FUJITSU

富士通 16 位微控制器

C 语言手册

应用篇

富士通复旦应用研究中心

目录

第一章　键盘接口及 C 编程

键盘是由若干按键组成的开关矩阵，它是微型计算机最常用的输入设备，用户可以通过键盘向计算机输入指令、地址和数据。键盘分为编码键盘和非编码键盘，非编码键盘是由软件来识别键盘上的闭合键，它具有结构简单，使用灵活等特点，因此被广泛应用于单片机系统。这里我们主要讨论未编码键盘的工作原理、接口技术和程序设计。

第一节　键盘工作原理

一个 4 x 4 的键盘结构如图 1.1 所示。行线通过电阻接正电源，并将行线接到单片机的输入口，而将列线接到单片机的输出口。这样，当按键没有按下时，行线都呈高电平。

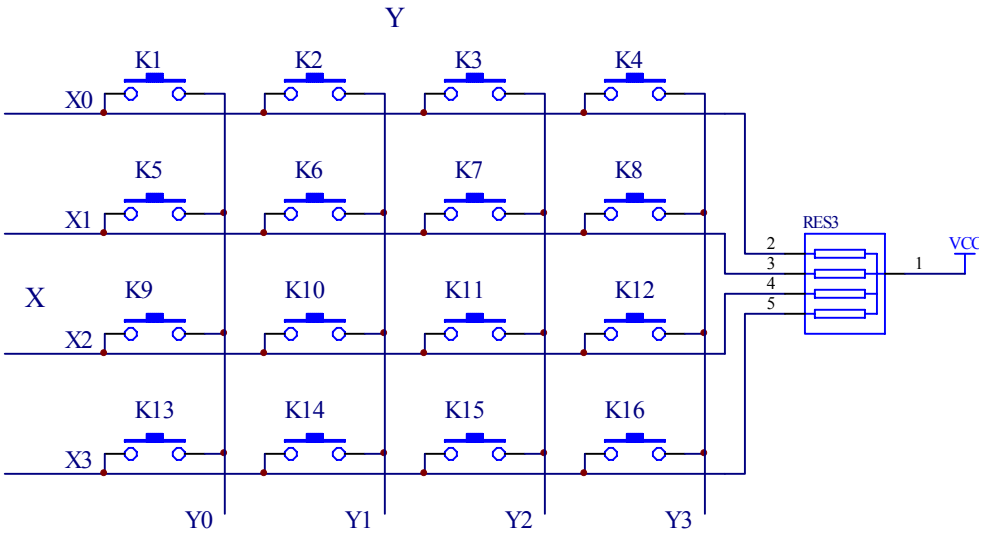


图 **1.1.** 键盘结构

如果列线全输出低电平，一旦有键按下，该键相应的行线和列线被短路，行线就会被拉低。这样，如果读入的行线状态不是全高，就表示有键按下了。

要确定是哪个键闭合，先使列线 Y0 为低电平，其余列线为高电平，读行线状态，如果不全为高，则被按下的键就是为低电平的行线和 Y0 相交的键；如果行线全为高，则 Y0 这一列上没有键闭合。接着使列线 Y1 为低电平，其余列线为高电平，用同样方法检查 Y1 这一列有无键闭合。以此类推，最后使列线 Y3 为低电平，其余列线为高电平，检查 Y3 这一列有无键闭合。这种逐行逐列的检查键盘状态的过程称为对键盘的一次扫描，用这种方法确定闭合键的键号。

单片机中应用的键盘一般是由机械触点构成的。由于按键是机械触点，当机械触点断开、闭合时，会有抖动，抖动时间的长短一般为 5~10ms，如图 1.2 所示。这种抖动对于人来说是感觉不到的，但对计算机来说，则是完全可以感应到的，因为计算机处理的速度是在微秒级，而机械抖动的时间至少是毫秒级。为了保证对每一次按键只作一次处理，就必须考虑去除抖动，在键的稳定闭合和断开时读键的状态。

图 **1.2.** 键按下和释放时的行线电压波形

第二节　键盘接口方法

图 1.3 是 4 x 4 键盘和 MB90560 的接口电路。P5 口的低 4 位作为 MB90560 的输输出口，控制键盘的列线 Y0~Y3 的电位；P5 口的高 4 位作为 MB90560 的输入口，接键盘的行线。

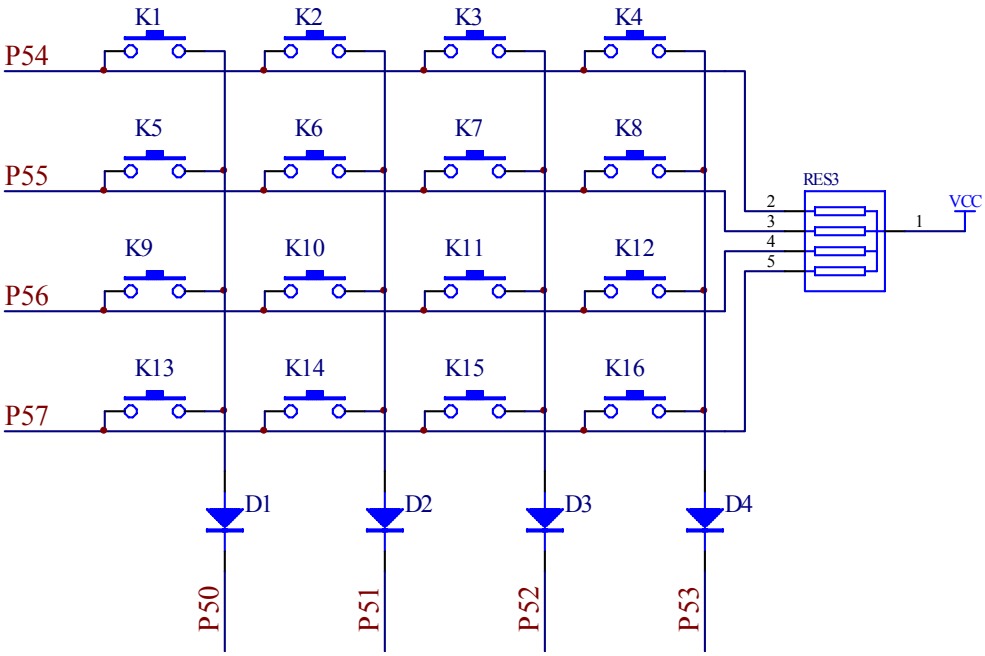


图 **1.3.** 键盘接口电路

第三节　键扫描 C 语言程序设计

1.3.1 键输入程序设计方法

键输入程序的功能应该包括以下 4 个方面的内容：

1． 判别键盘上有无键闭合：

判别方法为将键盘列线全部输出“0”，读入各行线的状态，如果全为“1”，则键盘上没有键闭合，若不全为“1”，则键盘上有键处于闭合状态。

2． 去除键的机械抖动：

方法为判别到键盘上有键闭合后，延迟一段时间在判别键盘的状态，若仍然有键闭合，则认为键盘上有一个键处于稳定的闭合状态，否则就认为是键的抖动。

3． 判别闭合键的键号：

对键盘的列线进行扫描，扫描口 P5.0~P5.3 依次输出：

P5.3	P5.2	P5.1	P5.0
1	1	1	0
1	1	0	1
1	0	1	1
0	1	1	1

依次读 P5.4~P5.7 的状态，如果 P5.4~P5.7 全为“1”，则这一列上没有键闭合。若不全为“1”，则为“0”的行线与这一列相交的那个键闭合了，闭合键的键号等于这一列的列号加上输入为“0”的行的首键号。

例如，P5 口低 4 位输出为 1101 时，读出 P5 口高 4 位的状态为 1101，则 1 行 1 列相交的键处于闭合状态，第 1 行的首键号为 8，第一列列号为 1，闭合键的键号就是：

$$N = \text{行首键号} + \text{列号} = 8+1 = 9$$

4． 对键的一次闭合只做一次处理：

采用的方法为等闭合键释放以后再做处理。

1.3.2 CPU 扫描方式

CPU 对键盘扫描可以采取程控扫描方式，也就是只有当 CPU 空闲时才调用键输入子程序，响应操作员的键输入请求；也可以采取定时控制方式，每隔一段时间，CPU 对键盘扫描一次，CPU 可随时响应键输入请求，这称为定时扫描方式；还可以采取中断方式，当键盘上有键闭合时，向 CPU 请求中断，CPU 响应键盘输入中断，对键盘扫描，以判别闭合键键号并做出处理。

1.3.2.1 程控扫描方式

程控扫描方式就是当 CPU 空闲时调用键输入程序，响应操作员的键入请求。程控扫描方式的硬件接线如图 1.3，程序流程如图 1.4 所示。

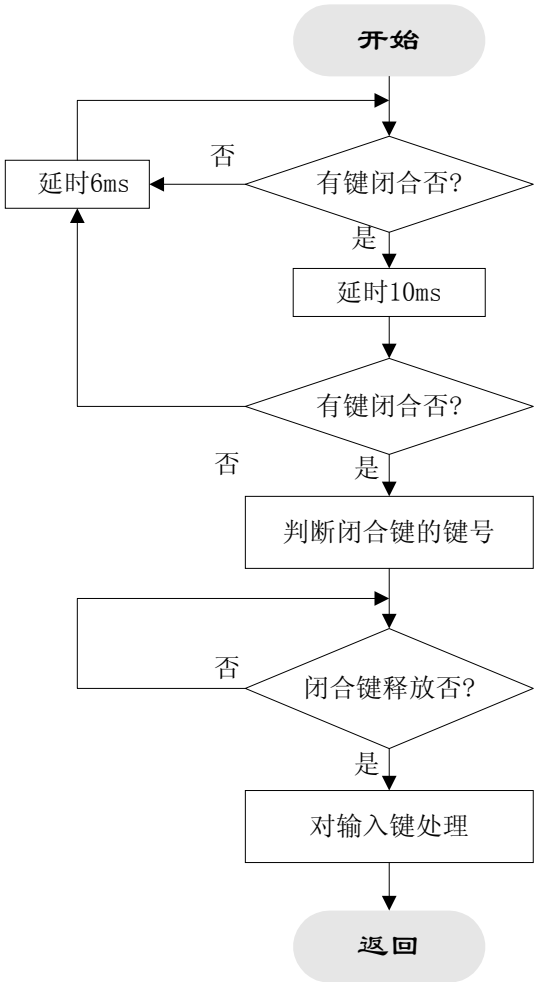


图 **1.4.** 程控扫描方式键输入程序流程

1.3.2.2 定时扫描方式

定时中断服务程序流程如图 1.5 所示。

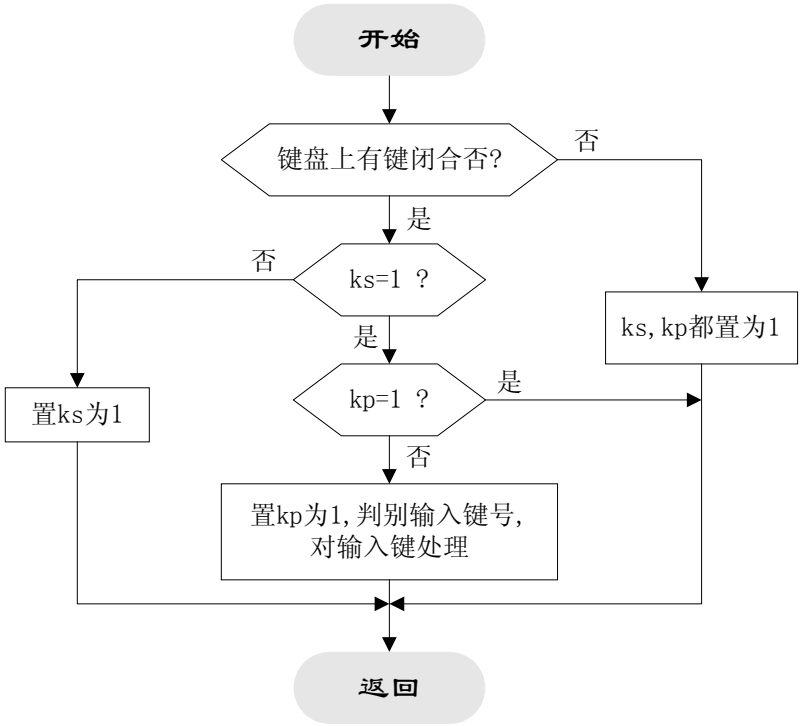


图 **1.5.** 定时扫描中断程序流程图

定时扫描方式利用内部的定时器，产生 10ms 的定时中断，CPU 响应中断时对键盘进行扫描，以响应键输入请求。流程图中包括了两个全局变量：去除键抖动标志 ks（为 1 表示已去除抖动）和处理标志 kp（为 1 表示闭合键已处理过）。

本例中采用 16 位重装入定时器 1 产生定时，程序开始时应定义定时器中断处理程序。

```
__interrupt void    timer_scan(void);
#pragma intvect    timer_scan 30
```

设置了三个全局变量，kp 是处理标志，ks 是去除键抖动标志，另外还设了一个全局变量：name，用于传递闭合键键号。

```
unsigned char name;
unsigned char kp;
unsigned char ks;
```

初始化重装入定时器 inittimer () 中对定时器重装入值做了设定，以产生 10ms 定时。有关重装入定时器的寄存器设置可参考富士通 16 位微控制器硬件手册重装入定时器章节。

```
void inittimer()
```

```
{
    IO_ICR09.byte = 0;                          /* 设置中断级别 */
    IO_TMCSR0.bit.CNTE = 0;                     /* 禁止计数 */
    IO_TMR0 = 0x1388;                           /* 设定重装入数值，产生定时 10ms */
    IO_TMCSR0.word = 0x81B;                     /* 采用内部时钟模式 */
                                                /* 禁止外部出发和外部输出 */
                                                /* 重装入模式，使能中断 */
}
```

kclose () 函数判断键盘上是否有键闭合。采用的方法如前所述：列线输出低电平，读出行线状态。若不是全为高电平，则有键闭合，返回值为 1。

```
char kclose()
{
    int a,z;
    IO_PDR5.byte=0xF0;                          /* 列线输出低电平 */
    a=IO_PDR5.byte;                             /* 读行线状态 */
    if(a==0xF0)         z=0;                    /* 若行线全为高电平，没有键闭合 */
    else z=1;
    return (z);                                 /* 返回值为 0，没有闭合键；为 1，有 */
}
```

keyname () 用于判断闭合键的键号，即进行一次扫描。逐列输出低电平，察看各行线状态，以确定闭合键的键号。

```
Char    keyname()
{
    int         i;
    char        a,b,x;
    a=0;                                        /* a 为行首键号 */
    b=0;                                        /* b 为列号 */
    IO_PDR5.byte=0xFE;
    i=0xFE;
    for(;b<16;)
    {
        if(!IO_PDR5.bit.P54)        a=0;                        /* 第 0 行 */
        else
        {   if(!IO_PDR5.bit.P55)        a=4;                    /* 第 1 行 */
            else {  if(!IO_PDR5.bit.P56)        a=8;            /* 第 2 行 */
                    else {  if(!IO_PDR5.bit.P57)    a=12;       /* 第 3 行 */
                            else a=16;                          /* 本列无闭合键 */
                         }
                 }
        }
        if(a==16)
            {   /* 列号加 1，对下一列进行扫描 */
```

```
                i=i*2+1;
                IO_PDR5.byte=i;
                b++;
                }
            else      break;
        }
        x=a+b;                                  /* 计算出闭合键键号 */
        if(a==16)x= 'U' ;                       /* 判断没有键闭合，出错，返回字符'U'*/
        return (x);
}

/* 主程序 */
void main()
{
        kp=0;
        ks=0;                                   /* kp: 键处理标志；ks: 键抖动标志 */
        name=0;
        __DI();                                 /* 禁止中断 */
        IO_ADER.byte=0x00;                      /* P5 口设为通用口 */
        IO_DDR5.byte=0x0F;                      /* 设 P5 口低 4 位(键盘列线接口) 为输出
                                                   P5 口高 4 位(键盘行线接口)为输入 */
        inittimer();
        __set_il(7);                            /* 设置 ILM 为 7 */
        __EI();                                 /* 使能中断 */
        while(1);
}
```

当 10ms 定时中断产生，进入中断处理程序时，首先判断是否有键闭合，没有则把两个标志位 kp，ks 置为 0。如果有键闭合，但是 ks=0，则表示没有去除键抖动，则置 ks 为 1 不做处理，等下一次在进入中断处理程序时，已经延时了 10ms，ks 为 1，表示去除了键抖动。此时仍然有键闭合，而 kp 为 0，则表示没有处理过，于是调用判断闭合键键号的程序 keyname（）以判断键号，并将 kp 置 1，标志已经处理了。变量 name 的值就是闭合键的键号。

```
/* 中断处理 */
__interrupt
void timer_scan(void)
{
        unsigned char close;
        close=kclose();
        if (close==0)                                   /* 没有键闭合，置 ks，kp 为 0 */
        {     ks=0;     kp=0;     }
        else                                            /* 有键闭合 */
        {if(ks==1)
            {     if(kp==0)                             /* 已去除键抖动，而且没有处理过 */
```

```
            {    name=keyname();              /* 得出键号 */
                 kp=1;                        /* 置处理标志 kp 为 1 */
            }
            else ;
        }
    else     ks=1;                            /* 未去除键抖动，等待下次中断处理*/
    };
    IO_TMCSR0.bit.UF = 0;                     /* 清除向下溢出标志 */
}
```

1.3.2.3 中断扫描方式

为了提高 CPU 的效率，可以采用中断方式，当键盘上有键闭合时产生中断请求，CPU 响应中断，执行中断服务程序，判别键盘上闭合键的键号，并作相应的处理。这就是中断扫描方式的处理方法。

中断扫描方式的硬件接线和定时扫描方式不同，需要将键盘的行线通过一个与门接到 CPU 的外部中断引脚 INT。初态时，列线都输出低电平，当键盘上没有键闭合时，INT 引脚为高电平；一旦键盘上有键闭合，INT 引脚变低电平，向 CPU 发出中断请求，若 CPU 开放外部中断，则响应中断请求，执行中断服务程序扫描键盘。

中断服务程序里，除了要识别闭合键的键号外，还要排除键抖动引起的误操作，以及避免对同一个键的一次闭合作多重处理的错误。这里的处理方法和定时扫描方式的处理方式类似，就不再重复了。

第二章　七段发光显示器应用及 C 编程

单片机系统中，通常用 LED 数码显示器来显示各种数字或符号。由于它具有显示清晰、亮度高、使用电压低、寿命长的特点，因此使用非常广泛。

第一节　七段发光显示器硬件知识

2.1.1 显示器的结构

七段 LED 显示器由 8 个发光二极管组成。其中 7 个长条形的发光管排列成“日”字形，另一个贺点形的发光管在显示器的右下角作为显示小数点用。当某一个发光二极管导通时，相应的一个点或一个笔划被点亮，控制不同组合的二极管导通，就能显示出各种数字字符。

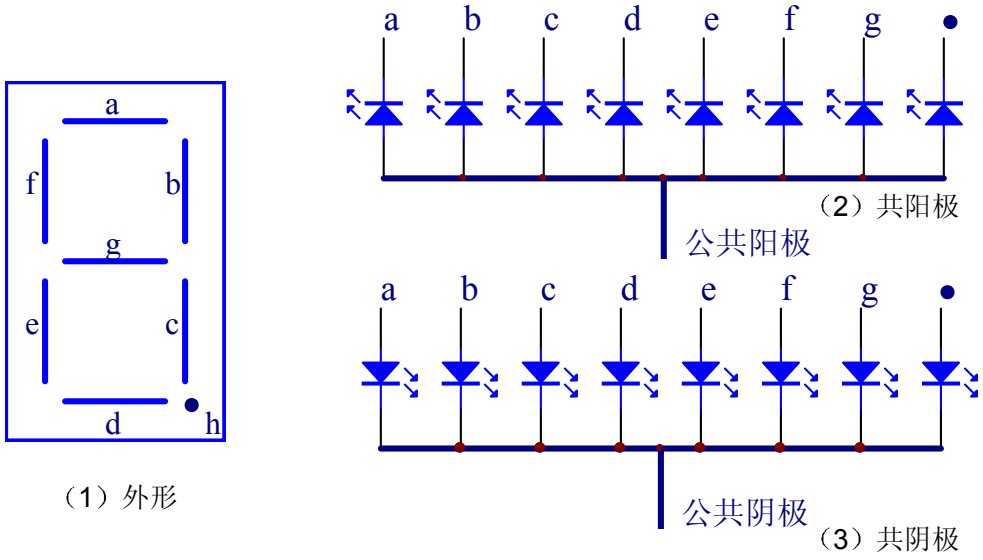


图 **2.1.** 七段发光显示器的结构

LED 显示器有两种不同的形式：一种是 8 个发光二极管的阳极都连在一起的，称之为共阳极 LED 显示器；另一种是 8 个发光二极管的阴极都连在一起的，称之为共阴极 LED 显示器。如图 2.1 所示。共阴和共阳结构的 LED 显示器各笔划段名和安排位置是相同的。

2.1.2 七段显示器的段选码

七段显示器与单片机接口非常容易，只要将一个 8 位并行输出口与显示器的发光二极管引脚相连即可。8 位并行输出口输出不同的字节数据即可获得不同的数字或字符显示，通常将控制发光二极管的 8 位字节数据称为段选码。表 1 给出了七段二极管的段选码，其中七段是共阴极情况下的对应值，同时，可以看到共阳极与共阴极的段选码互为补数。

在单片机应用系统中，使用 N 位 LED 显示器构成 LED 显示块。N 位 LED 显示器有 N 根位选线和 8N 根段选线。段选线控制字符选择；位选线控制显示位的亮、暗。

显示字符	共阴极段选码	共阳极段选码	D7	D6	D5	D4	D3	D2	D1	D0
			h	g	f	e	d	c	b	a
0	3FH	C0H	0	0	1	1	1	1	1	1
1	06H	F9H	0	0	0	0	0	1	1	0
2	5BH	A4H	0	1	0	1	1	0	1	1
3	4FH	B0H	0	1	0	0	1	1	1	1
4	66H	99H	0	1	1	0	0	1	1	0
5	6DH	92H	0	1	1	0	1	1	0	1
6	7DH	82H	0	1	1	1	1	1	0	1
7	07H	F8H	0	0	0	0	0	1	1	1
8	7FH	80H	0	1	1	1	1	1	1	1
9	6FH	90H	0	1	1	0	1	1	1	1
A	77H	88H	0	1	1	1	0	1	1	1
B	7CH	83H	0	1	1	1	1	1	0	0
C	39H	C6H	0	0	1	1	1	0	0	1
D	5EH	A1H	0	1	0	1	1	1	1	0
E	79H	86H	0	1	1	1	1	0	0	1
F	71H	84H	0	1	1	1	0	0	0	1
P	73H	82H	0	1	1	1	0	0	1	1
U	3EH	C1H	0	0	1	1	1	1	1	0
Γ	31H	CEH	0	0	1	1	0	0	0	1
Y	6EH	91H	0	1	1	0	1	1	1	0
8.	FFH	00H	1	1	1	1	1	1	1	1
.	80H	7FH	1	0	0	0	0	0	0	0
“灭”	00H	FFH	0	0	0	0	0	0	0	0

表 **2.1.** 七段显示器的段选码

2.1.3 显示器的工作方式

显示器显示常用两种方法：静态显示和动态扫描显示。

2.1.3.1 静态显示接口

所谓静态显示，就是每一个显示器都要占用单独的具有锁存功能的 I/O 接口用于笔划

段字形代码。这样单片机只要把要显示的字形代码发送到接口电路，就不用管它了，直到要显示新的数据时，再发送新的字形码，因此，使用这种方法节省了 CPU 的运行时间，提高 CPU 工作效率，但是位数较多时显示口也随之增加。为了节省 I/O 口线，常采用另外一种显示方式——动态显示方式。

2.1.3.2 动态显示接口

动态扫描显示接口是单片机中应用最为广泛的一种显示方式之一。其接口电路是把所有显示器的 8 个笔划段 a-h 同名端连在一起，而每一个显示器的公共极 COM 是各自独立地受 I/O 线控制。CPU 向字段输出口送出字形码时，所有显示器接收到相同的字形码，但究竟是那个显示器亮，则取决于 COM 端，而这一端是由 I/O 控制的，所以我们就可以自行决定何时显示哪一位了。而所谓动态扫描就是指我们采用分时的方法，轮流控制各个显示器的 COM 端，使各个显示器轮流点亮。

在轮流点亮扫描过程中，每位显示器的点亮时间是极为短暂的（约 1ms），但由于人的视觉暂留现象及发光二极管的余辉效应，尽管实际上各位显示器并非同时点亮，但只要扫描的速度足够快，给人的印象就是一组稳定的显示数据，不会有闪烁感。

第二节　C 语言编程实现对发光显示器的控制

从 LED 显示器的显示原理可知，为了显示字母与数字，必须最终转换成相应的段选码。这种转换可以由软件进行译码或通过硬件译码器。

2.2.1 软件译码显示器接口

在单片机应用系统中，由于单片机本身有较强的逻辑控制能力，采用软件译码并不复杂。而且软件译码其译码逻辑可随意编程设定，不受硬件译码逻辑限制。因此，在单片机应用系统中使用的最广的还是软件译码的显示器接口。

2.2.1.1 软件译码的动态显示接口

图 2.2 是软件译码的动态显示接口图。单片机的 P3 口输出位选码，经 8 位反相集成驱动芯片 MCT1413P 驱动后接各个显示器公共极，位选码占用输出口线数决定与显示器位数，共四位（LED1~4）。单片机 P2 口输出段选码，经 8 位同相集成驱动芯片 74HC244 驱动后接显示器的段选线（SEG1~8）。

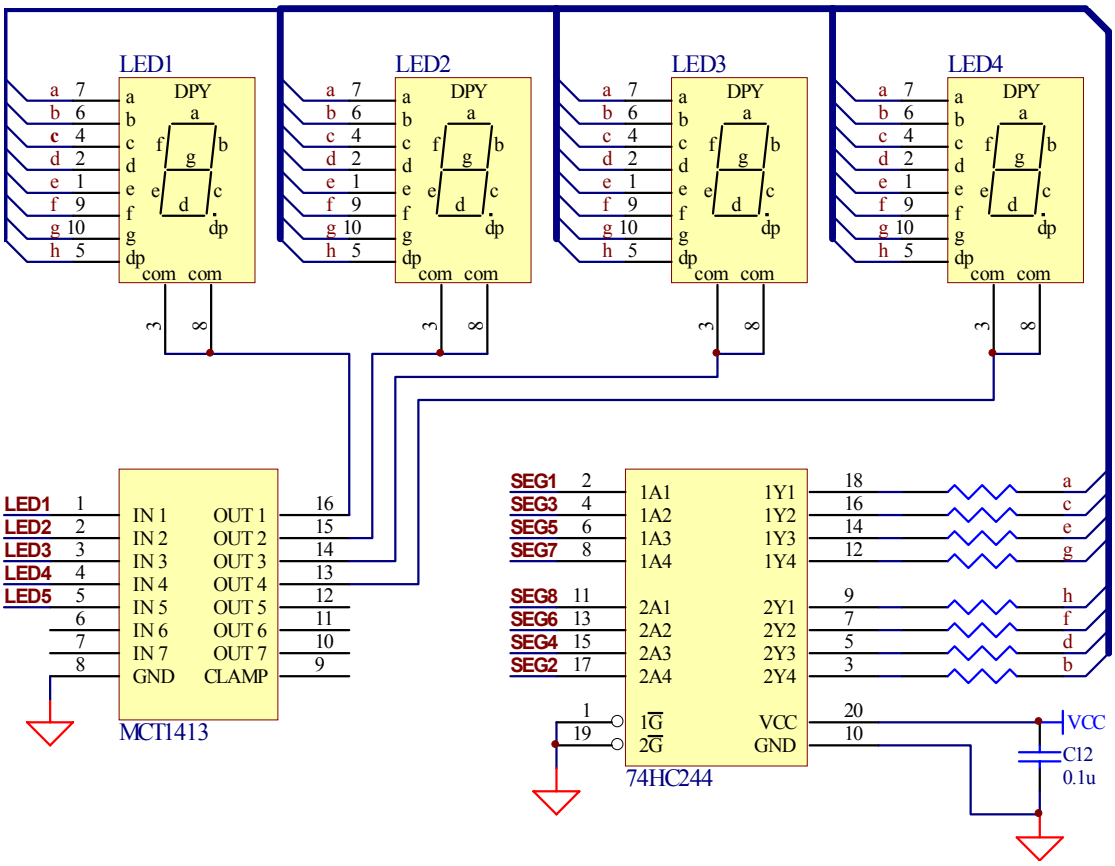


图 **2.2.** 软件译码接口电路图

2.2.1.2 软件译码的动态显示程序设计

图 2 的动态显示接口中，LED1~LED5 分别对应接到 P3.0~P3.4，输出位选码；SEG1~SEG8 分别对应接到 P2.0~P2.7，输出段选码。下面给出 C 语言例程。

动态扫描显示中，需不断循环送出相应的段选码、位选码，如果主程序里只做这样的循环，显然是浪费资源的。本例中采用定时器中断方式扫描显示，使用 16 位重装入定时器 1 产生 1ms 定时中断。在中断处理程序只点亮一位显示器，这位显示器就会亮到下一次进入中断处理程序点亮下一位显示器时熄灭。这样间隔 1ms 的轮流点亮，能使视觉上获得稳定的显示，同时并不占有太多资源。

首先将 P2 口和 P3.0~P3.4 口的段口寄存器和方向寄存器进行宏定义，如用 SEGDATA 表示输出的段选码，SEGDATA_DIRE 是输出段选码口的方向设置寄存器等。同时要定义重装入定时器 1 中断处理程序。

```
#define     SEGDATA       IO_PDR2.byte     /* 7 段数据 */
#define     LED1          IO_PDR3.bit.P30  /* LED1 位选码 */
#define     LED2          IO_PDR3.bit.P31  /* LED2 位选码 */
#define     LED3          IO_PDR3.bit.P32  /* LED3 位选码 */
#define     LED4          IO_PDR3.bit.P33  /* LED4 位选码 */
#define     LED5          IO_PDR3.bit.P34  /* LED5 位选码 */
#define     SEGDATA_DIRE  IO_DDR2.byte     /* 段选码输出口方向设置 */
```

```
#define        LED_DIRE          IO_DDR3.byte          /* 位选码输出口方向设置 */

/* 定义中断处理程序 */
__interrupt void    led_timer(void) ;
#pragma intvect   led_timer      32
```

定义全局变量时，首先定义一个常数数组 LEDHEX[ ]，用于存放共阴极七段 LED 的段选码，可以看到 LEDHEX[0]对应的显示字形就是“0”，0~F 都是对应的，后两个段选码对应的则分别是显示小数点和熄灭无显示。

变量 a，b，c，d 分别对应于数的千、百、十、个位，也就是 4 位显示器所需要显示的段数据。a、b、c、d 的赋值是在 disp_num ( )里完成的，而初始时 a=b=c=d=17，即对应于 LEDHEX[17]，没有显示值。led_counter 变量则是用来循环计数以确定点亮第几位显示器的。

```
/* 定义全局变量 */
const LEDHEX[] ={ 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D, 0x7D, 0x27, 0x7F, 0x6F,
                   /* 0      1     2       3     4       5       6     7       8      9   */
                   0x77, 0x7C,0x39, 0x5E, 0x79, 0x71,0x80,0x00   };
                  /* A       B      C      D      E      F     “.”  “熄灭”  */
unsigned char     a,b,c,d;
unsigned char     led_counter;
/* 函数列表 */
void       inittimer1();
void       initled244 ();
void       disp_num (int data);
```

初始化重装入定时器 1 函数 inittimer1() 里对中断优先级别进行了设置，同时设定重装入值，使定时 1ms，并启用重装入模式。详情参见富士通 16 位硬件手册。

```
void inittimer1()
{
    IO_ICR10.byte = 1;     /* 设置中断优先级别 */
    IO_TMCSR1.bit.CNTE = 0;/* 计数停止 */
    IO_TMR1 = 0x01ff;      /* 设置重装入值，使定时 1ms */
    IO_TMCSR1.word = 0x81B;      /* 预制 2us at 16 MHz */
                                 /* 禁止外部触发器和外部输出 */
                                 /* 重装入模式，使能中断 */
}
```

initled244 () 初始化显示器接口的 IO 口方向和初值。并设 led_counter 初值为 1，即从最高位——千位开始点亮；a、b、c、d 初值都为 17，即没有显示数值。

```
void initled244()
{
```

```
    LED1=0;                         /* 初始化 led：关断所有的 led */
    LED2=0;
    LED3=0;
    LED4=0;
    LED5=0;
    led_counter=1;
    a=17;b=17;c=17;d=17;            /* 各位缺省值为没有显示值 */
    SEGDATA = 0x00;                 /* 每个 led 都没有显示 */
    SEGDATA_DIRE = 0xFF;            /* 设置 IO 口方向寄存器为输出 */
    LED_DIRE = 0xFF;                /* 设置 IO 口方向寄存器为输出 */
}
```

disp_num () 是一个将数值以十进制形式显示出来的转换程序。参数 data 是要显示的数值，将其转换为十进制形式，千位、百位、十位和个位的值分别赋给 a、b、c、d。对于位数少的数，其高位赋值为 17（即显示器熄灭无显示）。例如，显示数 125，调用 disp_num() 后得到 a=17，b=1，c=2，d=5。

```
void disp_num(int data)
{   int i;
    i=data;
    if(i<10000)
    {
        a=i/1000;     i=i%1000;
        b=i/100;  i=i%100;
        c=i/10;
        d=i%10;
    }
    if(c==0&&b==0&&a==0)        c=17;
    if(b==0&&a==0)              b=17;
    if(a==0)                    a=17;
}
```

主程序里先调用 inittimer1 () 、initled244 () 进行一系列初始化后，调用 disp_num () 函数，要求显示数 1234。

```
/* 主程序 */
void main()
{
    int i;
    __DI();                     /* 关闭所有中断 */
    inittimer1();
    initled244();
    __set_il(7);            /* set ILM to 7 */
                            /* allow all interrupt levels       */
    __EI();                 /* 允许中断 */
```

```
    disp_num(1234);
    while(1);
}
```

中断处理程序 led_timer () 是定时器产生 1ms 中断时所做的处理程序。进入中断处理程序，要根据 led_counter 的值决定点亮哪一位显示器，并输出相应的段数据，当然不能忘记，首先要将上一位显示器关断。例如，led_counter 为 1，关断个位显示器 LED4，点亮千位显示器 LED1，输出的段选码为 LEDHEX[a]。Led_counter 为 5 时，需要重新赋值为 1，以循环计数。

```
/* 中断处理程序 */
__interrupt
void led_timer(void)
{
        switch(led_counter)
        {
        case 1:     /* 千位 */
                {   LED4=0;
                    SEGDATA=LEDHEX[a];
                    LED1=1;
                    break;
                }
        case 2:     /* 百位 */
                {   LED1=0;
                    SEGDATA=LEDHEX[b];
                    LED2=1;
                    break;
                }
        case 3:     /* 十位 */
                {   LED2=0;
                    SEGDATA=LEDHEX[c];
                    LED3=1;
                    break;
                }
        case 4:     /* 个位 */
                {   LED3=0;
                    SEGDATA=LEDHEX[d];
                    LED4=1;
                    break;
                }
        default :   ;
        }
    led_counter++;
    if(led_counter==5)    led_counter=1;        /* 循环计数 */
```

```
    IO_TMCSR1.bit.UF = 0;                    /* 重置下溢中断请求标志 */
}
```

2.2.2 硬件译码显示器接口

单片机应用系统中，通常要求 LED 显示器能显示十六进制及十进制带小数点的数。因此，在选择译码器时，要能够完成 BCD 码至十六进制的锁存、译码，并具有驱动功能，否则就不如采用软件译码接口。

MC14489 是 MOTOROLA 公司生产的 5 位/7 段 LED 译码/驱动芯片，它可以直接驱动 5 位共阴极 LED 数码显示器或 25 个 LED 指示灯，也可实现共阴极 LED 数码显示器和 LED 指示灯的组合驱动。使用 MC14489 只需几个 IO 口线，大大节约了 IO 口资源。

1．MC14489 的引脚

MC14489 的引脚排列如图 2.3 所示：

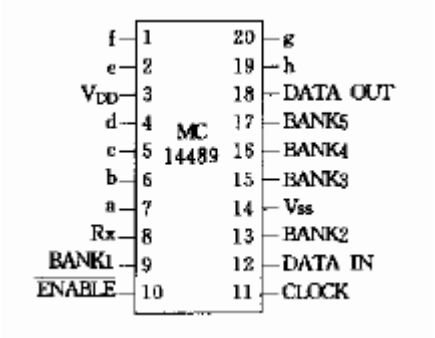


图 **2.3. MC14489** 的引脚排列

a~h：　各段输出驱动脚，接到 LED 的各阳极端。

Rx：　外接限流电阻，取值范围为 700Ω~无穷大。使用一个外接电阻 Rx 即可控制每一段的输出电流，而不需要在每一段上都加段限流电阻。

BANK1~BANK5：位扫描输出，与 LED 数码显示器的共阴极相连。

ENABLE：　输入使能。它为低电平时，使能 MC14489 的串行接口；为高电平时，禁止 MC14489 的串行接口。在 ENABLE 的上升沿时将数据寄存器的内容打入系统设置寄存器或显示寄存器中（取决于输入的字节数）。

DATA OUT：　串行数据输出，在移位时钟 CLOCK 的下降沿移出数据，不做级联使用时悬空。

DATA IN：　串行数据输入，在 ENABLE 为低电平期间，串行数据由 CLOCK 的上跳沿移入内部移位寄存器，移入时最高位（MSB）在前。

CLOCK：　时钟输入脚，是串行数据输入时所需的移位脉冲。

Vss，Vdd：　Vss 为电源地线；Vdd 为电源正端。

2．MC14489 的寄存器设置

MC14489 系统设置寄存器共 8 位，由 DATA IN 移入设置命令字的时序图如图 2.4 所示。

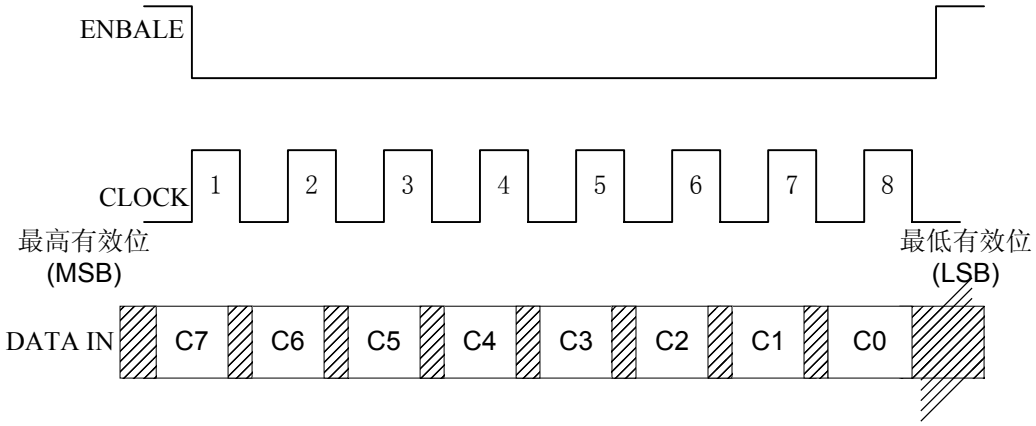


图 **2.4. MC14489** 系统设置寄存器命令字移入时序图

系统设置寄存器各位的作用如表 2.2 所示。

bit	各位作用
C0	0：低功耗模式（显示器全灭），上电复位处于该模式； 1：正常模式。
C1	控制 BANK1 译码方式：　0——16 进制译码； 1——取决于 C6。
C2	控制 BANK2 译码方式：　0——16 进制译码； 1——取决于 C6。
C3	控制 BANK3 译码方式：　0——16 进制译码； 1——取决于 C6。
C4	控制 BANK4 译码方式：　0——16 进制译码； 1——取决于 C7。
C5	控制 BANK5 译码方式：　0——16 进制译码； 1——取决于 C7。
C6	0：不译码； 1：特殊译码（见 C1、C2、C3 说明）
C7	0：不译码； 1：特殊译码（见 C4、C5 说明）

表 **2.2. MC14489** 系统设置寄存器各位作用

MC14489 显示寄存器共 24 位，各位的作用如图 2.5 所示。

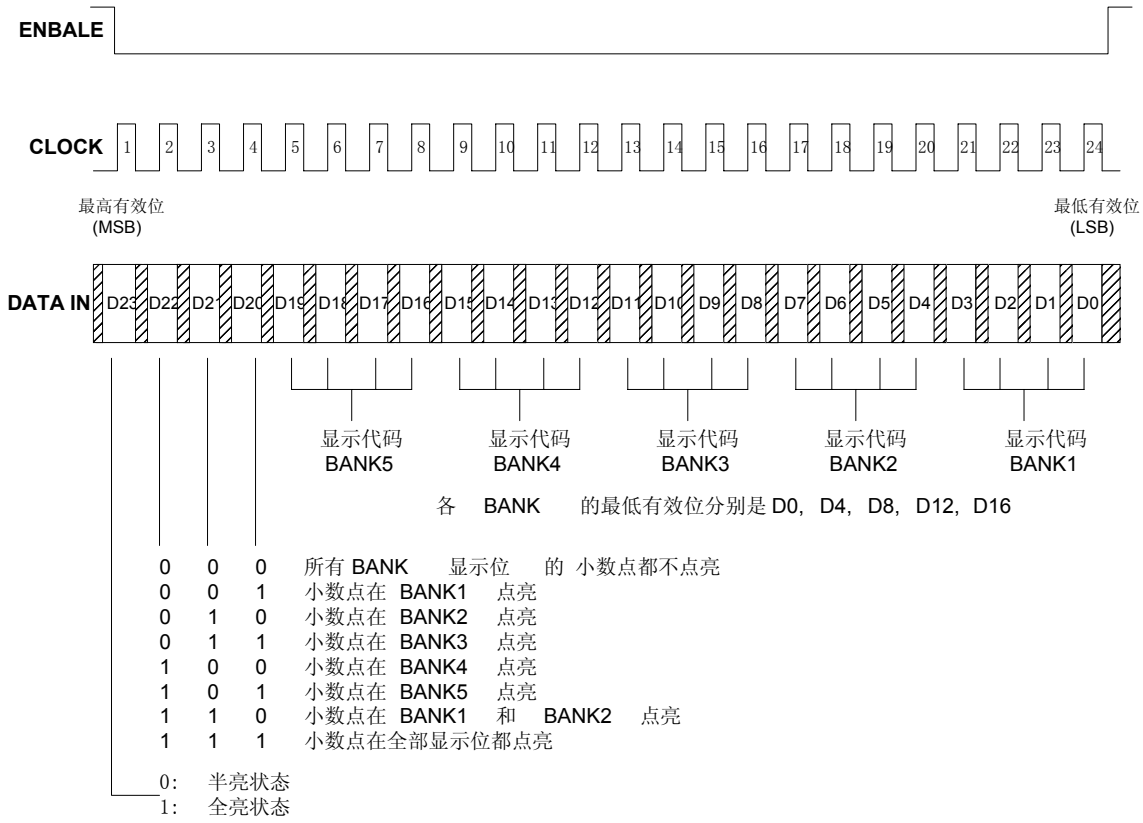


图 **2.5. MC14489** 内部显示寄存器中各位的作用

表 2.3 是三种不同译码方式的段译码表。

BANK 值					七段显示字符		指示灯			
16 进制数	2 进制数				**16** 进制译码	特殊译码	不译码			
	D3	D2	D1	D0			d	c	b	a
0H	0	0	0	0	0					
1H	0	0	0	1	1	C				亮
2H	0	0	1	0	2	H			亮	
3H	0	0	1	1	3	h			亮	亮
4H	0	1	0	0	4	J		亮		
5H	0	1	0	1	5	L		亮		亮
6H	0	1	1	0	6	n		亮	亮	
7H	0	1	1	1	7	o		亮	亮	亮
8H	1	0	0	0	8	P	亮			
9H	1	0	0	1	9	r	亮			亮
AH	1	0	1	0	A	U	亮		亮	
BH	1	0	1	1	B	u	亮		亮	亮
CH	1	1	0	0	C	y	亮	亮		
DH	1	1	0	1	D	-	亮	亮		亮
EH	1	1	1	0	E	=	亮	亮	亮	
FH	1	1	1	1	F	口	亮	亮	亮	亮

表 **2.3.** 三种不同译码方式的段译码表

图 2.6 是 MC14489 与 MB90560 的接口电路，其中 MC14489 控制 4 位 LED 七段显示器和 8 位 LED 指示灯。P3 口的低三位连接 MC14489 的控制线：P30 向 MC14489 串行输入要显示的数字或字符；P31 提供串行数据输入时所需的脉冲；P30 使能 MC14489 的串行接口。然后按照图 2.4 和图 2.5 所示的时序图就可以用软件实现串行数据的输入。

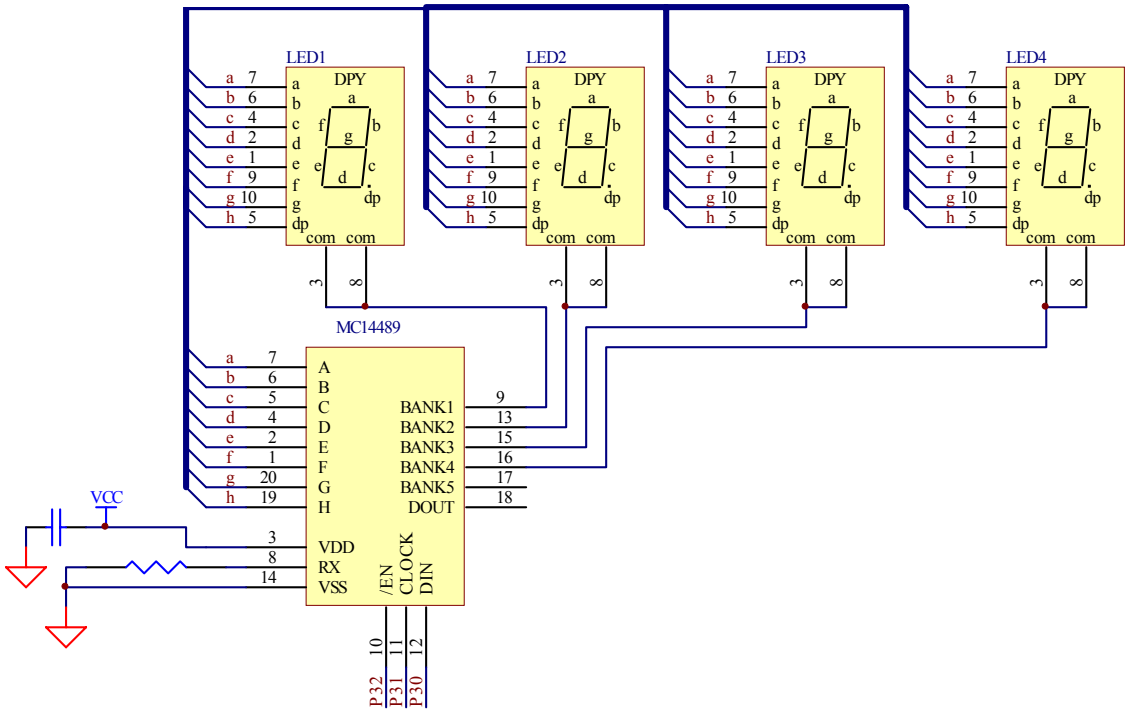


图 **2.6. MC14489** 与 **MB90560** 的接口电路

下面给出对于图 5 接口电路的 C 语言驱动程序：

首先将与 MC14489 连接的 IO 口端口寄存器和方向寄存器进行宏定义。DSPEN、DSPCLK、DSPDIN 既分别表示 MC14489 的 EN，CLK，DIN 引脚的状态。

```
#define  DSPEN      IO_PDR3.bit.P32 /*  定义 P32 端口寄存器为 DSPEN */
#define  DSPCLK     IO_PDR3.bit.P31 /*  定义 P31 端口寄存器为 DSPCLK */
#define  DSPDIN     IO_PDR3.bit.P30 /*  定义 P30 端口寄存器为 DSPDIN */
#define  EN_DIRE    IO_DDR3.bit.D32 /*  分别定义对应三根口线的方向寄存器  */
#define  CLK_DIRE   IO_DDR3.bit.D31
#define  DIN_DIRE   IO_DDR3.bit.D30
```

定义一个全局变量用来循环移位。

```
union    {
         unsigned char     byte;
         struct    {
                   unsigned char          :7;
                   unsigned char     bit7:1;
                   }bit;
         }temp;
```

dispcmd () 是写入单字节命令函数，用于写入内部设置寄存器。参数 data 是要写入 MC14489 内部设置寄存器的命令字，通过循环移位将命令写入。

```
void dispcmd(unsigned char data)
{
    char i;
    temp.byte=data;
    DSPEN=0;                    /* 使能 MC14489 */
    for(i=8;i>0;i--)
    {                           /* 移位写入单字节命令 */
    DSPDIN=temp.bit.bit7;
    temp.byte<<=1;
    DSPCLK=0;
    DSPCLK=1;
    }
    DSPEN=1;                    /* 禁止 MC14489 */
}
```

disdata () 是写入多字节命令函数，用于写入显示寄存器。多字节实际上是 24 位数据，前 8 位是显示寄存器命令字和 LED5 的显示内容，中间 8 位是 LED4 的显示内容和 LED3 的内容，最后 8 位是 LED2 的显示内容和 LED1 的显示内容。参数 cmd 是前 8 位，data2 是中间 8 位，data1 是最后 8 位，分三次写入。

```
void dispdata(unsigned char cmd,unsigned char data2,unsigned char data1)
{
    char i,j;
    j=0;
    DSPEN=0;                                /* 使能 MC14489 */
    while(j<24)
    {
    if(j<8)         temp.byte=cmd;      /* 显示寄存器命令字和 LED5 的显示内容 */
    else if(j<16)   temp.byte=data2;    /* LED4 的显示内容和 LED3 的内容 */
    else            temp.byte=data1;    /* LED2 的显示内容和 LED1 的显示内容 */
    for(i=8;i>0;i--)
        {                               /* 移位写入一个字节 */
        DSPDIN=temp.bit.bit7;
        temp.byte<<=1;
        DSPCLK=0;
        DSPCLK=1;
        }
    j=j+8;
    }
    DSPEN=1;                            /* 禁止 MC14489 */
}
```

主程序中先初始化 DSPEN，DSPCLK，DSPDIN。然后将写入命令字，采用 16 进制译码，所有 LED 都电亮，处于正常模式。LED1~LED5 分别显示 12345，半亮度显示。

```
void main()
{    /* 初始化 ：置 ENABLE 高电平，CLOCK 低电平，DATA IN 高电平 */
     DSPEN=1;
     DSPCLK=0;
     DSPDIN=1;
     /* 三根口线的端口方向寄存器都置为输出 */
     EN_DIRE=1;
     CLK_DIRE=1;
     DIN_DIRE=1;
     dispcmd(0x01);              /* 写 MC14489 内部设置寄存器，点亮所有 LED */
     dispdata(0x08,0x43,0x21); /*  LED1~LED4 分别显示 1234，半亮度显示  */
}
```

第三章　串行口通信的 C 编程

MB90V560 系列单片机上有 UART0 和 UART1 两个通道用于串行通信。UART 是通用串行数据通信接口，它用来和外部设备进行同步或异步（起停同步）通信。UART 具有标准的双向通信功能（标准方式）和仅适用于主系统的主一从通信功能（多处理器方式）。

第一节　与串行口有关的寄存器

3.1.1 串行控制寄存器 SCR0/1

该寄存器指定校验位，选择停止位和数据长度，选择方式 **1** 下的数据帧格式，清除接收错误标志，指定是否允许发送和接收。

Bit 15	bit 14	bit 13	bit 12	bit 11	bit 10	bit 9	bit 8	bit7…….bit 0
PEN	P	SBL	CL	A/D	REC	RXE	TXE	(SMR)
R/W	R/W	R/W	R/W	R/W	W	R/W	R/W	

寄存器 SCR0 和 SCR1 分别位于 000021H 和 000025H 地址单元，初始值为 00000100B。

下表为寄存器中各位的功能 ：

位　名		功　能
bit15	PEN: 校验使能位	该位选择在串行数据输入-输出方式下，发送过程中是否添加校验位，接收过程中是否检测校验位。 **<**注意**>** 在方式 1 和方式 2 下，不能使用校验，该位固定为“0”
bit14	P: 校验选择位	当允许校验时(PEN=1)，该位选择偶校验或奇校验
Bit13	SBL: 停止位长度选择位	该位选择停止位的长度或异步传输方式下发送数据的帧结束标志 **<**注意**>**　在接收过程中，只能检测到第一个停止位
bit12	CL: 数据长度选择位	该位指定发送和接收数据的长度 **<**注意**>** 在方式 0 下，可以选择 7 位(CL=0)；在方式 1 和方式 2 下，必须选择 8 位(CL=1)。

bit11	A/D: 地址/数据选择位	• 在方式 1 下，指定发送和接收帧的数据格式 • 该位为“0”，选择数据，该位为“1”,选择地址。
bit10	REC: 接收错误标志清除位	该位写入“0”，清除状态寄存器(SSR)的 FRE、ORE 和 PE 标志，写入“1”，没有任何影响。 **<注意>** 如果 UART 正在使用并且接收中断使能，仅当 FRE, ORE, 或 PE 标志为“1”时，清除 REC 位。
bit9	RXE: 接收使能位	• 该位控制 UART 接收 • 当该位为“0”，接收不允许，当该位为“1”，接收允许 **<注意>** 在接收过程中，如果禁止接收操作，完成帧的接收并存储数据于接收数据缓冲器(SIDR1)后，停止接收操作。
bit8	TXE: 发送使能位	• 该位控制 UART 发送 • 当该位为 0 时，发送禁止； 当该位为 1 时，发送允许 **<注意>** 在发送过程中，如果禁止发送操作，等待发送数据缓冲器(SODR1)中没有数据后，停止发送操作。

表 **3.1.** 串行控制寄存器 **SCR0/1** 设置

3.1.2 串行方式控制寄存器 **SMR0/1**

该寄存器选择操作方式和时钟输入源，设置专用的波特率发生器，指定是否允许串行数据和时钟输出到相应的引脚。

bit 15……...bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
SCR	MD1	MD0	CS2	CS1	CS0	BCH	SCKE	SOE
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

寄存器 SCR0 和 SCR1 分别位于 000020H 和 000024H 地址单元，初始值为 0000000B。

下表为寄存器各位的功能 ：

位　名		功　　能
Bit7 Bit6	MD1 和 MD0: 操作方式选择位	这些位选择操作方式
Bit5 Bit4 Bit3	CS2 到 CS0: 时钟选择位	• 这几位选择波特率时钟源。输入时钟可选择外部时钟(SCK0/1 引脚)、16 位可重装入定时器 0 和专用的波特率发生器。 • 当选择专用的波特率发生器时，波特率也同时被确定，有 8 种波特率可供选择：异步传输方式 5 种，同步传输方式 3 种。

Bit2	BCH:	
Bit1	SCKE: 串行时钟输出使能位	• 该位控制串行时钟输入-输出端口 • 当该位为“0”时， P40/SCK0 和 P62/SCK1 脚用作通用的输入-输出口（P40 和 P62）或串行时钟输入脚，当该位为“1”时，该脚用作串行时钟输出脚。 〈注意〉 • 当 P40/SCK0 和 P62/SCK1 用作串行时钟输入脚时(SCKE=0)，P40 和 P62 要设置为输入口，同时使用时钟选择位来选择外部时钟(SMR0/1: CS2 to CS0 = 111B)， • 当 P40/SCK0 和 P62/SCK1 用作串行时钟输出脚时，选择不同于外部时钟的时钟设置(不同于 SMR0/1: CS2 to CS0 = 111B)。 〈说明〉 当 SCK0/1 脚被指定为串行时钟输出时(SCKE=1)，它用作串行时钟输出脚，而不必考虑通用输入-输出口的状态。
Bit0	SOE: 串行数据输出使能位	• 该位使能或禁止串行数据输出 • 当该位为“0”时，P37/SOT0 和 P61/SOT11 脚用作通用的输入-输出口（P37 和 P61），当该位为“1”时，该脚用作串行数据输出脚(SOT0/1)。 〈说明〉 当串行数据使能输出时(SOE=1)，P37/SOT0 和 P61/SOT1 脚用作 SOT0/1，而不必考虑通用输入-输出口(P37 和 P61)的状态。

表 **3.2.** 串行方式控制寄存器 **SMR0/1** 设置

3.1.3 串行状态寄存器 **SSR0/1**

该寄存器检查发送、接收状态和错误状态，允许或禁止发送和接收中断。

Bit 15	bit 14	bit 13	bit 12	bit 11	bit 10	bit 9	bit 8	bit7…….bit 0
PE	ORE	PRE	PDRF	TDRE	BDS	RIE	TIE	(SIDR/SODR)
R	R	R	R	R	R/W	R/W	R/W	

寄存器 SSR0 和 SSR1 分别位于 0000023H 和 0000027H 地址单元，初始值为 0001000B。

下表为寄存器各位功能 ：

NO.	位　名	功　能
bit15	PE: 校验错标志位	• 在接收过程中，有校验错误发生时，该位被置“1”，串行控制寄存器(SCR0/1)REC位写入“0”时，该位被清“0”。

		• 当该位和RIE位都为“1”时，产生一个接收中断请求 • 该标志置位时，串行输入数据寄存器(SIDR0/1)中的数据无效
bit14	ORE：超越错标志位	• 在接收过程中，有超越错误发生时，该位被置“1”，串行控制寄存器(SCR0/1)REC位写入“0”时，该位被清“0”。 • 当该位和RIE位都为“1”时，产生一个接收中断请求 • 该标志置位时，串行输入数据寄存器(SIDR0/1)中的数据无效
bit13	FRE：帧格式错标志位	• 在接收过程中，有帧格式错误发生时，该位被置“1”，串行控制寄存器(SCR0/1)REC位写入“0”时，该位被清“0”。 • 当该位和RIE位都为“1”时，产生一个接收中断请求 • 该标志置位时，串行输入数据寄存器(SIDR0/1)中的数据无效
bit12	RDRF：接收数据 缓冲器满标志位	• 该标志指出输入数据寄存器(SIDR0/1)的状态 • 接收数据装入寄存器SIDR0/1，该位置“1”，读串行输入数据寄存器SIDR0/1，该位被清“0”。 • 当该位和RIE位都为“1”时，产生一个接收中断请求
bit11	TDRE： 发送数据 缓冲器空标志位	• 该标志指出输出数据寄存器(SIDR0/1)的状态 • 发送数据写入寄存器SIDR0/1，该位被清“0”，数据装入发送移位寄存器，启动发送时，该位置“1”。 • 当该位和RIE位都为“1”时，产生一个发送中断请求 〈注意〉 该位的初始值为“1”（寄存器SODR0/1空）
bit10	BDS：传输方向选择位	• 该位选择发送串行数据从最低位LSB开始（BDS=0），还是从最高位MSB开始（BDS=1） 〈注意〉 读或写串行数据寄存器时，串行数据的高位和低位是互相交换的。数据写入SDR寄存器后，如果该位设置为其它值，数据无效
bit9	RIE： 接收中断请求使能位	• 该为允许或禁止向CPU发出接收中断请求 • 当该位和接收数据标志位(RDRF)位都为“1”，或该位和一个或多个错误标志位(PE, ORE,和FRE)为“1”时，产生一个接收中断请求。
Bit8	TIE： 发送中断请求使能位	• 该位允许或禁止向CPU发出发送中断请求 • 当该位和TDRE位都为“1”时，产生一个发送中断请求

表 **3.3.** 串行状态寄存器 **SSR0/1** 设置

3.1.4 串行输入数据寄存器 **SIDR0/1**

串行输入数据寄存器（**SIDR0/1**）是串行数据接收寄存器，串行输出数据寄存器（**SODR0/1**）是串行数据发送寄存器，寄存器 **SIDR0/1** 和 **SODR0/1** 使用同一个地址。

3.1.4.1 串行输入数据寄存器（□□□□̃ ˜ ı ）

SIDR0/1 是用来保存接收数据的寄存器，串行数据从 SIN0/1 引脚传送到移位寄存器后，存储在寄存器 SIDR0/1 中。数据长度为 7 位时，最高位(D7)包含无效的数据，接收数据存入该寄存器后，接收数据满标志位(SSR0/1: RDRF)置“1”，如果此时接收中断请求使能，产生一个接收中断请求。

串行输入数据寄存器的位结构如图 3.1 所示。

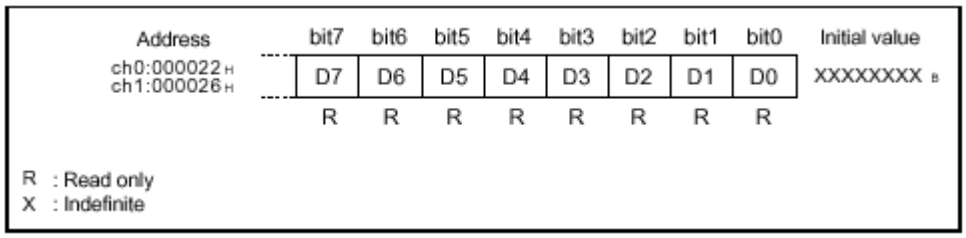


图 **3.1.**串行输入数据寄存器（**SIDR0/1**）

状态寄存器(SSR0/1)的RDRF位为“1”时，读寄存器SIDR01/1，RDRF位被自动清“0”。发生接收错误时(SSR0/1: PE, ORE, or FRE = 1)，SIDR0/1中的数据无效。

3.1.4.2 串行输出数据寄存器（□□□̃ ˜ ı ）

串行输出数据寄存器的位结构如图3.2 所示。

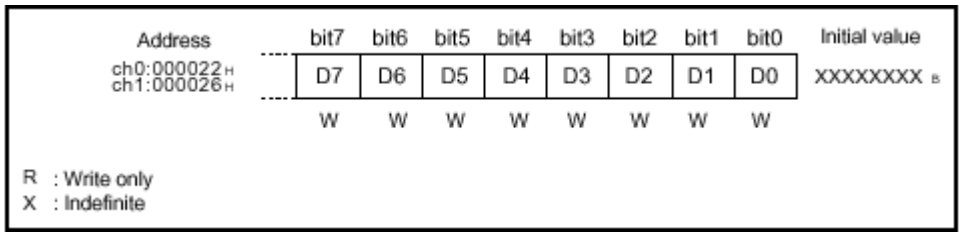


图 **3.2.** 串行输出数据寄存器（**SOR0/1**）

在发送使能状态下，发送数据写入该寄存器后，被传送到发送移位寄存器，然后转化为串行数据，发送到串行数据输出脚(SOT0/1脚)。数据长度为7位时，最高位(D7)包含无效的数据。

当发送数据写入该寄存器，发送数据缓冲器空标志位(SS0/1: TDRE)清“0”。数据全部传送到移位寄存器后，该位置“1”。TDRE位为“1”时，可以写入下一个发送数据。如果发送中断请求使能，产生一个发送中断请求。发生发送中断或TDRE位为“1”时，写入下一个发送数据。

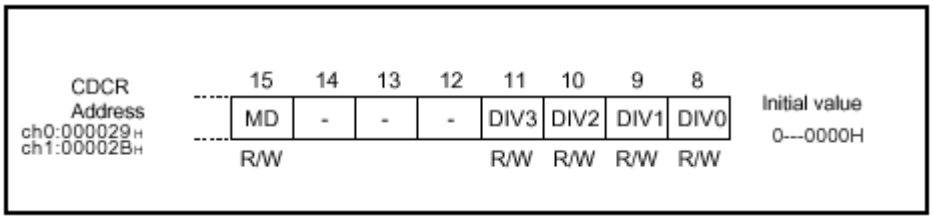
<注意>

SODR0/1 是只写寄存器，SIDR0/1 是只读寄存器。这两个寄存器具有相同的地址，读出值和写入的不同，因此对于执行读-修改-写(RMW)操作的指令，例如 INC/DEC，就不能使用。

3.1.5 通信预分频控制寄存器 **CDCR0/1**

3.1.5.1 通信预分频控制寄存器（□□□□˜ ı ）

UART的操作时钟可由机器时钟分频得到，对于不同的机器周期，UART可以获得确定的波特率。通信预分频器的输出可以用作扩充的I/O串行接口的操作时钟，寄存器CDCR的位结构如下图所示。



[Bit 15] MD(机器时钟分频模式选择)：通信预分频器操作使能位

0：停止通信预分频器

1：使用通信预分频器

[Bit 11,10,9,8]　DIV3~0(Divide3~0)：机器时钟分频系数，参照表3.4:

MD	DIV3	DIV2	DIV1	DIV0	Divided by
0	–	–	–	–	Stop
1	0	0	0	0	1
1	0	0	0	1	2
1	0	0	1	0	3
1	0	0	1	1	4
1	0	1	0	0	5
1	0	1	0	1	6
1	0	1	1	0	7
1	0	1	1	1	8

表 **3.4.** 通信预分频系数

<注意>

如果分频系数改变，在启动通信前，要等待两个定时周期作为时钟稳定时间。

第二节　串行口的工作方式

UART 在方式 **0** 和方式 **2** 下为标准的双向串行通信方式，在方式 **1** 下为主-从通信方式。

3.2.1 工作方式

串行口工作方式如表 3.5。

工作方式		数据长度		传输方式	停止位长度
		校验禁止	校验使能		
0	标准方式	7或8位		异步	1 or 2 bits *2
1	多处理器方式	8+1*1	–	异步	
2	标准方式	8	–	同步	无

表 **3.5.** 串行口工作方式

– : 不能设置

*1 : "+1"表明用于通信控制的地址/数据选择位(A/D)

*2 : 接收过程中，只能检测到一个停止位

<注意**>**

UART 方式 1 只能用于主-从连接方式中的主系统。UART 不能用于从系统，因为在接收过程中，它没有地址/数据检测功能。

3.2.2 CPU 间的连接方式

CPU间的连接方式有一对一的连接（标准方式）和主-从连接（多处理器方式）两种。对于任一连接方式，所有CPU的数据长度、是否使能校验和同步方式必须完全相同。操作方式的选择方法如下：

- 在一对一连接方式下，两个CPU必须使用方式0或方式2。异步传输方式选择方式0，同步传输方式选择方式2。
- 在主-从连接方式下，选择方式 1 用于主系统。该连接方式下，禁止奇偶校验。

各种操作方式都可以选择异步方式（起-停同步）或时钟同步方式。

3.2.3 操作使能位

UART使用操作使能位TXE（发送）和RXE（接收）控制发送和接收。如果某一操作被禁止，停止该操作的过程如下：

- 如果接收操作在接收过程中（已经有数据移入接收移位寄存器）中被禁止，待完成该帧的接收，并将接收数据存储在串行输入数据寄存器(SIDRI)后，停止接收操作。
- 如果发送操作在发送过程中（已经有数据从发送移位寄存器中移出）被禁止，要等到串行输出数据寄存器(SODR0/1)内没有数据后，发送操作才停止。

第三节　串行口的波特率

可以选择下列时钟之一用于确定 UART 发送和接收的波特率：
- 专用波特率发生器
- 内部时钟（16 位可重新装入定时器 0）
- 外部时钟（SCK 脚的时钟输入）

3.3.1 使用专用波特率发生器确定波特率

本节介绍专用波特率发生器时钟被选作 UART 传输时钟时，波特率的设置。

3.3.1.1 使用专用波特率发生器确定的波特率

当传输时钟由专用波特率发生器产生时，机器时钟被时钟预分频器分频。已分频的机器时钟被时钟选择器选择的传输时钟分频系数再次分频。异步和同步传输共用机器时钟预分频系数，但是可以选择不同的内部设置值作为异步和同步波特率的传输时钟分频系数。

实际的传输速率可有下面的公式计算得出：
异步传输波特率＝ϕ x (预分频系数) x (异步传输时钟分频系数)
同步传输波特率＝ϕ x (预分频系数) x (同步传输时钟分频系数)
ϕ: 机器时钟频率

3.3.1.2 预分频器的分频系数（异步和同步传输波特率共用）

机器时钟分频系数由CDCR寄存器的DIV3到DIV0位来选择，如表3.6所示。

MD	DIV3	DIV2	DIV1	DIV0	Divided by(div)
0	–	–	–	–	Stop
1	0	0	0	0	1
1	0	0	0	1	2
1	0	0	1	0	3
1	0	0	1	1	4
1	0	1	0	0	5
1	0	1	0	1	6
1	0	1	1	0	7
1	0	1	1	1	8

表 **3.6.** 机器时钟预分频器分频系数的选择

3.3.1.3 同步传输时钟分频系数

同步传输波特率分频系数由方式控制寄存器(SMR0/1)的CS2到CS0位来选择，如表3.7所示。

CS2	CS1	CS0	CLK 同步	计算公式	SCKI
0	0	0	16M	(ϕ ÷ div)/1	(ϕ ÷ div)/1
0	0	1	8M	(ϕ ÷ div)/2	(ϕ ÷ div)/2
0	1	0	4M	(ϕ ÷ div)/4	(ϕ ÷ div)/4
0	1	1	2M	(ϕ ÷ div)/8	(ϕ ÷ div)/8
1	0	0	1M	(ϕ ÷ div)/16	(ϕ ÷ div)/16
1	0	1	500K	(ϕ ÷ div)/32	(ϕ ÷ div)/32

表 **3.7.** 同步传输波特率分频系数的选择

上述计算假定机器时钟周期ϕ = 16 MHz ，div = 1

3.3.1.4 异步传输时钟分频系数

同步传输波特率分频系数由方式控制寄存器(SMR0/1)的CS2到CS0位来选择，如表3.8所示。

CS2	CS1	CS0	异步（起停同步）	计算公式	SCKI
0	0	0	76923	(ϕ ÷ div)/(8x13x2)	(ϕ ÷ div)/(13x1)
0	0	1	38461	(ϕ ÷ div)/(8x13x4)	(ϕ ÷ div)/(13x2)
0	1	0	19230	(ϕ ÷ div)/(8x13x8)	(ϕ ÷ div)/(13x4)
0	1	1	9615	(ϕ ÷ div)/(8x13x16)	(ϕ ÷ div)/(13x8)
1	0	0	500K	(ϕ ÷ div)/(8x2x2)	(ϕ ÷ div)/2
1	0	1	250K	(ϕ ÷ div)/(8x13x4)	(ϕ ÷ div)/4

表 **3.8.** 异步传输波特率分频系数的选择

上述计算假定机器时钟周期ϕ = 16 MHz ，div = 1

3.3.1.5 内部定时器

选用内部定时器，并且CS2到CS0设置为110时，波特率（使用可重装入定时其）的计算公式如下：

异步（起-停同步）：(ϕ ÷ N)/(16 x 2 x (n + 1))

时钟同步：(ϕ ÷ N)/(2 x (n + 1))

N: 定时器计数时钟源　n: 定时器重新装入值

<注意**>**

方式2下(时钟同步方式)，SCK0最多滞后SCK1三个时钟周期。理论上可达到的传输速率为1/3系统时钟频率。实际应用推荐采用1/4系统时钟频率。

3.3.1.6 外部时钟

选用外部时钟，并且CS2到CS0设置为111时， 注意以下事项：

如果外部时钟频率为f，波特率为：

异步（起-停同步）：f/16

时钟同步：f ’

注意 f最大可达1/2机器时钟，f’可达1/8机器时钟

3.3.2 使用内部定时器确定波特率

本节介绍内部定时器——**16** 位可重新装入定时器 **0** 提供的内部时钟用作 **UART** 传输时钟时，波特率的设置。同时给出了波特率的计算公式。

3.3.2.1 使用内部定时器（16 位可重新装入定时器 0）确定波特率

方式控制寄存器(SMR0/1) CS2到CS0写入110_B，选择由内部定时器确定通信波特率。通过选择预分频器分频系数和16位可重新装入定时器0的装入值，可设置多种波特率。

3.3.2.2 波特率计算公式

$$异步波特率 = \frac{\phi}{X(n+1)\times 2\times 16} bps$$

$$同步波特率 = \frac{\phi}{X(n+1)\times 2} bps$$

ϕ: 机器时钟频率
X: 16位可重新装入定时器0预分频系数(2^1, 2^3, or 2^5)
n: 16位可重新装入定时器0的装入值(0 to 65535)

3.3.2.3 重新装入值设置举例（机器时钟：7.3728 MHz）

波特率与定时器的装入值对应如表3.9：

波特率	重新装入值			
	时钟异步(起-停同步)		时钟同步	
	X=2^1 (机器周期2分频)	X=2^3 (机器周期8分频)	X=2^1 (机器周期2分频)	X=2^3 (机器周期8分频8)
38400	2	–	47	11
19200	5	–	95	23
9600	11	2	191	47
4800	23	5	383	95
2400	47	11	767	191
1200	95	23	1535	383
600	191	47	9071	767
300	383	95	6143	1535

表 **3.9.** 波特率与定时器的装入值

X : 16 位可重新装入定时器 0 预分频系数
– ：禁止设置

3.3.3 使用外部时钟确定波特率

本节介绍外部时钟用作 UART 传输时钟时波特率的设置。同时给出了波特率的计算公式。

3.3.3.1 使用外部时钟确定波特率

选择由外部时钟确定的波特率需要经过下面三步设置：

- 串行方式控制寄存器(SMR0/1)的CS2到CS0位写入111_B，选择由外部输入时钟确定波特率。
- 设置SCK0/P40 and SCK1/P62引脚为输入脚(DDR4: bit 0 = 0 和 DDR6: bit 2 = 0)。
- 串行方式控制寄存器(SMR0/1)的SCKE位写入“0”，设置该引脚为外部时钟输入脚。

外部时钟从SCK1引脚输入。因为内部分频系数是固定的，所以要改变通信波特率，就必须改变外部输入时钟的周期。

3.3.3.2 波特率计算公式

异步传输波特率= f/16
同步传输波特率= f
f: 外部时钟频率 (最高2 MHz)

第四节　串行口应用范例

本节给出 UART 编程的实例，分别采用查询和中断两种方式。

3.4.1 查询方式

这里使用 UART0 双向通信功能（标准方式）完成串行接收和发送。

首先定义一个全局变量——字符串 welcome，作为发出的第一个字符串。
unsigned char welcome[30] = "Welcome to Fujitsu";

Inituart0 () 函数初始化 UART0 通道。首先将 P60/SIN0 引脚用于通信输入，设置串行方式控制寄存器 SMR0：使能串行数据输出，使用专用波特率发生器产生波特率 9600bps，串行方式选择为方式 0。设置串行控制寄存器 SCR0：发送使能，接收使能，8 个数据位，1 个停止位，无校验。设置通信预分频控制寄存器 CDCR0：使用通信预分频器并且预分频系数为 1。

```
void inituart0()
{
/* initialize UART0 */
IO_DDR3.bit.D36 = 0 ;      /* 设置 P36/SIN0 引脚用于通信输入：SIN0 */
IO_SMR0.byte = 0x19;       /* 使能 P37/SOT0 引脚用于串行数据输出：SOT0 */
                           /* 使用专用波特率发生器，9600 bps */
                           /* 异步方式 */
IO_SCR0.byte = 0x17;       /* 发送使能 ，接收使能 */
                           /* 清除错误标志 */
                           /* 8 个数据位 ，1 个停止位 ，无校验 */
IO_CDCR0.byte = 0xF0;      /* 设置预分频器系数为 1 */
                           /* 使用通信预分频器 */
}
```

Putch () 函数用于发送一个字节。串行状态寄存器 SSR0 的 TDRE 位指出输出数据寄存器 SIDR0 的状态。TDRE 为 0 表示发送数据写入寄存器 SIDR0，当数据装入发送移位寄存器，启动发送时，该位置“1”。所以等 TDRE 为 1 时，才可继续向输出数据寄存器 SIDR0 写入数据。

```
void Putch (unsigned char ch)          /* 发送字节函数 */
{
while (IO_SSR0.bit.TDRE == 0);     /* 等待发送缓冲区空 */
IO_SIDR0.byte = ch;                /* 向数据寄存器写入数据 */
}
```

Getch () 用于查询接收并返回接收到的数据。SSR0的RDRF位指出输入数据寄存器SIDR0的状态。接收数据装入寄存器SIDR0，该位置“1”，从串行输入数据寄存器SIDR0读数据，该位被清“0”。所以等RDRF为1，表示接收到数据。即可将数据从SIDR0中读出。如果SSR0的ORE位为1，则表示发生超越错误，输入数据寄存器SIDR0的数据无效，返回“-1”。只有ORE为0时，才将从SIDR0中读出的数据作为返回值返回。

```
char Getch(void)                   /* 等待接收并将接收的数据返回 */
{
  unsigned ch;
while(IO_SSR0.bit.RDRF == 0);      /* 等待接收数据 */
if (IO_SSR0.bit.ORE)               /* 超越错误 */
{
    ch = IO_SIDR0.byte;            /* 清除错误标志  */
    return (-1);
}
else
    return (IO_SIDR0.byte);        /* 将接收到的数据返回 */
}
```

putstr () 函数用于发送一个字符串。计算字符串的长度后，逐个发送每个字符。

```
void putstr(unsigned char Name2[30])    /* 发送字符串 */
{
    char c;
    int i , len;
    len = strlen(Name2);                /* 计算字符串长度 */
    for (i=0; i<len; i++)               /* 逐个发送字符 */
    {
    c=(Name2[i]);
    Putch (c);
    }
}
```

主程序 main () 里首先输出字符串“Welcome to Fujitsu”，之后等待接收。一旦接收到数据，就将接收到的数据在发送出去，象反射一样。

```
/*=============================================================*/
void main(void)
{
    char ch;
    setClock(1);            /* set 16 MHz */
    inituart0();
    putstr(welcome);        /* 输出字符串 */
    Putch(10);              /* 回车 */
    Putch(13);              /* 换行 */
    while(1)                /* 等待接收，并将接收到的数据反射回去 */
    {   ch = Getch();
        Putch(ch);
        if (ch==13)
            Putch(10);
    }
}
```

这里需要注意的是：通信的波特率是在机器时钟为 16MHZ 的情况下设置的，所以必须确保机器时钟为 16MHZ，上述串行口通信程序才能正常运行。setClock () 函数根据倍频率来设置 CPU 时钟为 16MHZ，详情参见富士通 16 位硬件手册的时钟章节。

```
/*-------------------------------------时钟程序---------------------------------------*/
/* case1:   8 MHz input clock;          case2:   4MHz input clock;
   set PLL clock:                    cks = 0     1     2     3     4
case1:                  machine clock = 8    16    24    32    4 MHz
case2:                  machine clock = 4     8    12    16    2 MHz
/*------------------------------------------------------------------------------*/
void setClock(char cks)
{
```

```
IO_CKSCR.bit.MCS = 1;          /* PLL 时钟禁用 */
IO_CKSCR.bit.CS = cks & 3;     /* 倍频率选择 */

if (cks & 4)
    return;                    /* main clock selected */
IO_CKSCR.bit.MCS = 0;          /* 使用 PLL 时钟模式 */
while (IO_CKSCR.bit.MCM);      /* 等待 PLL 振荡稳定时间 */
}
```

3.4.2 中断方式

下面给出用中断方式实现标准方式串行口通信的例程。该例程使 UART0 和 UART1 形成一个通信通道：UART0 接收到的数据从 UART1 发送出去，反之亦然，UART1 接收到的数据从 UART0 发送出去。

首先定义 UART0 和 UART1 发送和接收的中断服务程序：

```
__interrupt   void    irqsrvUART0_RX(void);
__interrupt   void    irqsrvUART0_TX(void);
__interrupt   void    irqsrvUART1_RX(void);
__interrupt   void    irqsrvUART1_TX(void);

#pragma intvect   irqsrvUART0_RX 39      /* UART0 接收中断的中断向量 */
#pragma intvect   irqsrvUART0_TX 40      /* UART0 发送中断的中断向量 */
#pragma intvect   irqsrvUART1_RX 37      /* UART1 接收中断的中断向量 */
#pragma intvect   irqsrvUART1_TX 38      /* UART1 发送中断的中断向量 */
```

initUART0()是 UART0 的初始化程序，相关寄存器如 SMR0、SCR0、CDCR0 和查询方式设置的一样。除此之外，还需要设置中断优先级别，并将 UART0 的发送和接收中断禁止。

```
/*----------------------  UART0 初始化 ----------------------*/
void InitUART0(void)
{
IO_ICR14.byte = 5;          /* 设置中断优先级别 */

IO_DDR3.bit.D36 = 0 ;       /* 设置 P60/SIN0 引脚用于通信输入：SIN0 */
IO_SMR0.byte = 0x19;        /* 使能 P61/SOT0 引脚用于串行数据输出：SOT0 */
                            /* 使用专用波特率发生器，9600 bps *//* 异步方式 */
IO_SCR0.byte = 0x17;        /* 发送使能 ，接收使能 */
                            /* 清除错误标志 */
                            /* 8 个数据位 ，1 个停止位 ，无校验 */
IO_CDCR0.byte = 0xF0;       /* 设置预分频器系数为 1 */
                            /* 使用通信预分频器 */
IO_SSR0.bit.TIE     = 0;    /* 发送中断禁止 */
IO_SSR0.bit.RIE     = 0;    /* 接收中断禁止 */
```

```
}
```

initUART1() 是 UART1 的初始化函数。与串行口相关的寄存器 SMR1、SCR1、CDCR1 的设置也和 UART0 初始化函数的设置类似。当然，也要设置 UART1 中断的优先级别。UART1 的发送和接收中断初始设置为处于禁止状态。

```
/*----------------------   UART1 初始化   ----------------------*/
void InitUART1(void)
{
IO_ICR13.byte = 3;          /* 设置中断优先级别 */

IO_DDR6.bit.D60 = 0;        /* 设置 P60/SIN1 引脚用于通信输入：SIN1 */
IO_SMR1.byte = 0x19;        /* 使能 P61/SOT1 引脚用于串行数据输出：SOT1 */
                            /* 使用专用波特率发生器，9600 bps *//* 异步方式 */
IO_SCR1.byte = 0x17;        /* 发送使能 ，接收使能 */
                            /* 清除错误标志 */
                            /* 8 个数据位 ，1 个停止位 ，无校验 */
IO_CDCR1.byte = 0xF0;       /* 设置预分频器系数为 1 */
                            /* 使用通信预分频器 */
IO_SSR1.bit.TIE   = 0;      /* 发送中断禁止 */
IO_SSR1.bit.RIE   = 0;      /* 接收中断禁止 */
}
```

在主程序里除了初始化 UART0 和 UART1 外，还要注意通过时钟函数将机器时钟设为 16MHZ，这样才能保证通信正常。时钟函数 setClock()和查询方式的函数一样，这里不再复述。此外，在初始化之前最好将所有中断禁止。初始化完毕后，等待接收数据前允许中断，使能 UART0 和 UART1 的接收中断。

```
/*----------------------------   主程序   ----------------------------*/
void main(void)
{
     __DI();                          /* 禁止所有中断 */
    setClock(1);                      /* 设置机器时钟为 16MHz */
    InitUART0();                      /* 初始化 UART0 */
    InitUART1();                      /* 初始化 UART1 */
    __set_il(7);                      /* set ILM to 7 */
    __EI();                           /* 允许中断 */
    IO_SSR0.bit.RIE   = 1;            /* 使能 UART0 接收中断 */
    IO_SSR1.bit.RIE   = 1;            /* 使能 UART1 接收中断 */
    while(1);
}
```

下面我们来看看中断处理程序：

一旦 UART0 接收到数据，产生中断，进入 UART0 接收中断处理程序。RDRF 为 1，

表示接收到数据；ORE 为 0，表示没有超越错误；PE 为 0，表示无校验错误。这样判断接收到的是有效数据后，将数据从 UART0 输入数据寄存器 SIDR0 读出。然后使能 UART1 发送中断，并将刚从 UART0 接收到的数据写入 UART1 输出数据寄存器 SODR1。

```
/*------------- UART0 接收中断服务程序 ------------*/
__interrupt
void irqsrvUART0_RX(void)
{
unsigned char ch;
if((IO_SSR0.bit.RDRF==1)&&(IO_SSR0.bit.ORE==0)&&(IO_SSR0.bit.PE==0))
    {     /* 数据有效? */
    ch = IO_SIDR0.byte;             /* 从 UART0 输入数据寄存器读出数据 */
    IO_SSR1.bit.TIE   = 1;          /* UART1 发送中断使能 */
    IO_SIDR1.byte = ch;             /* 向 UART1 输出数据寄存器写入数据 */
    }
}
```

UART0 发送中断处理程序中无需做特别的处理，即是等 UART0 发送缓冲区空时，将 UART0 发送中断禁止。

```
/*------------- UART0 发送中断服务程序 ------------*/
__interrupt
void irqsrvUART0_TX(void)
{
while(IO_SSR0.bit.TDRE==0)   ;     /* 等待 UART0 发送缓冲区空 */
IO_SSR0.bit.TIE  = 0;              /* UART0 发送中断禁止 */
}
```

UART1 接收中断处理程序所做的处理和 UART0 接收中断服务程序类似。在确定接收到数据并且是没有超越错误和校验错误的有效数据后，将数据从 UART1 输入数据寄存器 SIDR1 中读出。然后使能 UART0 发送中断，并将数据写入 UART0 输出数据寄存器 SODR0 中。

```
/*------------- UART1 接收中断服务程序 ------------*/
__interrupt
void irqsrvUART1_RX(void)
{
unsigned char ch;
if((IO_SSR1.bit.RDRF == 1) && (IO_SSR1.bit.ORE == 0) && (IO_SSR1.bit.PE == 0))
    {    /* 接收到的数据有效吗? */
        ch=IO_SIDR1.byte;          /* 从 UART1 输入数据寄存器读出数据 */
        IO_SSR0.bit.TIE   = 1;     /* UART0 发送中断使能 */
        IO_SIDR0.byte = ch;        /* 向 UART0 输出数据寄存器写入数据 */
    }
}
```

UART1 发送中断处理程序中也无需做特别的处理，即是等 UART1 发送缓冲区空时，将 UART1 发送中断禁止。

```
/*------------- UART1 发送中断服务程序  ------------*/
__interrupt
void irqsrvUART1_TX(void)
{
while(IO_SSR1.bit.TDRE == 0);              /*  等待 UART1 发送缓冲区空  */
IO_SSR1.bit.TIE   = 0;
}
```

第四章　串行 **EEPROM** 的 **C** 编程

串行 EEPROM 是可在线电擦除和电写入的存储器，具有体积小、接口简单、数据保存可靠、可在线改写、功耗低等特点，而且为低电压写入，在单片机系统中应用十分普遍。

第一节　硬件原理

4.1.1 器件简介

串行 E2PROM 按总线形式分为三种，即 I2C 总线、Microwire 总线及 SPI 总线三种。I2C 总线采用时钟(SCL)和数据(SDA)两根线进行数据传输，接口十分简单。图 4.1 为 ATMEL 公司的产品 AT24C01A 型 1024 位 I2C 总线串行 E2PROM 的引脚图。

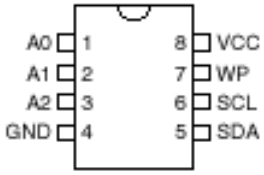


图 **4.1. EEPROM** 引脚图

SDA 是串行数据脚，用于地址、数据的输入和数据的输出，使用时需加上拉电阻。SCL 是时钟脚。该脚为器件数据传输的同步时钟信号。在单片机系统中，总线受单片机控制。单片机产生串行时钟(SCL)，控制总线的存取，发送 STRAT 和 STOP 信号。

4.1.2 总线协议

4.1.2.1 数据的保持和改变

发生在 SCL 低电平期间的数据线(SDA)电平跳变为有效的数据改变，SCL 高电平期间，SDA 上的数据保持稳定。如图 2 所示：

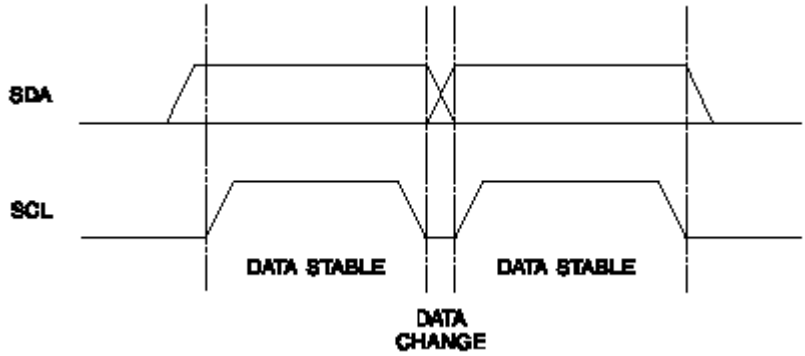


图 **4.2.** 数据的保持和改变时序图

4.1.2.2 数据的开始和结束

在 SCL 为高电平时数据线(SDA)从高电平跳变到低电平，为开始数据传输(START)的条件，开始数据传输条件后所有的命令有效；SCL 为高电平时，数据(SDA)从低电平跳变到高电平，为停止数据传输(STOP)的条件，停止数据传输条件后所有的操作结束。

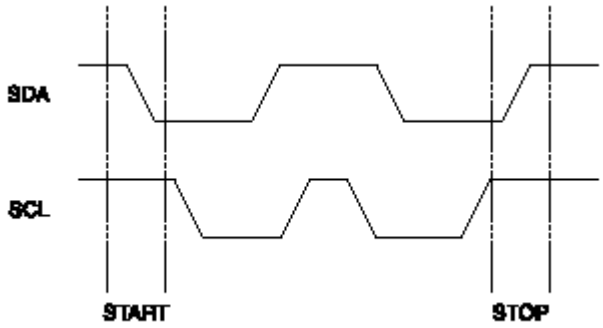


图 **4.3.** 开始**(START)**和停止**(STOP)**的信号时序图

4.1.2.3 ACK 信号

字节写入时，每写完一个字节，EEPROM 会发送一个传送结束信号 ACK 表明已收到——即 SCL 为高电平时 SDA 引脚输出一个低电平。连续读出时，读完一个字节，送一位传送结束信号 ACK，但 STOP 前一位结束时不需要送 ACK 信号。

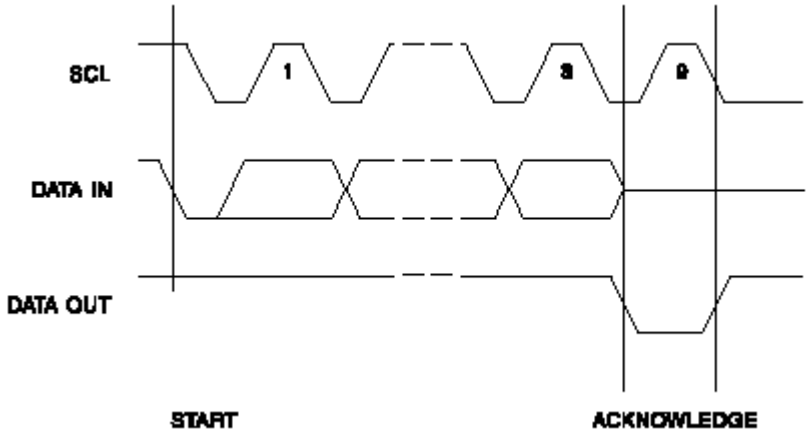


图 **4.4. ACK** 信号的时序图

4.1.3 器件地址

无论是读操作还是写操作，开始发送 START 信号后，单片机都要发送一个控制字，该控制字由 7 位器件地址和 1 位读写(R/W)选择位（“1”为读，“0”为写）组成。对于 ATMEL 公司的 24C01A 来说，器件地址的前四位固定为 1010，后三位跟引脚 A1,A2,A3 的接线有关，一般将 A1,A2,A3 都接地。24C01A 的控制字格式如下：

1　0　1　0　0　0　0　R/W

写控制字为 0xA0，读控制字为 0xA1。24C01A 随时监视总线上是否为有效地址，若器件地址正确且器件未处在编程方式下，传送结束时器件会产生传送结束位 ACK。单片机收到这个 ACK 后才可以继续传输。

4.1.4 写操作

单片机送出开始信号后，接着送写控制字，收到 ACK 后写入待写入数据的存放地址和待写入数据，然后结束一个字节的写操作。即 S＋写控制字(R/W 位为“0”)＋ACK(“0”)＋字地址＋ACK(“0”)＋写入数据＋ACK(“0”)＋STOP。

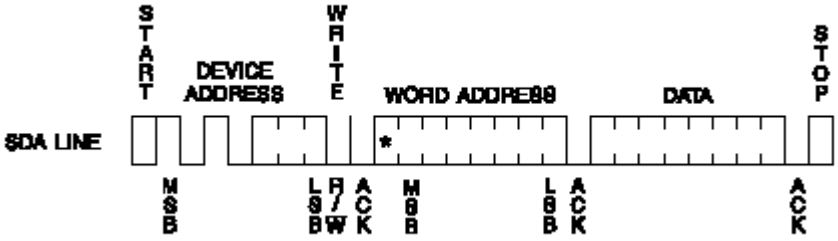


图 **4.5.** 写操作**(**写入一个字节**)**

4.1.5 读操作

读操作分为三种：读当前地址的内容、读指定地址的内容、读指定起始地址后的若干字节的内容，三种读操作的数据传输格式各不相同。

读当前地址的内容为：

S＋读控制字(R/W 位为“1”)＋ACK＋读出数据＋no ACK＋STOP

读指定地址的内容为：

S＋写控制字(R/W 位为“0”)＋ACK＋写入指定的地址字＋ACK＋读控制字(R/W 位为“1”)＋ACK＋读出数据＋no ACK＋STOP

读指定起始地址后的若干字节的内容为：

S＋写控制字(R/W 位为“0”)＋ACK＋写入指定的地址字＋ACK＋读控制字(R/W 位为“1”)＋ACK＋读出数据(1)＋ACK＋……＋读出数据(n＋x)＋noACK＋STOP

下面仅给出读指定地址数据的示意图，另外两种可以类推。

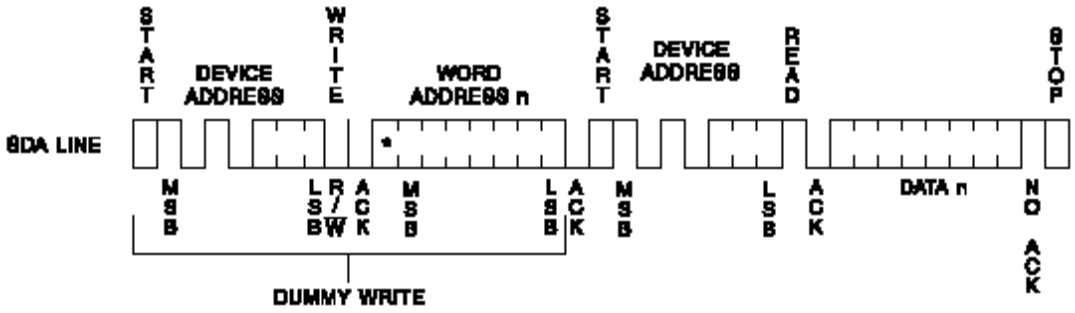


图 **4.6.** 读操作（读出指定地址的数据）

第二节　**C** 语言实现对 **EEPROM** 的读写

对 EEPROM 的接口线路很简单，将 SCL 和 SDA 分别连接到 P62 和 P63。下面通过软件实现对 EEPROM 的读写。

首先将 SDA、SCL 相连的 IO 口端口寄存器和方向寄存器进行宏定义，并定义一个共用体变量 trans 作为中转变量。

Initiic（）函数用于初始化连接 SDA 和 SCL 引脚的 IO 口，设置方向。

begintrans（）和 stoptrans ( ) 函数分别发送一个 START 信号和 STOP 信号，ack ( ) 则是等待给出 ACK 信号。bytein ( ) 通过循环移位向 EEPROM 写入一个字节。byteout ( ) 也通过循环移位从 EEPROM 读出一个字节来。

writebyte ( ) 和 readbyte ( ) 分别是向 EEPROM 进行写操作和读操作（读出指定地址的数据）。为数据显示直观，程序中将读出的数据通过串行口发送出去，调用了串行口初始化程序 inituart0（）、发送程序 Putch（）以及时钟设置程序 setClock（）。

下面给出用 C 语言实现对 EEPROM 读写的程序。

```
#define     SCL         IO_PDR6.bit.P62     /* 定义 P62 端口寄存器为 SCL */
#define     SDA         IO_PDR6.bit.P63     /* 定义 P63 端口寄存器为 SDA */
#define     SCL_DIRE    IO_DDR6.bit.D62     /* 定义 P62 方向寄存器为 SCL_DIRE */
#define     SDA_DIRE    IO_DDR6.bit.D63     /* 定义 P63 方向寄存器为 SDA_DIRE */
/* 定义一个中间变量 trans，通过移位来实现串行输入  */
union   {
        char        byte;
        struct
            {char   bit0:1;
             char       :6;
             char   bit7:1;
             } bit;
        } trans;
```

```
void initiic()                          /*  初始化 SDA,SCL */
{
    SCL=0;                              /* SCL=0 */
    SDA=0;                              /* SDA=0 */
    SCL_DIRE=1;                         /* SCL  方向: 输出  */
    SDA_DIRE=1;                         /* SDA  方向: 输出  */
}

void begintrans()                       /*  发送 START 信号  */
{
    SDA=1;                              /* SDA=1 */
    SDA_DIRE=1;                         /* SDA 方向为输出到 EEPROM */
    SCL=1;                              /* SCL=1 */
    SDA=0;                              /* SDA=0 */
}

void stoptrans()                        /*  发送 STOP 信号  */
{
    SDA=0;                              /* SDA=0 */
    SDA_DIRE=1;                         /* SDA 方向为输出到 EEPROM */
    SCL=1;                              /* SCL=1 */
    SDA=1;                              /* SDA=1 */
    delay(0x02);
}

void ack()                              /*  等待 EEPROM 的 ACK 信号  */
{
    char d;
    SDA_DIRE=0;                         /* SDA 方向为从 EEPROM 输入  */
    SCL=1;                              /* SCL=1 */
    do
    {    d=SDA;  }
    while(d==1);                        /*  等待 EEPROM 输出低电平  */
    SCL=0;                              /* SCL=0 */
}

void bytein(char ch)                    /*  向 EEPROM 写入一个字节  */
{
    int i;
    trans.byte=ch;
    SCL=0;                              /* SCL=0 */
    SDA=trans.bit.bit7;                 /*  数据首位 MSB */
    SDA_DIRE=1;                         /* SDA 方向为输出到 EEPROM   */
```

```
    for(i=8;i>0;i- -)
    {
    SDA=trans.bit.bit7;                 /* 数据通过 SDA 串行移入 EEPROM */
    SCL=1;                              /* SCL=1 */
    trans.byte=trans.byte*2;
    SCL=0;                              /* SCL=0 */
    }
    ack();
}

void writebyte(char addr,char data)     /* 向 EEPROM 指定地址写入一个字节的数据 */
{
    begintrans();
    bytein(0xA0);                       /* 写入写控制字：0xA0 */
    bytein(addr);                       /* 写入指定地址 */
    bytein(data);                       /* 写入待写入 EEPROM 的数据 */
    stoptrans();
    delay(0x5FF);                       /* 延时 10ms */
}

char readbyte(char addr)                /* 从 EEPROM 指定地址读取一个字节的数据 */
{
    char c;
    begintrans();                       /* START */
    bytein(0xA0);                       /* 写入写控制字：0xA0 */
    bytein(addr);                       /* 写入指定地址 */
    begintrans();                       /* START */
    bytein(0xA1);                       /* 写入读控制字：0xA1 */
    c=byteout();                        /* 读出 EEPROM 输出的数据 */
    stoptrans();                        /* STOP */
    return(c);
}

char byteout()                          /* 从 EEPROM 输出一个字节 */
{
    int i;
    char ch;
    SDA_DIRE=0;                         /* SDA 的方向为从 EEPROM 输出 */
    for(i=8;i>0;i--)
    {
    trans.byte=trans.byte*2;
    SCL=1;                              /* SCL=1 */
    trans.bit.bit0=SDA;                 /* 数据通过 SDA 串行移出 EEPROM */
    SCL=0;                              /* SCL=0 */
```

```
    }
    ch=trans.byte;
    return(ch);
}

void delay(int   time )                          /* 延时函数 */
{
    for(；time>0；time--)
        ;
}

void inituart0()                                 /* 初始化 UART0 ——参见串行口应用一节 */
{
    IO_DDR3.bit.D36 = 0 ;
    IO_SMR0.byte = 0x19;
    IO_SCR0.byte = 0x17;
    IO_CDCR0.byte = 0xF0;
}
void Putch (unsigned char ch)            /* 发送字符函数——细节参见串行口 UART */
{
    while (IO_SSR0.bit.TDRE == 0);       /* 等待发送缓冲区空 */
    IO_SIDR0.byte = ch;                  /* 向数据寄存器写入数据 */
}
void setClock(char cks)
    {
    IO_CKSCR.bit.MCS = 1;          /* PLL 时钟禁用 */
    IO_CKSCR.bit.CS = cks & 3;     /* 倍频率选择 */

    if (cks & 4)
        return;                    /* main clock selected */
    IO_CKSCR.bit.MCS = 0;          /* 使用 PLL 时钟模式 */
    while (IO_CKSCR.bit.MCM);      /* 等待 PLL 振荡稳定时间 */
}

void main(void)
{
    char ch1，ch2；
    setClock(1);
    inituart0();
    initiic();
    writebyte（0x10，‘a’）；          /* 向 EEPROM 的地址 0x20 写入数据‘a’ */
    writebyte（0x11，‘b’）；          /* 向 EEPROM 的地址 0X21 写入字符‘b’*/
    ch1=readbyte（0x10）；            /* 读出 EEPROM 地址单元 0x20 的数据 */
    ch2=readbyte（0x11）；            /* 读出 EEPROM 地址单元 0X21 的数据 */
```

```
    Putch（ch1）;
    Putch（ch2）;                    /* 将读出的数值发送出去 */
}
```

第五章　液晶显示的 C 编程

在单片机应用中，液晶（LCD）显示是重要的一个部分。液晶显示器以其低功耗、重量轻、体积小等诸多优点，在袖珍式仪表和低功耗应用系统中，得到越来越广泛的应用。

第一节　液晶显示模块概述

液晶显示器分很多种类，按显示方式可分为段式，行点阵式和全点阵式。段式与数码管类似，行点阵式一般是英文字符，全点阵式可显示任何信息，如汉字、图形、图表等。这里我们使用一种行点阵式字符型 LCD 显示模块，该模块集成了 LCD 板、PCB 板、控制器驱动器。

液晶板上排列着若干 5 x 7 或 5 x 10 点阵的字符显示位，每个显示位显示 1 个字符。根据能显示的行数以及每行的位数不同分为不同的规格，这里使用的是 16 位 x 2 行模块。在接口方面，该模块通过 8 条数据线以及 3 条控制线与微控制器相连，通过送入的数据和指令进行工作。

LCD 控制器内包含有显示数据 RAM，字符发生器 ROM，字符发生器 RAM。详细解释如下：

DDRAM：

显示数据 RAM，用来寄存待显示的代码。容量：80 X 8 bits（80 字符）。

DDRAM 的地址：

LCD 控制器的指令系统规定，在送待显示字符代码的指令前，先要送 DDRAM 的地址，实际上就是待显示的字符显示位置。若 LCD 为双行字符显示，每行 40 个显示位置，第一行地址为 00H~27H；第二行地址为 40H~67H。双行显示的 DDRAM 地址与显示位置的对应关系如表 1 所示。

显示位置		**1**	**2**	**3**	**4**	**5**	**6**	**7**		**39**	**40**
DDRAM 地址	第一行	00H	01H	02H	03H	04H	05H	06H	…	26H	27H
	第二行	40H	41H	42H	43H	44H	45H	46H	…	46H	67H

表 **5.1.** 显示位置与 **DDRAM** 地址的对应关系

CGROM：字符发生器 ROM，用于提供用户所需字符库或标准库。

CGROM 字符容量：192 个字符（5 x 7 点字形）；32 个字符（5 x 10 点字形）。

CGRAM：字符发生器 RAM，它是 8 个允许用户自定义的字符图形 RAM。

CGRAM 和 CGROM 的字符图形对应关系入表 2 所示。

AC：地址计数器。AC 的内容是 DDRAM 或 CGRAM 的单元地址。当对 DDRAM 或 CGRAM 进行读写操作后，AC 自动加 1 或减 1。

光标/闪烁控制：此控制可产生光标或使光标在显示位置处闪烁，显示位置为 AC 中的 DDRAM 地址。

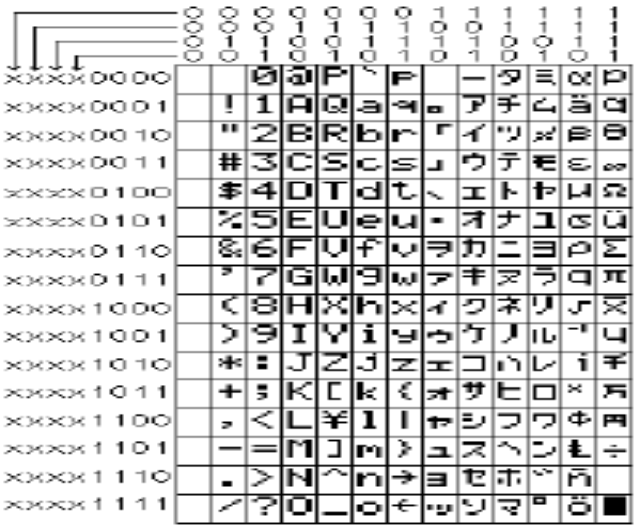


表 **5.2. CGROM** 和 **CGRAM** 中字符代码与字符图形对应关系

第二节　液晶显示模块引脚功能和寄存器选择功能

字符型 LCD 模块有 14 个引脚线，其接线如下图所示：

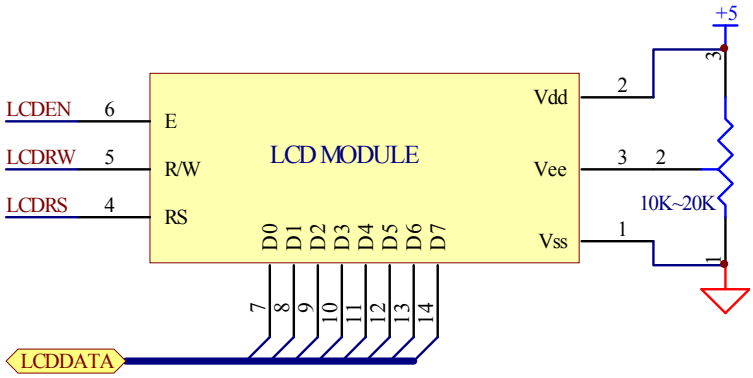


图 **5.1.** 液晶模块引脚接线图

这 14 个引脚线的定义是标准的，其功能如表 3 所示：

引脚线号	符号	名称	备注
1	Vss	接地	0V
2	Vdd	电路电源	5V±10%
3	Vee	液晶驱动电压	通过可调电阻输入电压，在 0~5V 内
4	RS	寄存器选择信号	H：数据寄存器；L：指令寄存器
5	R/W	读/写信号	H：读；L：写
6	E	片选信号	下降沿触发
7~14	D0~D7	数据线	数据传输

表 **5.3.** 液晶模块引脚功能

备注：

指令寄存器：用来接收 CPU 送来的指令码；也寄存 DDRAM 和 CGRAM 的地址。

数据寄存器：用来寄存 CPU 发来的字符代码数据。

寄存器操作的选择是通过 RS 和 R/W 两个引脚线的输入组合控制的。对应关系如表 4：

RS	**R/W**	寄存器操作
0	0	指令寄存器（IR）写入
0	1	忙标志和地址计数器读出
1	0	数据寄存器（DR）写入
1	1	数据寄存器读出

表 **5.4.** 寄存器选择操作

备注：

状态标志位：LCD 控制器有一个忙信号标志位 BF。当 BF=1 时，表示 LCD 正在进行内部操作，此时不能输入指令或数据，要等内部操作结束，BF 为“0”时才可以。

第三节　液晶显示模块指令系统

字符型 LCD 模块由 11 条指令，它的读写指令、屏幕和光标的操作等等都是通过指令编程来实现的。其指令格式如下所示：

RS　R/W　　D7　　D6　　D5　　D4　　D3　　D2　　D1　　D0

指令	**RS**	**R/W**	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
清屏	0	0	0	0	0	0	0	0	0	1
光标返回	0	0	0	0	0	0	0	0	1	*
输入方式设置	0	0	0	0	0	0	0	1	I/D	S
显示开/关控制	0	0	0	0	0	0	1	D	C	B
光标或显示移位	0	0	0	0	0	1	S/C	R/L	*	*
模式设置	0	0	0	0	1	DL	N	F	*	*
字符发生存储器(CGRAM)地址设置	0	0	0	1	字符发生存储器（CGRAM）地址 AGG					
数据存储器(DDRAM)地址设置	0	0	1	显示数据存储器（DDRAM）地址 ADD						
读忙标志(BF)和地址计数器	0	1	BF	用于 DD、CGRAM 地址的地址计数器 AC						
写数据到 CGRAM 或 DDRAM	1	0	待写的数据							
从 CGRAM 或 DDRAM 读数据	1	1	读出的数据							

表 **5.5.** 指令表一览

上表中*位无关项。

下面详细解释各指令的功能：

指令 1：清屏，光标复位到地址 00H 位置。
指令 2：DDRAM 地址为 0，光标返回到地址 00H；DDRAM 内容不变，显示的内容不变。
指令 3：读/写方式下光标移动方向以及整体显示是否移动的设定。
　　I/D：地址计数器的变化方向，及光标移动的方向。
　　I/D 为 1 是增量方式，表示 AC 自动加 1，光标右移一字符位；I/D 为 0 是减量方式，表示 AC 自动减 1，光标左移一字符位。
　　S： 整体显示是否移动。S 为 0，不移动；S 为 1，移动。
　　S=1，I/D=1：整体显示左移；S=1，I/D=0：整体显示右移。
指令 4：显示开关控制，控制整体显示、光标显示、光标位字符闪烁的开关。
　　D： D=1 时显示开，D=0 时显示关闭；DDRAM 中内容不变。
　　C： C=1 时显示光标，C=0 时不显示光标。
　　B： B=1 时光标位字符闪烁，B=0 时无闪烁。
指令 5：光标或整体显示移位，同时 DDRAM 的内容保持不变。
　　S/C：S/C=1，整体显示移动；S/C=0，光标移动。
　　R/L：R/L=1，右移；R/L=0，左移。
指令 6：模式设置命令。
　　DL：接口数据位数。DL=1，8 位；DL=0，4 位。
　　N： 显示行数。N=1，双行显示；N=0，单行显示。
　　F： 显示字形点阵样式。F=1，5 X 10 点阵；F=0，5 X 7 点阵。
指令 7：CGRAM 地址设置。设置后 CGRAM 数据被发送和接收。
指令 8：DDRAM 地址设置。设置后 DDRAM 数据被发送和接收。
指令 9：读忙信号位(BF)以及 AC 中的地址。
指令 10：写数据到 CGRAM 或 DDRAM。
指令 11：从 CGRAM 或 DDRAM 读数据。

第四节　**LCD** 显示模块的接口以及 **C** 语言编程

下面提供一个用字符型 LCD 模块显示字符和数字的示例。LCD 的数据线 D0~D7 接 MB90560 的 P1 口，LCD 模块 RS 引脚接 P44 口，R/W 引脚接 P45 口，E 引脚接 P46 口。

首先将这些接口的端口寄存器和方向寄存器进行宏定义：

```
#define     LCDEN       IO_PDR4.bit.P46     /* LCD 模块 E 引脚 */
#define     LCDRS       IO_PDR4.bit.P44     /* LCD 模块 RS 引脚 */
#define     LCDRW       IO_PDR4.bit.P45     /* LCD 模块 R/W 引脚 */
#define     LCDDATA     IO_PDR1.byte        /* LCD 数据接口 */
#define     EN_DIRE     IO_DDR4.bit.D46     /* 各接口方向寄存器设置 */
#define     RS_DIRE     IO_DDR4.bit.D44
#define     RW_DIRE     IO_DDR4.bit.D45
#define     DATA_DIRE   IO_DDR1.byte
```

checkbusy ( ) 函数用于检查 LCD 模块的忙标志位 BF，以确定 LCD 是否仍在忙于内部操作。需要提醒的是 checkbusy 函数里首先需要将连接 LCD 数据引脚的 IO 口方向设为输入口，这样才能读出数据。同样的道理，读出 BF=0 返回时，将 IO 口方向设回输出口，以便继续向 LCD 发送命令或数据。

```
void checkbusy ( )
    {
    unsigned char i ;
    i = 1;
    DATA_DIRE = 0x00;   /* 首先将 IO 口方向改为输入，用以读数据 */
    while(i)
        {
        LCDRS = 0 ;          /* 选择命令寄存器 */
        LCDRW = 1 ;          /* 读操作 */
        LCDEN = 1 ;          /* lcdEN 置高电平 */

        i = IO_PDR1.bit.P17;       /* 检查忙标志位 */

        LCDEN = 0;           /* lcdEN 置低，下降沿触发 */
        LCDEN = 1;           /* lcdEN 置高 */
        LCDEN = 0;           /* lcdEN 置低，下降沿触发 */

        LCDRW = 0 ;
        }
        DATA_DIRE = 0xFF;   /* IO 口方向改为输出 */
    }
```

sendbyte ( ) 函数用于向 LCD 模块写入一个字节 data，发送的是数据还是命令是由 flag 决定的，flag 为 1 表示写入数据寄存器；flag 为 0 表示写入命令寄存器。值得一提的是写入字节后应该调用 checkbusy () 以检查 LCD 模块是否忙于内部操作。

```
void sendbyte(unsigned char data , int flag)
    {
    int i,dataflag;
    i=data;
    dataflag=flag;
    LCDRW = 0 ;              /* 写入字节 */
    LCDRS = dataflag;        /* 选择要写入的寄存器
                                1：数据寄存器     0：命令寄存器 */
    LCDDATA = i;
    LCDEN = 1;               /* lcdEN 置高电平*/
    LCDEN = 0;               /* lcdEN 置低，一个下降沿触发 */
    checkbusy();             /* 检查忙标志 */
```

```
}
```

initlcd（）函数对是初始化函数。除了对 MB90560 相应的 IO 口做一些初始化工作外，还要设置 LCD 模块的工作模式。比如，设置为 8 位数据操作，2 行显示，字符为 5x10 点阵等，这些命令可以参照前面的指令系统。

```
void initlcd()
    {
        /* 设置所使用的 IO 口方向 */
        DATA_DIRE = 0xFF;       /* 初始时为输出数据到 LCD 模块 */
        EN_DIRE = 1;            /* 输出 */
        RW_DIRE = 1;            /* 输出 */
        RS_DIRE = 1;            /* 输出 */

        sendbyte(0x3C,0);       /* 模式设置：8 位操作，2 行显示，5x10 点阵 */
        sendbyte(0x08,0);       /* 关显示 */
        sendbyte(0x06,0);       /* AC 增量方式显示，整体不移位 */
        sendbyte(0x01,0);       /* 清屏 */

        sendbyte(0x0E,0);       /* 开显示 */
    }
```

displaystr（）是一个从指定位置开始显示字符串的函数。第一个参数是要显示的字符串，第二个参数是显示位置的行数，最后一个参数是显示位置的列数。函数的流程，简而言之就是先将光标移到指定位置（即通过行数和列数计算出 DDRAM 的地址 AC），计算字符串长度后逐个显示字符，直至全部显示。

```
void displaystr(char *string1,int row,int column)
    {
    unsigned char c;
    int i,l;
    l=strlen(string1);          /* 计算字符串长度 */
    i=row*0x40+column+0x80;     /* 字符串的显示位置 */
    sendbyte(i,0);
    for(i=0;i<l;i++)
        {
        c=string1[i];
        sendbyte(c,1);
        }
    }
```

displaynum（）函数是以十进制的形式显示数字的函数。参数和上面显示字符串的函数类似，首先是要显示的数字，然后分别是要显示位置的行数和列数。这里第一个显示位的位置确定也同上面所说的一样。将数字反复除 10，并将余数存入缓冲区，然后从缓冲区

逐个输出就是十进制形式的数字。

```
void displaynum(int number1,int numrow,int numcolumn)
{
    int i,a,position,count;
    int buffer[20]={0};
    position=numrow*0x40+numcolumn+0x80;    /* 显示位置 */
    a=number1;                              /* display number */
    if(a<10)                                /* 如果是个位数，直接输出 */
    {   sendbyte(position,0);
        sendbyte(a+48,1);
    }
    else                            /* 如果是大于 10 的数，通过反复除 10 得出每个位数值 */
    {   for(count=0,a=number1;a>=10;count++)
        {
        buffer[count]=a%10;         /* 将余数存入 buffer ，及每个位数 */
        a=a/10;
        }
        buffer[count]=a;
        sendbyte(position,0);
        for(i=count ; i>=0 ; i--)   /* 逐个显示每位数 */
        sendbyte(buffer[i]+48,1);
    }
}
```

主程序里首先调用初始化函数 initlcd () ，然后从第一行第一个位置开始显示字符串“ Have a nice day! ”，并第二行第 6 位显示数字 365。

```
/* 主程序 */
void main()
{
    initlcd();
    displaystr("Have a nice day!",0,0);
    displaystr("Total : ",1,1);
    displaynum(365,1,10);
}
```

第六章　步进电机控制的 C 编程

第一节　步进电机及其工作方式

步进电机也称为脉冲电机，它可以直接接收来自计算机的数字脉冲，使电机旋转过相应的角度。在要求精确定位的场合作为执行部件，步进电机得到广泛采用。

步进电机有如下特点：给步进脉冲电机就转，不给步进脉冲电机就不转；步进脉冲的频率越高，步进电机转的越快；改变各相的通电方式，可以改变电机的运行方式；改变通电顺序，可以控制步进电机的正、反转。

这里采用的是四相步进电机，其工作方式为四相四拍，顺时钟转动（以轴的方向看）的绕组通电顺序为：AB→BC→CD→DA→AB；逆时钟转动的绕组通电顺序为：AB→AD→CD→BC→AB。

第二节　用 C 语言控制步进电机

图 1 是步进电机和 MB90560 的接口电路。这里所采用的是四相步进电机 SMR40，48 步一周。驱动器 MCT1413P 实现对步进脉冲的驱动，P4.0~P4.3 四位用来控制步进电机定子 A、B、C、D 三相控制绕组通电与断电，例如：P40 置低电平，A 绕组不通电；P40 置高电平，A 绕组通电；

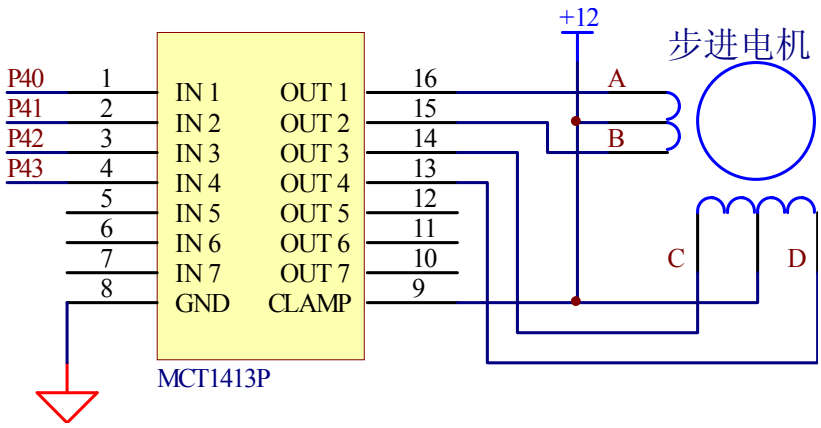


图 **6.1.** 步进电机接线图

下面通过软件实现步进脉冲的产生及脉冲在各相绕组的分配和电机的顺时钟、逆时钟转动控制。四相四拍顺时钟转动脉冲顺序为：AB→BC→CD→DA→AB，P4 口低 4 位输出的控制字为：0x03→0x06→0x0C→0x09→0x03；逆时钟转动脉冲顺序为：AB→AD→CD→BC→AB，P4 口低 4 位输出的控制字为：0x03→0x09→0x0C→0x06→0x03。

定义 PHASEPORT 为 P4 口端口寄存器的内容，PHASE_DIRE 为 P4 口方向寄存器的内容，以提高程序的可读性和可移植性。

```
#define     PHASEPORT       IO_PDR4.byte
#define     PHASE_DIRE      IO_DDR4.byte
```

顺时钟转动和逆时钟转动的 4 个控制字分别放在两个数组 clkwise [ ]，antiwise [ ]中。数组以 00 作为结尾字节便于判断 4 个控制字的结束以恢复初值，重新循环。

```
/* 定义全局变量 */
unsigned char clkwise[5]={0x03,0x06,0x0C,0x09,0x00};        /* 顺时钟转动 */
unsigned char antiwise[5]={0x03,0x09,0x0C,0x06,0x00};       /* 逆时钟转动 */
unsigned char portinit;
unsigned char *p;
```

initmotor () 函数用于初始化与各相相连的 IO 口，设置方向和初始输出值。

```
void initmotor()                                    /* 初始化 */
{
    PHASEPORT=PHASEPORT&0xF0;                       /* 各相均不加电 */
    PHASE_DIRE=PHASE_DIRE|0x0F;                     /* 方向为输出 */
    portinit=PHASEPORT;
}
```

产生控制脉冲的函数 control（）包含两个参数：步进电机的转动方向 direct，转动的步数 n。根据转动方向的值，指针 p 指向不同的转动控制字数组。direct 为 0 时，顺时钟转动，direct 为 1 时，逆时钟转动。然后 IO 口依次输出 4 个控制字，4 个控制字结束后恢复初值，重新循环，这样一直转动到指定的步数时结束。函数结束前，不要忘了将 IO 口恢复输出初始值，使各绕组处于不通电状态，否则绕组会导电发热。

```
void control(char direct,int n)
{
    int i;
    if(direct==0)  p=&clkwise[0];
    else           p=&antiwise[0];
    for(i=0;i<n;i++)
    {
        PHASEPORT=portinit+*p;
        p++;
        if(*p==0)      p=p-4;                       /* 4 个控制字结束后恢复初值 */
        else ;
        delay(0x3FFF);                              /* 通过改变延时可以改变转速 */
    }
```

```
    PHASEPORT=portinit;                          /* 切断所有绕组的通电 */
}
```

延时函数 delay（）用于延时一段时间，改变其中的参数即改变步进脉冲的频率，可以改变步进电机转动的快慢。

```
void delay(int i)                                /* 延时函数 */
{
    for( ;i>0;i--)
    ;
}
```

主程序里先使电机顺时针转动 48 步即一圈，然后逆时针转动一圈。

```
/* 主程序 */
void main()
{
    initmotor();
    control (0,48);                              /* 顺时钟转动 48 步 */
    control (1,48);                              /* 逆时钟转动 48 步 */
}
```