第十章 综合实践（二）

本章的综合实践将综合前 4 章的内容，指导读者完成以下的实践：

- 如何实现频率测量和简单频率计的设计实现。
- 完成一个比较完善的实时时钟的设计和实现。
- 介绍 LED 显示及键盘接口芯片—HD7279A。

10.1.2 测频法测量频率

测频法的基本思想，就是采用在已知限定的时间内对被测信号输入的脉冲个数进行计数的方法来实现对信号频率的测量。当被测信号的频率比较高时，采用这种方法比较适合，因为在一定时间内，频率越高，计数脉冲的个数也越多，测量也越准确。

例 10.1　采用测频法的频率计设计与实现

1）　硬件电路

硬件电路的显示部分与图 9-7 相同，PA 口为 8 个 LED 数码管的段输出，PC 口控制 8 个 LED 数码管的位扫描。被测频率信号由 PB0（T0）输入。

2）　软件设计

我们首先给出系统程序，然后做必要的说明。

```
/*********************************************
File name          : demo_10_1.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*********************************************/

#include <mega16.h>
flash char led_7[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
flash char position[8]={0x7f,0xbf,0xdf,0xef,0xf7,0xfb,0xfd,0xfe};

char dis_buff[8];                    // 显示缓冲区，存放要显示的 8 个字符的段码值
char posit;
bit  time_1ms_ok,display_ok=0;
char time0_old,time0_new,freq_time;
unsigned int  freq;

void display(void)                   // 8 位 LED 数码管动态扫描函数
{
    PORTC = 0xff;
    PORTA = led_7[dis_buff[posit]];
    if (posit==5) PORTA = PORTA | 0x80;
    PORTC = position[posit];
```

```
    if (++posit >=8 ) posit = 0;
}

// Timer 2 output compare interrupt service routine
interrupt [TIM2_COMP] void timer2_comp_isr(void)
{
    time0_new = TCNT0;                  // 1ms 到，记录当前 T/C0 的计数值
    time_1ms_ok = 1;
    display_ok = ~display_ok;
    if (display_ok) display();
}

void freq_to_disbuff(void)                  // 将频率值转化为 BCD 码并送入显示缓冲区
{
    char i,j=7;
    for (i=0;i<=4;i++)
    {
        dis_buff[j-i] = freq % 10;
        freq = freq / 10;
    }
    dis_buff[2] = freq;
}

void main(void)
{
    char i;
    DDRA=0xFF;                          // LED 数码管驱动
    DDRC=0xFF;
        // T/C0 初始化，外部计数方式
    TCCR0=0x06;                         // 外部 T0 脚下降沿触发计数，普通模式
    TCNT0=0x00;
    OCR0=0x00;
        // T/C2 初始化
    TCCR2=0x0B;                         // 内部时钟，32 分频（4M/32=125KHz），CTC 模式
    TCNT2=0x00;
    OCR2=0x7C;                          // OCR2 = 0x7C(124),(124+1)/125=1ms

    TIMSK=0x80;                         // 允许 T/C2 比较匹配中断

    for (i=0;i<=7;i++) dis_buff[i] = 0;
    time0_old = 0;

    #asm("sei")                         // 开放全局中断
```

```
        while (1)
        {
            if (time_1ms_ok)
            {   // 累计 T/C0 的计数值
                if (time0_new >= time0_old) freq = freq + (time0_new - time0_old);
                else freq = freq + (256 - time0_old + time0_new);
                time0_old = time0_new;
                if (++freq_time >= 100)
                {
                    freq_time = 0;          // 100ms 到,
                    freq_to_disbuff();      // 将 100ms 内的脉冲计数值送显示
                    freq = 0;
                }
                time_1ms_ok = 0;
            }
        };
    }
```

程序中 LED 扫描形式函数 desplay()，以及脉冲计数值转换成 BCD 码并送显示缓冲区函数 freq_to_disbuff()比较简单，请读者自己分析。

在该程序中，使用了两个定时/计数器。T/C0 工作在计数器方式，对外部 T0 引脚输入的脉冲信号计数（下降沿触发）。T/C2 工作在 CTC 方式，每隔 1ms 中断一次，该定时时间即作为 LED 的显示扫描，同时也是限定时间的基时。每一次 T/C2 的中断中，都首先记录下 T/C0 寄存器 TCNT0 当前的计数值，因此前后两次 TCNT0 的差值(time0_new – time0_old)或(256 - time0_old + time0_new)就是 1ms 时间内 T0 脚输入的脉冲个数。为了提高测量精度，程序对 100 个 1ms 的脉冲个数进行了累计（在变量 freq 中），即已知限定的时间为 100ms。

读者还应该注意频率的连续测量与 LED 扫描、BCD 码转换之间的协调问题。T/C2 中断间隔为 1ms，因此在 1sm 时间内，程序必须将脉冲个数进行的累计、BCD 码转换和送入显示缓冲区，以及 LED 的扫描工作完成掉，否则就会影响到下一次中断到来后的处理。

在本实例的 T/C2 中断中，使用了 display_ok 标志，将 LED 扫描分配在奇数 ms（1、3、5、7、……），而将 1ms 的 TCNT0 差值计算、累积和转换等处理放在主程序中完成。另外由于计算量大的 BCD 码转换是在偶数 ms（100ms）处理，所以程序中 LED 的扫描处理和 BCD 码转换处理不会同时进行（不会在两次中断间隔的 1ms 内同时处理 LED 扫描和 BCD 码转换），这就保证了在下一次中断到达时，前一次的处理已经全部完成，使频率的连续测量不受影响。

该实例程序的性能和指标为（假定系统时钟没有误差 = 4MHz）：

- ✓ 频率测量绝对误差：±10Hz。限定的时间为 100ms，T/C0 的计数值有±1 的误差，换算成频率为±10Hz。
- ✓ 被测最高频率值：255KHz。由于 T/C0 的长度 8 位，所以在 1ms 中，T0 输入的脉冲个数应小于 255 个，大于 255 后造成 T/C0 的自动清另，丢失脉冲个数。
- ✓ 测量频度：10 次/秒。限定的时间为 100ms，连续测量，所以为 10 次/秒。
- ✓ 使用资源：两个定时器，一个中断。

3) 思考与实践

根据上面采用测频法的思路，如何修改程序提高测量精度和被测最高频率？参考提示如下：

- ✓ 延长限定的时间，如采用 1s，可提高频率的测量精度。但测量频度减小，同时注意变量 freq 应定义为长整型变量。

✓ 将 T/C0 换成 16 位的 T/C1，可以提高被测最高频率值。注意此时 time0_new、time0_old 应定义为整型变量。

10.1.3 测周法测量频率

测周法的基本思想，就是测量在限定的脉冲个数之间的时间间隔，然后再换算成频率（需要时）。当被测信号的频率比较低时，采用这种方法比较适合，因为频率越低，在限定的脉冲个数之间的时间间隔也也长，因此定时计数的个数也越多，测量也越准确。

例 10.2　采用测周法的频率计设计与实现

1） 硬件电路

硬件电路的显示部分与图 9-7 相同，PA 口为 8 个 LED 数码管的段输出，PC 口控制 8 个 LED 数码管的位扫描。被测频率信号由 PB0（T0）输入。

2） 软件设计

我们首先给出系统程序，然后做必要的说明。

```
/*********************************************
File name          : demo_10_2.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
*********************************************/

#include <mega16.h>
flash char led_7[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
flash char position[8]={0x7f,0xbf,0xdf,0xef,0xf7,0xfb,0xfd,0xfe};

char dis_buff[8];               // 显示缓冲区，存放要显示的 8 个字符的段码值
char posit;
bit  freq_ok = 0;
char time2_new;
unsigned int  freq;
unsigned long int freq_temp;

void display(void)              // 8 位 LED 数码管动态扫描函数
{
    PORTC = 0xff;
    PORTA = led_7[dis_buff[posit]];
    if (posit==5) PORTA = PORTA | 0x80;
    PORTC = position[posit];
    if (++posit >=8 ) posit = 0;
}

// T/C0 比较匹配中断服务，250 个计数脉冲中断一次
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
    time2_new = TCNT2;
    TCNT2 = 0;
```

```
    TIFR |= 0x02;
    freq_temp = freq;
    freq = 0;
    freq_ok = 1;
}

// T/C2 比较匹配中断服务，500us 一次
interrupt [TIM2_COMP] void timer2_comp_isr(void)
{
    freq++;
    #asm("sei")        // 开中断，允许中断嵌套，T/C0 中断可打断该中断服务
    display();
}

void freq_to_disbuff(void)        // 频率值转化为 BCD 码送显示缓冲区
{
    char i,j=7;
    for (i=0;i<=7;i++)
    {
        dis_buff[j-i] = freq_temp % 10;
        freq_temp = freq_temp / 10;
    }
}

void main(void)
{
    char i;
    DDRA=0xFF;                    // LED 数码管
    DDRC=0xFF;
        // T/C2 初始化
    TCCR2=0x0A;                   // 内部时钟，8 分频（4M/8=500KHz），CTC 模式，
    TCNT2=0x00;                   // 基时为 2us
    OCR2=0xF9;                    // OCR2 = 0xF9(249),(249+1)/500 = 0.5ms
        // T/C0 初始化
    TCCR0=0x0E;                   // 外部 T0 脚下降沿触发计数，CTC 模式
    TCNT0=0x00;
    OCR0=0xF9;                    // OCR0 = 0xF9(249),(249 + 1) = 250

    TIMSK=0x82;                   // 允许 T/C2、T/C0 比较匹配中断

    for (i=0;i<=7;i++)dis_buff[i]=0;

    #asm("sei")                   // 开放全局中断

    while (1)
   {
        if (freq_ok)
        {
            freq_temp = freq_temp * 250 + time2_new;  // 累计 250 个脉冲的时间间隔
```

```
                freq_temp = 12500000000/freq_temp;        // 换算成频率
                freq_to_disbuff();                        // 频率值送显示
                freq_ok = 0;
            }
        };
    }
```

程序中 LED 扫描形式函数 desplay()，以及脉冲计数值转换成 BCD 码并送显示缓冲区函数 freq_to_disbuff()比较简单，请读者自己分析。

在该程序中，同样使用了两个定时/计数器。T/C2 仍旧工作在 CTC 方式，每隔 500us 中断一次，该定时时间即作为 LED 的显示扫描，同时也用于时间累计。在每一次 T/C2 的中断中，将累计中断的次数（在 freq 中），然后马上开放全局中断（由于在进入 T/C0 中断时，系统硬件已经自动关闭了全局中断允许），保证系统能及时响应 T/C0 的中断。

该程序的核心是 T/C0 的中断。T/C0 工作在计数器 + CTC 方式，它负责对外部 T0 引脚输入的脉冲信号计数（下降沿触发），一旦计数值（限定脉冲个数）到达 250 产生中断。进入 T/C0 中断后，立即记录当前 T/C2 寄存器 TCNT2 的值（在 time2_new 中），然后清另 TCNT2 和 T/C2 的中断标志位，为下一次计时做初始化准备。接下来同样需要把 T/C2 产生中断的次数累计值备份到 freq_temp 中，此时变量 freq_temp 和 time2_new 中的值就是 T0 输入的 250 个限定脉冲之间的时间间隔。

当 T/C0 中断产生后，系统应该立即响应，马上读取 T/C2 的值。由于 T/C2 的计时过程不会停止，所以拖延 T/C0 中断的响应时间就会影响测量的精度。因此需要把 T/C2 的中断设计成支持中断嵌套，使系统仅可能的马上响应 T/C0 中断。

计算 250 个限定脉冲之间的时间间隔是在主程序中完成的。计算公式为：250 个脉冲之间的时间间隔 = T/C2 中断次数 * 250 + T/C2 当前值（计时时基个数）；1 计时时基个数 = 2us（注：T/C2 计时时基 = 4M/8）。换算成频率值：1000000/（250 个脉冲之间的时间间隔 * 2us/250）* 100 = 12500000000/250 个脉冲之间的时间间隔，单位为 Hz。乘上 100 是为了保留 2 位小数。程序中全部使用了整型数的运算，它比采用浮点数运算的速度要快的多，同时也保证了在 T/C0 两次中断的间隔中，能全部完成频率换算、LED 扫描等处理任务，不造成对频率连续测量的影响。

该实例程序的性能和指标为（假定系统时钟没有误差 = 4MHz）：

- ✓ 周期测量绝对误差：±(2us/250)。如果不考虑中断响应时间的影响，由于 T/C2 的计数值有±1 的误差，所以周期测量绝对误差为±(2us/250)。当考虑中断响应时间的影响使，周期测量绝对误差在±(2～5/250)us。
- ✓ 被测最低频率值：8Hz。考虑 freq 的长度为 16 位，最大计数值为 65535，所以可以记录的 250 个脉冲之间的时间间隔最大为 65535*250*2us = 32767500us。那么最长 1 个脉冲周期为 32767500us/250 = 131070us，换算成频率为 1/131070 = 7.63Hz。
- ✓ 测量频度：与被测频率有关。如被测频率为 125Hz，测量频度 = 1 次/2 秒；被测频率为 250Hz，测量频度 = 1 次/秒；被测频率为 2K，测量频度 = 8 次/秒。
- ✓ 使用资源：两个定时器，两个中断，其中一个支持嵌套。

下面我们进一步讨论测量的精度问题，在测频法中，由于频率测量的绝对误差是±10Hz，因此被测频率越高（仅受系统时钟限制），测量精度也就越好，这一点是明显的。而在测周法中，由于其周期测量绝对误差是固定的，因此被测频率越低，精度越好。这一特点不容易直接看出，我们以测量 1K 频率和 4K 频率为例，分别计算出它们的精度结果，并进行比较。

首先我们取测周法的周期测量绝对误差为±(2us/250)，即±0.008us。对于 1K 频率，其标准周期为 1000us。考虑测量误差：1000.008us～999.992us，对应频率为：999.992Hz～

1000.008Hz，有效位数为 6 位。而对于 4K 频率，其标准周期为 250us。考虑测量误差：250.008us～249.992us，对应频率为：3999.872Hz～4000.128Hz，此时有效位数降为 5 位了。可见，当被测频率越高时，有效位数越少，测量的精度也越差了。

10.1.4 频率测量小结

以上我们介绍了两种频率的测量方法，通过分析我们知道，频率的测量还是比较复杂的。如果你要设计制作一个频率计，要求的被测频率范围比较宽，变化大时，单一的使用某一种方法都是不能满足需要的。所以，一个完善的频率计，需要设计一个智能的测量过程，即其系统程序能够根据每次的测试数据，自动转换使用正确的测量方法，以及能够自动调节限定的时间（测频法），或调节限定脉冲数（测周法），或调整计时的时间基时等。这样经过几次自动的调整后，系统测出的频率达到最高的测量精度。

此外，上面的频率测量方法都必须占用 MCU 的 2 个硬件资源，这也是一般单片机测频所采用的方法（或采用 1 个 T/C 加 1 个外部中断，同样占用 2 个硬件资源）。但 AVR 单片机的 T/C1 增加了捕捉功能，利用该功能进行频率的测量时，不仅只需要使用 1 个硬件资源 T/C1 就能完成频率的测量，而且还能获得更好的测量的精度。关于 T/C1 的使用，将在下一章介绍。

10.2 带校时功能的实时时钟设计与实现

在前面的章节中分别介绍了 I/O 口输入/出的应用、中断的应用、8 位 T/C 的应用，以及基于状态机思想的系统分析和系统程序设计方法等。在本节里，将给出一个功能比较完整的“带校时功能的实时时钟”系统，作为上面各种基本应用的综合设计示例。

例 10.3　带校时功能的实时时钟设计与实现

1）　硬件电路

硬件电路如图 10-1 所示，PA 口为 LED 数码管的 8 段码输出，PC0-PC5 共 6 个 I/O 口控制时间显示 6 个 LED 数码管的位扫描。PC6、PC7 分别接连接两个按键，用于设置时钟的工作状态和校时时间的设置。

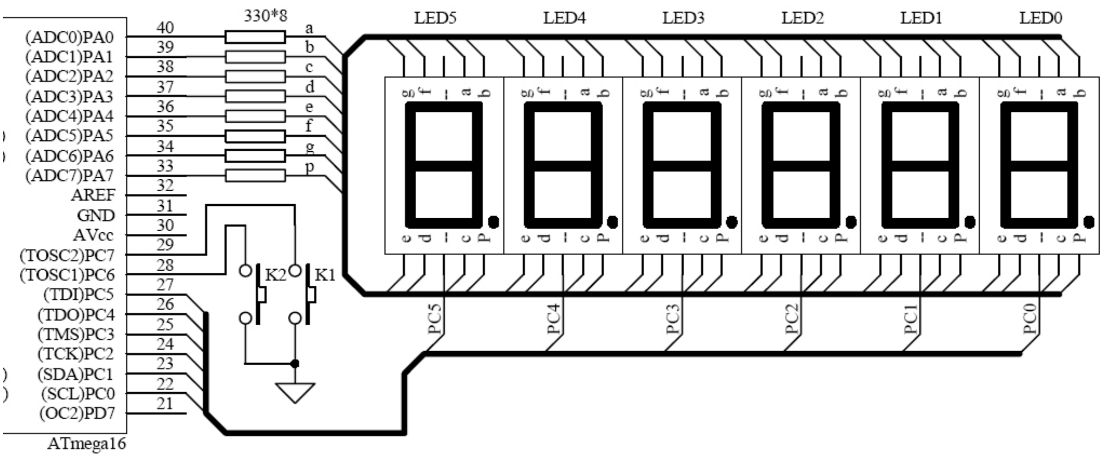

图 10-1 带校时功能的实时时钟电路图

定义两个按键的功能为：K1 用于设置转换时钟工作状态，K2 用于设置校时时间（加 1）。

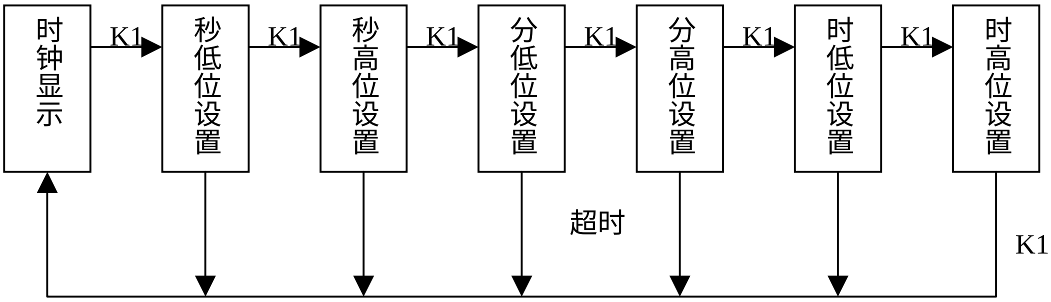

图 10-2 时钟工作状态转换图

时钟工作状态转换图如图 10-2 所示，具体每个状态的定义和功能如下：

- ✓ 平时时钟工作在时钟显示状态，每按一下 K1 键，时钟依次进入校时时间的设置状态。
- ✓ 时钟由“时钟显示”进入“秒低位设置”时，校时时间的初始值为转换时刻的时钟值。
- ✓ 时钟由“时高位设置”回到（K1 作用下）“时钟显示”时，时钟时间由校时时间代替，确认完成校时的设置。
- ✓ 当时钟处在时间设置的 6 个状态时，每按一次 K2 键，相应的位加 1，并且自动做相应的调整。
- ✓ 当时钟处在时间设置的 6 个状态时，在 20 秒内无任何键按下，系统自动返回“时间显示”状态，设置的时间无效，不改变原计时时间。
- ✓ 在效时时间设置过程中，原显示时间不停止时间的计时过程，除非当时钟由“时高位设置”回到（K1 作用下）“时钟显示”时，时钟时间由校时时间代替而改变。
- ✓ 时钟显示亮度均匀、无闪烁。当设置相应时间位时，该位应闪烁提示。

2） 软件设计

```
/***********************************************
File name          : demo_10_3.c
Chip type           : ATmega16
Program type        : Application
Clock frequency     : 4.000000 MHz
Memory model        : Small
External SRAM size  : 0
Data Stack size     : 256
***********************************************/

#include <mega16.h>
flash char led_7[10]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07,0x7F,0x6F};
flash char position[6]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf};

char time[3],time_set[3];              // 时、分、秒计数和设置单元
char dis_buff[6];                      // 显示缓冲区，存放要显示的 6 个字符的段码值
char time_counter,key_stime_counter;  // 时间计数单元，
char clock_state = 6,return_time;
bit  point_on,set_on,time_1s_ok,key_stime_ok;
```

```
void display(void)                   // 6 位 LED 数管动态扫描函数
{
    static char posit=0;
    PORTC = 0xff;
    PORTA = led_7[dis_buff[posit]];
    if (set_on && (posit==clock_state)) PORTA= 0x00;        // 校时闪烁
    if (point_on && (posit==2||posit==4)) PORTA |= 0x80;    // 秒闪烁
    PORTC = position[posit];
    if (++posit >=6 ) posit = 0;//  (3)
}
    // Timer 0 比较匹配中断服务，2ms 定时
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
    display();                            // LED 扫描显示
    if (++key_stime_counter >=5)
    {
        key_stime_counter = 0;
        key_stime_ok = 1;                               // 10ms 到
        if (!(++time_counter % 25)) set_on = !set_on;  // 设置校时闪烁标志
        if (time_counter >= 100)
        {
            time_counter = 0;
            time_1s_ok = 1;                             // 1s 到
        }
    }
}

void time_to_disbuffer(char *time)                  // 时钟时间送显示缓冲区函数
{
    char i,j=0;
    for (i=0;i<=2;i++)
    {
        dis_buff[j++] = time[i] % 10;
        dis_buff[j++] = time[i] / 10;
    }
}

#define key_input  PINC         // 按键输入口
#define key_mask   0b11000000   // 按键输入屏蔽码
#define key_no     0
#define key_k1     1
#define key_k2     2
#define key_state_0    0
#define key_state_1    1
```

```
#define key_state_2    2

char read_key(void)
{
    static char key_state = 0,key_press;
    char key_return = key_no;

    key_press = key_input & key_mask;                  // 读按键 I/O 电平
    switch (key_state)
    {
        case key_state_0:               // 按键初始态
            if (key_press != key_mask) key_state = key_state_1;
            break;                      // 键被按下，状态转换到键确认态
        case key_state_1:               // 按键确认态
            if (key_press == (key_input & key_mask))
            {
                if (key_press == 0b01000000) key_return = key_k1;
                else if (key_press == 0b10000000) key_return = key_k2;
                key_state = key_state_2;    // 状态转换到键释放态
            }
            else
                key_state = key_state_0;    // 按键已抬起，转换到按键初始态
            break;
        case key_state_2:
            if (key_press == key_mask) key_state = key_state_0;
            break;                          //按键已释放，转换到按键初始态
    }
    return key_return;
}

void main(void)
{
    char key_temp,i;

    PORTA=0x00;
    DDRA=0xFF;
    PORTC=0xFF;
    DDRC=0x3F;
        // T/C0 初始化
    TCCR0 = 0x0B;          // 内部时钟，64 分频 (4M/64=62.5KHz)，CTC 模式
    TCNT0 = 0x00;
    OCR0 = 0x7C;           // OCR0 = 0x7C(124),(124+1)/62.5=2ms
    TIMSK = 0x02;          // 允许 T/C0 比较匹配中断
```

```
time[2] = 23; time[1] = 58; time[0] = 55; // 设时间初值 23:58:55

#asm("sei")            // 开放全局中断

while (1)
{
    if (time_1s_ok)                 // 1 秒到
    {
        time_1s_ok = 0;
        point_on = ~point_on;       // 秒闪烁标志
        if (++time[0] >= 60)        // 秒加 1，以下为时间调整
        {
            time[0] = 0;
            if (++time[1] >= 60)
            {
                time[1] = 0;
                if (++time[2] >= 24) time[2] = 0;
            }
        }
        if ((++return_time >= 20) && (clock_state != 6)) clock_state = 6;
        if (clock_state == 6) time_to_disbuffer(time);
    }
    if (key_stime_ok)               // 10ms 到，键处理
    {
        key_stime_ok = 0;
        key_temp = read_key();      // 调用按键接口程序
        if (key_temp)               // 确认有键按下
        {
            return_time = 0;
            if (key_temp == key_k1)     // K1 键按下，状态转换
            {
                if (++clock_state >= 7) clock_state = 0;
                if (clock_state == 0)
                {
                    for (i=0;i<=2;i++) time_set[i] = 0;
                    time_to_disbuffer(time_set);
                }
                if (clock_state == 6)
                {
                    for (i=0;i<=2;i++) time[i] = time_set[i];
                    time_to_disbuffer(time);
                }
            }
            if ((clock_state != 6) && (key_temp == key_k2))     // K2 键按下
```

```
                    {
                        if (clock_state%2) time_set[clock_state/2] += 10;
                        else
                        {
                            if ((time_set[clock_state/2] % 10) == 9)
                                time_set[clock_state/2] -= 9;
                            else
                                time_set[clock_state/2]+=1;
                        }
                        if (time_set[0] >= 60) time_set[0]-= 60;  // 以下设置时间调整
                        if (time_set[1] >= 60) time_set[1]-= 60;
                        if (time_set[2] >= 24) time_set[2]-= 10;
                        time_to_disbuffer(time_set);            // 设置时间送显示缓存
                    }
                }
            }
        }
    }
```