9.2　按键输入接口设计

在单片机嵌入式系统中，按键和键盘是一个基本和常用的接口，它是构成人机对话通道的一种常用的方式。按键和键盘能实现向嵌入式系统输入数据、传输命令等功能，是人工干预、设置和控制系统运行的主要手段。

9.2.1 简单的按键输入硬件接口与分析

键盘是由一组按键组合构成的，所以我们先讨论简单的单个按键的输入。

图 9-2 是简单按键输入接口硬件连接电路图，图中单片机的三个 I/O 口 PC7、PC6、PC5 作为输入口（输入方式），分别与 K3、K2、K1 三个按键连接。其中 K2 是标准的连接方式，当没有按下 K2 时，PC6 的输入为高电平，按下 K2 输入为低电平。PC6 引脚上的电平值反映了按键的状态。

按键 K1 是一种经济的接法，它充分利用了 AVR 单片机 I/O 口的内部上拉特点。在 K1 的连接中，除了把 PC5 定义为输入方式时（DDRC.5=0），同时设置 PC5 口的上拉电阻有效(PORTC.5=1)，这样当 K1 处在断开状态时，PC5 引脚在内部上拉电阻的作用下为稳定的高电平（如果上拉电阻无效，则 PC5 处在高阻输入态，PC5 的输入易受到干扰，不稳定），按下 K1 输入为低电平。与 K2 连接方式比较，K1 连接电路中省掉了一个外部上拉电阻，而在 K2 的连接方法中，由于外部使用了上拉电阻，所以只要设置 PC6 口为输入方式即可，该口内部的上拉电阻有效与否则不必考虑了。

而对于 K3 的连接方式，我们不提倡使用，因当 K3 按下闭合时，PC7 口直接与 Vcc 接通了，有可能会造成大的短路电流流过 PC7 引脚，从而把端口烧毁。因此电阻 R2 不仅起到上拉的作用，还有限流的作用，通常在 5K-50K 之间。

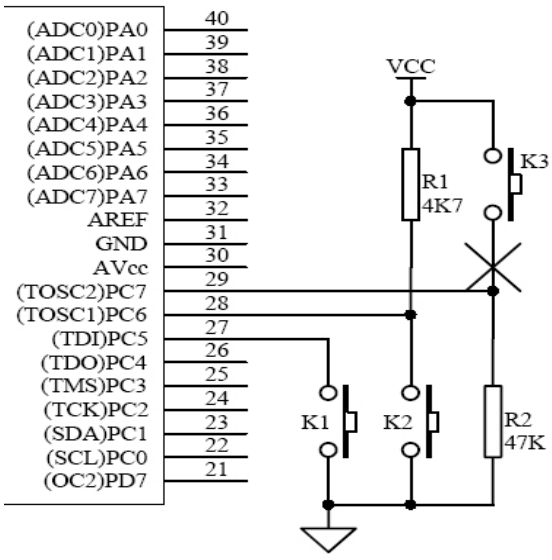

图 9-2 简单按键输入电路图

根据按键连接电路可知，按键状态的确认就是判别按键是否闭合，反映在输入口的电平就是和按键相连的 I/O 引脚呈现出高电平或低电平。如果输入高电平表示断开的话，那么低电平则表示按键闭合，所以简单的讲，在程序中通过检测引脚电平的高低，便可确认按键是否按下。

但对于实际的按键确认并不是象上面描述的那么简单。首先要考虑的是按键消抖的问题。通常，按键的开关为机械弹性触点开关，它是利用机械触点接触和分离实现电路的通、

断。由于机械触点的弹性作用，加上人们按键时的力度、方向的不同，按键开关从按下到接触稳定要经过数毫秒的弹跳抖动，既在按下的几十毫秒时间里会连续产生多个脉冲。释放按键时，电路也不会一下断开，同样会产生抖动（图 9-3）。这两次抖动的时间分别为 10-20ms 左右，而按键的稳定闭合期通常大于 0.3-0.5 秒。因此，为了确保 MCU 对一次按键动作只确认一次，在确认按键是否闭合时，必须要进行消抖处理。否则，由于 MCU 软件执行的速度很快，非常可能将抖动产生的多个脉冲误认为多次的按键。在第七章的例 7.1 中，采用了简单的中断输入按键接口，没有消抖动的功能，所以出现了按键输入控制不稳定的现象。

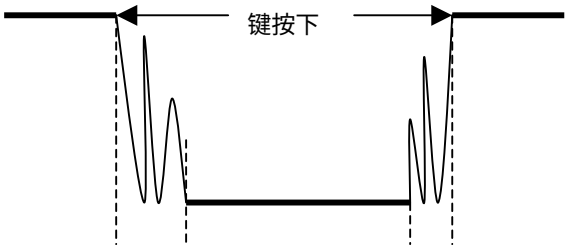

图 9-3 按键操作波形

消除按键的抖动既可采用硬件方法，也可采用软件的方法。使用硬件消抖的方式，需要在按键连接的硬件设计上增加硬件消抖电路，如采用 R-S 触发器或 RC 积分电路等。采用硬件消抖方式增加了系统的成本，而利用软件方式消抖则是比较经济的做法，但增加了软件设计的复杂性。

软件方式消抖的基本原理是在软件中对按键进行两次测试确认，既在第一次检测到按键按下后，间隔 15ms 左右再次检测该按键是否按下，只有在两次都测到按键按下时才最终确认有键按下，从而消除了抖动的影响。

在按键接口软件中，除了要考虑按键消抖外，一般还要判别按键的释放，只有检测到按键释放以后，才能确定为一次完整的按键动作完成。

9.2.2 基于状态机的按键输入软件接口设计

一般的教课书中给出的按键输入软件接口程序通常非常简单，在程序中一旦检测到按键输入口为低电平时（图 9-2），便采用（调用）软件延时程序延时 15ms。然后再次检测按键输入口，如果还是低电平则表示按键按下，转入执行按键处理程序。如果第二次检测按键输入口为高电平，则放弃本次按键的检测，从头开始一次新的按键检测过程。这种方式实现的按键输入接口，作为基础学习和一些简单的系统中可以采用，但在多数的实际产品设计中，这种按键输入软件的实现方法有很大的缺陷和不足。

上面所提到简单的按键检测处理方法，不仅是由于采用了软件延时而使的 MCU 的效率降低，而且也不容易同系统中其它功能模块协调工作，系统的实时性也差。另外，由于在不同的产品系统中对按键功能的定义和使用方式也会不同，而且是多变的，加上在测试和处理按键的同时，MCU 还要同时处理其它的任务（如显示、计算、计时等），因此编写键盘和按键接口的处理程序需要掌握有效的分析方法，具备较高的软件设计能力和程序编写的技巧。

读者可以先仔细观察一下实际产品中各种按键的功能和使用。如一般的电子手表上只有 2-3 个的按键，却要实现时间、日期、闹钟时间的设置和查看显示等多种功能，因此这些按键是多功能（或复用）的，在不同的状态下，按键的功能也不同。更典型的是手机的键盘，就拿手机键盘上的数字键“2”来将，当手机用于打电话需要拨出电话号码时，按“2”键代表数值“2”。而使用手机发短信用于输入短信文字信息时（英文输入），第一次按下“2”键为字母“A”，紧接着再次按下为字母“B”，连续短时间按下该键，它的输入代表的符号不同，但在同一个位置，而稍微等待一段时间后，光标的位置就会右移，表示对最后输入字符的确认。

因此，按键输入接口设计和实现的核心，更多的体现在软件接口处理程序的设计中。下面将以此为例，介绍有限状态机的分析设计原理，以及基于状态机思想进行程序设计的基本

方法与技巧。

一、有限状态机分析设计的基本原理

对于电子技术和电子工程类的读者，最先接触和使用到状态机应该是在数字逻辑电路课程里，状态机的思想和分析方法被应用于时序逻辑电路设计。其实，有限状态机（FSM）是实时系统设计中的一种数学模型，是一种重要的、易于建立的、应用比较广泛的、以描述控制特性为主的建模方法，它可以应用于从系统分析到设计（包括硬件、软件）的所有阶段。

很多实时系统，特别是实时控制系统，其整个系统的分析机制和功能与系统的状态有相当大的关系。有限状态机由有限的状态和相互之间的转移构成，在任何时候只能处于给定数目的状态中的一个。当接收到一个输入事件时，状态机产生一个输出，同时也可能伴随着状态的转移。

一个简单的有限状态机在数学上可以描述为：

（1） 一个有限的系统状态的集合 $S_i(t_k)=\{S_1(t_k),S_2(t_k),\cdots,S_q(t_k)\}$

其中(i = 1，2，…，q)。该式表示系统可能存（处）在的状态有 q 个，而在时刻 Tk，系统的状态为其中之一 Si（唯一性）。

（2） 一个有限的系统输入信号的集合 $I_j(t_k)=\{I_1(t_k),I_2(t_k),\cdots,I_m(t_k)\}$

其中(j = 1，2，…，m)，表示系统共有 q 个输入信号。该式表示在时刻 Tk，系统的输入信号为输入集合的全集或子集（集合性）。

（3） 一个状态转移函数 F： $S_{i'}(t_{k+1})=S_i(t_k)\times I_j(t_k)$

状态转移函数也是一个状态函数，它表示对于时刻 Tk，系统在某一状态 Si 下，相对给定输入 Ij 后，FSM 转入该函数产生的新状态，这个新状态就是系统在下一时刻（K+1）的状态。这个新的状态也是唯一确定的（唯一性）。

（4） 一个有限的输出信号集合 $O_l(S_i(t_k))=\{O_1(S_i(t_k)),O_2(S_i(t_k)),\cdots,O_n(S_i(t_k))\}$

其中(l = 1，2，…，n)，表示系统共有 n 个输出信号。该式表示对于在时刻 Tk，系统的状态为 Si 时，其输出信号为输出集合的全集或子集（集合性）。这里需要注意的是，系统的输出只与系统所处的状态有关。

（5） 时间序列 $T=\{t_0,t_1,\cdots,t_k,t_{k+1},\cdots\}$

在状态机中，时间序列也是非常重要的一个因素，从硬件的角度看，时间序列如同一个触发脉冲序列或同步信号，而从软件的角度看，时间序列就是一个定时器。状态机由时间序列同步触发，定时检测输入，以及根据当前的状态输出相应的信号，并确定下一次系统状态的转移。在时间序列进入下一次触发时，系统的状态将根据前一次的状态和输入情况发生状态的转移。其次，作为时间序列本身也可能是一个系统的输入信号，影响到状态的改变，进而也影响到系统的输出。所以对于时间序列，正确分析和考虑选择合适的时间段的间隔也是非常重要的。间隔太短的话，对系统的速度、频率响应要求高，并且可能减低系统的效率；间隔太长时，系统的实时性差，响应慢，还有可能造成外部输入信号的丢失。一般情况下，时间序列的时间间隔的选取，应稍微小于外部输入信号中变化最快的周期值。

通常主要有两种方法来建立有限状态机，一种是“ 状态转移图 ”，另一种是“ 状态转移表 ”，分别用图形方式和表格方式建立有限状态机。实时系统经常会应用于比较大型的系统中，这时采用图形或表格方式对理解复杂的系统具有很大的帮助。

总的来说，有限状态机的优点在于简单易用，状态间的关系能够直观看到。应用在实时

系统中时，便于对复杂系统进行分析。

下面将给出两个按键与显示相结合的应用设计实例，结合设计的例子，讨论如何使用有限状态机进行系统的分析和实际，以及如何在软件中进行描述和实现。

二、基于状态机分析的简单按键设计（一）

我们把单个按键作为一个简单的系统，根据状态机的原理对其动作和确认的过程进行分析，并用状态图表示出来，然后根据状态图编写出按键接口程序。

把单个按键看成是一个状态机话，首先需要对一次按键操作和确认的实际过程进行分析，根据实际的情况和系统的需要确定按键在整个过程的状态，每个状态的输入信号和输出信号，以及状态之间的转换关系。最后还要考虑时间序列的间隔。

采用状态机对一个系统进行分析是一项非常细致的工作，它实际上是建立在对真实系统有了全面深入的了解和认识的基础之上，进行综合和抽象化的模型建立的过程。这个模型必须与真实的系统相吻合，既能正确和全面的对系统进行描述，也能够适合使用软件或硬件方式来实现。

在一个嵌入式系统中，按键的操作是随机的，因此系统软件对按键需要一直循环查询。由于按键的检测过程需要进行消抖处理，因此取状态机的时间序列的周期为 10ms 左右，这样不仅可以跳过按键抖动的影响，同时也远小于按键 0.3-0.5 秒的稳定闭合期（图 9-3），不会将按键操作过程丢失。很明显，系统的输入信号是与按键连接的 I/O 口电平，“ 1 ”表示按键处于开放状态，“ 0 ”表示按键处于闭合状态（图 9-2）。而系统的输出信号则表示检测和确认到一次按键的闭合操作，用“ 1 ”表示。

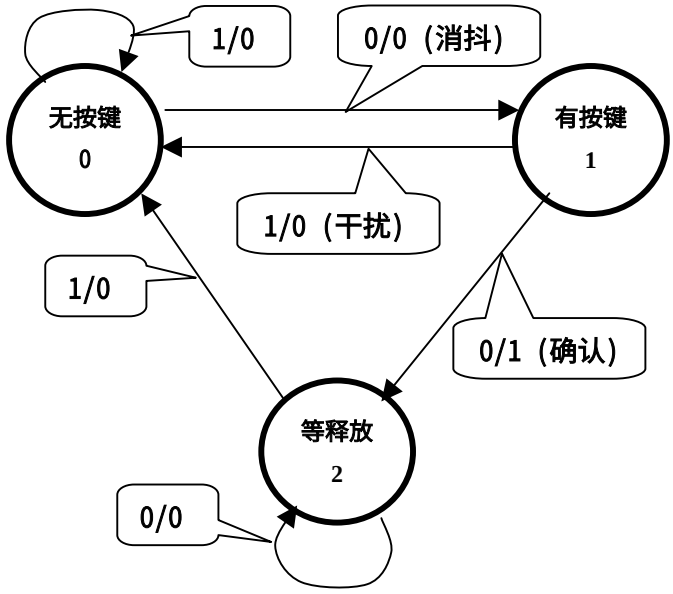

图 9-4 简单按键状态机的状态转换图

图 9-4 给出了一个简单按键状态机的状态转换图。在图中，将一次按键完整的操作过程分解为 3 个状态，采用时间序列周期为 10ms。下面对该图做进一步的分析和说明，并根据状态图给出软件的实现方法。

首先，读者要充分体会时间序列的作用。在这个系统中，采用的时间序列周期为 10ms，它意味着，每隔 10ms 检测一次按键的输入信号，并输出一次按键的确认信号，同时改变按键的状态也发生一次转换。

图中“ 状态 0 ”为按键的初始状态，当按键输入为“ 1 ”时，表示按键处于开放，输出“ 0 ”(1/0)，下一状态仍旧为“ 状态 0 ”。当按键输入为“ 0 ”，表示按键闭合，但输出还是“ 0 ”(0/0)（没有经过消抖，不能确认按键真正按下），下一状态进入“ 状态 1 ”。

“ 状态 1 ”为按键闭合确认状态，它表示了在 10ms 前按键为闭合的，因此当再次检测

到按键输入为“1”时，可以确认按键被按下了，输出“1”表示确认按键闭合（0/1），下一状态进入“状态2”。而当再次检测到按键的输入为“0”时，输出为“0”（1/0），下一状态返回到“状态0”。这样，利用状态2，实现了按键的消抖处理。

“状态3”为等待按键释放状态，因为只有等按键释放后，一次完整的按键操作过程才算完成。

从对图9-4的分析中可以知道，在一次按键操作的整个过程，按键的状态是从“状态0”->“状态1”->“状态2”，最后返回到“状态0”的。并且在整个过程中，按键的输出信号仅在“状态 1”时给出了唯一的一次确认按键闭合的信号“1”（其它状态均输出“0”）。所以上面状态机所表示的按键系统，不仅克服了按键抖动的问题，同时也确保在一次按键整个的过程中，系统只输出一次按键闭合信号（“1”）。换句话讲，不管按键被按下的时间保持多长，在这个按键的整个过程中都只给出了一次确认的输出，因此在这个设计中，按键没有“连发”功能，它是一个最简单和基本的按键。

一旦有了正确的状态转换图，就可以根据状态转换图编写软件了。在软件中实现状态机的方法和程序结构通常使用多分支结构（IF-ELSEIF-ELSE、CASE 等）实现。下面是根据图9-4、基于状态机方式编写的简单按键接口函数read_key()。

```
#define key_input      PIND.7            // 按键输入口
#define key_state_0    0
#define key_state_1    1
#define key_state_2    2

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

    key_press = key_input;               // 读按键I/O电平
    switch (key_state)
    {
        case key_state_0:                // 按键初始态
            if (!key_press) key_state = key_state_1; // 键被按下，状态转换到键确认态
            break;
        case key_state_1:                // 按键确认态
            if (!key_press)
            {
                key_return = 1;              // 按键仍按下，按键确认输出为“1”
                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_state = key_state_0;   //按键已释放，转换到按键初始态
            break;
    }
```

```
    return key_return;
}
```

该简单按键接口函数 read_key()在整个系统程序中应每隔 10ms 调用执行一次，每次执行将先读取与按键连接的 I/O 的电平到变量 key_press 中，然后进入用 switch 结构构成的状态机。switch 结构中的 case 语句分别实现了 3 个不同状态的处理判别过程，在每个状态中将根据状态的不同，以及 key_press 的值（状态机的输入）确定输出，和确定下一次按键的状态值。

函数 read_key()的返回参数提供上层程序使用。返回值为 0 时，表示按键无动作；而返回 1 表示有一次按键闭合动作，需要进入按键处理程序做相应的键处理。

在函数 read_key()中定义了 3 个局部变量，其中 key_press 和 key_return 为一般普通的局部变量，每次函数执行时，key_press 中保存着刚检测的按键值。key_return 为函数的返回值，总是先初始化为 0，只有在状态 1 中重新置 1，作为表示按键确认的标志返回。变量 key_state 非常重要，它保存着按键的状态值，该变量的值在函数调用结束后不能消失，必须保留原值，因此在程序中定义为“局部静态变量”，用 static 声明。如果使用的语言环境不支持 static 类型的局部变量，则应将 key_state 定义为全局变量（关于局部静态变量的特点请参考相关介绍 C 语言程序设计的书籍）。

例 9.1　单按键的实时时钟秒校时设置设计（一）

1）　硬件电路

在前面的章节中曾几次给出了简易实时时钟的设计例子，但都没有加入按键，不能实现时钟校时的设置。下面结合上面的按键接口的设计，实现对时钟的校时设置。在该例子中，只是实现了秒单元的校时设置，其重点是让读者体会和实践按键输入接口和处理的实现。

时钟系统的硬件电路与图 6-15 基本相同，仅在 I/O 口 PD7 上连接一个按键 K1，该按键的功能为秒加 1，既每按下一次，秒加 1，到 60 秒时分加 1，秒回到为 0。图 9-6 为电原理图。

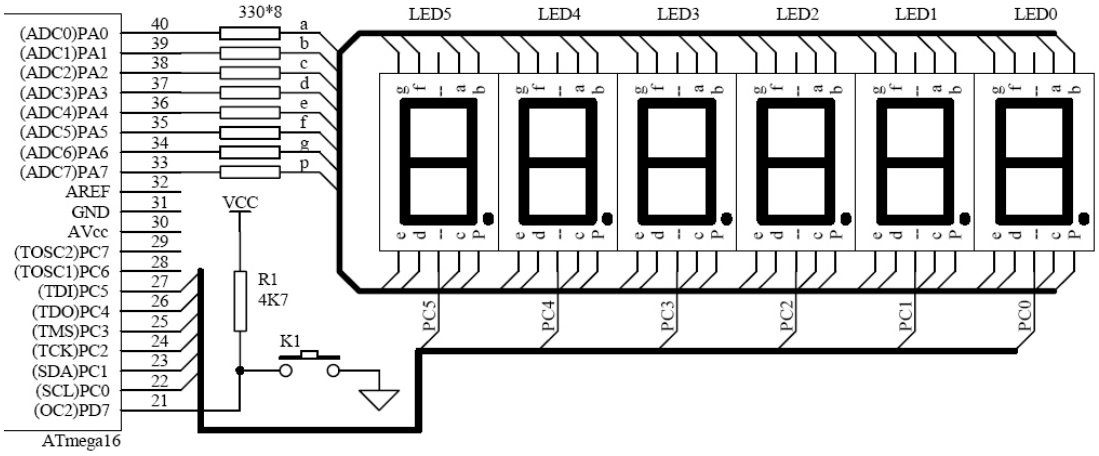

图 9-5 可实现秒校时的简单时钟电路

2）　软件设计

以下是采用一个简单按键实现秒单元的校时设置系统的程序。程序中，给出了采用一个简单按键实现时钟校时（仅秒位）设置功能的基本设计，时钟本身的计时显示方法用秒闪烁表示（每秒种 LED 数码管的小数点段闪烁一次）。

```
/*********************************************
File name          : demo_9_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[6]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf};

char time[3];                   // 时、分、秒计数单元
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 posit;
bit  point_on, time_1s_ok,key_stime_ok;

void display(void)                  // 6 位 LED 数码管动态扫描函数
{
         PORTC = 0xff;
         PORTA = led_7[dis_buff[posit]];
         if (point_on && (posit==2||posit==4)) PORTA |= 0x80;
         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 >= 100)
              {
                   time_counter = 0;
                   time_1s_ok = 1;            // 1s 到
              }
         }
}

void time_to_disbuffer(void)                // 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  PIND.7            // 按键输入口
#define key_state_0    0
#define key_state_1    1
#define key_state_2    2

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

    key_press = key_input;              // 读按键 I/O 电平
    switch (key_state)
    {
        case key_state_0:               // 按键初始态
            if (!key_press) key_state = key_state_1; // 键被按下，状态转换到键确认态
            break;
        case key_state_1:               // 按键确认态
            if (!key_press)
            {
                key_return = 1;             // 按键仍按下，按键确认输出为"1"  (1)
                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_state = key_state_0;  //按键已释放，转换到按键初始态
            break;
    }
    return key_return;
}

void main(void)
{
        PORTA = 0x00;                   // 显示控制 I/O 端口初始化
```

```
    DDRA = 0xFF;
    PORTC = 0x3F;
    DDRC = 0x3F;
    DDRD = 0x00;                    // PD7 为输入方式
    // 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
    posit = 0;
    time_to_disbuffer();

    #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 (key_stime_ok)               // 10ms 到，键处理
        {
            key_stime_ok = 0;
            if (read_key())             // 调用按键接口程序
            {                           // 按键确认按下
                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;
                    }
                }
            }
            time_to_disbuffer();        // 新调整好的时间送显示缓冲区
        }
    };
}
```

该程序中 LED 数码管动态扫描等部分与前面介绍的例子相同，T/C0 的工作与 CTC 方式，每 2ms 产生中断。在 T/C0 中断服务中增加了 10ms 到的标志变量 key_time_ok，作为按键状

态机的时间触发序列信号。在主程序中，每隔 10ms 调用 read_key()按键接口程序，当函数返回值为 1 时，说明产生了一次按键操作过程，秒位加 1，然后进行时间调整。

3） 思考与实践

- ✓ 仔细分析 read_key()函数的调用和执行情况，说明为什么该按键接口程序可以正确的处理一次按键的操作过程？
- ✓ 如果将 read_key()中标有（1）的语句 key_return = 1 去掉，而将状态 2 的处理程序改为：

```
case key_state_2:
        if (key_press) key_state = key_state_0;   //按键已释放，转换到按键初始态
    else  key_return = 1;
   break;
```

或：

```
case key_state_2:
    if (key_press)
    {     key_state = key_state_0;  //按键已释放，转换到按键初始态
          key_return = 1;
    }
    break;
```

时，按键处理会产生什么变化？为什么？

三、基于状态机分析的简单按键设计（二）

在上面的例子中，对秒的校时设置还是不方便，例如，如果将秒从 00 设置为 59，需要按键 59 次。如果将按键设计成一个具有“连发”功能的系统，就能很好的解决这个问题。

现在考虑这样的一个按键系统：当按键按下后在 1 秒内释放了，此时秒计时加 1，而当按键按下后在 1 秒内没有释放，那么每隔 0.5 秒，秒计时就会自动加上 10，直到按键释放为止。这样的按键系统就具备了一种“连发”功能，其中时间也成为系统的一个输入参数了。

参考图 9-4，根据状态机的分析方法，可以得到具有上述“连发”功能的按键系统状态转换图（图 9-6）。

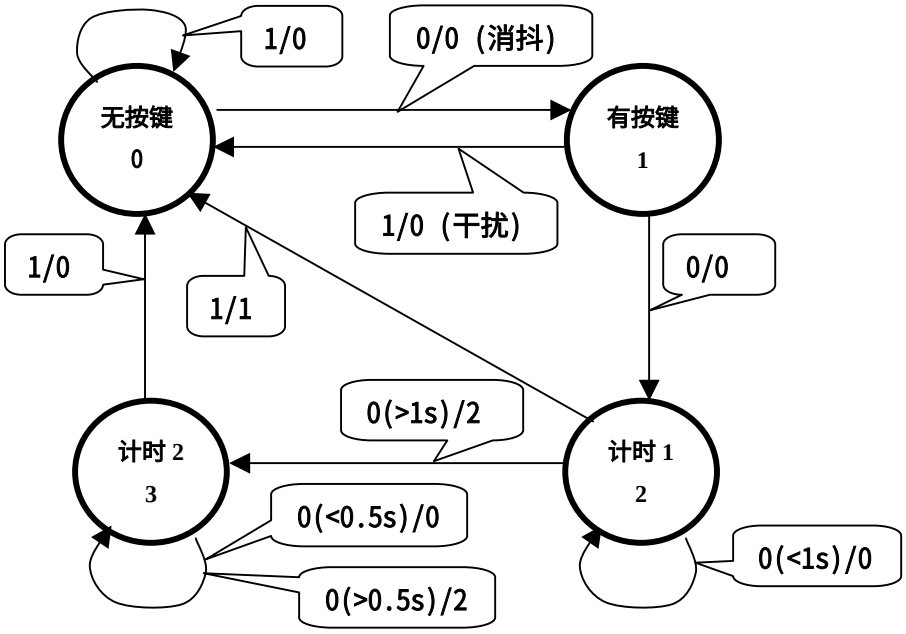

图 9-6 具有连发功能按键状态机的状态转换图

在图 9-6 所示的图中，按键系统输出“ 1 ”表示按键在 1 秒内释放了，输出“ 2 ”表示按键产生一次“ 连发 ”的效果。根据图 9-4，下面是基于状态机方式编写的带“ 连发 ”功能按键接口函数 read_key_n()。

```
#define key_input      P1ND.7            // 按键输入口
#define key_state_0    0
#define key_state_1    1
#define key_state_2    2
#define key_state_3    3

char read_key_n(void)
{
    static char key_state = 0, key_time = 0;
    char key_press, key_return = 0;

    key_press = key_input;               // 读按键 I/O 电平
    switch (key_state)
    {
        case key_state_0:                // 按键初始态
            if (!key_press) key_state = key_state_1; // 键被按下，状态转换到键确认态
            break;
        case key_state_1:                // 按键确认态
            if (!key_press)
            {
                key_state = key_state_2;    // 按键仍按下，状态转换到计时 1
                key_time = 0;               // 清另按键时间计数器
            }
            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_state = key_state_0;    // 按键已释放，转换到按键初始态
                key_return = 1;             // 输出“ 1 ”
            }
            else if (++key_time >= 100)     // 按键时间计数
            {
                key_state = key_state_3;    // 按下时间>1s，状态转换到计时 2
                key_time = 0;               // 清按键计数器
                key_return = 2;             // 输出“ 2 ”
            }
            break;
        case key_state_3:
            if (key_press) key_state = key_state_0; //按键已释放，转换到按键初始态
            else
```

```
            {
                if (++key_time >= 50)       // 按键时间计数
                {
                    key_time = 0;           // 按下时间>0.5s，清按键计数器
                    key_return = 2;         // 输出“2”
                }
            }
            break;
        }
        return key_return;
    }
```

函数 read_key_n()与前面的 read_key()结构非常类似，设计为每隔 10ms 被调用执行一次，在构成状态机的 switch 结构中使用了局部静态变量 key_time 作为按键时间计数器，记录按键按下的时间值。

函数 read_key_n()的返回参数提供上层程序使用。返回值为 0 时，表示按键无动作；返回 1 表示一次按键的“单发”(<1s）动作；返回 2 表示一次按键的“连发”动作，提供按键处理程序做相应的键处理。

例 9.2　单按键的实时时钟秒校时设置设计（二）

1）　硬件电路

硬件电路还是图 9-6，在 I/O 口 PD7 上连接一个按键 K1，该按键为具备“连发”功能，按键按下后并在 1 秒内释放，此时秒计时加 1；而当按键按下后在 1 秒内没有释放，那么每隔 0.5 秒，秒计时就会自动加上 10，直到按键释放为止。

2）　软件设计

以下是采用具备“连发”功能按键实现秒单元的校时设置系统的程序，程序中的显示等部分与 demo_9_1.c 相同，仅仅在主程序中做了相应的改变。

```
/*****************************************
File name         : demo_9_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[6]={0xfe,0xfd,0xfb,0xf7,0xef,0xdf};

char time[3];                   // 时、分、秒计数单元
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;
char posit;
```

```
bit  point_on, time_1s_ok,key_stime_ok;

void display(void)                    // 6 位 LED 数码管动态扫描函数
{
        ……
}

// Timer 0 比较匹配中断服务,2ms 定时
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
        ……
}

void time_to_disbuffer(void)                    // 时钟时间送显示缓冲区函数
{
        ……
}

#define key_input      PIND.7             // 按键输入口
#define key_state_0    0
#define key_state_1    1
#define key_state_2    2
#define key_state_3    3

char read_key_n(void)
{
        ……
}

void main(void)
{
        PORTA = 0x00;                                // 显示控制 I/O 端口初始化
        DDRA = 0xFF;
        PORTC = 0x3F;
        DDRC = 0x3F;
        DDRD = 0x00;
        // 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
        posit = 0;
```

```
        time_to_disbuffer();

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

        while (1)
        {
            if (time_1s_ok)
            {
                time_1s_ok = 0;
                point_on = ~point_on;           // 1 秒到，秒闪烁标志取反
        }
            if (key_stime_ok)
            {
                key_stime_ok = 0;               // 10ms 到
                switch (read_key_n())
                {
                    case 1:
                        ++time[0];
                        break;
                    case 2:
                        time[0] += 10;
                        break;
                }
                if (time[0] >= 60)      // 按键确认按下，秒加 1，以下时间调整
                {
                    time[0] -= 60;
                    if (++time[1] >= 60)
                    {
                        time[1] = 0;
                        if (++time[2] >= 24) time[2] = 0;
                    }
                }
                time_to_disbuffer();        // 新调整好的时间送显示缓冲区
            }
        };
    }
```

主程序中，每隔 10ms 调用 read_key_n()按键接口程序，并根据其返回值对秒位的时间进行加 1 或加 10 的处理。

9.3　键盘输入接口设计

在上节中按键的连接方法采用的是独立式按键接口方式。独立式按键就是各按键相互独立，每个按键各占用一位 I/0 的口线，它们之间状态是独立的，相互之间没有影响，只要单独测试口线电平的高低就能判断键的状态。独立式按键电路简单、配置灵活，软件结构也相对简单。此种接口方式适用于系统需要按键数目较少的场合。在按键数量较多的情况下，如

系统需要 12 或 16 个按键的键盘时，采用独立式接口方式就会占用太多的 I/O 口，因此一般适用于按键数量多的硬件连接方式往往使用矩阵式键盘接口。

9.3.1 矩阵键盘的工作原理和扫描确认方式

当键盘中按键数量交多时，为了介绍对 I/O 口的占用，通常将按键排列成矩阵形式，也称为行列键盘，这是一种常见的连接方式。矩阵式键盘接口见图 9-7 所示，它由行线和列线组成，按键位于行、列的交叉点上。当键被按下时，其交点的行线和列线接通，相应的行线或列线上的电平发生变化，MCU 通过检测行或列线上的电平变化可以确定哪个按键被按下。

图 9-7 为一个 4 x 3 的行列结构，可以构成 12 个键的键盘。如果使用 4 x 4 的行列结构，就能组成一个 16 键的键盘。很明显，在按键数量多的场合，矩阵键盘与独立式按键键盘相比可以节省很多的 I/O 口线。

矩阵键盘不仅在连接上比单独式按键复杂，它的按键识别方法也比单独式按键复杂。在矩阵键盘的软件接口程序中，常使用的按键识别方法有行扫描法和线反转法。这两种方法的基本思路是采用循环查循的方法，反复查询按键的状态，因此会大量占用 MCU 的时间，所以较好的方式也是采用状态机的方法来设计，尽量减少键盘查询过程对 MCU 的占用时间。

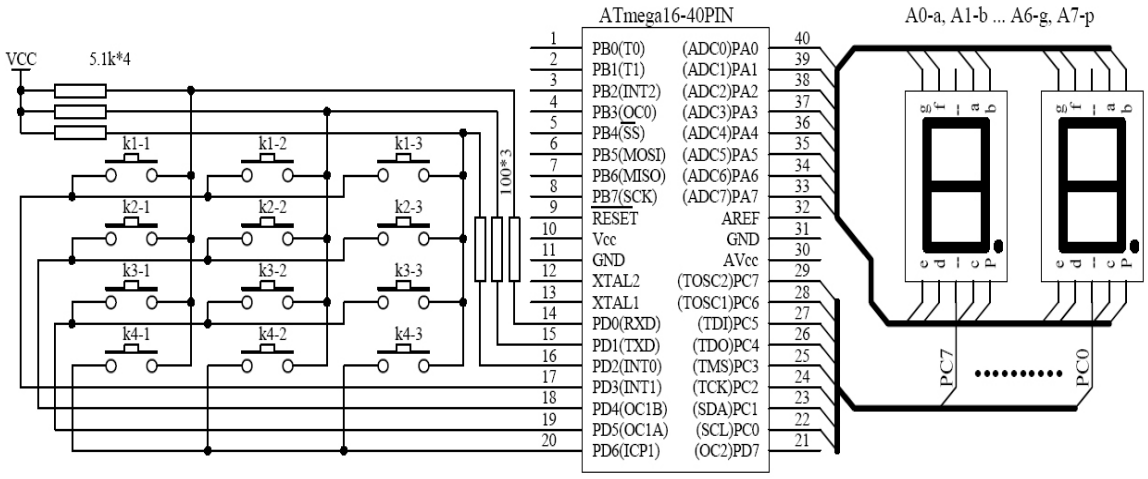

图 9-7 4*3 矩阵键盘的组成

下面以图 9-7 为例，介绍采用行扫描法对矩阵键盘进行判别的思路。图 9-7 中，PD0、PD1、PD2 为 3 根列线，作为键盘的输入口（工作于输入方式）。PD3、PD4、PD5、PD6 为 4 根行线，工作于输出方式，由 MCU（扫描）控制其输出的电平值。行扫描法也称为逐行扫描查询法，其按键识别的过程如下。

- ✓ 将全部行线 PD3－PD6 置低电平输出，然后读 PD0－PD2 三根输入列线中有无低电平出现。只要有低电平出现，则说明有键按下（实际编程时，还要考虑按键的消抖）。如读到的都是高电平，则表示无键按下。
- ✓ 在确认有键按下后，需要进入确定具体哪一个键闭合的过程。其思路是：依次将行线置为低电平，并检测列线的输入（扫描），进而确认是具体的按键位置。如当 PD5 输出低电平时（PD3=1、PD4=1、PD5=0、PD6=1），测到 PD1 的输入为低电平（PD0=1、PD1=0、PD2=1），则可确认按键 K3-2 处于闭合状态。通过以上分析可以看出，MCU 对矩阵键盘的按键识别，是采用扫描方式控制行线的输出和检测列线的信号相配合实现的。
- ✓ 矩阵按键的识别仅仅是确认和定位了行和列的交叉点上的按键，接下来还要考虑键盘的编码，即对各个按键进行编号。在软件中常通过计算的方法或查表的方法对按

键进行具体的定义和编号。

在单片机嵌入式系统中，键盘扫描只是 MCU 的工作内容之一。MCU 除了要检测键盘和处理键盘操作之外，还要进行其他事物的处理，因此，MCU 如何响应键盘的输入需要在实际系统程序设计时认真考虑。

键盘扫描处理的设计原则是：既要保证 MCU 能及时的判别按键的动作，处理按键输入的操作，又不能过多占用 MCU 的工作时间，让它有充裕的时间去处理其它的操作。

通常，完成键盘扫描和处理的程序是系统程序中的一个专用子程序，MCU 调用该键盘扫描子程序对键盘进行扫描和处理的方式有三种：程序控制扫描、定时扫描和中断扫描。

- ✓ 程序控制扫描方式。在主控程序中的适当位置调用键盘扫描程序，对键盘进行读取和处理。
- ✓ 定时扫描方式。在该方式中，要使用 MCU 的一个定时器，使其产生一个 10ms 的定时中断，MCU 响应定时中断，执行键盘扫描，当在连续两次中断中都读到相同的按键按下（间隔 10ms 作为消抖处理），MCU 才执行相应的键处理程序。
- ✓ 中断方式。使用中断方式时，键盘的硬件电路要做一定的改动，增加一个按键产生中断信号的输入线，当键盘有按键按下时，键盘硬件电路产生一个外部的中断信号，MCU 响应外部中断，进行键盘处理。

下面我们介绍基于状态机并采用定时键盘扫描的键盘处理系统的设计方法。

9.3.2 定时扫描方式的键盘接口程序

根据图 9-7，下面的键盘接口函数 read_keyboaed()完成了对 4*3 键盘的扫描识别和键盘的编码。编码键盘的定义使用 define 语句定义，键盘的形式类似电话和手机键盘，如图 9-8

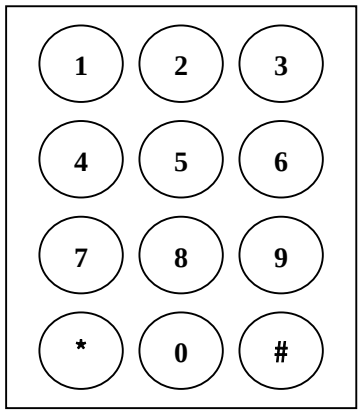

图 9-8 手机键盘

所示。

```
#define No_key  255
#define K1_1    1
#define K1_2    2
#define K1_3    3
#define K2_1    4
#define K2_2    5
```

```
#define K2_3    6
#define K3_1    7
#define K3_2    8
#define K3_3    9
#define K4_1    10
#define K4_2    0
#define K4_3    11
#define Key_mask    0b00000111

char read_keyboard()
{
  static char key_state = 0, key_value, key_line;
  char key_return = No_key,i;
  switch (key_state)
  {
      case 0:
          key_line = 0b00001000;
          for (i=1; i<=4; i++)                    // 扫描键盘
          {
              PORTD = ~key_line;                  // 输出行线电平
              PORTD = ~key_line;                  // 必须送 2 次 !!!
               key_value = Key_mask & PIND;       // 读列电平
              if (key_value == Key_mask)
                  key_line <<= 1;                 // 没有按键，继续扫描
                  else
                  {
                      key_state++;                    // 有按键，停止扫描
                      break;                          // 转消抖确认状态
                  }
          }
          break;
      case 1:
          if (key_value == Key_mask & PIND)       // 再次读列电平，
          {
              switch (key_line | key_value)       // 与状态 0 的相同，确认按键
              {                                   // 键盘编码，返回编码值
                  case 0b00001110:
                      key_return = K1_1;
                      break;
                  case 0b00001101:
                      key_return = K1_2;
                      break;
                  case 0b00001011:
                      key_return = K1_3;
```

```
                    break;
                case 0b00010110:
                    key_return = K2_1;
                    break;
                case 0b00010101:
                    key_return = K2_2;
                    break;
                case 0b00010011:
                    key_return = K2_3;
                    break;
                case 0b00100110:
                    key_return = K3_1;
                    break;
                case 0b00100101:
                    key_return = K3_2;
                    break;
                case 0b00100011:
                    key_return = K3_3;
                    break;
                case 0b01000110:
                    key_return = K4_1;
                    break;
                case 0b01000101:
                    key_return = K4_2;
                    break;
                case 0b01000011:
                    key_return = K4_3;
                    break;
            }
            key_state++;                // 转入等待按键释放状态
        }
        else
            key_state--;                // 两次列电平不同返回状态 0，(消抖处理)
        break;
    case 2:                             // 等待按键释放状态
        PORTD = 0b00000111;             // 行线全部输出低电平
        PORTD = 0b00000111;             // 重复送一次
        If ( (Key_mask & PIND) == Key_mask)
            key_state=0;                // 列线全部为高电平返回状态 0
        break;
  }
  return key_return;
}
```

系统主程序应每隔 10ms 调用该键盘接口函数 read_keyboaed()，函数返回值为 255 时

表示无按键按下。检测和确认按键按下时，函数返回值为 0 到 11 之间的一个，该返回值已经是经过了键盘编码的值。

键盘接口函数 read_keyboaed()是基于状态机实现的，将键盘扫描处理过程化分成三个状态，每个状态的功能为：

- ✓ 状态 0，键盘扫描检测。控制 PD3-PD4，4 根行线逐行输出低电平，对键盘进行扫描检测。一旦检测到有键按下（key_value），立即停止键盘的扫描，状态转换到状态 1。注意此时变量 key_value 中保存着读到的列线输入值，而且该行线低电平的输出是保持不变的。
- ✓ 状态 1，消抖处理和键盘编码。再次检测键盘列线的输入，并与状态 0 时的 key_value 比较，不相等则返回状态 0，实现了消抖处理。相等则确认该键的输入，进行键盘编码和设置函数的返回值，状态转化到状态 2。
- ✓ 状态 2，等待按键释放。控制 PD3-PD4，4 根行线全部输出低电平，检测 3 根列线输入全部为高电平（无按键按下）时状态返回到状态 0。

读者在阅读该段程序时，请注意 key_mask、key_value、key_line 的作用和使用，它们不仅与键盘的硬件连接有关系，同时还要注意他们在程序中是如何使用的，其值的保存等等。

通过这个例子也可以看出，在硬件设计的过程中，也需要认真考虑软件的编写。比如，你也可以将 4 根键盘行线与 PD0、PD2、PD3、PD6 连接，列线使用 PD1、PD4、PD5，从硬件的角度看是完全可以的，但会给软件编写造成很多麻烦。所以，一个好的嵌入式系统工程师必须同时具备良好的软件设计编写能力和硬件设计能力。

实际上 read_keyboaed()还是一个比较简单的键盘接口函数，还不能处理类似多键按下的识别、按键“连发”等功能。但当你真正掌握了基本键盘接口的设计思想和方法后，就可以在这个简单的键盘接口函数基础上，设计出功能更加完善的键盘系统了。

例 9.3　简单电话拨号键盘的设计

1）　硬件电路

在这个例子中，结合图 9-7 的硬件电路和图 9-8 定义的键盘，实现一个简单的电话拨号键盘。系统由 PA、PC 口控制的 8 个 LED 数码管和 PD 口的 4*3 键盘组成，系统上电时，8 个 LED 数码管显示“--------”8 条横线，每按下一个号码后，原 8 位 LED 数码管的显示内容向左移动一位，最右边一位则显示键盘上刚按下的数字（“*”键用“A”表示，“#”键用“b”表示）。要求：对键盘按键操作的反应迅速而且无误，同时按键操作过程中应保证 LED 的扫描显示均匀、连续不间断。

2）软件设计

```
/*********************************************
File name          : demo_9_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[13]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,   // 字型码
                      0x7D,0x07,0x7F,0x6F,0x77,0x7C,0x40};
```

```
                                        // 后 3 位为 “ A ”, “ b ”, “ - ”
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 key_stime_counter;
char posit;
bit  key_stime_ok;

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

// Timer 0 比较匹配中断服务,2ms 定时
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
         display();                             // 调用 LED 扫描显示
         if (++key_stime_counter >=5)
         {
              key_stime_counter = 0;
              key_stime_ok = 1;                 // 10ms 到
         }
}
#define No_key     255
#define K1_1  1
#define K1_2  2
#define K1_3  3
#define K2_1  4
#define K2_2  5
#define K2_3  6
#define K3_1  7
#define K3_2  8
#define K3_3  9
#define K4_1  10
#define K4_2  0
#define K4_3  11
#define Key_mask   0b00000111

char read_keyboard()
{
         static char key_state = 0, key_value, key_line;
```

```
            char key_return = No_key,i;
            ……                          // 同上面 read_keyboard()
        }

    void main(void)
    {
            char i, key_temp;

            PORTA = 0x00;                           // 显示控制 I/O 端口初始化
            DDRA = 0xFF;
            PORTC = 0xFF;
            DDRC = 0xFF;
            PORTD = 0xFF;                           // 键盘接口初始化
            DDRD = 0xF8;                            // PD2、PD1、PD0 列线，输入方式，上拉
有效
            // 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

            for (i=0; i<8 ;i++)
            {dis_buff[i]= 12;}      // LED 初始显示 8 个“ - ”
            #asm("sei")             // 开放全局中断

            while (1)
            {
                if (key_stime_ok)
                {
                    key_stime_ok = 0;               // 10ms 到
                    key_temp = read_keyboard();     // 调用键盘接口函数读键盘
                    if (key_temp != No_key)
                    {                               // 有按键按下
                        for (i=0; i<7; i++)
                        {dis_buff[i] = dis_buff[i+1];}  // LED 显示左移一位
                        dis_buff[7] = key_temp;         // 最右显示新按下键的键值
                    }
                }
            };
    }
```

3）思考与实践

程序中 LED 扫描和定时器的使用与以前的方式相同，每隔 2ms 进行 LED 的扫描，同时每隔 10ms 调用 read_keyboard()键盘接口程序读键盘，并根据返回结果调整 LED 显示的内容。程序本身不复杂，主要体会键盘接口程序的功能和使用，请读者自己分析并实现。

- ✓ 本例中，在 T/C0 的中断服务中进行了 LED 的扫描，而读键盘和键盘处理是在主程序中完成的。如果将读键盘和键盘处理也放在 T/C0 中断中完成是否可以？请深入分析这两种处理方式的优点和缺点，说明原因。
- ✓ 在 read_keyboard()中，行线输出语句为什么重复 2 次？
- ✓ 说明在 read_keyboard()中，key_mask 的作用，另外是否可以将变量 key_line 和 key_value 定义成普通的局部动态变量？为什么？
- ✓ 修改程序，键盘上数字键功能不变，而“ # ”键的功能为总清除（即清除 LED 上的全部的数字显示，显示复原为 8 个“ - ”），“ * ”键的功能为修改键（表示最后输入的数字有误，LED 显示全部左移一位，清楚最后输入的数字，最左边一位补入“ - ”）。