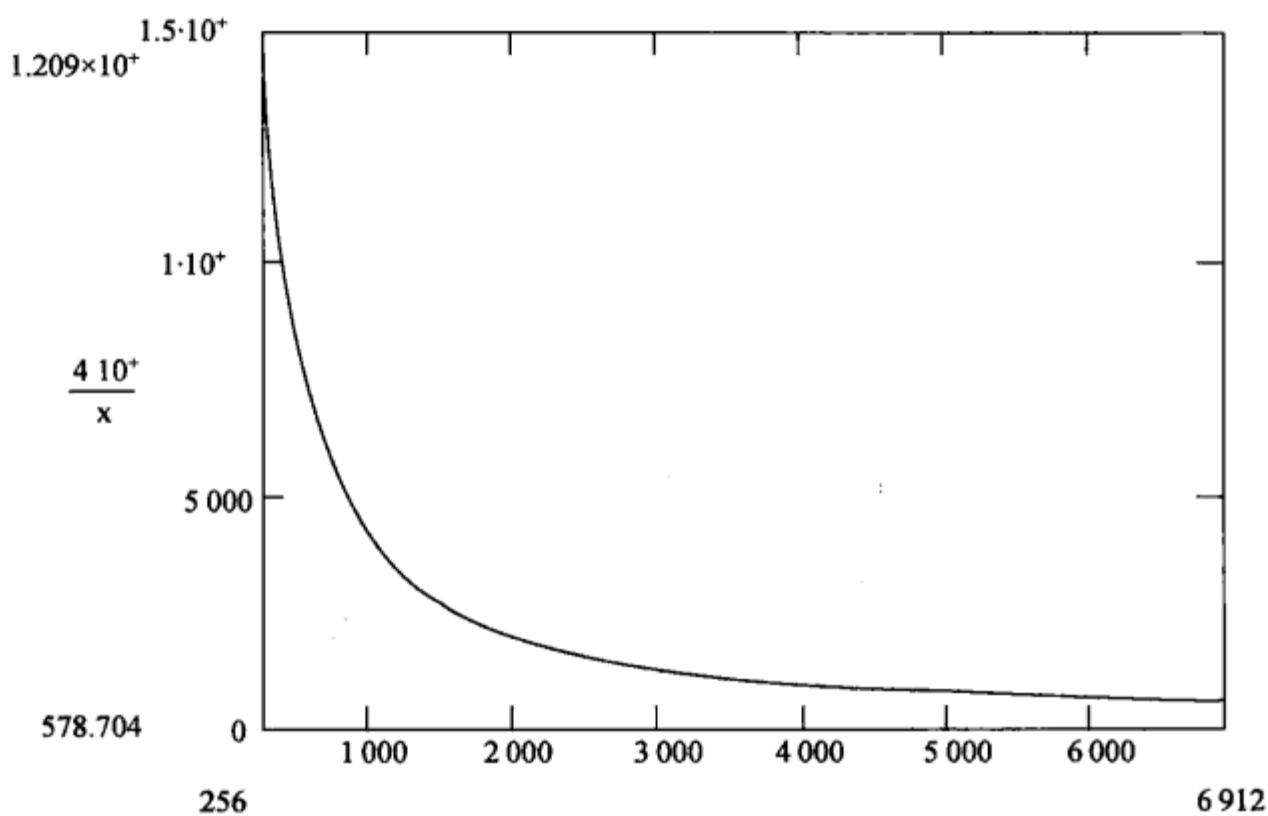


图 5-32 模拟输出的枪声频率的衰减曲线

## 2. 实训要求

① 本例直接用 GB-2312 字库中的全角字符“■”代替被射击目标,完成本例调试后重新绘制被射击目标的原始图形及被击中时的图形(例如飞碟等),改写本例程序,实现更逼真的射击游戏效果。

② 重新设计本例游戏模式,将原有矩阵区域内的水平射击改成枪支可在某个固定点对扇形区域的目标进行射击,原有的上下移动键改成射击角度加减键,在瞄准过程中允许根据当前枪支所指方向在屏幕上绘制虚线弹道。

③ 基于本例硬件环境设计太空入侵者(Space Invaders)游戏,编写时可参考 Proteus 所提供的 PIC 单片机汇编版的该套游戏的运行效果。

### 3. 源程序代码

```

001 //----- main.c -----
002 // 名称:射击训练游戏
003 //----- 
004 // 说明:程序启动时液晶屏显示游戏封面,然后显示游戏区,默认弹药为 20 发,
005 // K1、K2 键用于向上或向下移动枪支,跟踪目标,K3 用于发射并模拟枪声,
006 // 在每次发射时,如果击中则加 1 分,在击中后如果目标物体尚未移动时,
007 // 程序不重复加分. 弹药用完后可按下 K4 重新开始
008 //----- 
009 #define F_CPU 4000000UL
010 #include <avr/io.h>
011 #include <avr/pgmspace.h>
012 #include <avr/interrupt.h>
013 #include <util/delay.h>
014 #include <stdio.h>
015 #include <stdlib.h>
016 #include "PG160128.h"
017 #include "PictureDots.h"
018 #define INT8U unsigned char
019 #define INT16U unsigned int
020
021 //LCD 显示相关函数及相关变量
022 extern void Clear_Screen(); //清屏
023 extern INT8U LCD_Initialise(); //LCD 初始化
024 extern INT8U LCD_Write_Command(INT8U cmd); //写无参数的命令
025 //写双参数命令
026 extern INT8U LCD_Write_Command_P2(INT8U cmd, INT8U para1, INT8U para2);
027 extern INT8U LCD_Write_Data(INT8U dat); //写数据
028 extern void Set_LCD_POS(INT8U row, INT8U col); //设置当前地址
029 //绘制线条
030 extern void Line(INT8U x1, INT8U y1, INT8U x2, INT8U y2, INT8U Mode); //显示字符串(wb 设置反相)
031
032 extern void Display_Str_at_xy(INT8U x, INT8U y, char * Buffer, INT8U wb); //从指定位置开始显示图像
033
034 extern void Draw_Image(prog_uchar * G_Buffer, INT8U Start_Row, INT8U Start_Col);
035 extern INT8U LCD_WIDTH, LCD_HEIGHT; //LCD 宽度与高度
036 extern prog_uchar Game_Surface[], Gun_Image[]; //游戏封面与枪支图像点阵数组
037
038 //按键定义
039 #define K1_DOWN() (PINB & _BV(PB4)) == 0x00 //上移
040 #define K2_DOWN() (PINB & _BV(PB5)) == 0x00 //下移
041 #define K3_DOWN() (PINB & _BV(PB6)) == 0x00 //发射

```

```

042 #define K4_DOWN() (PINB & _BV(PB7)) == 0x00           //重新开始
043 //蜂鸣器
044 #define BEEP()    PORTB ^= _BV(PB0)
045
046 INT8U HCount = 0,LCount = 0;                         //控制模拟枪声的延时变量
047 INT8U Score = 0, Bullet_Count = 20;                  //得分,剩余弹药数
048 INT8U Target_x = 0, Target_y = 0;                    //目标物体位置
049 INT8U Pre_Target_y = 0;                             //目标物体上次所在纵坐标位置
050 INT8U gun_y = 20;                                  //枪支纵坐标(其中横坐标固定为 16 * 8)
051 //-----
052 // 显示成绩与剩余弹药数
053 //-----
054 void Show_Score_and_Bullet()
055 {
056     char dat_str[4] = {' ',0,0,0};
057     //显示成绩
058     dat_str[1] = Score / 10 + '0';
059     dat_str[2] = Score % 10 + '0';
060     Display_Str_at_xy(37,117,dat_str,1);
061
062     //显示剩余弹药数
063     dat_str[1] = Bullet_Count / 10 + '0';
064     dat_str[2] = Bullet_Count % 10 + '0';
065     Display_Str_at_xy(134,117,dat_str,1);
066 }
067
068 //-----
069 // 键盘中断(INT1)
070 //-----
071 ISR (INT1_vect)
072 {
073     //枪支位置上移-----
074     if (K1_DOWN())
075     {
076         if (gun_y != 0) Display_Str_at_xy(16 * 8,gun_y," ",0);
077         gun_y -= 8;
078         if (gun_y < 20 ) gun_y = 20;
079         Draw_Image(Gun_Image,gun_y,16);
080     }
081     //枪支位置下移-----
082     else if (K2_DOWN())
083     {
084         if (gun_y != 0) Display_Str_at_xy(16 * 8,gun_y," ",0);

```

```

085
086     gun_y += 8;
087     if (gun_y > 100) gun_y = 100;
088     Draw_Image(Gun_Image, gun_y, 16);
089 }
090 //发射,绘制与擦除弹道线条,模拟枪声,判断成绩-----
091 else if (K3_DOWN())
092 {
093     //如果有剩余弹药则为T/C1提供8分频时钟(设0x02),模拟枪声输出,否则直接返回
094     if (Bullet_Count != 0) TCCR1B = 0x02; else return;
095     //绘制弹道线条
096     Line(10, gun_y + 4, 125, gun_y + 4, 1);
097     _delay_ms(80);
098     //80ms后擦除弹道线条
099     Line(10, gun_y + 4, 125, gun_y + 4, 0);
100    //弹药递减
101    if (Bullet_Count != 0)
102    {
103        Bullet_Count--;
104        //判断成绩
105        //Pre_Target_y用于保存目标物体上次所在纵坐标位置
106        //避免物体在同一位置被反复多次击中而多次得分
107        if ((gun_y + 4) > Target_y && (gun_y + 4) < Target_y + 11 &&
108            Pre_Target_y != Target_y)
109        {
110            Score++; Pre_Target_y = Target_y;
111        }
112    }
113    //刷新显示成绩与弹药数
114    Show_Score_and_Bullet();
115 }
116 //成绩与弹药数复位-----
117 else if (K4_DOWN())
118 {
119     Score = 0; Bullet_Count = 20;
120     Show_Score_and_Bullet();
121 }
122 }
123
124 //-----
125 // 定时器0溢出中断控制目标物体随机移动
126 //-----
127 ISR (TIMERO_OVF_vect)

```

```

128 {
129     static INT8U tCount = 0;
130     TCNT0 = 256 - F_CPU / 1024.0 * 0.05;
131     //累加延时 30 * 0.05s
132     if (++tCount != 30) return;
133     tCount = 0;
134
135     //清除原位置目标物体
136     if (Target_x != 0 && Target_y != 0)
137         Display_Str_at_xy(Target_x, Target_y, " ", 0);
138     Target_x = random() % 60 + 8;      //随机生成新坐标位置
139     Target_y = random() % 80 + 20;
140     while (abs(Pre_Target_y - Target_y) < 4)
141     {
142         Target_y = random() % 80 + 20;
143     }
144     //在新位置绘制目标物体
145     Display_Str_at_xy(Target_x, Target_y, "■", 0);
146 }
147
148 //-----
149 // 定时器1溢出中断模拟枪声输出
150 //-----
151 ISR (TIMER1_OVF_vect)
152 {
153     static INT16U Tcnt1x = 0x0000;
154     BEEP();
155     TCNT1 = --Tcnt1x;
156     if (Tcnt1x == 0xFE50)
157     {
158         //重设定时初值变量
159         Tcnt1x = 0x0000;
160         //T/C1 无时钟,枪声输出停止
161         TCCR1B = 0x00;
162     }
163 }
164
165 //-----
166 // 主程序
167 //-----
168 int main()
169 {
170     DDRB = 0x0F; PORTB = 0xFF;           //配置端口

```



```
171     DDRC = 0xFF;
172     DDRD = 0xF7;
173     LCD_Initialise();           //液晶初始化
174     Clear_Screen();            //清屏
175     Draw_Image(Game_Surface,6,0); //显示游戏封面
176     _delay_ms(3000);
177     Clear_Screen();            //随后清除封面
178     //显示固定文字
179     Display_Str_at_xy(12,1,"★★ 射击训练游戏 ★★",1);
180     Display_Str_at_xy(2,117,"得分:",0);
181     Display_Str_at_xy(75,117,"剩余弹药:",0);
182     Show_Score_and_Bullet();    //显示成绩与弹药
183     Line(0,18,159,18,1);       //用 4 条直线绘制游戏区边框
184     Line(159,18,159,112,1);
185     Line(159,112,0,112,1);
186     Line(0,112,0,18,1);
187     Draw_Image(Gun_Image,gun_y,16); //在初始位置绘制枪支
188     //设置 T/C0,T/C1,INT1 中断
189     TCCR0 = 0x05;              //T/C0 预分频:1024
190     TCNT0 = 256 - F_CPU / 1024.0 * 0.05; //T/C0 在 4 MHz/1024 时钟下设置0.05 s定时
191     TCCR1B = 0x00;             //T/C1 无时钟(枪声无输出)
192     //允许 T/C0,T/C1 定时器溢出中断
193     //前者控制目标随机移动,后者模拟枪声输出,初始时无输出
194     TIMSK = _BV(TOIE0) | _BV(TOIE1);
195     MCUCR = 0x08;              //INT1 为下降沿触发
196     GICR = _BV(INT1);         //INT1 中断使能
197     SREG = 0x80;               //使能中断
198     while(1);
199 }
```

```
001 //----- LCD_160128.c -----
002 // 名称: PG12864LCD 显示驱动程序(T6963C) (不带字库)
003 //-----
.....限于篇幅,这里省略了部分类似代码
158 //本例汉字点阵库
159 const struct typFNT_GB16 GB_16[] = { //12 * 12 点阵,宋体小五号,用 Zimo 软件取得点阵
160 {{"得"},{0x27,0xC0,0x24,0x40,0x57,0xC0,0x94,0x40,0x27,0xC0,0x60,0x00,
161           0xAF,0xE0,0x20,0x80,0x2F,0xE0,0x24,0x80,0x21,0x80,0x00,0x00}},
162 {{"分"},{0x11,0x00,0x11,0x00,0x20,0x80,0x20,0x80,0x40,0x40,0xBF,0xA0,
163           0x08,0x80,0x08,0x80,0x10,0x80,0x20,0x80,0xC7,0x00,0x00,0x00}},
164 {{":"}, {0x00,0x00,0x00,0x00,0x0C,0x00,0x0C,0x00,0x00,0x00,0x00,0x00,
165           0x00,0x00,0x0C,0x00,0x0C,0x00,0x00,0x00,0x00,0x00,0x00,0x00}},
166 {{"★"}, {0x04,0x00,0x04,0x00,0x0E,0x00,0x0E,0x00,0xFF,0xE0,0x7F,0xC0,
```

```

167         0x1F,0x00,0x1F,0x00,0x3B,0x80,0x20,0x80,0x40,0x40,0x00,0x00} },
168  {{"■"},{0x00,0x00,0x7F,0xC0,0x7F,0xC0,0x7F,0xC0,0x7F,0xC0,
169      0x7F,0xC0,0x7F,0xC0,0x7F,0xC0,0x7F,0xC0,0x00,0x00,0x00,0x00}},
170
171  {"射"},{0x20,0x40,0x78,0x40,0x48,0x40,0x7F,0xE0,0x48,0x40,0x7A,0x40,
172      0x49,0x40,0xF9,0x40,0x28,0x40,0x48,0x40,0x99,0xC0,0x00,0x00}},
173  {"击"},{0x04,0x00,0x04,0x00,0x7F,0xC0,0x04,0x00,0x04,0x00,0xFF,0xE0,
174      0x04,0x00,0x44,0x40,0x44,0x40,0x44,0x40,0x7F,0xC0,0x00,0x00}},
175  {"训"},{0x44,0x40,0x25,0x40,0x05,0x40,0x05,0x40,0xC5,0x40,0x45,0x40,
176      0x45,0x40,0x45,0x40,0x55,0x40,0x68,0x40,0x10,0x40,0x00,0x00}},
177  {"练"},{0x22,0x00,0x4F,0xE0,0x42,0x00,0x9F,0x80,0xE4,0x80,0x44,0x80,
178      0xAF,0xE0,0xC0,0x80,0x34,0xC0,0xA0,0x13,0xA0,0x00,0x00}},
179  {"游"},{0x91,0x00,0x49,0xE0,0x3E,0x00,0x93,0xE0,0x5C,0x40,0x54,0x80,
180      0x55,0xE0,0x94,0x80,0x94,0x80,0xA4,0x80,0x4D,0x80,0x00,0x00}},
181  {"戏"},{0x02,0x80,0xF2,0x40,0x12,0x40,0x13,0xE0,0x9E,0x00,0x52,0x40,
182      0x22,0x80,0x31,0x00,0x49,0x20,0x42,0xA0,0x8C,0x60,0x00,0x00}},
183
184  {"剩"},{0x7C,0x20,0x10,0xA0,0xFE,0xA0,0x54,0xA0,0xD6,0xA0,0x54,0xA0,
185      0xD6,0xA0,0x38,0xA0,0x54,0xA0,0x92,0x20,0x10,0xE0,0x00,0x00}},
186  {"余"},{0x04,0x00,0x0A,0x00,0x11,0x00,0x20,0x80,0xDF,0x60,0x04,0x00,
187      0x7F,0xC0,0x15,0x00,0x24,0x80,0x44,0x40,0x9C,0x40,0x00,0x00}},
188  {"弹"},{0x04,0x40,0xE2,0x80,0x2F,0xC0,0x29,0x40,0xEF,0xC0,0x89,0x40,
189      0xEF,0xC0,0x21,0x00,0x3F,0xE0,0x21,0x00,0xC1,0x00,0x00,0x00}},
190  {"药"},{0x11,0x00,0xFF,0xE0,0x11,0x00,0x22,0x00,0x4B,0xE0,0x74,0x20,
191      0x22,0x20,0x59,0x20,0x61,0x20,0x18,0x40,0xE1,0xC0,0x00,0x00}}
192  };

```

.....限于篇幅,这里省略了部分类似代码

```

598
599 //-----
600 // 绘制图像(图像数据来自于 Flash 程序 ROM 空间)
601 //-----
602 void Draw_Image(prog_uchar * G_Buffer, INT8U Start_Row, INT8U Start_Col)
603 {
604     INT16U i,j,W,H;
605     //图像行数控制(G_Buffer 的前 2 个字节分别为图像宽度与高度)
606     W = pgm_read_byte(G_Buffer + 1);
607     for(i = 0;i < W;i++)
608     {
609         Set_LCD_POS(Start_Row + i,Start_Col);
610         LCD_Write_Command(LC_AUT_WR);
611
612         //绘制图像每行像素
613         H = pgm_read_byte(G_Buffer);

```

```

614     for( j = 0; j < H / 8; j ++ )
615         LCD_Write_Data(pgm_read_byte(G_Buffer + i * ( H / 8 ) + j + 2));
616         LCD_Write_Command(LC_AUT_OVR);
617     }
618 }

001 //----- PictureDots.h -----
002 // 游戏封面数据存放于 Flash 程序空间
003 //-----
004 prog_uchar Game_Surface[] = { 160,110, //游戏封面:160 * 110
005 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
006 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
.....限于篇幅,这里省略了游戏封面的大部分点阵数据
141 0x00,0xF8,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x80,0x00,0xE0,
142 0x00,0x00,0xC0,0x00,0x00,0x20,0x00,0x00
143 };
144
145 //-----
146 // 枪支图像数据存放于 Flash 程序空间
147 //-----
148 prog_uchar Gun_Image[] = { 24,12, //枪支图像,W/H:24 * 12
149 0x03,0x00,0x00,0x07,0x80,0x00,0x07,0x80,0x00,0x7F,0xFF,0xFE,0xFF,0xFF,0xFF,0xFF,
150 0xFF,0xFC,0x7F,0xFF,0xFC,0x00,0x01,0xFC,0x00,0x01,0xFC,0x00,0x00,0x7F,0x00,0x00,
151 0x7F,0x00,0x00,0x1F
152 };

```

## 5.29 PC 机通过 485 远程控制单片机

基于 RS - 232 总线的串行通信系统设计简单、易于使用,但其传输速度慢、传输距离短、易受外界电气干扰的缺点,使其无法完全满足工业应用系统的需求。RS - 485 总线作为其替代标准,在工业应用系统中地位十分重要,它在通信速率、传输距离、多机连接等方面均有很大提高。本例运行时,通过上位机软件(本例直接使用串口助手软件)发送的数字将通过 485 总线传输给单片机,显示在 4 位数码管上。本例电路及部分运行效果如图 5 - 33 所示。

### 1. 程序设计与调试

RS - 485 总线采用平衡发送和差分接收方式实现通信,发送端将串口的 TTL 电平信号转换成差分信号由 A、B 两路输出,接收端再将差分信号还原成 TTL 电平信号。由于传输线通常使用双绞线,且采用差分方式传输,因而具有很强的抗共模干扰能力,总线收发器灵敏度很高,可以检测到低至 200 mV 的电压。

RS - 485 最大通信距离约为 1219 m,最大传输速率为 10 Mb/s,要传输更长距离时,需添加 485 中继器。

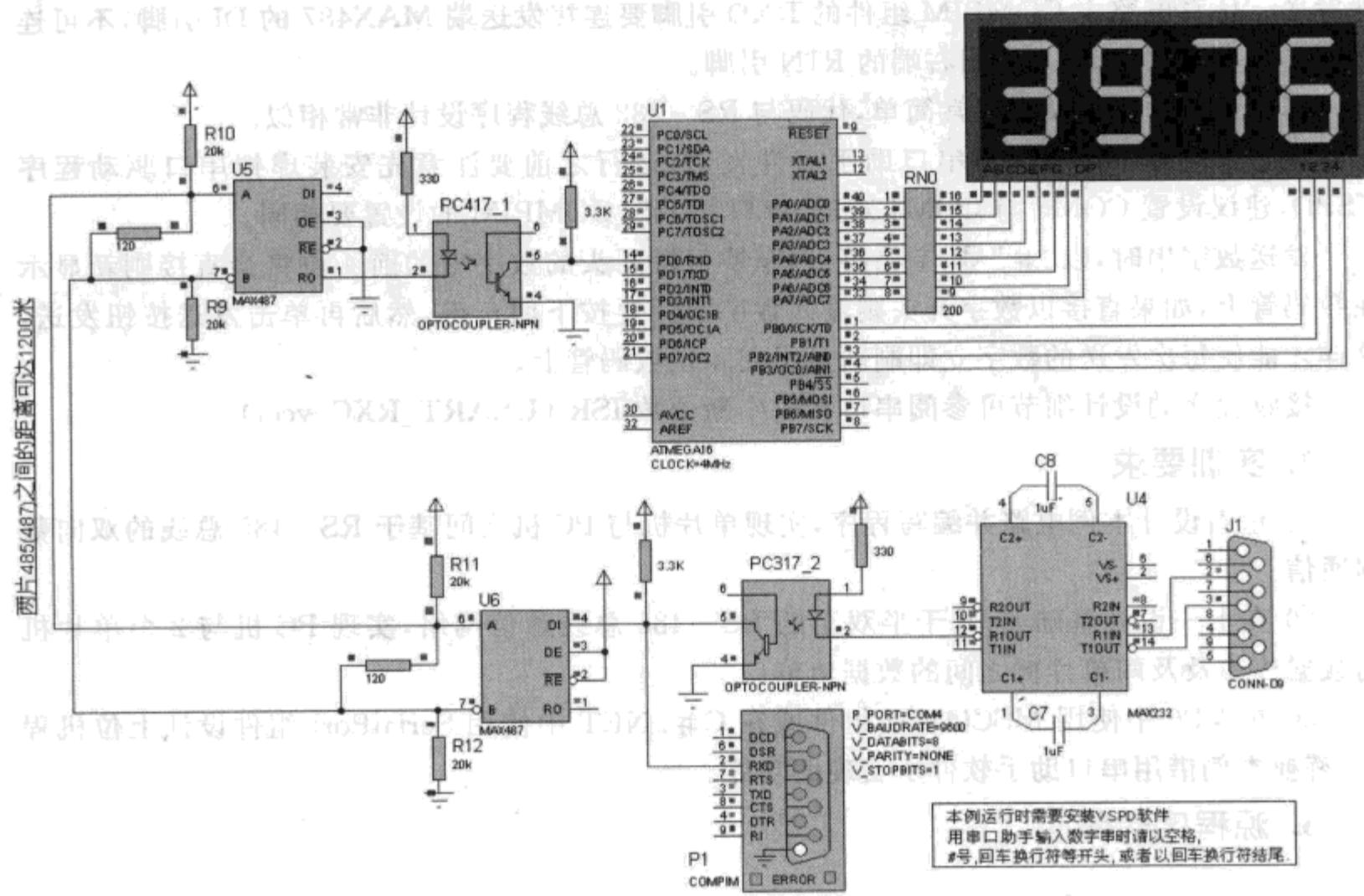


图 5-33 PC 机通过 485 远程控制单片机

RS-485 采用半双工工作方式，支持多点数据通信，总线拓扑结构多采用终端匹配的总线型结构，即采用一条总线将各个节点串接起来，RS-485 总线一般最大支持 32 个节点。

本例所使用的 MAX487 芯片是 MAXIM 公司生产的用于 RS-485 总线通信的低功耗半双工收发器件，芯片内集成了一个驱动器和一个接收器，符合 RS-422A 和 RS-485 总线通信标准。

下面是 MAX487 芯片的引脚功能：

RO——接收器输出(Receiver Output);

DI——驱动器输入(Driver Input);

**RE**——接收器输出使能(Receiver Output Enable),低电平允许,高电平禁止;

DE——驱动器输出使能(Driver Output Enable),高平电允许,低电平禁止;

A——接收器非反向输入端和驱动器非反向输出端；

B——接收器反向输入端和驱动器反向输出端。

本例以 PC 机为主控机,通过 RS-232/485 转换电路接入 RS-485 总线,由于本例 PC 机仅使用 485 总线向单片机单向发送数据,发送端 MAX487 的 DE、RE 引脚固定连接高电平,使能发送;接收端的 DE、RE 固定连接低电平,使能接收。

为避免干扰,收发双方均使用了光耦隔离器件 PC417,在实物电路中 PC417 的引脚 6 接高电平。另外,由于 Proteus 组件限制,本例未进一步在电路添加 DC-DC 电源隔离模块,在实际应用设计中建议添加电源隔离模块。

在实物电路图中,PC 机串口连接 CONN - D9 发送数据,在仿真时数据通过 COMPIM 组

件发送。仿真电路中 COMPIM 组件的 RXD 引脚要连接发送端 MAX487 的 DI 引脚,不可连接到本例电路中 MAX232 的右端的 R1N 引脚。

本例电路的程序设计非常简单,代码与 RS-232 总线程序设计非常相似。

运行本例时,PC 端使用串口助手软件发送,运行之前要注意先安装虚拟串口驱动程序 VSPD,建议设置 COM3 与 COM4 对连,串口助手与 COMPIM 的设置要相同。

发送数字串时,以“#”号、空格符、回车换行符开头的数字串的前 4 位将会直接刷新显示在数码管上,如果直接以数字开头则输入数字串后要按下回车键,然后再单击发送按钮发送,这样才能使每次发送的数字立即刷新显示在 4 位数码管上。

接收程序的设计细节可参阅串口接收中断函数 ISR (USART\_RXC\_vect)。

## 2. 实训要求

① 重新设计本例电路并编写程序,实现单片机与 PC 机之间基于 RS-485 总线的双向数据通信。

② 在上一设计基础上,基于半双工的 RS-485 总线通信网络,实现 PC 机与 2 个单片机的数据传输及两单片机之间的数据传输。

③ 在 VB6 中使用 MSComm 组件或在 C#.NET 中使用 SerialPort 组件设计上位机程序,替换本例借用串口助手软件所实现的功能。

## 3. 源程序代码

```

01 //-----
02 // 名称: 基于 485 通信的单片机程序
03 //-----
04 // 说明: 本例运行时,PC 机通过串口发送的数字串的前 4 位将显示在数码管上
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 #define MAXLEN 4                  //可接收数字的最大个数
15 struct                      //数字串接收缓冲结构类型
16 {
17     INT8U Buf_array[MAXLEN];    //缓冲空间
18     INT8U Buf_Len;             //当前缓冲长度
19 } Receive_Buffer;            //接收缓冲结构类型变量
20
21 INT8U Clear_Buffer_Flag = 0; //清空缓冲标志
22
23 //共阳数码管 0~9 的段码表,最后一位为黑屏

```

```

24 const INT8U SEG_CODE[] =
25 {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90,0x00};
26
27 //-----
28 // USART 初始化
29 //-----
30 void Init_USART()
31 {
32     UCSRB = _BV(RXEN) | _BV(RXCIE);           //允许接收,接收中断使能
33     UCSRC = _BV(URSEL)| _BV(UCSZ1)| _BV(UCSZ0); //8位数据位,1位停止位
34     UBRRH = (F_CPU / 9600 / 16 - 1) % 256;      //波特率:9600
35     UBRLR = (F_CPU / 9600 / 16 - 1) / 256;
36 }
37
38 //-----
39 // 显示所接收数字串的前 4 位(数字串由 PC 串口发送,单片机串口接收)
40 //-----
41 void Show_Received_Digits()
42 {
43     INT8U i;
44     for (i = 0 ; i < 4; i++)
45     {
46         PORTB = _BV(i);                         //发送位码
47         PORTA = SEG_CODE[ Receive_Buffer.Buf_array[i] ]; //发送段码
48         _delay_ms(4);
49     }
50 }
51
52 //-----
53 // 主程序
54 //-----
55 int main()
56 {
57     Receive_Buffer.Buf_Len = 0;                //缓冲长度清零
58     DDRA = 0xFF; PORTA = 0xFF;                 //配置端口
59     DDRB = 0xFF; PORTB = 0x00;
60     DDRD = 0xF6; PORTD = 0xFF;
61     Init_USART();                            //串口初始化
62     sei();                                //开中断
63     while(1) Show_Received_Digits();        //显示所接收到数字
64 }
65
66 //-----

```



```
67 // 串口接收中断函数(接收 PC 机数据)
68 //-----
69 ISR (USART_RXC_vect)
70 {
71     INT8U c = UDR;
72     //接收到回车符、换行符、空格符或"#"字符则设置清空缓冲标志
73     if (c == '\r' || c == '\n' ||
74         c == ' ' || c == '#')    Clear_Buffer_Flag = 1;
75     if (c >= '0' && c <= '9')
76     {
77         //如果收到清空缓冲标志,则本次从缓冲开始位置存放
78         if (Clear_Buffer_Flag == 1)
79         {
80             Receive_Buffer.Buf_Len = 0;
81             Clear_Buffer_Flag = 0;
82         }
83         //如果缓冲未满则存入新数字
84         if (Receive_Buffer.Buf_Len < MAXLEN)
85         {
86             //缓存新接收的数字(由 ASCII 字符转换为数字)
87             Receive_Buffer.Buf_array[Receive_Buffer.Buf_Len] = c - '0';
88             //刷新缓冲长度(不超过最大长度)
89             Receive_Buffer.Buf_Len++;
90         }
91     }
92 }
```

## 5.30 用 IE 访问 AVR+RTL8019 设计的以太网应用系统

Ethernut 是一个开放硬件和软件设计方案的嵌入式系统协议栈的统称,德国 egnite Software GmbH 公司负责 Ethernut 的硬件与软件升级,在网站 <http://www.ethernut.de> 可找到 Ethernut 的全部硬件及软件源代码,网站还提供了大量实例供参考,用户可以方便地对其进行增删或修改,定制出适合自己的以太网(Ethernet)解决方案。

运行本例时,通过 IE 浏览器访问以 Atmega128 与 RTL8019 为核心设计的以太网应用系统,通过单击所设计 WEB 页中的超链接,可实现 LED 状态查询、电机启/停控制、admin 用户密码设置等功能。本例电路如图 5-34 所示。

### 1. 程序设计与调试

本例基于 Ethernut-4.9.7 的 http 应用案例进行设计,设计与调试涉及 6 个方面的内容:

#### (1) 以太网控制器 RTL8019AS 简介

RTL8019AS 是一种全双工即插即用的以太网控制器,它在一块芯片上集成了 RTL8019

内核和一个 16 KB 的 SDRAM 存储器, 兼容 RTL8019 控制软件和 NE2000 8-bit 或 16-bit 的传输, 支持 UTP、AUI、BNC 和 PNP 自动检测模式, 支持外接闪存读/写操作, 支持 I/O 地址的完全解码, 支持 4 个诊断 LED 可编程输出。其接口符合 Ethernet2 和 IEEE802.3 (10Base5、10Base2、10BaseT) 标准。

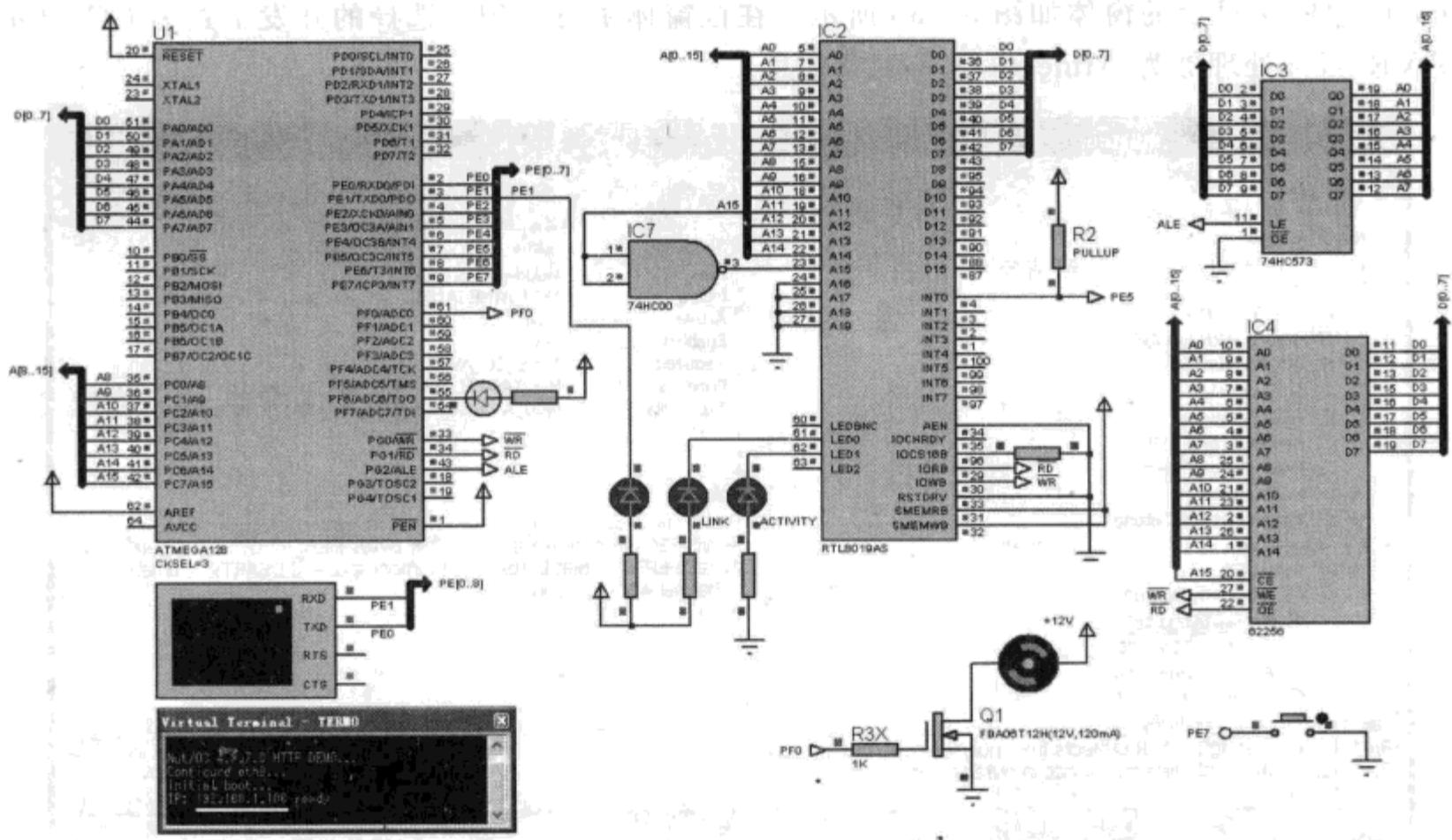


图 5-34 用 IE 访问 AVR+RTL8019 设计的以太网应用系统

以太网控制器 RTL8019AS 采用 100 脚 PQFP 封装, 其主要引脚说明如下:

- A0~A19——地址总线;
- D0~D15——数据总线;
- INT0~INT7——中断控制总线;
- AEN——地址使能端;
- SMEMRB——存储器读控制;
- SMEMWB——存储器写控制;
- LED0——网络通信冲突指示;
- LED1——RX 接收数据指示;
- LED2——TX 发送数据指示;
- LEDBNC——介质类型指示;
- IOCS16B——16 位 I/O 口方式;
- IORB——端口读控制;
- IOWB——端口写控制;
- RSTDRV——复位驱动器。

## (2) Nut/OS 的安装、配置与编译

进行本例开发设计之前, 需要首先从 [www.ethernut.de](http://www.ethernut.de) 下载安装包 ethernut-4.9.7.exe



并安装，默认安装目录为 c:\etternut - 4.9.7。

完成安装后，安装程序会接着自动提示否运行配置工具软件 configurator，如果忽略该操作，也可以在开始程序菜单中的“Ethernut. 4.9.7”下运行 configurator 工具。

在配置工具软件中单击 open 菜单，打开 C:\etternut - 4.9.7\nut\conf\ethernut13h.conf 文件，所显示的窗体如图 5-35 所示。在该窗体中可看到所选择的开发工具为 GCC for AVR，目标处理器为 Atmega128 等配置信息。

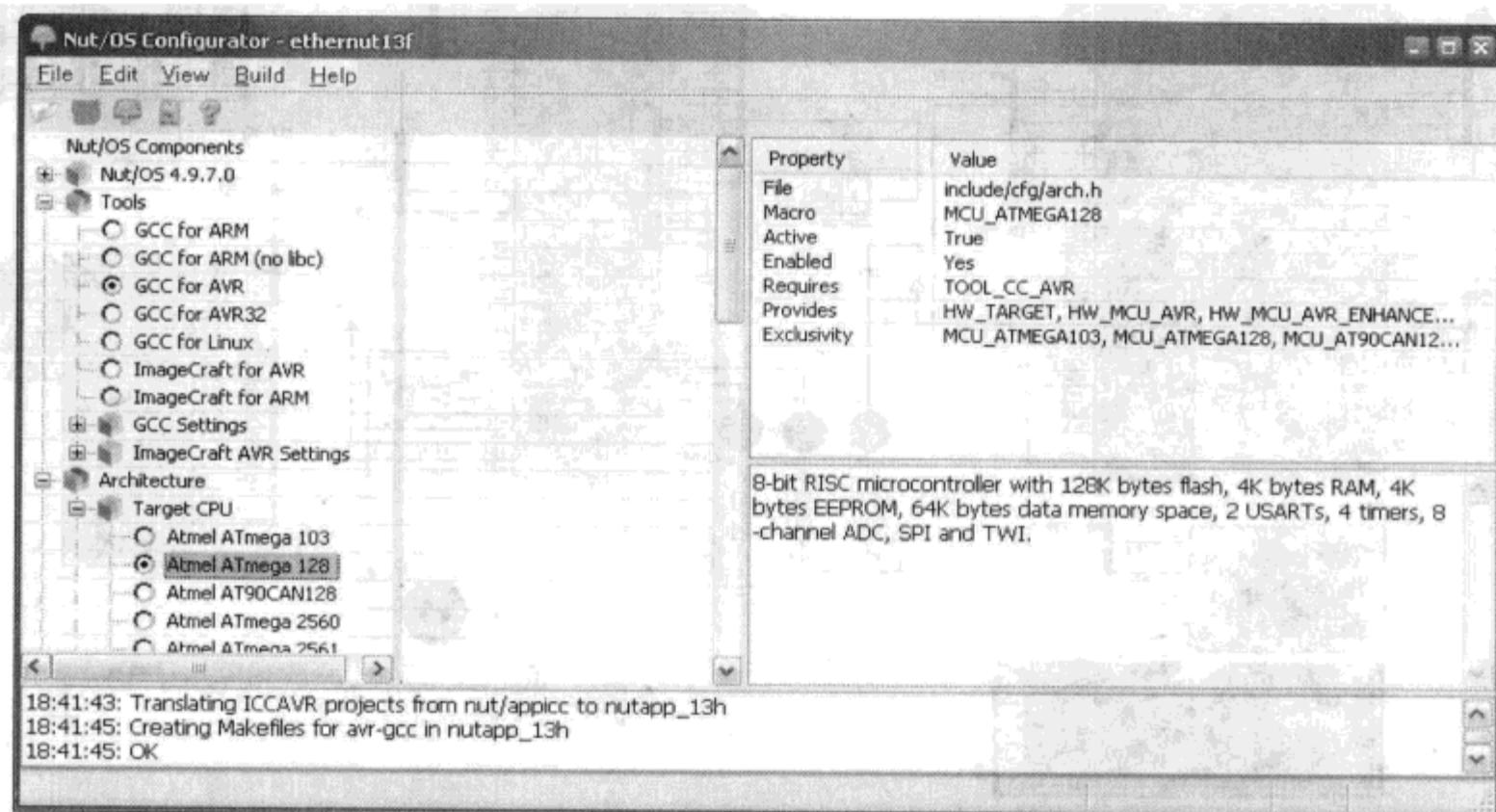


图 5-35 Nut/OS 配置

接下来单击 Edit→Settings，打开 NutConf Settings 窗体，在该窗体中完成如下设置：

- ① Repository 中的库文件(Repository File)默认为 nut\conf\repository.nut；
- ② Build 中的源目录 (Source Directory) 默认为 nut，平台 (Platform) 选择 avr/gcc，构建目录 (Build Directory) 设为 nutbld\_13h；
- ③ Tools 中的工具目录 (Tool Directory) 默认为 C:\etternut - 4.9.7\nut\tools\win32；
- ④ Samples 中的应用案例目录 (Application Directory) 设为 nutapp\_13h，编程器软件 (Programmer) 默认选择 avr - dude。

完成上述设置后单击 OK，然后再单击菜单 Build→Build Nut/OS 构建应用于本目标系统的 Nut/OS，完成后再单击 Build→Create Sample Directory 生成案例文件夹。

完成上述操作后，c:\etternut - 4.9.7 下会出现两个文件夹，它们分别是 nutbld\_13h 和 nutapp\_13h，前者是所生成的目标系统的 Nut/OS 所有库函数文件，后者是该目标系统平台的应用案例集。

应用案例文件夹 nutapp\_13h 有 8 个 make 文件及 2 个.mk 文件，余下的就是大量的应用案例文件夹，其中之一是 httpd，本例基于 httpd 设计完成。

为避免破坏原始案例文件及文件夹，设计本例程序时，可先在 c:\etternut - 4.9.7 下建立新文件夹“用 IE 访问 AVR 以太网应用系统”，然后将 nutapp\_13h 下的所有 make 文件及 ht-

tpd 文件夹复制到“c:\ethernut - 4.9.7\用 IE 访问 AVR 以太网应用系统”下,当然,其中 makevars.avr32-gcc 和 makevars.arm-gcc 可不必复制。

要特别注意的是,新目录下的所有 make 文件及 httpd 文件夹的深度必须与原始文件及文件夹深度相同,否则会使后续操作出现大量编译错误。

### (3) 嵌入式系统中的 WEB 页设计

在开始本例程序设计之前,要先设计出将存放于 Atmega128 单片机的 WEB 页文件,这些 WEB 页文件通过网页设计工具 DreamWeaver 设计,先存放于 httpd/web\_files 下。下面是该 WEB site 中的所有相关文件:

index.html——首页文件,它实际上是由顶框架 top 与主框架 main 构成的框架集页面;  
top.html——顶框架中的页面文件,显示 Flash 动画及标题内容、所有操作链接等;  
main.html——初始时显示在主框架中的页面文件,显示 Ethernut 4.9.7 的相关信息;  
setpassword.html——设置 admin 用户密码的 form 页面文件;  
flash/surface.swf——为加强页面视觉效果而设计的 Flash 动画文件;  
run\_status.html——返回 LED 状态,电机运行状态的页面。

本例压缩包中提供了上述文件,限于篇幅,这里不讨论使用 DreamWeaver 工具软件设计上述页面文件的操作方法与操作过程。

上述文件中的前 5 个在运行过程所显示的内容保持不变(包括 Flash 动画文件),这些文件也被称为静态文件;第 6 个文件所显示的是当前 LED 及 MOTOR 的状态,其内容是变化的,这类文件也被称为动态文件。

对于 1~5 号文件,它们以静态文件形式被固定生成到 urom.c,每个文件所有字节将都被生成到该文件内的一个数组中。

urom.c 由 C:\ethernut - 4.9.7\nut\tools\win32 下的工具程序 crurom.exe 生成,在后续编译过程中,该工具将会被自动调用,它根据指定的 web\_files 中的相关文件生成 urom.c。

第 6 号文件仅仅是为了便于后续 C 程序设计中动态生成状态页面的编写而提供的一个样板页面,它的所有 HTML 标记语言将被移植到 httpdserv.c 程序内的相关函数中,标记语言中有关 LED 及 MOTOR 状态的部分将会由 fprintf\_P 函数动态生成,然后返回到客户端的 IE 浏览器内。

### (4) HTTP 服务程序设计

使用 configurator 创建案例文件夹时,nutapp\_13h/httpd 下会自动出现项目文件 httpd.prj,但该文件并非 AVRStudio 的项目文件,编写本例 C 程序前,需要首先在 AVRStudio 中创建项目文件“用 IE 访问基于 RTL8019 设计的以太网应用系统.aps”,然后添加 C 程序文件 httpdserv.c。

对于静态页面的返回,C 程序中不需要单独提供动态生成响应页面的函数,Nut/OS 会自动响应并返回客户端单击链接所请求的 html 页面。

本例程序设计要点在于动态页面的返回,单片机程序对带参数链接的处理,对所传送过来的表单数据的处理。

下面首先讨论获取 LED 状态、启动电机及停止电机链接的处理,这 3 个链接所指向的 URL 分别为:

admin/mcu\_control.cgi? para = GETLEDSTATUS



```
admin/mcu_control.cgi? para = STARTMOTOR  
admin/mcu_control.cgi? para = STOPMOTOR
```

为了保护 mcu\_control.cgi 程序,链接前面添加了 admin/,程序中通过调用 Nut/OS 的 API 函数 NutRegisterAuth 来保护 admin 路径下的文件,调用语句如下:

```
NutRegisterAuth("admin", admin_password);
```

上述 3 个链接中的某一个被最先请求时,由于 URL 前面添加了路径 admin/,IE 会弹出对话框,要求输入拥有该目录权限的用户账号和密码。本例中将初始账号和密码分别设为“root”、“123456”。

由于主程序中将 cgi 请求 mcu\_control.cgi 注册给函数 mcu\_control,执行语句如下:

```
NutRegisterCgi("mcu_control.cgi", mcu_control);
```

通过密码验证后,对 mcu\_control.cgi 的请求将交由函数 mcu\_control 来处理,该函数的参数为文件流对象 stream 和请求对象 req,通过调用 Nut/OS 的 API 函数 NutHttpGetParameterName 与 NutHttpGetParameterValue 可分别获得 URL 中“?”后面所带的参数名及参数值,由所获取的参数信息即可得知客户端的 IE 用户发出了何种请求:

```
para_name = NutHttpGetParameterName(req, 0);  
para_value = NutHttpGetParameterValue(req, 0);
```

在本例以太网应用系统案例中,LED 的开关是由硬件系统中的按键控制的,客户端的 IE 浏览器不控制 LED 的开关,仅查询当前 LED 开关状态。对于应用系统中的电机,其启停操作则完全由客户端的 IE 浏览器控制。

由于所收到的 3 种请求最终都要求返回操作状态,因而 mcu\_control 函数中 LED 的判断分支内不执行任何处理,只有电机的 2 个判断才分别完成启停操作,最后再根据 LED 的开关状态及 MOTOR 的运行状态动态构造返回到客户端的 WEB 页,这个任务通过调用创建状态 WEB 页的函数 create\_status\_webpage 完成。

以其中构造 LED 状态返回标记语言部分为例,其核心部分如下:

```
static char * html_x[] = //待输出状态 WEB 页的 HTML 标记语言  
{  
    .....  
    ".red_style {font-family: '黑体'; font-size: 60px; color: #FF0000;}",  
    ".blk_style {font-family: '黑体'; font-size: 60px; color: #000000;}",  
    .....  
    "<div align = 'center' class = '% s'> % s</div></td><td>", //10. LED 状态格式串  
    .....  
};  
if (led) fprintf (stream, html_x[10], "red_style", "ON");  
else    fprintf (stream, html_x[10], "blk_style", "OFF");
```

当 LED 开启时,fprintf 向流对象 stream 中写入标记语言,前一个占位符 %s 对应于样式名称,后一个占位符 %s 对应于输出的文字“ON”或“OFF”。上述语句执行后,根据变量 led 的值,可动态返回红色黑体 60px 的“ON”或黑色黑体 60px 的“OFF”,显示在客户端的 IE 浏览器中。

通过流对象 stream 返回 LED 与 MOTOR 状态页之前,还需要先执行以下两行语句:

```
NutHttpSendHeaderTop(stream, req, 200, "Ok");
NutHttpSendHeaderBottom(stream, req, html_mt, -1);
```

它们向返回的 HTTP 流中写入标准的 HTML 头信息,前者发送的是 HTTP 及服务器版本信息,后者发送的是 Content-Type、Content-Length 等信息。

对本例的第 4 个链接,单击后将返回填写密码的静态页面:setpassword.html。当按下保存密码按钮时,它所请求的 URL 地址为 admin/cgi-bin/setpassword.cgi。显然,主程序中为与上面的为 cgi 程序注册对应的处理函数一样,对于该请求中设置密码的操作,同样通过指定的函数 setpassword 来完成,该函数通过获取参数的 API 取得 2 次输入的密码,如果为合法密码则调用 eeprom 块写函数将新密码写入 0x00A0 地址,语句如下:

```
eeprom_write_block((uint8_t *)pwd1,(uint8_t *)0x00A0,strlen(pwd1));
```

阅读有关设备注册、CGI 注册、以太网配置等 API 调用及 HTTP 服务器线程编写,可参考源代码中的详细注释,或查看 Ethernut-4.9.7 的 HTML API Reference。

### (5) 编译设置

完成 httpserv.c 的编写以后,还需要一个包括所有 WEB 页文件的 urom.c,将 web\_files 下的所有各文件字节分别以数组形式写入 urom.c 时,可先在开始/运行窗口中输入 cmd 命令,然后进入 httpd 文件夹(不要进入 web\_files 文件夹),在 httpd 文件夹下输入以下命令并执行即可生成 urom.c 文件:

```
crurom -r -o urom.c web_files
```

编译本例时并不需要通过命令行生成 urom.c 文件,因为 eternut 已经为 httpd 的编译准备好了 makefile 文件,使用 makefile 来创建 urom.c 并编译本例程序的方法如下:

单击 AVRStudio 工具栏上的“Edit Current Configuration Options”(编辑当前配置选项)按钮,打开“Httsperv Project Options”对话框,在如图 5-36 所示的 Httsperv 项目选项配置窗口中选中“Use External Makefile”(使用外部 Makefile 文件)选项,然后单击其后边的按钮,选中 httpd 文件下的 Makefile 文件即可。

为了便于修改该 Makefile 文件中的配置,Makefile 文件要添加到 AVRStudio 左边的 AVR GCC 窗口中的 Other Files 分支(在该分支上单击,选择 Add Existing File 即可),添加后双击打开该文件。下面对其中的几个重要部分分别加以说明:

- ① PROJ = httsperv;
- ② WEBDIR=web\_files;
- ③ WEBFILE=urom.c;
- ④ \$(WEBFILE): \$(WEBDIR)/index.html \$(WEBDIR)/main.html \
\$(WEBDIR)/top.html \$(WEBDIR)/setpassword.html \
\$(WEBDIR)/flash/Surface.swf;
- ⑤ \$(CRUROM)-r-o \$(WEBFILE) \$(WEBDIR).

上述 5 个部分中,①将项目名称设为 httsperv;②定义了 WEB 目录,本例所有 WEB 相关文件存放于 web\_files 下;③定义生成的 WEB 文件为 urom.c;④定义了 web\_files 下所有待

添加到 urom.c 中的文件,一行过长时可在后面添加“\”;⑤调用 crurom 工具生成 urom.c 文件。

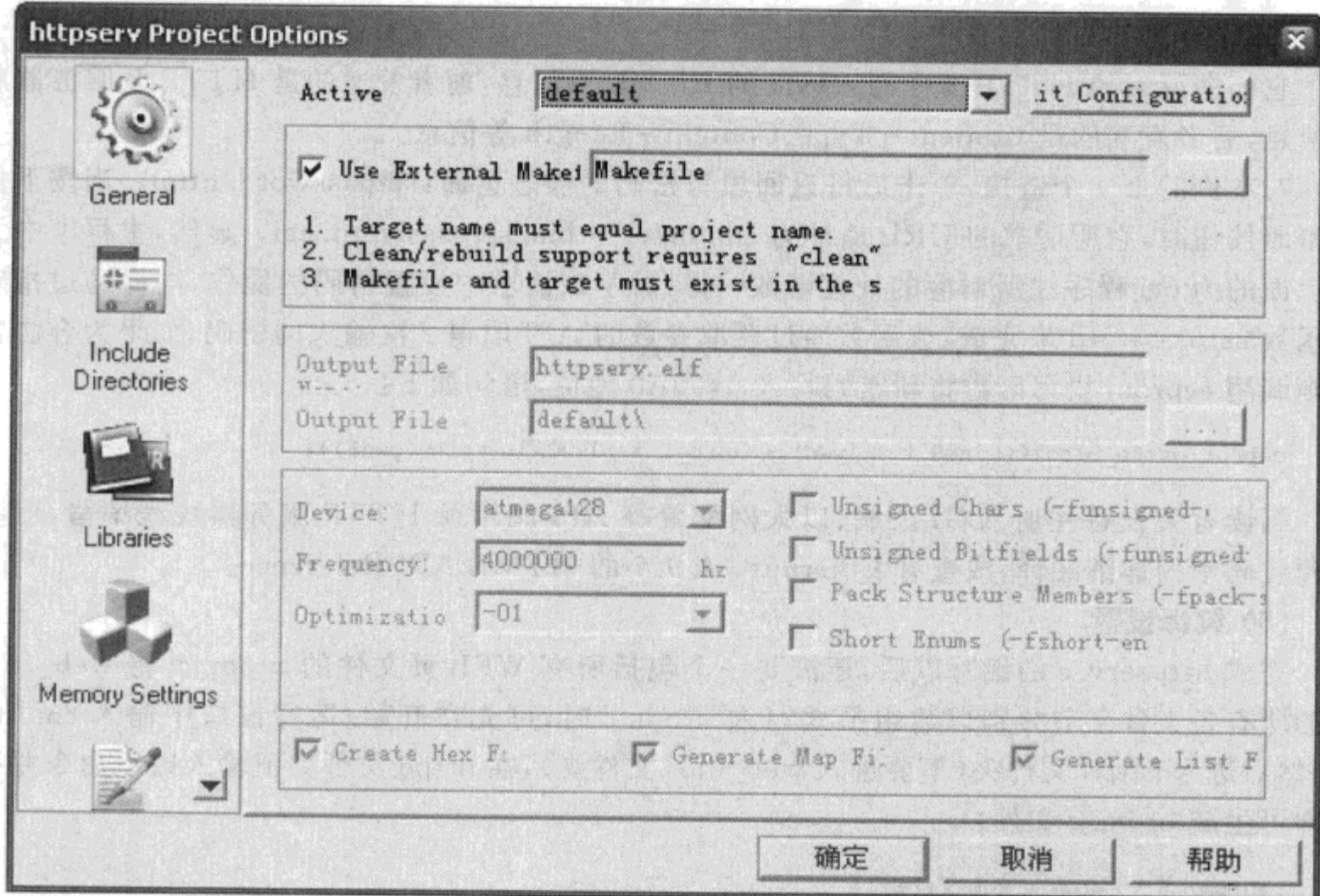


图 5-36 Httpserv 项目选项配置窗口

完成上述设置后即可编译生成 httsperv.hex 文件。

#### (6) WinPcap 安装调试运行

仿真调试运行本例时,需要以下 3 个部分同时运行:

- ① 安装并运行 Windows 包捕获程序 WinPcap(或 libpcap);
- ② 在 Proteus 中运行以 ATmega128+RTL8019 为核心设计的以太网应用系统;
- ③ 在本机打开 IE 浏览器,根据 Proteus 虚拟终端的提示输入 WEB 地址。

本例仿真运行时,以太网控制器仿真组件 RTL8019AS 不是通过高层协议访问网络,它需要一个底层环境去直接操纵网络通信,因此要有一个不需要协议栈支持的原始的访问网络的方法,而 WinPcap 即提供了具有这种访问能力的编程接口。

WinPcap 是用于捕获网络封包的一套工具,适用于 32 位操作平台上的网络封包解析,它包含了核心的封包过滤器(packet filter)、1 个底层动态链接库(packet.dll)、1 个高层系统函数库(wpcap.dll)及可用来直接存取封包的应用程序接口。

通过 packet.dll 提供的底层 API 可直接访问网络设备驱动,它独立于视窗操作系统。作为强大的捕获程序库,wpcap.dll 与 Unix 下的 libpcap 兼容,它独立于下层的网络硬件和操作系统。

当 WinPcap 未安装时,在 Proteus 中仿真运行以太网应用系统时,Proteus 会提示“无法获取网络适配器列表:未能找到接口! 运行本例要确保 libpcap 或 WinPcap 被完整地安装到本

机”(Cannot get network adapters list: No interface found! Make sure libcap/WinPcap is properly installed on the local machine.)。

完成 WinPcap 安装，并运行 Proteus 中的以太网应用仿真系统后，通过虚拟终端可看到动态分配的 IP 地址，这个 IP 地址与本机 IP 处于同一网段，例如 192.168.1.102。在 IE 浏览器中输入 <http://192.168.1.102> 回车即可看到从以 Atmega128 为核心的以太网应用系统返回的 WEB 页，默认打开的是首页文件 index.html。单击首页中的 5 个链接，可分别打开 HOME 页、查询 LED 状态、启动电机、停止电机、设置 admin 用户密码。

图 5-37 显示的是 LED 当前状态及电机被远程启动后的返回状态，图 5-38 是设置以太网应用系统管理员密码的 WEB 界面，图 5-39 是保存在 EEPROM 中的管理员密码。

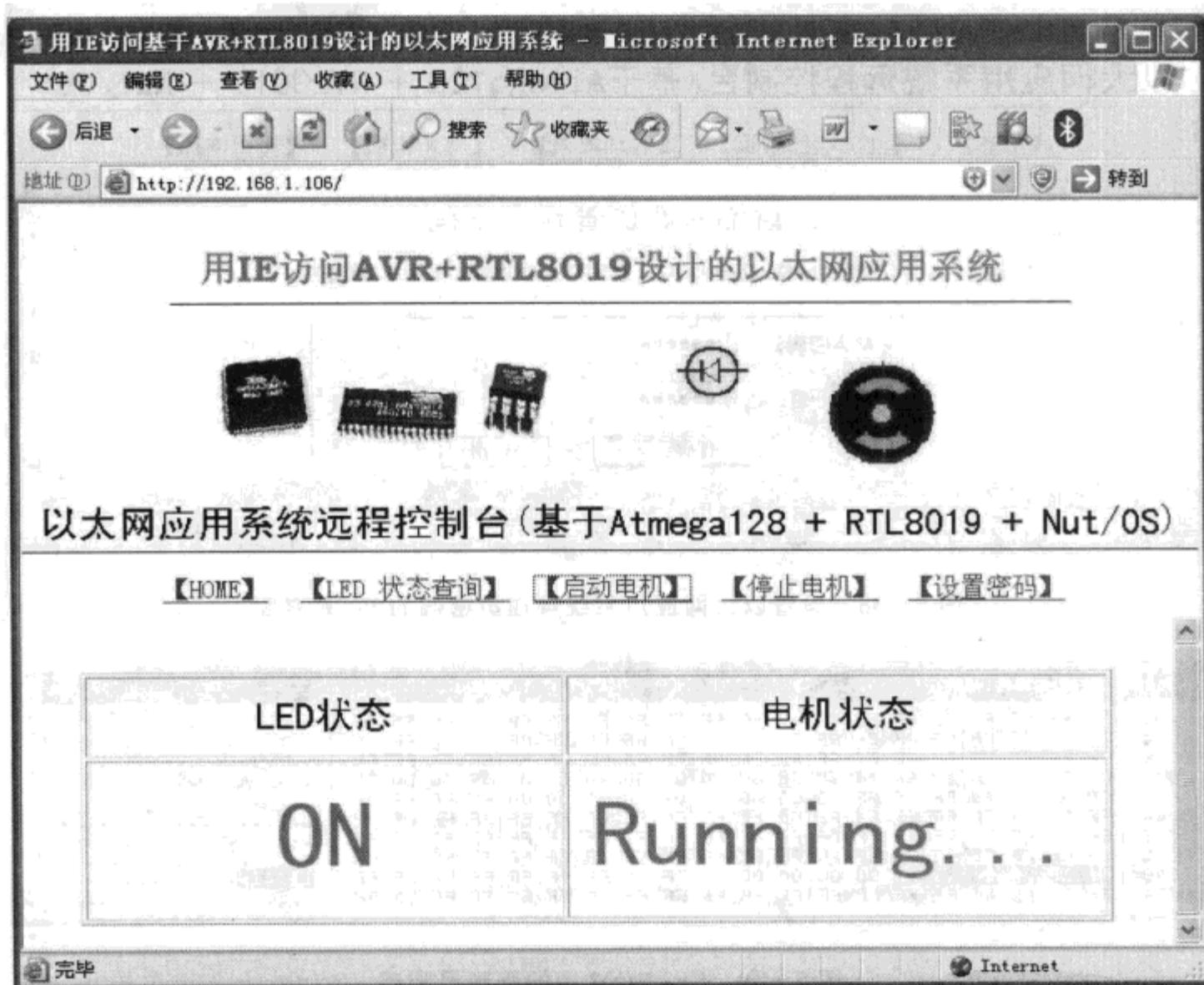


图 5-37 LED 当前状态及电机被远程启动后的返回状态

## 2. 实训要求

① 本例 IP 是动态分配的，完成本例调试后修改程序，首先给应用系统分配固定 IP 地址 192.168.1.100，子网掩码 255.255.255.0，如果 PC 机 IP 与以太网应用系统 IP 不在同一网段则修改 PC 机 IP 及子网掩码，然后通过 PC 机 IE 浏览器以 <http://192.168.1.100> 访问应用系统，在应用系统 WEB 页中添加手动设置与保存 IP 地址的功能。更改 IP 后如果 IE 访问没有响应，可进入 DOS 命令行界面，输入 arp /d 清除主机设置。另外，进行该项设计时，建议先参考 Nut/OS 的以下 API 相关资料及相关符号常量定义：

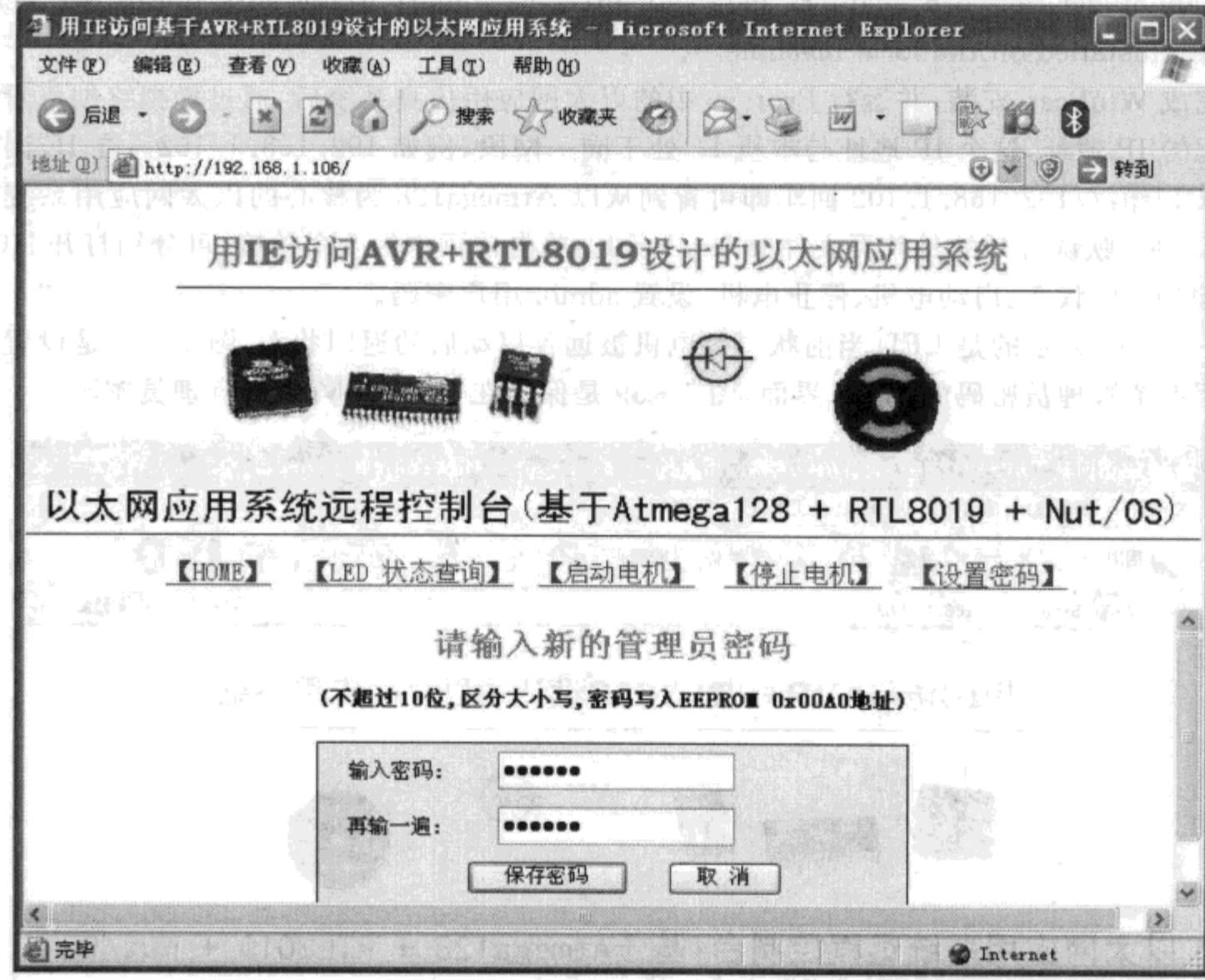


图 5-38 设置以太网应用系统管理员密码的 WEB 界面

AVR EEPROM Memory - U1															
0000	FF														
0014	FF														
0028	FF														
003C	FF	FF	FF	FF	20	65	74	68	30	00	00	00	00	98	30
0050	C0	A8	01	69	FF	FF	FF	00	CO	A8	01	01	00	00	35
0064	FF														
0078	FF														
008C	FF														
00A0	31	32	33	34	35	36	00	00	00	FF	FF	FF	FF	FF	FF
00B4	FF														

eth0.....0.5  
1.....  
123456.....

图 5-39 EEPROM 中的管理员密码

- NutNetLoadConfig(从 EEPROM 加载网络配置);
- NutNetSaveConfig(将网络配置保存到 EEPROM);
- CONFNET\_EE\_OFFSET (以太网配置数据在 EEPROM 内的偏移地址, 默认为 64, 即 0x0040);
- NutNvMemLoad(从 EEPROM 加载数据);
- NutNvMemSave(将数据保存到 EEPROM)。

其中前 2 个 API 根据所设置的符号常量 CONFNET\_EE\_OFFSET 分别调用后 2 个 API。

② 进一步修改程序，在本例电路中添加 DS18B20，通过单击 IE 浏览器超链接可刷新显示当前温度信息。如果熟悉 JavaScript 脚本编程，可在 WEB 页中添加 JS 程序，使浏览器可每隔 1 min 自动刷新显示当前外界环境温度数据。

③ 在仿真电路中添加 80×16 点阵的 LED 点阵屏，重新设计页面并编写程序，使以太网应用系统可接收并在点阵屏上滚动显示客户端 IE 浏览器内文本框中所填写的信息。

### 3. 源程序代码

```

001 //-----
002 // 名称：用 IE 访问 AVR 以太网应用系统
003 //-----
004 // 功能：本例运行时，客户端通过 IE 浏览器可以查询 LED 状态，启停电机
005 //       设置管理员密码等
006 //
007 //-----
008 //MAC 地址，如果 EEPROM 包含了有效配置则忽略此行
009 #define MY_MAC "\x00\x06\x98\x30\x00\x35"
010 //IP 地址(如果启用了 DHCP 则忽略)
011 #define MY_IPADDR "192.168.1.100"
012 //IP 网络掩码(如果启用了 DHCP 则忽略)
013 #define MY_IPMASK "255.255.255.0"
014 //网关 IP(如果启用了 DHCP 则忽略)
015 #define MY_IPGATE "192.168.1.1"
016 //是否使用 DHCP
017 #define USE_DHCP
018 //定义文件系统设备
019 #ifndef MY_FSDEV
020 #define MY_FSDEV devUrom
021 #endif
022
023 #include <cfg/os.h>
024 #include <string.h>
025 #include <io.h>
026 #include <fcntl.h>
027 #include <dev/board.h>
028 #include <dev/urom.h>
029 #include <dev/irqreg.h>
030 #include <arch/avr32/ihndlr.h>
031 #include <avr/eeprom.h>
032 #include <sys/version.h>
033 #include <sys/thread.h>
034 #include <sys/timer.h>
035 #include <sys/heap.h>
```



```
036 # include <sys/confnet.h>
037 # include <sys/socket.h>
038 # include <arpa/inet.h>
039 # include <net/route.h>
040 # include <pro/httpd.h>
041 # include <pro/dhcp.h>
042
043 //服务器线程堆栈大小
044 # ifndef HTTPD_SERVICE_STACK
045 # define HTTPD_SERVICE_STACK ((580 * NUT_THREAD_STACK_MULT) + NUT_THREAD_STACK_ADD)
046 # endif
047
048 static char * html_mt = "text/html";
049 static char admin_password[16] = "root:";           //管理员账号密码
050 //-----
051 // 根据 LED 与 MOTOR 状态构造创建 WEB 页
052 //-----
053 void create_status_webpage(FILE * stream, int led, int motor)
054 {
055     u_char i;
056     //待输出状态 WEB 页的 HTML 标记
057     static char * html_x[] =
058     {
059         "<html><head><style type='text/css'>", //此行开始的 0~9 行为固定部分
060         ".title_style {font-family: '黑体';font-size: 24px; }",
061         ".red_style {font-family: '黑体';font-size: 60px;color: #FF0000;}",
062         ".blk_style {font-family: '黑体';font-size: 60px;color: #000000;}",
063         "</style></head><body><br />",
064         "<table width='630' height='160' border='1' align='center'>",
065         "<tr><td width='290' height='49'><div align='center'>",
066         "<span class='title_style'>LED 状态</span></div></td><td width='326'>",
067         "<div align='center'><span class='title_style'>电机状态</span>",
068         "</div></td></tr><tr><td height='98'>",
069         "<div align='center' class='%s'>%s</div></td><td>,//10.LED 状态格式串
070         "<div align='center' class='%s'>%s</div></td></tr>,//11.MOTOR 状态格式串
071         "</table></body></html>"                                //12. 结尾部分
072     };
073
074     //将固定的 HTML 标记写入 stream(0~9 行).
075     for (i = 0; i < 10; i++) fputc(html_x[i], stream);
076
077     //向流中写入红色"ON"字符串标记(红色样式 red_style 定义在 61 行)
078     if (led) fprintf(stream, html_x[10], "red_style", "ON");
```

```

079 //否则向流中写入黑色"OFF"字符串标记(黑色样式 blk_style 定义在 62 行)
080 else     fprintf(stream, html_x[10], "blk_style", "OFF");
081
082 //向流中写入红色"Running..."字符串标记
083 if (motor) fprintf(stream, html_x[11], "red_style", "Running..."); 
084 //否则向流中写入黑色" * STOP * "字符串标记
085 else     fprintf(stream, html_x[11], "blk_style", " * STOP * ");
086
087 //输出结尾部分
088 fputs(html_x[12], stream);
089 }
090
091 //-----
092 // LED 状态查询与电机控制函数
093 // 该函数必须由 NutRegisterCgi() 注册, 当客户端请求 cgi-bin/mcu_Control.cgi 时
094 // 自动被 NutHttpProcessRequest() 调用
095 //-----
096 static int mcu_control(FILE * stream, REQUEST * req)
097 {
098     //led 及 motor 状态
099     int led, motor;
100    //参数名及参数值变量, 根据参数决定返回 LED 状态或启/停电机
101    char * para_name, * para_value;
102
103    //3 个用户请求超链接格式
104    //admin/mcu_control.cgi? para = GETLEDSTATUS
105    //admin/mcu_control.cgi? para = STARTMOTOR
106    //admin/mcu_control.cgi? para = STOPMOTOR
107    //读取所接收到的参数名及参数值
108    para_name = NutHttpGetParameterName (req, 0);
109    para_value = NutHttpGetParameterValue(req, 0);
110
111    //根据不同参数值完成不同操作
112    if (!strcmp(para_name, "para"))
113    {
114        //根据不同参数完成不同操作
115        if (!strcmp(para_value, "GETLEDSTATUS")) //获取 LED 状态
116        {
117            //因为无论是查询 LED 还是启停电机, 该调用都要返回 LED 状态
118            //故将这里的 LED 状态查询放在 if 语句外面
119        }
120        else
121            if (!strcmp(para_value, "STARTMOTOR")) //启动电机

```

```

122     {
123         PORTF |= _BV(PF0);
124     }
125     else
126         if (!strcmp(para_value,"STOPMOTOR")) //停止电机
127     {
128         PORTF &= ~_BV(PF0);
129     }
130 }
131
132 //LED 状态由 PF6 位判断(注意 led 与 motor 的状态判断返回值是相反的)
133 led = (PORTF & _BV(PF6)) ? 0:1;
134 //MOTOR 状态由 PF0 判断
135 motor = (PORTF & _BV(PF0)) ? 1:0;
136 //以下两行发送 HTTP 头部,创建 HTTP 响应
137 //发送 HTTP 及版本行
138 NutHttpSendHeaderTop(stream, req, 200, "Ok");
139 //发送 Content - Type, Content - Length 等
140 NutHttpSendHeaderBottom(stream, req, html_mt, -1);
141 //根据 LED 与 MOTOR 状态构造返回 WEB 页
142 create_status_webpage(stream,led,motor);
143 //刷新返回的流
144 fflush(stream);
145 return 0;
146 }
147
148 //-----
149 // 设置管理员密码
150 // 该函数必须由 NutRegisterCgi()注册,当客户端请求
151 // admin/cgi-bin/setpassword.cgi 时,该函数将自动
152 // 被 NutHttpProcessRequest()调用
153 //-----
154 static int setpassword(FILE * stream, REQUEST * req)
155 {
156     u_char save_OK = 0; //是否保存成功
157     char * pwd1, * pwd2; //2 次输入的密码字符串指针
158
159     //调用获取参数 API,根据文本框的名称 pass1 与 pass2 分别获取 2 个密码
160     pwd1 = NutHttpGetParameter(req, "pass1");
161     pwd2 = NutHttpGetParameter(req, "pass2");
162
163     //检查 2 次输入的密码是否相同,且长度是否在 10 以内
164     if (!strcmp(pwd1,pwd2) && strlen(pwd1) > 0 && strlen(pwd1) < 11)

```

```

165  {
166      //将新输入的密码保存到字符串 admin_password 的"root:"后面
167      strcpy(admin_password + 5, pwd1);
168      //新密码写入 EEPROM 中 0x00A0 地址处(包括末尾的'\0')
169      eeprom_write_block((uint8_t *)pwd1,(uint8_t *)0x00A0,strlen(pwd1));
170      //清除所有授权条目
171      NutClearAuth();
172      //注册新设置的账号密码
173      NutRegisterAuth("admin", admin_password);
174      save_OK = 1; //保存成功
175  }
176
177  //以下两行发送 HTTP 头部,创建 HTTP 响应
178  NutHttpSendHeaderTop (stream, req, 200, "Ok");
179  NutHttpSendHeaderBottom(stream, req, html_mt, -1);
180
181  //待输出密码保存成功与否信息 WEB 页的 HTML 标记
182  static char * html_x[] =
183  {
184      "<html><head><title>设置管理员密码</title></head>",
185      "<body><br><H1>返回信息:</H1><br>",
186      "<font color = '%s'> %s</font><br></body></html>"
187  };
188
189  fputs(html_x[0], stream);           //发送 HTML 中的固定部分
190  fputs(html_x[1], stream);           //同上
191
192  //根据 save_OK 输出蓝色的保存成功信息或红色的保存失败信息.
193  if (save_OK)
194      fprintf(stream,html_x[2],"#0000FF","密码被成功保存!!!!");
195  else
196      fprintf(stream,html_x[2],"#FF0000","* * 密码未能保存 * *");
197
198  fflush(stream);                   //刷新输出
199  return 0;
200 }
201
202 //-----
203 // HTTP 服务线程(循环等待客户连接,处理 HTTP 请求并断开连接)
204 //-----
205 THREAD(Service, arg)
206 {
207     TCPSOCKET * sock;           //套接字

```

```

208     FILE * stream;                                //文件流对象
209     u_char id = (u_char)((uptr_t)arg);
210
211     while (1)
212     {
213         //创建套接字
214         if ((sock = NutTcpCreateSocket()) == 0)
215         {
216             printf("[ % u] Creating socket failed\n", id);
217             NutSleep(5000);
218             continue;
219         }
220         //在 80 号端口监听,在获得来自客户端的连接之前该调用将被阻塞
221         NutTcpAccept(sock, 80);
222         printf("[ % u] Connected, % u bytes free\n", id, NutHeapAvailable());
223
224         //等待至少具备 8K 的自由 RAM,这可在低内存条件下保持客户连接
225         while (NutHeapAvailable() < 8192)
226         {
227             printf("[ % u] Low mem\n", id);
228             NutSleep(1000);
229         }
230
231         //创建与套接字关联的流对象,以便于使用标准 I/O 调用
232         if ((stream = _fdopen((int) ((uptr_t) sock), "r + b")) == 0)
233         {
234             printf("[ % u] Creating stream device failed\n", id);
235         }
236         else
237         {
238             //此 API 调用解析客户请求,发送已经注册的文件,
239             //系统中的被请求的文件,或通过调用注册的 CGI 例程处理 CGI 请求等
240             NutHttpProcessRequest(stream);
241             //注销虚拟流设备
242             fclose(stream);
243         }
244
245         NutTcpCloseSocket(sock);                      //关闭套接字
246         printf("[ % u] Disconnected\n", id);          //提示连接断开
247     }
248 }
249
250 //-----

```

```

251 // 外部中断 INT7 控制 LED 开关
252 //-
253 static void External_Interrupt7_IRQHandler(void * arg)
254 {
255     PORTF ^= _BV(PF6);                                //切换 LED 开关
256 }
257
258 //-
259 // 主程序(Nut/OS 在初始化后自动调用该入口函数)
260 //-
261 int main(void)
262 {
263     u_long baud = 115200;                            //波特率 115200
264     u_char i;
265     //注册设备 DEV_DEBUG, 初始化 UART 设备
266     NutRegisterDevice(&DEV_DEBUG, 0, 0);
267     //将流 stdout 指定给 DEV_DEBUG_NAME, 操作模式为写(w)
268     freopen(DEV_DEBUG_NAME, "w", stdout);
269     //调用 I/O 设备控制函数(设置 stdout 的波特率)
270     _ioctl(_fileno(stdout), UART_SETSPEED, &baud);
271     //临时挂起当前线程 200ms
272     NutSleep(200);
273     printf("\n\nNut/OS %s HTTP DEMO... \n", NutVersionString());
274     //注册以太网控制器设备
275     if (NutRegisterDevice(&DEV_ETHER, 0, 0))
276     {
277         puts("Registering device failed\n");
278     }
279     printf("Configure %s... \n", DEV_ETHER_NAME);
280
281     u_char mac[] = MY_MAC;
282     printf("initial boot... \n");
283     //如果设置使用 DHCP 则动态配置以太网接口
284     // (mac 为介质访问地址, 60000 ms 为超时设置)
285     #ifdef USE_DHCP
286     if (NutDhcpIfConfig(DEV_ETHER_NAME, mac, 60000))
287     #endif
288     {
289         //在未使用 DHCP 时, 或动态配置失败时使用静态配置
290         u_long ip_addr = inet_addr(MY_IPADDR); //IP 地址
291         u_long ip_mask = inet_addr(MY_IPMASK); //掩码
292         u_long ip_gate = inet_addr(MY_IPGATE); //网关
293         printf("No DHCP... \n");

```



```
294     //按设定参数配置以太网接口
295     if (NutNetIfConfig(DEV_ETHER_NAME,mac,ip_addr,ip_mask) == 0)
296     {
297         if(ip_gate) //无 DHCP 时需要手动设置默认网关
298         {
299             printf("hard coded gate... ");
300             //向 IP 路由表中添加新项
301             NutIpRouteAdd(0, 0, ip_gate, &DEV_ETHER);
302         }
303         puts("OK\n");
304     }
305     else puts("failed\n");
306 }
307 //显示就绪信息及动态或静态配置的最终所使用的以太网 IP 地址
308 printf("IP: %s ready\n", inet_ntoa(confnet.cdn_ip_addr));
309
310 //注册文件系统设备
311 NutRegisterDevice(&MY_FSDEV, 0, 0);
312 //从 EEPROM 的 0x00A0 地址处读取 admin 用户的初始密码
313 //存入字符串 admin_password 中的 ":" 之后.
314 eeprom_read_block((uint8_t *) (admin_password + 5),(uint8_t *) 0x00A0,11);
315 //如是第一字节为 0x00 或 0xFF 则表示无初始密码, 系统默认设置"root:123456"
316 if (admin_password[5] == 0xFF || admin_password[5] == 0x00)
317 {
318     //初始密码写入 EEPROM 中 0x00A0 地址处
319     eeprom_write_block((uint8_t *)"123456\0\0\0\0\0", (uint8_t *) 0x00A0,11);
320     //设置管理员 admin 的账号 root 及初始密码 123456(root:123456)
321     strcpy((admin_password + 5), "123456");
322 }
323
324 //注册授权用户 admin 的账号密码 root:XXXXXXXXXX
325 //保护 admin/cgi-bin 下的程序文件
326 NutRegisterAuth("admin", admin_password);
327 //注册 cgi-bin 路径(前两条路径本例未用)
328 NutRegisterCgiBinPath("cgi-bin/;user/cgi-bin/;admin/cgi-bin/");
329 //注册 LED 状态查询与电机控制 cgi 程序
330 NutRegisterCgi("mcu_control.cgi", mcu_control);
331 //注册设置管理员新密码的 cgi 程序
332 NutRegisterCgi("setpassword.cgi", setpassword);
333 //创建 4 个服务线程
334 for (i = 1; i <= 4; i++)
335 {
336     char thname[] = "httpd0";
```

```
337     thname[5] = '0' + i;
338     NutThreadCreate(thname, Service, (void *) (uptr_t) i,
339                     (HTTPD_SERVICE_STACK * NUT_THREAD_STACK_MULT) + NUT_THREAD_STACK_ADD);
340 }
341 //设置线程优先级
342 NutThreadSetPriority(254);
343 //PF 端口的 PF0,PF6 设为输出
344 DDRF |= _BV(PF0) | _BV(PF6);
345 //初始时关闭 LED,停止电机
346 PORTF |= _BV(PF6); PORTF &= ~_BV(PF0);
347 //PE 端口 PE7 设为输入,内部上拉
348 DDRE &= ~_BV(PE7); PORTE |= _BV(PE7);
349 //注册 INT7 外部中断请求函数 External_Interrupt7_IRQ
350 NutRegisterIrqHandler(&sig_INTERRUPT7, External_Interrupt7_IRQ, 0);
351 //使能 INT7 中断
352 NutIrqEnable(&sig_INTERRUPT7);
353 //或使用 sbi(EIMSK, INT7);
354 while (1) NutSleep(60000);
355 return 0;
356 }
```

## 参考文献

- [1] 马潮. 高档 8 位单片机 ATmega128 原理与应用指南[M]. 北京:北京航空航天大学出版社,2004.
- [2] 佟长福. AVR 单片机的 GCC 程序设计[M]. 北京:北京航空航天大学出版社,2006.
- [3] 周兴华. AVR 单片机 C 语言高级程序设计[M]. 北京:中国电力出版社,2008.
- [4] 杨正忠. AVR 单片机应用开发指南及实例精解[M]. 北京:中国电力出版社,2008.
- [5] 胡汉才. 高档 AVR 单片机原理及应用[M]. 北京:清华大学出版社,2008.
- [6] 金钟夫. AVR ATmega128 单片机 C 程序设计与实践[M]. 北京:北京航空航天大学出版社,2008.
- [7] 张军. AVR 单片机 C 语言程序设计实例精粹[M]. 北京:电子工业出版社,2009.
- [8] 沈建良. ATmega128 单片机入门与提高[M]. 北京:北京航空航天大学出版社,2009.
- [9] 朱飞. AVR 单片机 C 语言开发入门与典型实例[M]. 北京:人民邮电出版社,2009.

