单片机教程(二)

极客学院出版

前言

单片机又称单片微控制器,它不是完成某一个逻辑功能的芯片,而是把一个计算机系统集成到一个芯片上。相当于一个微型的计算机，和计算机相比，单片机只缺少了I/O设备。本文是全程套教程第二部分。

适用人群

本文手把手教你学51单片机，可让您夯实单片机基础。

学习前提

- 本教程中，单片机的功能由C语言实现，如果您不了解C语言，看清阅读C语言入门教程。
- 在学习本教程之前，你需要对 单片机教程(一)有一定了解。

> 致谢：http://c.biancheng.net/cpp/shell/

更新日期	更新内容
2015-08-7	单片机教程(二)

目录

1

7. 变量进阶与点阵 LED

当我们走在马路上的时候，经常会看到马路两侧有一些 LED 点阵的广告牌，这些广告牌看起来绚烂夺目，非常吸引人，而且还会变化很多种不同的显示方式。本章我们就会学习到点阵 LED 的控制方式，但是首先得来了解一点 C 语言变量的进阶知识——变量的作用域和存储类别。

7.1 C 语言变量的作用域

所谓的作用域就是指变量起作用的范围，也是变量的有效范围。变量按他的作用域可以分为局部变量和全局变量。

局部变量

在一个函数内部声明的变量是内部变量，它只在本函数内有效，在本函数以外是不能使用的，这样的变量就是局部变量。此外，函数的形参也是局部变量，形参我们会在后面再详细解释。

比如上节课程序中定义的 unsigned long sec 这个变量，它是定义在 main 函数内部的，所以只能由 main 函数使用，中断函数就不能使用这个变量。同理，我们如果在中断函数内部定义的变量，在 main 函数中也是不能使用的。

全局变量

在函数外声明的变量就是全局变量。一个源程序文件可以包含一个或者多个函数，全局变量的作用范围是从它开始声明的位置一直到程序结束。

比如上节课程序中定义的 unsigned char LedBuff[6]这个数组，它的作用域就是从开始定义的位置一直到程序结束，不管是 main 函数，还是中断函数 InterruptTimer0，都可以直接使用这个数组。

局部变量只有在声明它的函数范围内可以使用，而全局变量可以被作用域内的所有的函数直接使用。所以在一个函数内既可以使用本函数内声明的局部变量，也可以使用全局变量。

从编程规范上讲，一个程序文件内所有的全局变量都应定义在文件的开头部分，在文件中所有函数之前。

由于 C 语言函数只有一个返回值，但是我们却经常会希望一个函数可以提供或影响多个结果值，这时我们就可以利用全局变量来实现。但是考虑到全局变量的一些特征，应该限制全局变量的使用，过多使用全局变量也会带来一些问题。

1) 全局变量可以被作用域内所有的函数直接引用，可以增加函数间数据联系的途径，但同时加强了函数模块之间的数据联系，使这些函数的独立性降低，对其中任何一个函数的修改都可能会影响到其它函数的执行结果，函数之间过于紧密的联系不利于程序的维护的。

2) 全局变量的应用会降低函数的通用性，函数在执行的时候过多依赖于全局变量，不利于函数的重复利用。目前我们编写的程序还都比较简单，就一个 .c 文件，但以后我们要学到一个程序中有多个 .c 文件，当一个函数被

另外一个 .c 文件调用的时候，必须将这个全局变量的变量值一起移植，而全局变量不只被一个函数调用，这样会引起一些不可预见的后果。

3) 过多使用全局变量会降低程序的清晰度，使程序的可读性下降。在各个函数执行的时候都可能改变全局变量值，往往难以清楚的判断出每个时刻各个全局变量的值。

4) 定义全局变量会永久占用单片机的内存单元，而局部变量只有进入定义局部变量的函数时才会占用内存单元，函数退出后会自动释放所占用的内存。所以大量的全局变量会额外增加内存消耗。

综上所述之原因，在编程规范上有一条原则，就是尽量减少全局变量的使用，能用局部变量代替的就不用全局变量。

还有一种特殊情况，大家在看别人程序的时候请注意，C 语言是允许局部变量和全局变量同名的，他们定义后在内存中占有不同的内存单元。如果在同一源文件中，全局变量和局部变量同名，在局部变量作用域范围内，只有局部变量有效，全局变量不起作用，也就是说局部变量具有更高优先级。但是从编程规范上讲，是要避免全局变量与局部变量重名的，从而避免不必要的误解和误操作。

7.2 C 语言变量的存储类别

变量的存储类别分为自动、静态、寄存器和外部这四种。其中后两种我们暂不介绍，主要是自动变量和静态变量这两种。

函数中的局部变量，如果不加 static 这个关键字来修饰，都属于自动变量，也叫做动态存储变量。这种存储类别的变量，在调用该函数的时候系统会给他们分配存储空间，在函数调用结束后会自动释放这些存储空间。动态存储变量的关键字是 auto，但是这个关键字是可以省略的，所以我们平时都不用。

那么与动态变量对应的就是静态变量。首先，全局变量均是静态变量，此外，还有一种特殊的局部变量也是静态变量。即我们在定义局部变量时前边加上 static 这个关键字，加上这个关键字的变量就称之为静态局部变量，它的特点是，在整个生存期中只赋一次初值，在第一次执行该函数时，它的值就是给定的那个初值，而之后在该函数所有的执行次数中，它的值都是上一次函数执行结束后的值，即它可以保持前次的执行结果。

有这样一种情况，某个变量只在一个函数中使用，但是我们却想在函数多次调用期间保持住这个变量的值而不丢失，也就是说在该函数的本次调用中该变量值的改变要依赖与上一次调用函数时的值，而不能每次都从初值开始。如果我们使用局部动态变量的话，每次进入函数后上一次的值就丢失了，它每次都从初值开始，如果定义成全局变量的话，又违背了我们上面提到的尽量减少全局变量的使用这条原则，那么此时，局部静态变量就是最好的解决方案了。

比如第六章最后的例程中有一个控制数码管动态扫描显示用的索引变量 i，我们当时就是定义成了全局变量，现在我们就可以改成局部静态变量来试试。

```
 #include <reg52.h>



sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;

unsigned char code LedChar[] = {  //数码管显示字符转换表
    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
    0x80, 0x90, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E
};
unsigned char LedBuff[6] = {  //数码管显示缓冲区，初值 0xFF 确保启动时都不亮
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
};
unsigned int cnt = 0;//记录 T0 中断次数
```

```
void main(){
    unsigned long sec = 0; //记录经过的秒数


    EA = 1;  //使能总中断
    ENLED = 0;  //使能 U3，选择控制数码管
    ADDR3 = 1;  //因为需要动态改变 ADDR0-2 的值，所以不需要再初始化了
    TMOD = 0x01;  //设置 T0 为模式1
    TH0 = 0xFC;  //为 T0 赋初值 0xFC67，定时 1 ms
    TL0 = 0x67;
    ET0 = 1;  //使能 T0 中断
    TR0 = 1;  //启动 T0

    while (1){
        if (cnt >= 1000){  //判断 T0 溢出是否达到1000次
            cnt = 0;  //达到1000次后计数值清零
            sec++;  //秒计数自加1

            //以下代码将 sec 按十进制位从低到高依次提取并转为数码管显示字符
            LedBuff[0] = LedChar[sec%10];
            LedBuff[1] = LedChar[sec/10%10];
            LedBuff[2] = LedChar[sec/100%10];
            LedBuff[3] = LedChar[sec/1000%10];
            LedBuff[4] = LedChar[sec/10000%10];
            LedBuff[5] = LedChar[sec/100000%10];
        }
    }
}
/* 定时器0中断服务函数 */
void InterruptTimer0() interrupt 1{
    static unsigned char i = 0; //动态扫描的索引，定义为局部静态变量
    TH0 = 0xFC; //重新加载初值
    TL0 = 0x67;
    cnt++;  //中断次数计数值加1

    //以下代码完成数码管动态扫描刷新
    P0 = 0xFF;  //显示消隐
    switch (i){
        case 0: ADDR2=0; ADDR1=0; ADDR0=0; i++; P0=LedBuff[0]; break;
        case 1: ADDR2=0; ADDR1=0; ADDR0=1; i++; P0=LedBuff[1]; break;
        case 2: ADDR2=0; ADDR1=1; ADDR0=0; i++; P0=LedBuff[2]; break;
        case 3: ADDR2=0; ADDR1=1; ADDR0=1; i++; P0=LedBuff[3]; break;
        case 4: ADDR2=1; ADDR1=0; ADDR0=0; i++; P0=LedBuff[4]; break;
        case 5: ADDR2=1; ADDR1=0; ADDR0=1; i=0; P0=LedBuff[5]; break;
```

```
        default: break;
    }
}
```

大家注意看程序中中断函数里的局部变量 i，我们为其加上了 static 关键字来修饰，就成为了静态局部变量。它的初始化 i = 0 操作只进行一次，程序执行代码中会进行 i++等操作，那么下次再进入中断函数的时候，i 会保持上次中断函数执行完毕后的值。如果去掉 static 这个关键字，那么每次进入中断函数后，i 都会被初始化成0，大家可以自己修改程序看一下实际效果是否和理论相符。

7.3 单片机 LED 点阵的介绍

点阵 LED 显示屏作为一种现代电子媒体，具有灵活的显示面积（可任意分割和拼装）、高亮度、长寿命、数字化、实时性等特点，应用非常广泛。

前边学了 LED 小灯和 LED 数码管后，学 LED 点阵就要轻松得多了。一个数码管是8个 LED 组成，同理，一个8*8*的点阵就是由64个 *LED 小灯组成。图7-1就是一个点阵 LED 最小单元，即一个*8*8*的点阵 LED，图7-2是它的内部结构原理图。

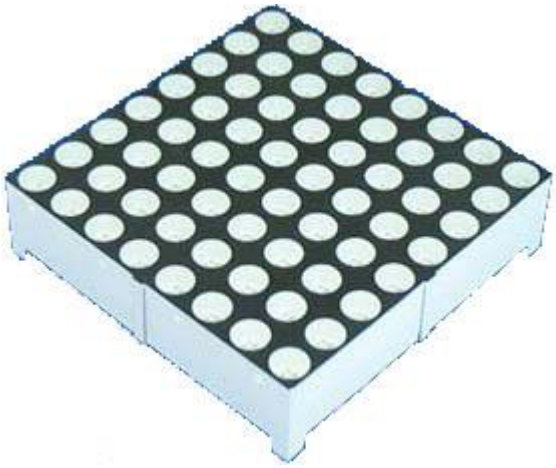

图7-1 8*8 LED 点阵外观

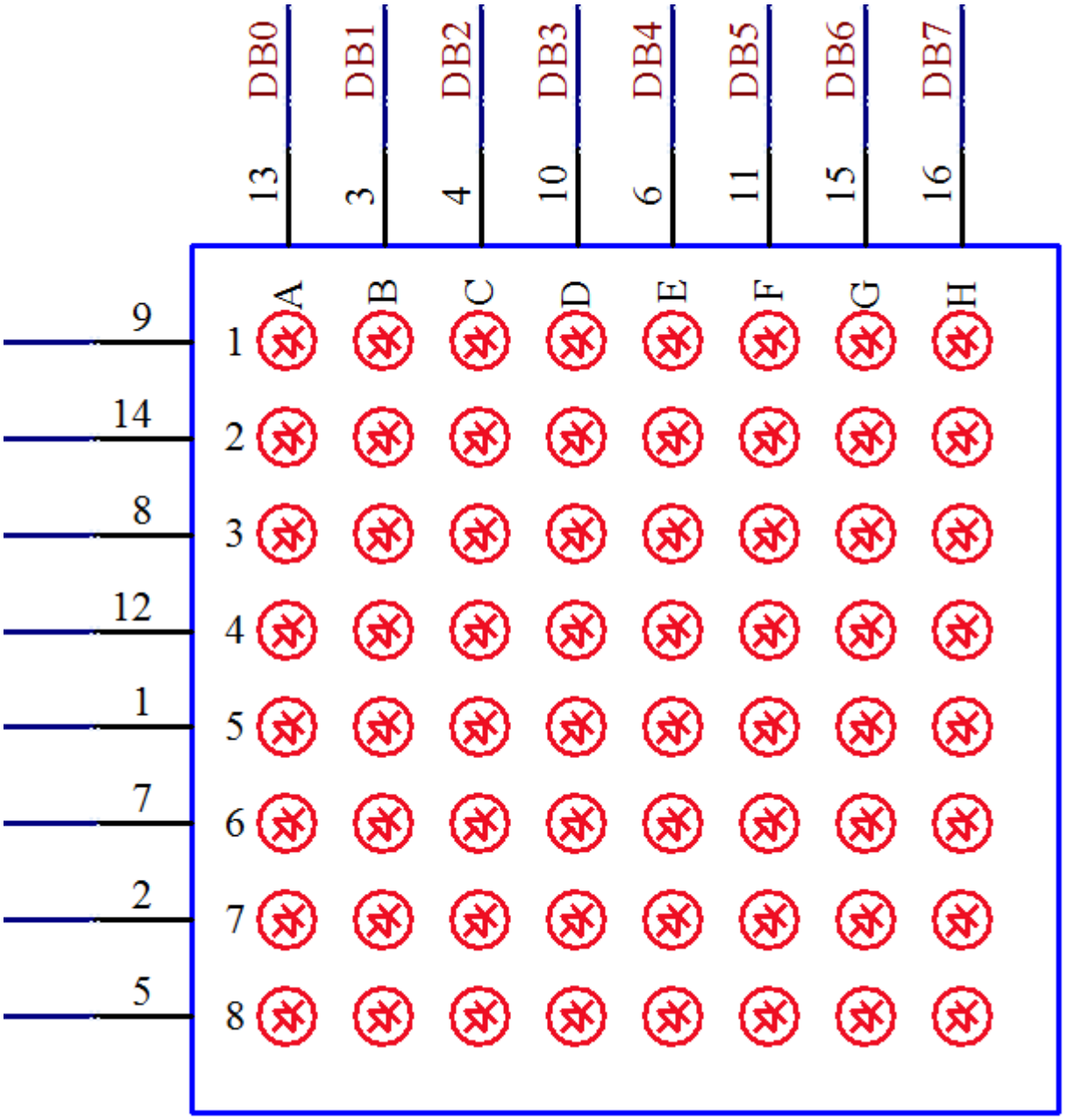

图7-2 8*8点阵结构原理图

从图7-2上可以看出，其实点阵 LED 点亮原理还是很简单的。在图中大方框外侧的就是点阵 LED 的引脚号，左侧的8个引脚是接的内部 LED 的阳极，上侧的8个引脚接的是内部 LED 的阴极。那么如果我们把9脚置成高电平、13脚置成低电平的话，左上角的那个 LED 小灯就会亮了。下面我们就用程序来实现一下，特别注意，控制点阵左侧引脚的 74HC138 是原理图上的 U4，8个引脚自上而下依次由 U4 的 Y0～Y7 输出来控制。

```
 #include <reg52.h>


sbit LED = P0^0;
sbit ADDR0 = P1^0;
```

```
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;

void main(){
    ENLED = 0;  //U3、U4 两片 74HC138 总使能
    ADDR3 = 0;  //使能 U4 使之正常输出
    ADDR2 = 0;  //经 U4 的 Y0 输出开启三极管 Q10
    ADDR1 = 0;
    ADDR0 = 0;
    LED = 0;  //向 P0.0 写入0来点亮左上角的一个点
    while(1);  //程序停止在这里
}
```

那么同样的方法，通过对 P0 的整体赋值我们可以一次点亮点阵的一行，那么这次我们用程序来点亮点阵的第二行，对应的就需要编号 U4 的 74HC138 在其 Y1 引脚输出低电平了。

```
 #include <reg52.h>



sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;

void main(){
    ENLED = 0;  //U3、U4 两片 74HC138 总使能
    ADDR3 = 0;  //使能 U4 使之正常输出
    ADDR2 = 0;  //经 U4 的 Y1 输出开启三极管 Q11
    ADDR1 = 0;
    ADDR0 = 1;
    P0 = 0x00;  //向 P0 写入0来点亮一行
    while(1);  //程序停止在这里
}
```

从这里我们可以逐步发现点阵的控制原理了。我们前面讲了一个数码管就是8个 LED 小灯，一个点阵是64个 LED 小灯。同样的道理，我们还可以把一个点阵理解成是8个数码管。经过前面的学习已经掌握了6个数码管同时显示的方法，那8个数码管也应该轻轻松松了。下面我们就利用定时器中断和数码管动态显示的原理来把这个点阵全部点亮。

```
 #include <reg52.h>
```

```
sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;

void main(){
    EA = 1;  //使能总中断
    ENLED = 0;  //使能 U4，选择 LED 点阵
    ADDR3 = 0;  //因为需要动态改变 ADDR0-2 的值，所以不需要再初始化了
    TMOD = 0x01; //设置 T0 为模式1
    TH0 = 0xFC; //为 T0 赋初值 0xFC67，定时 1 ms
    TL0 = 0x67;
    ET0 = 1;  //使能 T0 中断
    TR0 = 1;  //启动 T0
    while (1);  //程序停在这里，等待定时器中断
}
/* 定时器0中断服务函数 */
void InterruptTimer0() interrupt 1{
    static unsigned char i = 0; //动态扫描的索引

    TH0 = 0xFC;  //重新加载初值
    TL0 = 0x67;
    //以下代码完成 LED 点阵动态扫描刷新
    P0 = 0xFF;  //显示消隐
    switch (i){
        case 0: ADDR2=0; ADDR1=0; ADDR0=0; i++; P0=0x00; break;
        case 1: ADDR2=0; ADDR1=0; ADDR0=1; i++; P0=0x00; break;
        case 2: ADDR2=0; ADDR1=1; ADDR0=0; i++; P0=0x00; break;
        case 3: ADDR2=0; ADDR1=1; ADDR0=1; i++; P0=0x00; break;
        case 4: ADDR2=1; ADDR1=0; ADDR0=0; i++; P0=0x00; break;
        case 5: ADDR2=1; ADDR1=0; ADDR0=1; i++; P0=0x00; break;
        case 6: ADDR2=1; ADDR1=1; ADDR0=0; i++; P0=0x00; break;
        case 7: ADDR2=1; ADDR1=1; ADDR0=1; i=0; P0=0x00; break;
        default: break;
    }
}
```

7.4 单片机 LED 点阵的图形显示

独立的 LED 小灯可以实现流水灯，数码管可以显示多位数字，那点阵 LED 就得来显示一点花样了。

我们要显示花样的时候，往往要先做出来一些小图形，这些小图形的数据要转换到我们的程序当中去，这个时候就需要取模软件。给大家介绍一款简单的取模软件，这种取模软件在网上都可以下载到，大家来了解一下如何使用，先看一下操作界面，如图7-3所示。

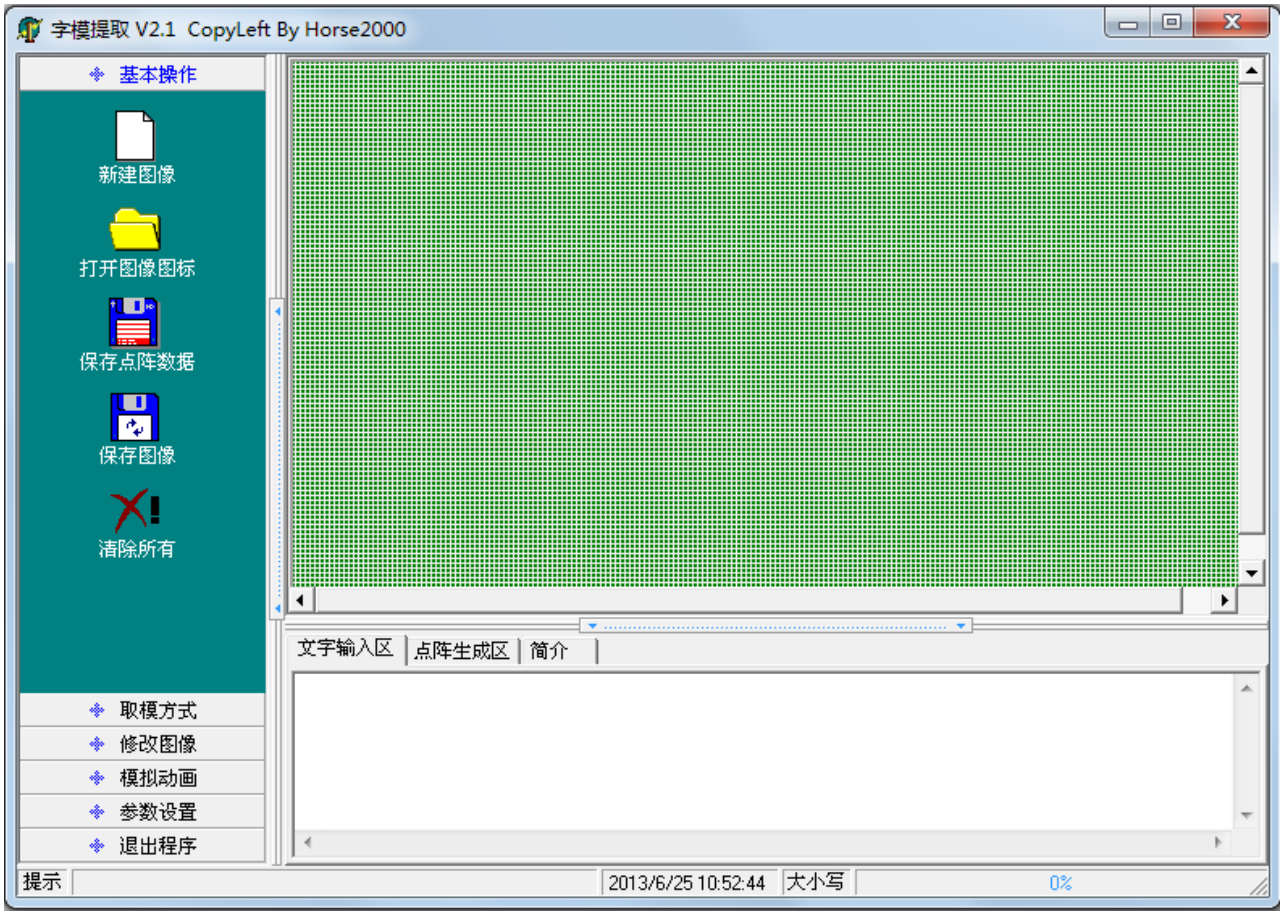

图7-3 字模提取软件界面

鼠标点一下“新建图形”，根据我们板子上的点阵，把宽度和高度分别改成8，然后点确定，如图7-4所示。

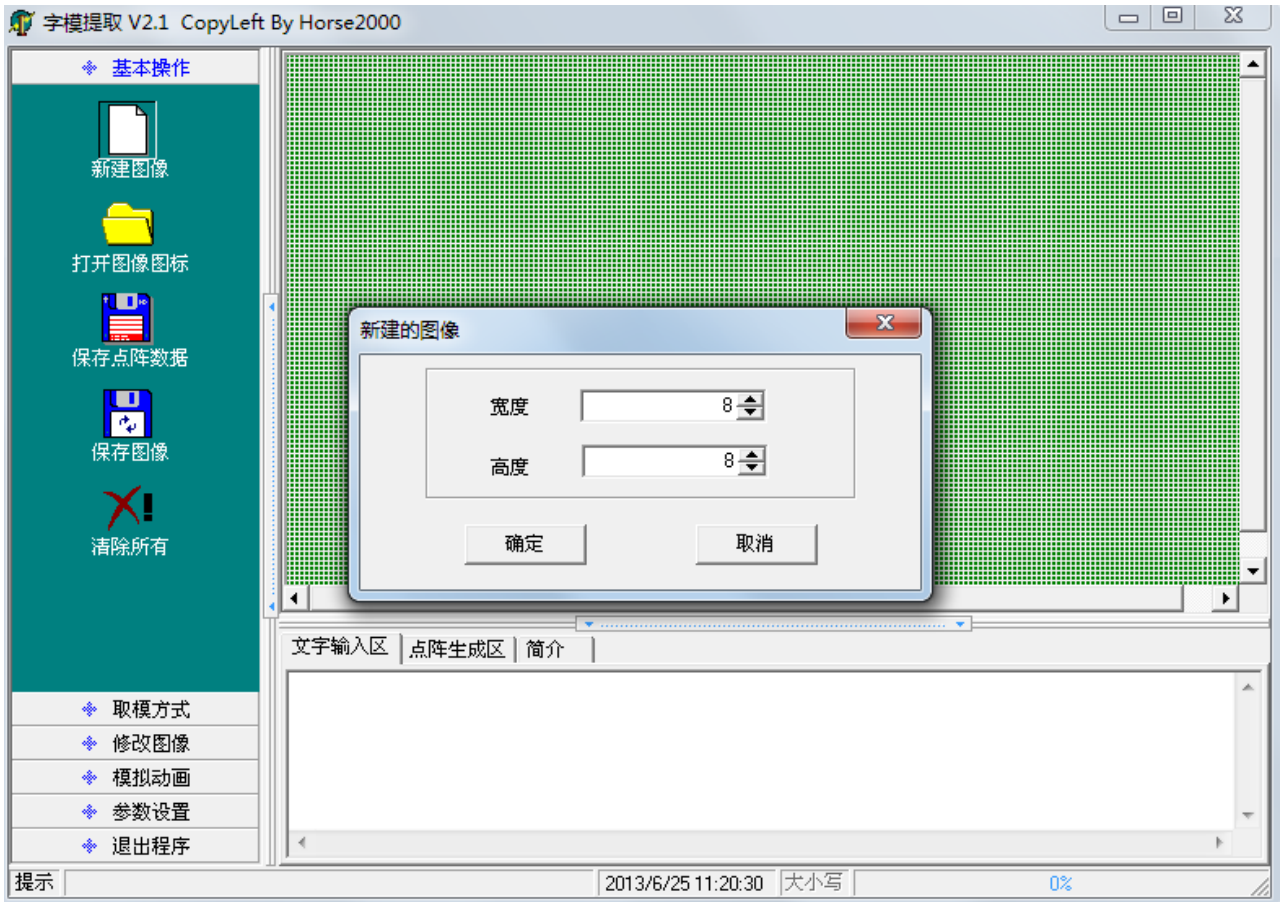

图7-4 新建图形

点击左侧的“模拟动画”菜单，再点击“放大格点”选项，一直放大到最大，那我们就可以在我们的8*8的点阵图形中用鼠标填充黑点，就可以画图形了，如图7-5所示。

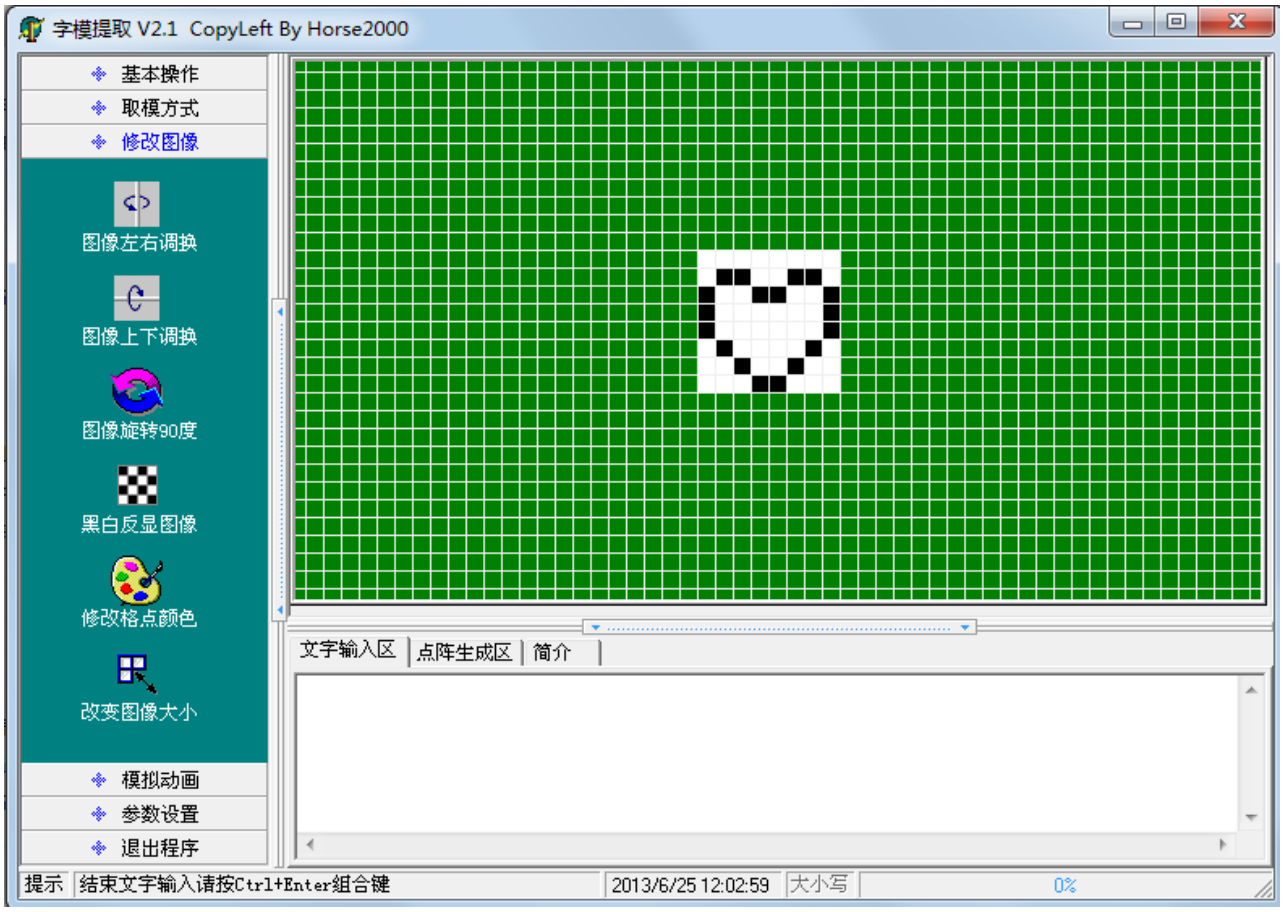

图7-5 字模提取软件画图

经过我们的一番精心设计，画出来一个心形图形，并且填充满，最终出现我们想要的效果图，如图7-6所示。

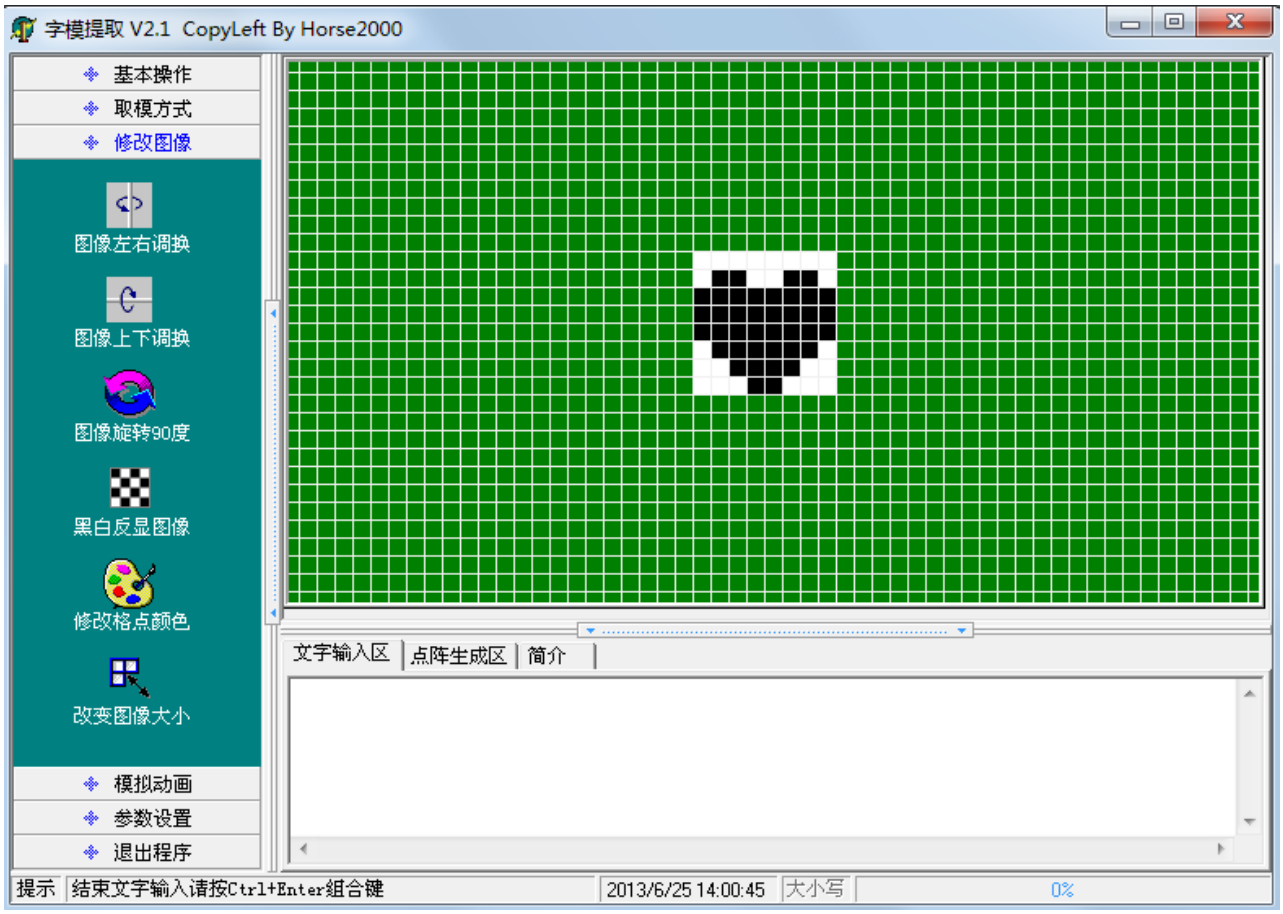

图7-6 心型图形

由于取模软件是把黑色取为1，白色取为0，但我们点阵是1对应 LED 熄灭，0对应 LED 点亮，而我们需要的是一颗点亮的“心”，所以我们要选“修改图像”菜单里的“黑白反显图像”这个选项，再点击“基本操作”菜单里边的“保存图像”可以把我们设计好的图片进行保存，如图7-7所示。

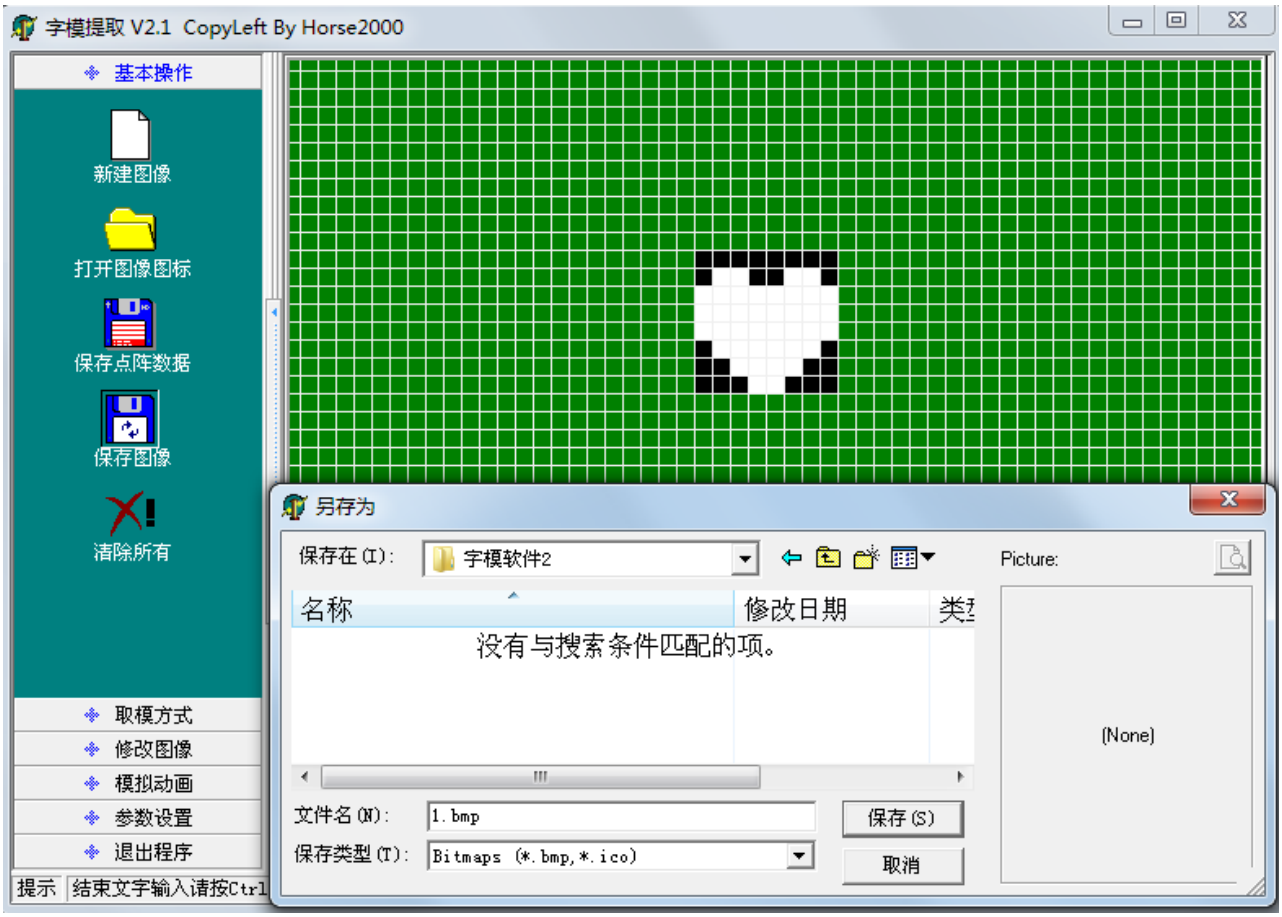

图7-7 保存图形

保存文件只是为了再次使用或修改使方便方便，当然你也可以不保存。操作完了这一步后，点击“参数设置”菜单里的“其他选项”，如图7-8所示。

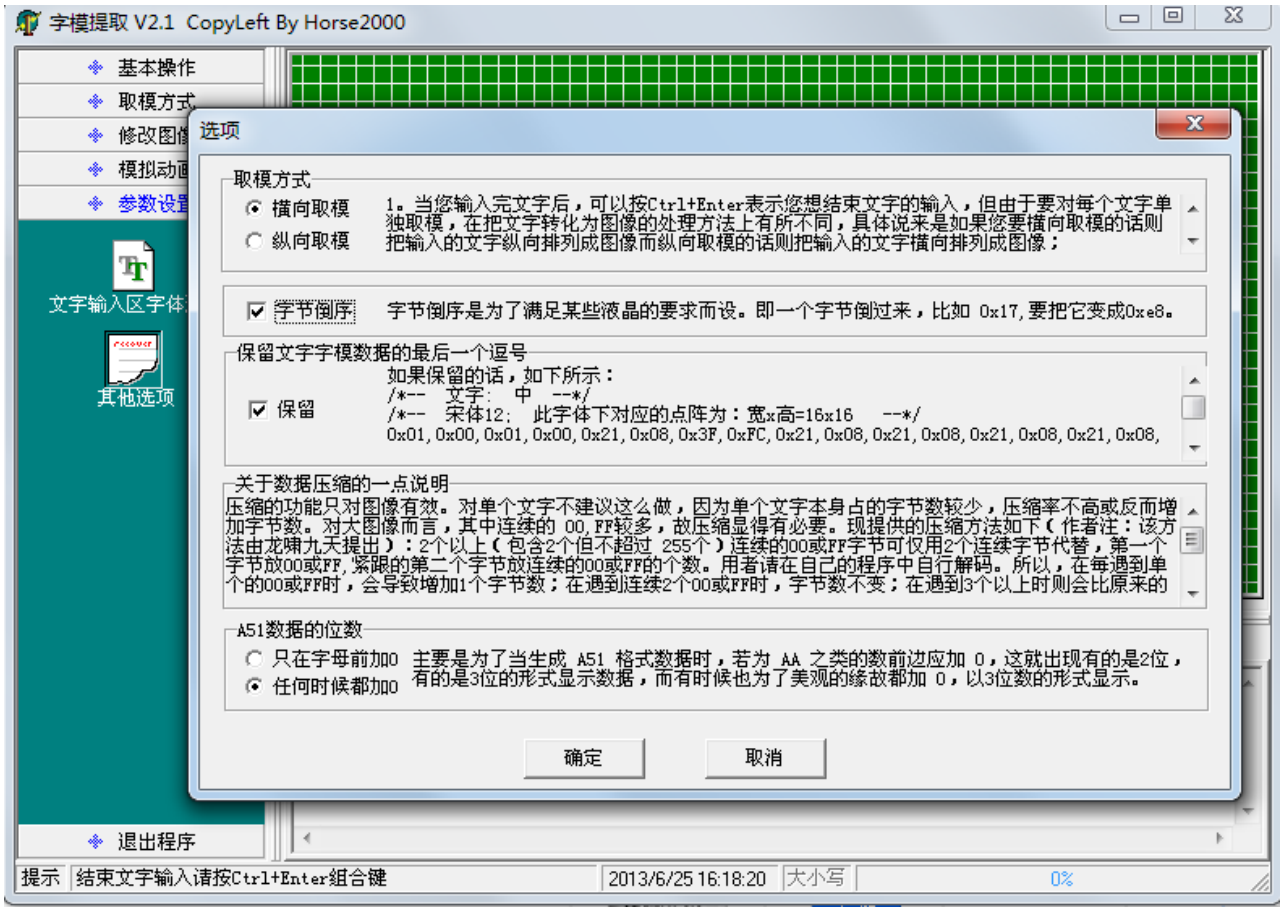

图7-8 选项设置

这里的选项，要结合图7-2来进行设置，大家可以看到 P0 口控制的是一行，所以用“横向取模”，如果控制的是一列，就要选“纵向取模”。选中“字节倒序”这个选项，是因为图7-2中左边是低位 DB0，右边是高位 DB7，所以是字节倒序，其它两个选项大家自己了解，点确定后，选择“取模方式”这个菜单，点一下“C51 格式”后，在“点阵生成区”自动产生了8个字节的数据，这8个字节的数据就是取出来的“模”，如图7-9所示。

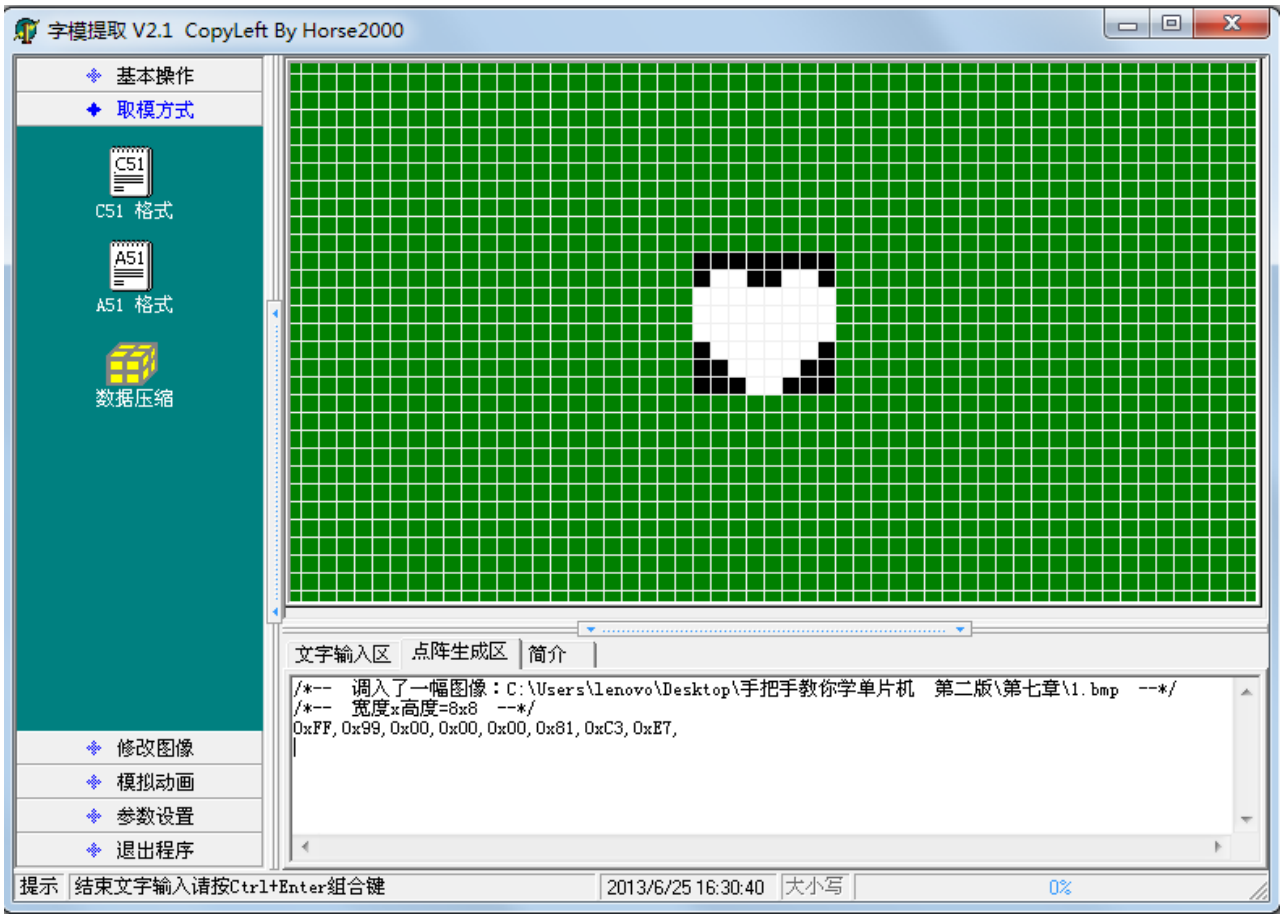

图7-9 取模结果

大家注意，虽然我们用了软件来取模，但是也得知道其原理是什么，在这个图片里，黑色的一个格子表示一位二进制的1，白色的一个格子表示一位二进制的0。第一个字节是 0xFF，其实就是这个8*8图形的第一行，全黑就是0xFF；第二个字节是 0x99，低位在左边，高位在右边，大家注意看，黑色的表示1，白色的表示0，就组成了 0x99 这个数值。同理其它的数据大家也就知道怎么来的了。

那么下面我们就用程序把这些数据依次送到点阵上去，看看运行效果如何。

```
#include <reg52.h>


sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;

unsigned char code image[] = { //图片的字模表
    0xFF, 0x99, 0x00, 0x00, 0x00, 0x81, 0xC3, 0xE7
};
```

```
void main(){
    EA = 1;  //使能总中断
    ENLED = 0;  //使能 U4，选择 LED 点阵
    ADDR3 = 0;
    TMOD = 0x01;  //设置 T0 为模式1
    TH0 = 0xFC;  //为 T0 赋初值 0xFC67，定时 1 ms
    TL0 = 0x67;
    ET0 = 1;  //使能 T0 中断
    TR0 = 1;//启动 T0
    while (1);
}
/* 定时器0中断服务函数 */
void InterruptTimer0() interrupt 1{
    static unsigned char i = 0;  //动态扫描的索引

    TH0 = 0xFC; //重新加载初值
    TL0 = 0x67;
    //以下代码完成 LED 点阵动态扫描刷新
    P0 = 0xFF;  //显示消隐
    switch (i){
        case 0: ADDR2=0; ADDR1=0; ADDR0=0; i++; P0=image[0]; break;
        case 1: ADDR2=0; ADDR1=0; ADDR0=1; i++; P0=image[1]; break;
        case 2: ADDR2=0; ADDR1=1; ADDR0=0; i++; P0=image[2]; break;
        case 3: ADDR2=0; ADDR1=1; ADDR0=1; i++; P0=image[3]; break;
        case 4: ADDR2=1; ADDR1=0; ADDR0=0; i++; P0=image[4]; break;
        case 5: ADDR2=1; ADDR1=0; ADDR0=1; i++; P0=image[5]; break;
        case 6: ADDR2=1; ADDR1=1; ADDR0=0; i++; P0=image[6]; break;
        case 7: ADDR2=1; ADDR1=1; ADDR0=1; i=0; P0=image[7]; break;
        default: break;
    }
}
```

对于8*8的点阵来说，我们可以显示一些简单的图形，字符等。但大部分汉字通常来说要用到16*16个点，而8*8的点阵只能显示一些简单笔画的汉字，大家可以自己取模做出来试试看。使用大屏显示汉字的方法和小屏的方法是类似的，所需要做的只是按照相同的原理来扩展行数和列数而已。

7.5 单片机 LED 点阵的纵向移动(动态显示)

点阵的动画显示，说到底就是对多张图片分别进行取模，使用程序算法巧妙的切换图片，多张图片组合起来就成了一段动画了，我们所看到的动画片、游戏等等，它们的基本原理也都是这样的。

上一节我们学了如何在点阵上画一个?形，有时候我们希望这些显示是动起来的，而不是静止的。对于点阵本身已经没有多少的知识点可以介绍了，主要就是编程算法来解决问题了。比如我们现在要让点阵显示一个 I ？U 的动画，首先我们要把这个图形用取模软件画出来看一下，如图7-10所示。

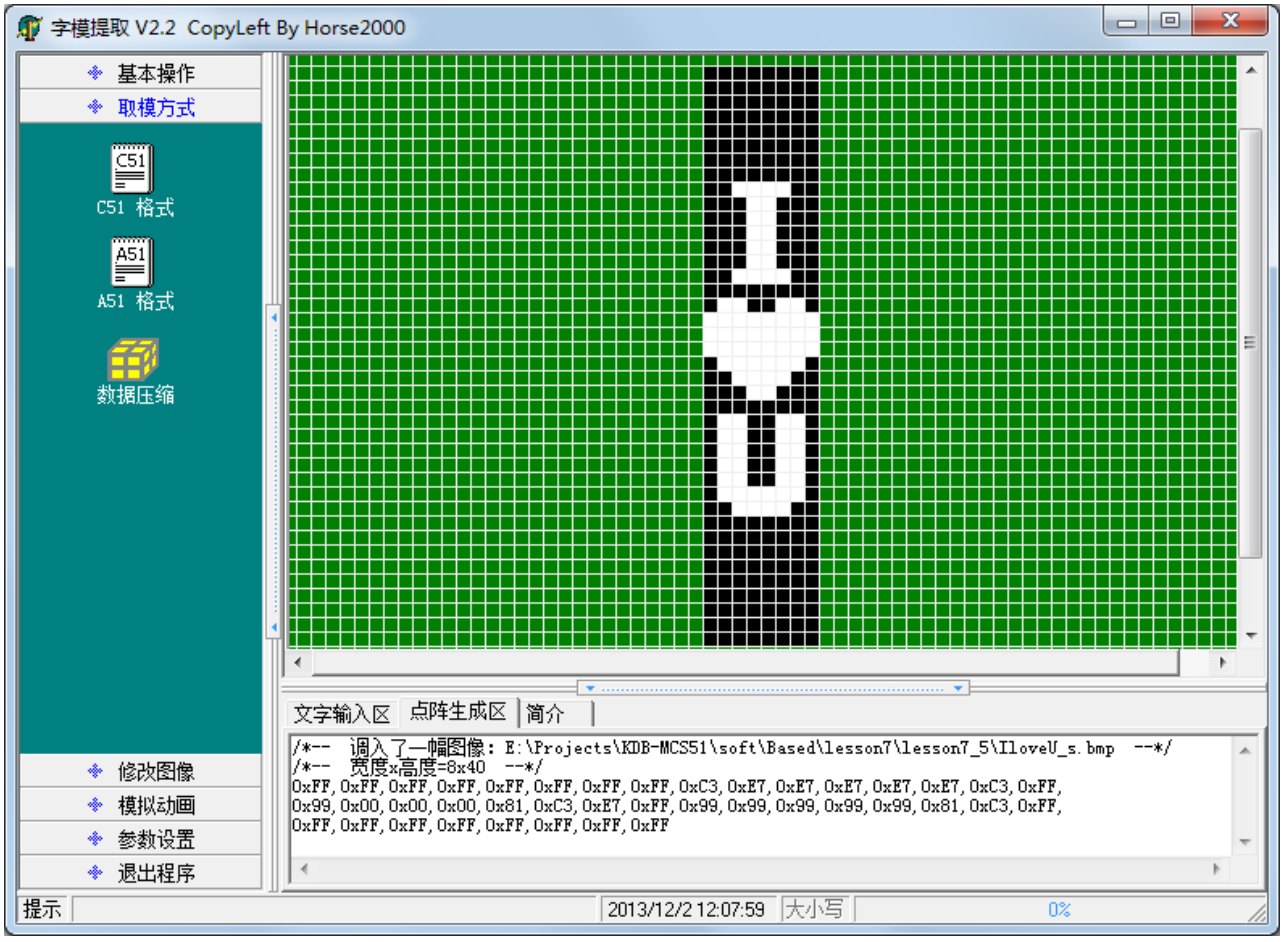

图7-10 上下移动横向取模

这张图片共有40行，每8行组成一张点阵图片，并且每向上移动一行就出现了一张新图片，一共组成了32张图片。

用一个变量 index 来代表每张图片的起始位置，每次从 index 起始向下数8行代表了当前的图片，250 ms 改变一张图片，然后不停的动态刷新，这样图片就变成动画了。首先我们要对显示的图片进行横向取模，虽然这是32张图片，由于我们每一张图片都是和下一行连续的，所以实际的取模值只需要40个字节就可以完成，我们来看看程序。

```
 #include <reg52.h>


sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;

unsigned char code image[] = { //图片的字模表
    0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
    0xC3,0xE7,0xE7,0xE7,0xE7,0xE7,0xC3,0xFF,
    0x99,0x00,0x00,0x00,0x81,0xC3,0xE7,0xFF,
    0x99,0x99,0x99,0x99,0x99,0x81,0xC3,0xFF,
    0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF
};

void main(){
    EA = 1;  //使能总中断
    ENLED = 0;  //使能 U4，选择 LED 点阵
    ADDR3 = 0;
    TMOD = 0x01; //设置 T0 为模式1
    TH0 = 0xFC; //为 T0 赋初值 0xFC67，定时 1 ms
    TL0 = 0x67;
    ET0 = 1;  //使能 T0 中断
    TR0 = 1;  //启动 T0
    while (1);
    }
    /* 定时器0中断服务函数 */
    void InterruptTimer0() interrupt 1{
    static unsigned char i = 0; //动态扫描的索引
    static unsigned char tmr = 0; //250 ms 软件定时器
    static unsigned char index = 0; //图片刷新索引

    TH0 = 0xFC; //重新加载初值
    TL0 = 0x67;
    //以下代码完成 LED 点阵动态扫描刷新
    P0 = 0xFF;  //显示消隐
    switch (i){
        case 0: ADDR2=0; ADDR1=0; ADDR0=0; i++; P0=image[index+0]; break;
        case 1: ADDR2=0; ADDR1=0; ADDR0=1; i++; P0=image[index+1]; break;
        case 2: ADDR2=0; ADDR1=1; ADDR0=0; i++; P0=image[index+2]; break;
        case 3: ADDR2=0; ADDR1=1; ADDR0=1; i++; P0=image[index+3]; break;
        case 4: ADDR2=1; ADDR1=0; ADDR0=0; i++; P0=image[index+4]; break;
        case 5: ADDR2=1; ADDR1=0; ADDR0=1; i++; P0=image[index+5]; break;
```

```
        case 6: ADDR2=1; ADDR1=1; ADDR0=0; i++; P0=image[index+6]; break;
        case 7: ADDR2=1; ADDR1=1; ADDR0=1; i=0; P0=image[index+7]; break;
        default: break;
    }
    //以下代码完成每 250 ms 改变一帧图像
    tmr++;
    if (tmr >= 250){  //达到 250 ms 时改变一次图片索引
        tmr = 0;
        index++;
        if (index >= 32){  //图片索引达到32后归零
           index = 0;
        }
    }
}
```

大家把这个程序下载到单片机上看看效果，一个 I ？U 一直往上走动的动画就出现了，现在还有哪位敢说我们工科同学不懂浪漫的？还需要用什么玫瑰花取悦女朋友吗？一点技术含量都没有，要玩就玩点高科技，呵呵。

当然，别光图开心，学习我们还要继续。往上走动的动画我写出来了，那往下走动的动画，大家就要自己独立完成了，不要偷懒，一定要去写代码调试代码。瞪眼看只能了解知识，而能力是在真正的写代码、调试代码这种实践中培养起来的。

7.6 单片机 LED 点阵的横向移动(动态显示)

上下移动我们会了，那我们还想左右移动该如何操作呢？

方法一、最简单，就是把板子侧过来放，纵向取模就可以完成。

这里大家是不是有种头顶冒汗的感觉？我们要做好技术，但是不能沉溺于技术。技术是我们的工具，我们在做开发的时候除了用好这个工具外，也得多拓展自己解决问题的思路，要慢慢培养自己的多角度思维方式。

那把板子正过来，左右移动就完不成了吗？当然不是。大家慢慢的学多了就会培养了一种感觉，就是一旦硬件设计好了，我们要完成一种功能，大脑就可以直接思考出来能否完成这个功能，这个在我们进行电路设计的时候最为重要。我们在开发产品的时候，首先是设计电路，设计电路的时候，工程师就要在大脑中通过思维来验证板子硬件和程序能否完成我们想要的功能，一旦硬件做好了，做好板子回来剩下的就是靠编程来完成了。只要是硬件逻辑上没问题，功能上软件肯定可以实现。

当然了，我们在进行硬件电路设计的时候，也得充分考虑软件编程的方便性。因为我们的程序是用 P0 来控制点阵的整行，所以对于我们这样的电路设计，上下移动程序是比较好编写的。那如果我们设计电路的时候知道我们的图形要左右移动，那我们设计电路画板子的时候就要尽可能的把点阵横过来放，有利于我们编程方便，减少软件工作量。

方法二、利用二维数组来实现，算法基本上和上下移动相似。

二维数组，前边提过一次，他的使用其实也没什么复杂的。它的声明方式是：

```
    数据类型 数组名[数组长度1][数组长度2];
```

与一位数组类似，数据类型是全体元素的数据类型，数组名是标识符，数组长度1和数组长度2分别代表数组具有的行数和列数。数组元素的下标一律从0开始。

例如：unsigned char a[2][3];声明了一个具有2行3列的无符号字符型的二维数组 a。

二维数组的数组元素总个数是两个长度的乘积。二维数组在内存中存储的时候，采用行优先的方式来存储，即在内存中先存放第0行的元素，再存放第一行的元素.....，同一行中再按照列顺序存放，刚才定义的那个a[2][3]的存放形式就如表7-1所示。

表7-1 二维数组的物理存储结构

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]

二维数组的初始化方法分两种情况，我们前边学一维数组的时候学过，数组元素的数量可以小于数组元素个数，没有赋值的会自动给0。当数组元素的数量等于数组个数的时候，如下所示：

```
    unsigned char a[2][3] = {{1,2,3}, {4,5,6}};
```

或者是

```
    unsigned char a[2][3] = {1,2,3,4,5,6};
```

当数组元素的数量小于数组个数的时候，如下所示：

```
    unsigned char a[2][3] = {{1,2}, {3,4}};
```

等价于

```
    unsigned char a[2][3] = {1,2,0,3,4,0};
```

而反过来的写法

```
    unsigned char a[2][3] = {1,2,3,4};
```

等价于

```
    unsigned char a[2][3] = {{1,2,3}, {4,0,0}};
```

此外，二维数组初始化的时候，行数可以省略，编译系统会自动根据列数计算出行数，但是列数不能省略。

讲这些，只是为了让大家了解一下，看别人写的代码的时候别发懵就行了，但是我们今后写程序的时候，按照规范，行数列数都不要省略，全部写齐，初始化的时候，全部写成unsigned char a[2][3] = {{1,2,3}, {4,5,6}};的形式，而不允许写成一维数组的格式，防止大家出错，同时也是提高程序的可读性。

那么下面我们要进行横向做 I ？U 的动画了，先把我们需要的图片画出来，再逐一取模，和上一张图片类似的是，我们这个图形共有30张图片，通过程序每 250 ms 改变一张图片，就可以做出来动画效果了。但是不同的是，我们这个是要横向移动，横向移动的图片切换时的字模数据不是连续的，所以这次我们要对30张图片分别取模，如图7-11所示。

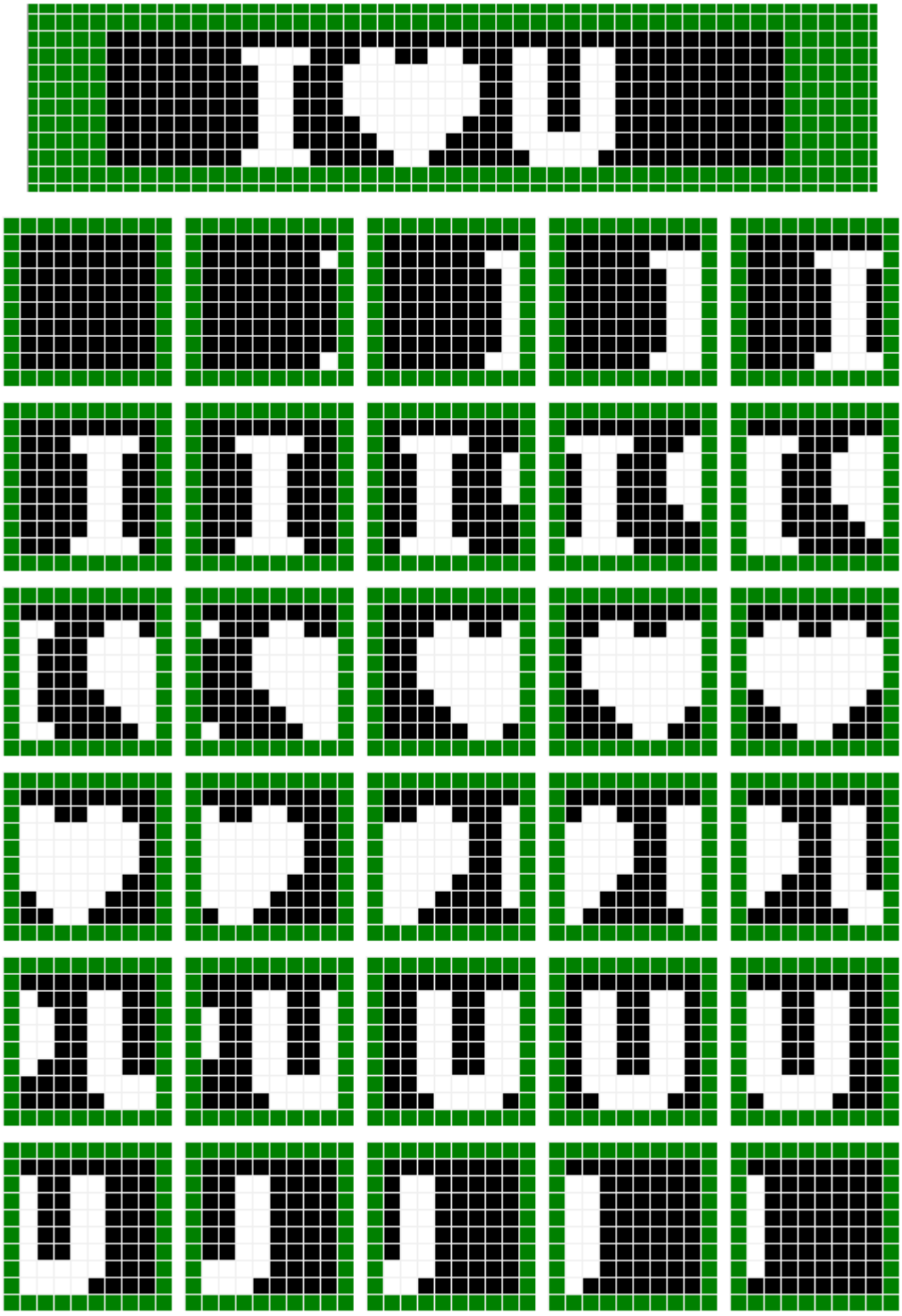

图7-11 横向动画取模图片

图7-11中最上面的图形是横向连在一起的效果，而实际上我们要把它分解为30个帧，每帧图片单独取模，取出来都是8个字节的数据，一共就是30*8个数据，我们用一个二维数组来存储它们。

```
 #include <reg52.h>


sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;

unsigned char code image[30][8] = {
    {0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF}, //动画帧1
    {0xFF,0x7F,0xFF,0xFF,0xFF,0xFF,0xFF,0x7F}, //动画帧2
    {0xFF,0x3F,0x7F,0x7F,0x7F,0x7F,0x7F,0x3F}, //动画帧3
    {0xFF,0x1F,0x3F,0x3F,0x3F,0x3F,0x3F,0x1F}, //动画帧4
    {0xFF,0x0F,0x9F,0x9F,0x9F,0x9F,0x9F,0x0F}, //动画帧5
    {0xFF,0x87,0xCF,0xCF,0xCF,0xCF,0xCF,0x87}, //动画帧6
    {0xFF,0xC3,0xE7,0xE7,0xE7,0xE7,0xE7,0xC3}, //动画帧7
    {0xFF,0xE1,0x73,0x73,0x73,0xF3,0xF3,0xE1}, //动画帧8
    {0xFF,0x70,0x39,0x39,0x39,0x79,0xF9,0xF0}, //动画帧9
    {0xFF,0x38,0x1C,0x1C,0x1C,0x3C,0x7C,0xF8}, //动画帧10
    {0xFF,0x9C,0x0E,0x0E,0x0E,0x1E,0x3E,0x7C}, //动画帧11
    {0xFF,0xCE,0x07,0x07,0x07,0x0F,0x1F,0x3E}, //动画帧12
    {0xFF,0x67,0x03,0x03,0x03,0x07,0x0F,0x9F}, //动画帧13
    {0xFF,0x33,0x01,0x01,0x01,0x03,0x87,0xCF}, //动画帧14
    {0xFF,0x99,0x00,0x00,0x00,0x81,0xC3,0xE7}, //动画帧15
    {0xFF,0xCC,0x80,0x80,0x80,0xC0,0xE1,0xF3}, //动画帧16
    {0xFF,0xE6,0xC0,0xC0,0xC0,0xE0,0xF0,0xF9}, //动画帧17
    {0xFF,0x73,0x60,0x60,0x60,0x70,0x78,0xFC}, //动画帧18
    {0xFF,0x39,0x30,0x30,0x30,0x38,0x3C,0x7E}, //动画帧19
    {0xFF,0x9C,0x98,0x98,0x98,0x9C,0x1E,0x3F}, //动画帧20
    {0xFF,0xCE,0xCC,0xCC,0xCC,0xCE,0x0F,0x1F}, //动画帧21
    {0xFF,0x67,0x66,0x66,0x66,0x67,0x07,0x0F}, //动画帧22
    {0xFF,0x33,0x33,0x33,0x33,0x33,0x03,0x87}, //动画帧23
    {0xFF,0x99,0x99,0x99,0x99,0x99,0x81,0xC3}, //动画帧24
    {0xFF,0xCC,0xCC,0xCC,0xCC,0xCC,0xC0,0xE1}, //动画帧25
    {0xFF,0xE6,0xE6,0xE6,0xE6,0xE6,0xE0,0xF0}, //动画帧26
    {0xFF,0xF3,0xF3,0xF3,0xF3,0xF3,0xF0,0xF8}, //动画帧27
    {0xFF,0xF9,0xF9,0xF9,0xF9,0xF9,0xF8,0xFC}, //动画帧28
    {0xFF,0xFC,0xFC,0xFC,0xFC,0xFC,0xFC,0xFE}, //动画帧29
    {0xFF,0xFE,0xFE,0xFE,0xFE,0xFE,0xFE,0xFF} //动画帧30
};
```

```
void main(){
    EA = 1;  //使能总中断
    ENLED = 0;  //使能 U4，选择 LED 点阵
    ADDR3 = 0;
    TMOD = 0x01; //设置 T0 为模式1
    TH0 = 0xFC; //为 T0 赋初值 0xFC67，定时 1 ms
    TL0 = 0x67;
    ET0 = 1;  //使能 T0 中断
    TR0 = 1;  //启动 T0
    while (1);
}
/* 定时器0中断服务函数 */
void InterruptTimer0() interrupt 1{
    static unsigned char i = 0; //动态扫描的索引
    static unsigned char tmr = 0; //250 ms 软件定时器
    static unsigned char index = 0; //图片刷新索引
    TH0 = 0xFC; //重新加载初值
    TL0 = 0x67;
    //以下代码完成 LED 点阵动态扫描刷新
    P0 = 0xFF;  //显示消隐

    switch (i){
        case 0: ADDR2=0; ADDR1=0; ADDR0=0; i++; P0=image[index][0]; break;
        case 1: ADDR2=0; ADDR1=0; ADDR0=1; i++; P0=image[index][1]; break;
        case 2: ADDR2=0; ADDR1=1; ADDR0=0; i++; P0=image[index][2]; break;
        case 3: ADDR2=0; ADDR1=1; ADDR0=1; i++; P0=image[index][3]; break;
        case 4: ADDR2=1; ADDR1=0; ADDR0=0; i++; P0=image[index][4]; break;
        case 5: ADDR2=1; ADDR1=0; ADDR0=1; i++; P0=image[index][5]; break;
        case 6: ADDR2=1; ADDR1=1; ADDR0=0; i++; P0=image[index][6]; break;
        case 7: ADDR2=1; ADDR1=1; ADDR0=1; i=0; P0=image[index][7]; break;
        default: break;
    }
    //以下代码完成每 250 ms 改变一帧图像
    tmr++;
    if (tmr >= 250){ //达到 250 ms 时改变一次图片索引
        tmr = 0;
        index++;
        if (index >= 30){ //图片索引达到30后归零
           index = 0;
        }
    }
}
```

下载进到板子上瞧瞧，是不是有一种帅到掉渣的感觉呢。技术这东西，外行人看的是很神秘的，其实我们做出来会发现，也就是那么回事而已，每 250 ms 更改一张图片，每 1 ms在定时器中断里刷新单张图片的某一行。

不管是上下移动还是左右移动，大家要建立一种概念，就是我们是对一帧帧的图片的切换，这种切换带给我们的视觉效果就是一种动态的了。比如我们的 DV 拍摄动画，实际上就是快速的拍摄了一帧帧的图片，然后对这些图片的快速回放，把动画效果给显示了出来。因为我们硬件设计的缘故，所以在写上下移动程序的时候，数组定义的元素比较少，但是实际上大家也得理解成是32张图片的切换显示，而并非是真正的“移动”。

2

8. C 语言函数进阶与单片机按键

用户与单片机之间的信息交互需要依赖于两类设备：输入设备和输出设备。前边讲的 LED 小灯、数码管、点阵都是输出设备，本章我们就来学习一下最常用的输入设备——按键，同时还会学到一些硬件电路的基础知识与 C 语言函数的一些进阶知识。

8.1 单片机最小系统解析(电源、晶振和复位电路)

电源

我们在学习过程中，很多指标都是直接用的概念指标，比如我们说 +5 V 代表1，GND 代表0等等。但在实际电路中的电压值并不是完全精准的，那这些指标允许范围是什么呢？随着我们所学的内容不断增多，大家要慢慢培养一种阅读数据手册的能力。

比如，我们要使用 STC89C52RC 单片机的时候，找到它的数据手册第11页，看第二项——工作电压：5.5 V～3.4 V（5 V 单片机），这个地方就说明这个单片机正常的工作电压是个范围值，只要电源 VCC 在 5.5 V～3.4 V 之间都可以正常工作，电压超过 5.5 V 是绝对不允许的，会烧坏单片机，电压如果低于 3.4 V，单片机不会损坏，但是也不能正常工作。而在这个范围内，最典型、最常用的电压值就是 5V，这就是后面括号里“5 V 单片机”这个名称的由来。除此之外，还有一种常用的工作电压范围是 2.7 V～3.6 V、典型值是 3.3 V 的单片机，也就是所谓的“3.3 V 单片机”。日后随着大家接触更多的器件，对这点会有更深刻的理解。

现在我们再顺便多了解一点，大家打开 74HC138 的数据手册，会发现 74HC138 手册的第二页也有一个表格，上边写了 74HC138 的工作电压范围，最小值是 4.75 V，额定值是 5 V，最大值是 5.25 V，可以得知它的工作电压范围是 4.75 V～5.25 V。这个地方讲这些目的是让大家清楚的了解，我们获取器件工作参数的一个最重要、也是最权威的途径，就是查阅该器件的数据手册。

晶振

晶振通常分为无源晶振和有源晶振两种类型，无源晶振一般称之为 crystal（晶体），而有源晶振则叫做 oscillator（振荡器）。

有源晶振是一个完整的谐振振荡器，它是利用石英晶体的压电效应来起振，所以有源晶振需要供电，当我们把有源晶振电路做好后，不需要外接其它器件，只要给它供电，它就可以主动产生振荡频率，并且可以提供高精度的频率基准，信号质量也比无源信号要好。

无源晶振自身无法振荡起来，它需要芯片内部的振荡电路一起工作才能振荡，它允许不同的电压，但是信号质量和精度较有源晶振差一些。相对价格来说，无源晶振要比有源晶振价格便宜很多。无源晶振两侧通常都会有个电容，一般其容值都选在 10 pF~40 pF 之间，如果手册中有具体电容大小的要求则要根据要求来选电容，如果手册没有要求，我们用 20 pF 就是比较好的选择，这是一个长久以来的经验值，具有极其普遍的适用性。

我们来认识下比较常用的两种晶振的样貌，如图8-1和图8-2所示。

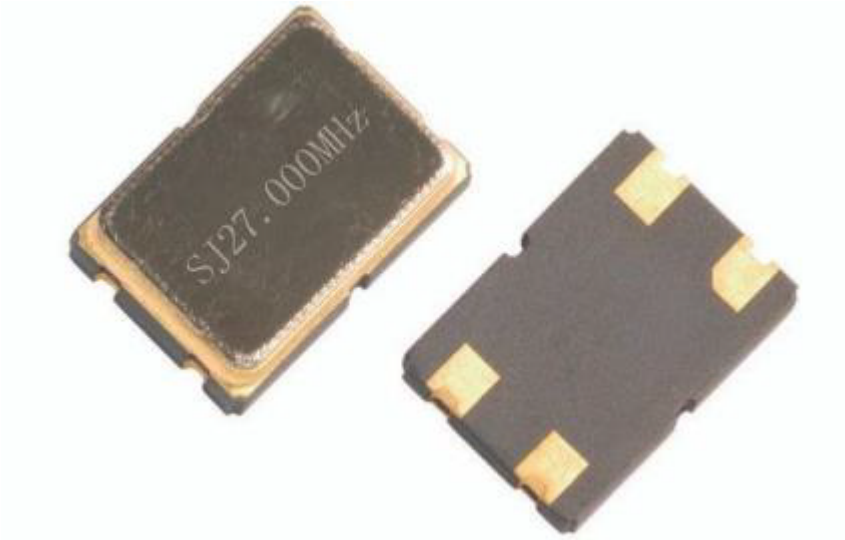

图8-1 有源晶振实物图

图8-2 无源晶振实物图

有源晶振通常有4个引脚，VCC，GND，晶振输出引脚和一个没有用到的悬空引脚（有些晶振也把该引脚作为使能引脚）。无源晶振有2个或3个引脚，如果是3个引脚的话，中间引脚接是晶振的外壳，使用时要接到 GND，两侧的引脚就是晶体的2个引出脚了，这两个引脚作用是等同的，就像是电阻的2个引脚一样，没有正负之分。对于无源晶振，用我们的单片机上的两个晶振引脚接上去即可，而有源晶振，只接到单片机的晶振的输入引脚上，输出引脚上不需要接，如图8-3和图8-4所示。

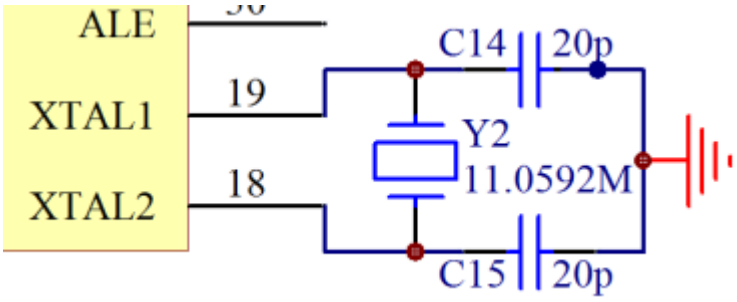

图8-3 无源晶振接法

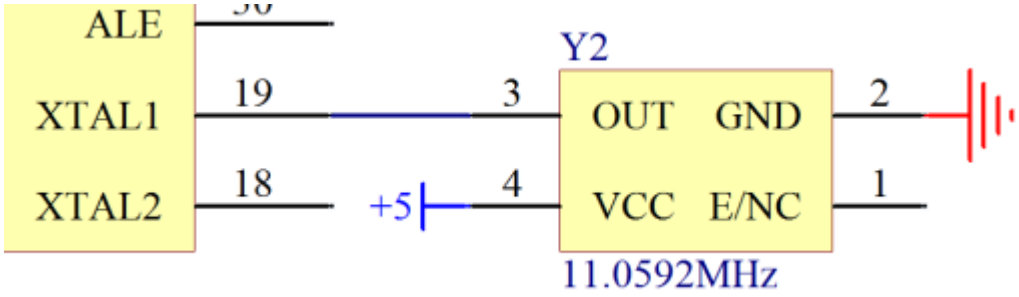

图8-4 有源晶振接法

复位电路

我们先来分析一下 KST-51 开发板上的复位电路，如图8-5所示。

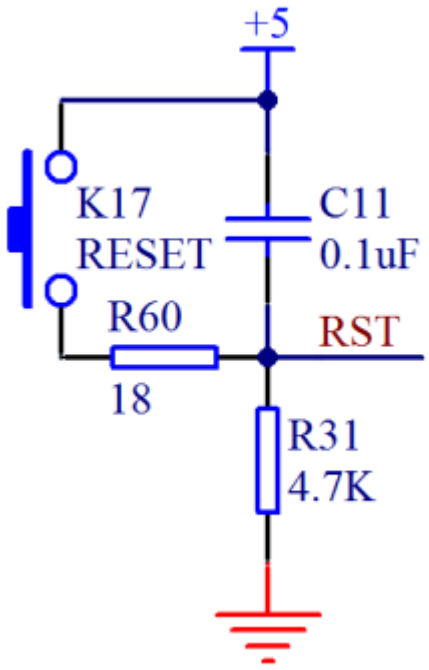

图8-5 单片机复位电路

当这个电路处于稳态时，电容起到隔离直流的作用，隔离了 +5 V，而左侧的复位按键是弹起状态，下边部分电路就没有电压差的产生，所以按键和电容 C11 以下部分的电位都是和 GND 相等的，也就是 0 V。我们这个单片机是高电平复位，低电平正常工作，所以正常工作的电压是 0 V，没有问题。

我们再来分析从没有电到上电的瞬间，电容 C11 上方电压是 5 V，下方是 0 V，根据我们初中所学的知识，电容 C11 要进行充电，正离子从上往下充电，负电子从 GND 往上充电，这个时候电容对电路来说相当于一根导线，全部电压都加在了 R31 这个电阻上，那么 RST端口位置的电压就是 5 V，随着电容充电越来越多，即将充满的时

候，电流会越来越小，那 RST 端口上的电压值等于电流乘以 R31 的阻值，也就会越来越小，一直到电容完全充满后，线路上不再有电流，这个时候 RST 和 GND 的电位就相等了也就是 0 V 了。

从这个过程上来看，我们加上这个电路，单片机系统上电后，RST 引脚会先保持一小段时间的高电平而后变成低电平，这个过程就是上电复位的过程。那这个“一小段时间”到底是多少才合适呢？每种单片机不完全一样，51单片机手册里写的是持续时间不少于2个机器周期的时间。复位电压值，每种单片机不完全一样，我们按照通常值0.7 VCC 作为复位电压值，复位时间的计算过程比较复杂，我这里只给大家一个结论，时间 t=1.2 RC，我们用的R 是4700欧，C 是0.0000001法，那么计算出 t 就是 0.000564秒，即 564 us，远远大于2个机器周期(2 us)，在电路设计的时候一般留够余量就行。

按键复位（即手动复位）有2个过程，按下按键之前，RST 的电压是 0 V，当按下按键后电路导通，同时电容也会在瞬间进行放电，RST 电压值变化为 4700 VCC/(4700+18)，会处于高电平复位状态。当松开按键后就和上电复位类似了，先是电容充电，后电流逐渐减小直到 RST 电压变 0 V 的过程。我们按下按键的时间通常都会有几百毫秒，这个时间足够复位了。

按下按键的瞬间，电容两端的 5 V 电压（注意不是电源的 5 V 和 GND 之间）会被直接接通，此刻会有一个瞬间的大电流冲击，会在局部范围内产生电磁干扰，为了抑制这个大电流所引起的干扰，我们这里在电容放电回路中串入一个18欧的电阻来限流。

如果有的同学已经想开始 DIY 设计自己的电路板，那单片机最小系统的设计现在已经有了足够的理论依据了，可以考虑尝试了。基础比较薄弱的同学先不要着急，继续跟着往下学，把课程都学完了再动手操作也不迟，磨刀不误砍柴工。

8.2 C 语言函数的调用

在一个程序的编写过程中，随着代码量的增加，如果把所有的语句都写到 main 函数中，一方面程序会显得的比较乱，另外一个方面，当同一个功能需要在不同地方执行时，我们就得再重复写一遍相同的语句。此时，如果把一些零碎的功能单独写成一个函数，在需要它们时只需进行一些简单的函数调用，这样既有助于程序结构的清晰条理，又可以避免大块的代码重复。

在实际工程项目中，一个程序通常都是由很多个子程序模块组成的，一个模块实现一个特定的功能，在 C 语言中，这个模块就用函数来表示。一个 C 程序一般由一个主函数和若干个其他函数构成。主函数可以调用其它函数，其它函数也可以相互调用，但其它函数不能调用主函数。在我们的51单片机程序中，还有中断服务函数，是当相应的中断到来后自动调用的，不需要也不能由其它函数来调用。

函数调用的一般形式是：

```
    函数名 (实参列表);
```

函数名就是需要调用的函数的名称，实参列表就是根据实际需求调用函数要传递给被调用函数的参数列表，不需要传递参数时只保留括号就可以了，传递多个参数时参数之间要用逗号隔开。

那么我先举例看一下函数调用使程序结构更加条理清晰方面的作用。回顾一下图6-1所示的程序流程图和为实现它而编写的程序代码，相对来说这个主函数的结构就比较复杂了，

很难一眼看清楚它的执行流程。那么如果我们把其中最重要的两件事——秒计数和数码管动态扫描功能都用单独的函数来实现会怎样呢？来看程序。

```
 #include <reg52.h>


sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;

unsigned char code LedChar[] = {  //数码管显示字符转换表
    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
    0x80, 0x90, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E
};
unsigned char LedBuff[6] = {  //数码管显示缓冲区，初值 0xFF 确保启动时都不亮
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
};
```

```
void SecondCount();
void LedRefresh();

void main(){
    ENLED = 0;  //使能 U3，选择控制数码管
    ADDR3 = 1;  //因为需要动态改变 ADDR0-2 的值，所以不需要再初始化了
    TMOD = 0x01;  //设置 T0 为模式1
    TH0 = 0xFC;  //为 T0 赋初值 0xFC67，定时 1 ms
    TL0 = 0x67;
    TR0 = 1;  //启动 T0
    while (1){
        if (TF0 == 1){  //判断 T0 是否溢出
            TF0 = 0;  //T0 溢出后，清零中断标志
            TH0 = 0xFC;  //并重新赋初值
            TL0 = 0x67;
            SecondCount();  //调用秒计数函数
            LedRefresh();  //调用显示刷新函数
        }
    }
}
/* 秒计数函数，每秒进行一次秒数+1，并转换为数码管显示字符 */
void SecondCount(){
    static unsigned int cnt = 0;  //记录 T0 中断次数
    static unsigned long sec = 0;  //记录经过的秒数
    cnt++;  //计数值自加 1
    if (cnt >= 1000){  //判断 T0 溢出是否达到1000次
        cnt = 0;  //达到1000次后计数值清零
        sec++;  //秒计数自加1

        LedBuff[0] = LedChar[sec%10];
        LedBuff[1] = LedChar[sec/10%10];
        LedBuff[2] = LedChar[sec/100%10];
        LedBuff[3] = LedChar[sec/1000%10];
        LedBuff[4] = LedChar[sec/10000%10];
        LedBuff[5] = LedChar[sec/100000%10];
    }
}
/* 数码管动态扫描刷新函数 */
void LedRefresh(){
    static unsigned char i = 0; //动态扫描的索引

    switch (i){
        case 0: ADDR2=0; ADDR1=0; ADDR0=0; i++; P0=LedBuff[0]; break;
        case 1: ADDR2=0; ADDR1=0; ADDR0=1; i++; P0=LedBuff[1]; break;
```

```
        case 2: ADDR2=0; ADDR1=1; ADDR0=0; i++; P0=LedBuff[2]; break;
        case 3: ADDR2=0; ADDR1=1; ADDR0=1; i++; P0=LedBuff[3]; break;
        case 4: ADDR2=1; ADDR1=0; ADDR0=0; i++; P0=LedBuff[4]; break;
        case 5: ADDR2=1; ADDR1=0; ADDR0=1; i=0; P0=LedBuff[5]; break;
        default: break;
    }
}
```

看一下，主函数的结构是不是清晰的多了——每隔 1 ms 就去干两件事，至于这两件事是什么交由各自的函数去实现。还请大家注意一点：原来程序中的 i、cnt、sec 这三个变量在放到单独的函数中后，都加了 static 关键字而变成了静态变量。因为原来的 main()永远不会结束所以它们的值也总是得到保持的，但现在它们在各自的功能函数内，如不加 static 修饰那么每次函数被调用时它们的值就都成了初值了，借此也把静态变量再加深一下理解吧。

当然，这是我们刻意把程序功能做了这样的划分，主要目的还是来讲解函数的调用，对于这个程序即使你不划分函数也复杂不到哪里去，但继续学下去你就能领会到划分功能函数的必要了。现在我们还是把注意力放在学习函数调用上，有以下几点需要大家注意：

1) 函数调用的时候，不需要加函数类型。我们在主函数内调用 SecondCount()和 LedRefresh()时都没有加 void。

2) 调用函数与被调用函数的位置关系，C 语言规定：函数在被调用之前，必须先被定义或声明。意思就是说：在一个文件中，一个函数应该先定义，然后才能被调用，也就是调用函数应位于被调用函数的下方。但是作为一种通常的编程规范，我们推荐 main 函数写在最前面（因为它起到提纲挈领的作用），其后再定义各个功能函数，而中断函数则写在文件的最后。那么主函数要调用定义在它之后的函数怎么办呢？我们就在文件开头，所有函数定义之前，开辟一块区域，叫做函数声明区，用来把被调用的函数声明一下，如此，该函数就可以被随意调用了。如上述例程所示。

3) 函数声明的时候必须加函数类型，函数的形式参数，最后加上一个分号表示结束。函数声明行与函数定义行的唯一区别就是最后的分号，其它的都必须保持一致。这点请尤其注意，初学者很容易因粗心大意而搞错分号或是修改了定义行中的形参却忘了修改声明行中的形参，导致程序编译不过。

8.3 C 语言函数的形参和实参

上一个例程中在进行函数调用的时候，不需要任何参数传递，所以函数定义和调用时括号内都是空的，但是更多的时候我们需要在主调函数和被调用函数之间传递参数。在调用一个有参数的函数时，函数名后边括号中的参数叫做实际参数，简称 `实参` 。而被调用的函数在进行定义时，括号里的参数叫做形式参数，简称 `形参` 。我们用个简单程序例子做说明。

```
unsigned char add(unsigned char x, unsigned char y);  //函数声明
void main(){
    unsigned char a = 1;
    unsigned char b = 2;
    unsigned char c = 0;
    //调用时，a 和 b 就是实参，把函数的返回值赋给 c
    //执行完后，c 的值就是 3
    c = add(a, b);
    while(1);
}
//函数定义，这里括号中的 x 和 y 就是形参
unsigned char add(unsigned char x, unsigned char y){
    unsigned char z = 0;
    z = x + y;
    return z;  //返回值 z 的类型就是函数 add 的类型
}
```

这个演示程序虽然很简单，但是函数调用的全部内容都囊括在内了。主调函数 main 和被调用函数 add 之间的数据通过形参和实参发生了传递关系，而函数运算完后把值传递给了变量 c，函数只要不是 void 类型，就都会有返回值，返回值类型就是函数的类型。

关于形参和实参，还有以下几点需要注意：

1. 函数定义中指定的形参，在未发生函数调用时不占内存，只有函数调用时，函数 add中的形参才被分配内存单元。在调用结束后，形参所占的内存单元也被释放，这个前边讲过了，形参是局部变量。
2. 实参可以是常量，也可以是简单或者复杂的表达式，但是要求他们必须有确定的值，在调用发生时将实参的值传递给形参。如上边这个程序也可以写成：c = add(1, a+b);
3. 形参必须要指定数据类型，和定义变量一样，因为它本来就是局部变量。
4. 实参和形参的数据类型应该相同或者赋值兼容。和变量赋值一样，当形参和实参出现不同类型时，则按照不同类型数值的赋值规则进行转换。
5. 主调函数在调用函数之前，应对被调函数做原型声明。

6. 实参向形参的数据传递是单向传递，不能由形参再回传给实参。也就是说，实参值传递给形参后，调用结束，形参单元被释放，而实参单元仍保留并且维持原值。

8.4 单片机按键介绍

独立按键

常用的按键电路有两种形式，独立式按键和矩阵式按键，独立式按键比较简单，它们各自与独立的输入线相连接，如图8-6所示。

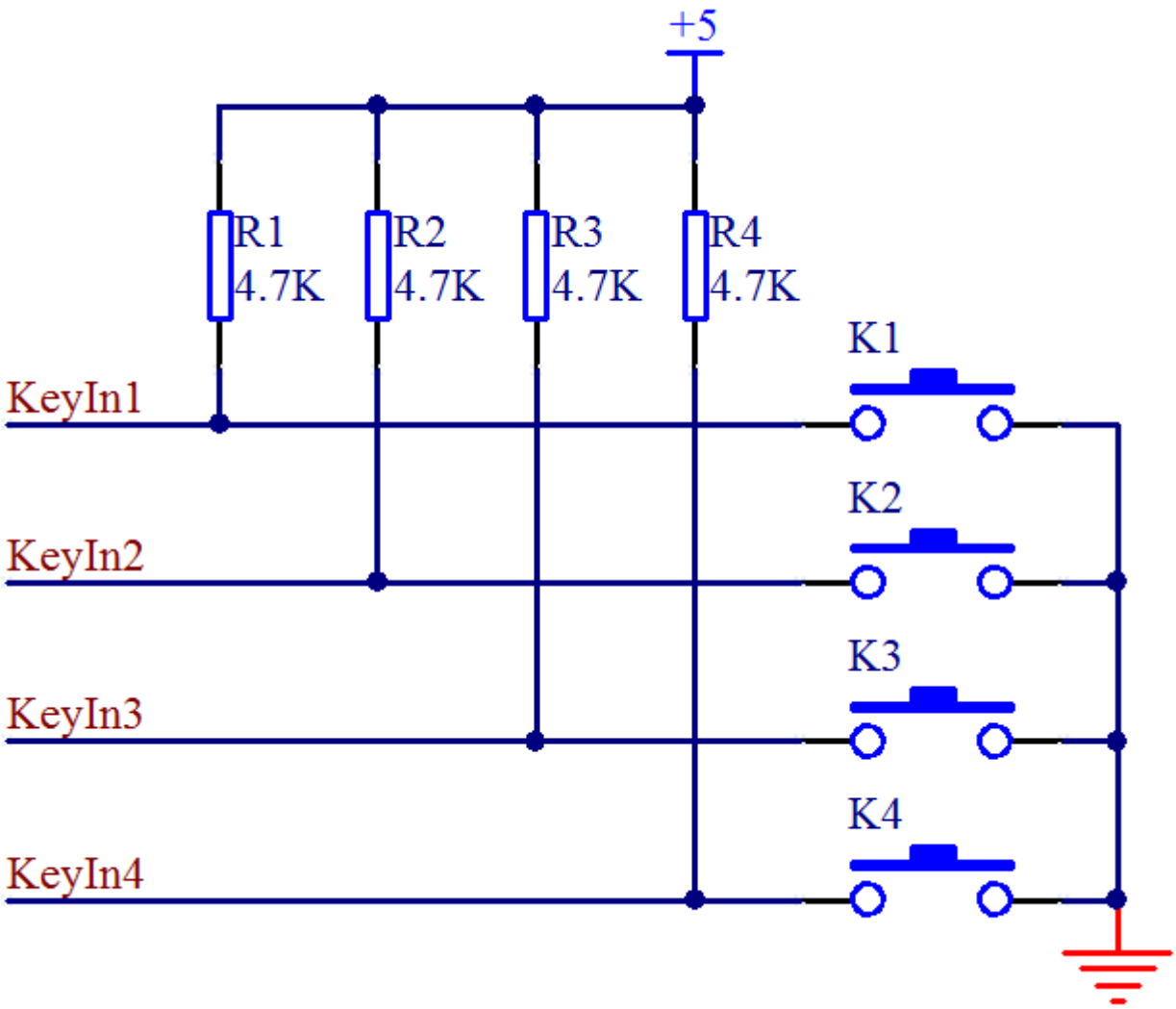

图8-6 独立式按键原理图

4条输入线接到单片机的 IO 口上，当按键 K1 按下时，+5 V 通过电阻 R1 然后再通过按键 K1 最终进入 GND 形成一条通路，那么这条线路的全部电压都加到了 R1 这个电阻上，KeyIn1 这个引脚就是个低电平。当松开按键后，线路断开，就不会有电流通过，那么 KeyIn1 和 +5 V 就应该是等电位，是一个高电平。我们就可以通过 KeyIn1 这个 IO 口的高低电平来判断是否有按键按下。

这个电路中按键的原理我们清楚了，但是实际上在我们的单片机 IO 口内部，也有一个上拉电阻的存在。我们的按键是接到了 P2 口上，P2 口上电默认是准双向 IO 口，我们来简单了解一下这个准双向 IO 口的电路，如图8-7所示。

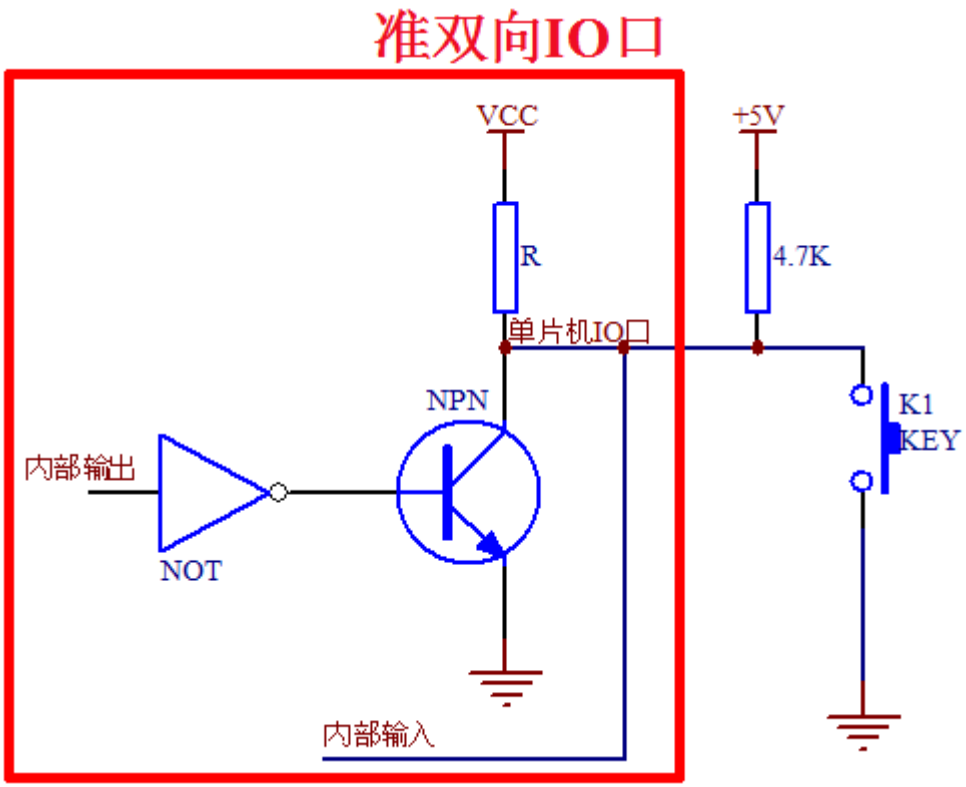

图8-7 准双向 IO 口结构图

首先说明一点，就是我们现在绝大多数单片机的 IO 口都是使用 MOS 管而非三极管，但用在这里的 MOS 管其原理和三极管是一样的，因此在这里我用三极管替代它来进行原理讲解，把前面讲过的三极管的知识搬过来，一切都是适用的，有助于理解。

图8-7方框内的电路都是指单片机内部部分，方框外的就是我们外接的上拉电阻和按键。这个地方大家要注意一下，就是当我们要读取外部按键信号的时候，单片机必须先给该引脚写“1”，也就是高电平，这样我们才能正确读取到外部按键信号，我们来分析一下缘由。

当内部输出是高电平，经过一个反向器变成低电平，NPN 三极管不会导通，那么单片机 IO 口从内部来看，由于上拉电阻 R 的存在，所以是一个高电平。当外部没有按键按下将电平拉低的话，VCC 也是 +5 V，它们之间虽然有2个电阻，但是没有压差，就不会有电流，线上所有的位置都是高电平，这个时候我们就可以正常读取到按键的状态了。

当内部输出是个低电平，经过一个反相器变成高电平，NPN 三极管导通，那么单片机的内部 IO 口就是个低电平，这个时候，外部虽然也有上拉电阻的存在，但是两个电阻是并联关系，不管按键是否按下，单片机的 IO 口上输入到单片机内部的状态都是低电平，我们就无法正常读取到按键的状态了。

这个和水流其实很类似的，内部和外部，只要有一边是低电位，那么电流就会顺流而下，由于只有上拉电阻，下边没有电阻分压，直接到 GND 上了，所以不管另外一边是高还是低，那电平肯定就是低电平了。

从上面的分析就可以得出一个结论，这种具有上拉的准双向 IO 口，如果要正常读取外部信号的状态，必须首先得保证自己内部输出的是1，如果内部输出0，则无论外部信号是1还是0，这个引脚读进来的都是0。

矩阵按键

在某一个系统设计中，如果需要使用很多的按键时，做成独立按键会大量占用 IO 口，因此我们引入了矩阵按键的设计。如图8-8所示，是我们的 KST-51 开发板上的矩阵按键电路原理图，使用8个 IO 口来实现了16个按键。

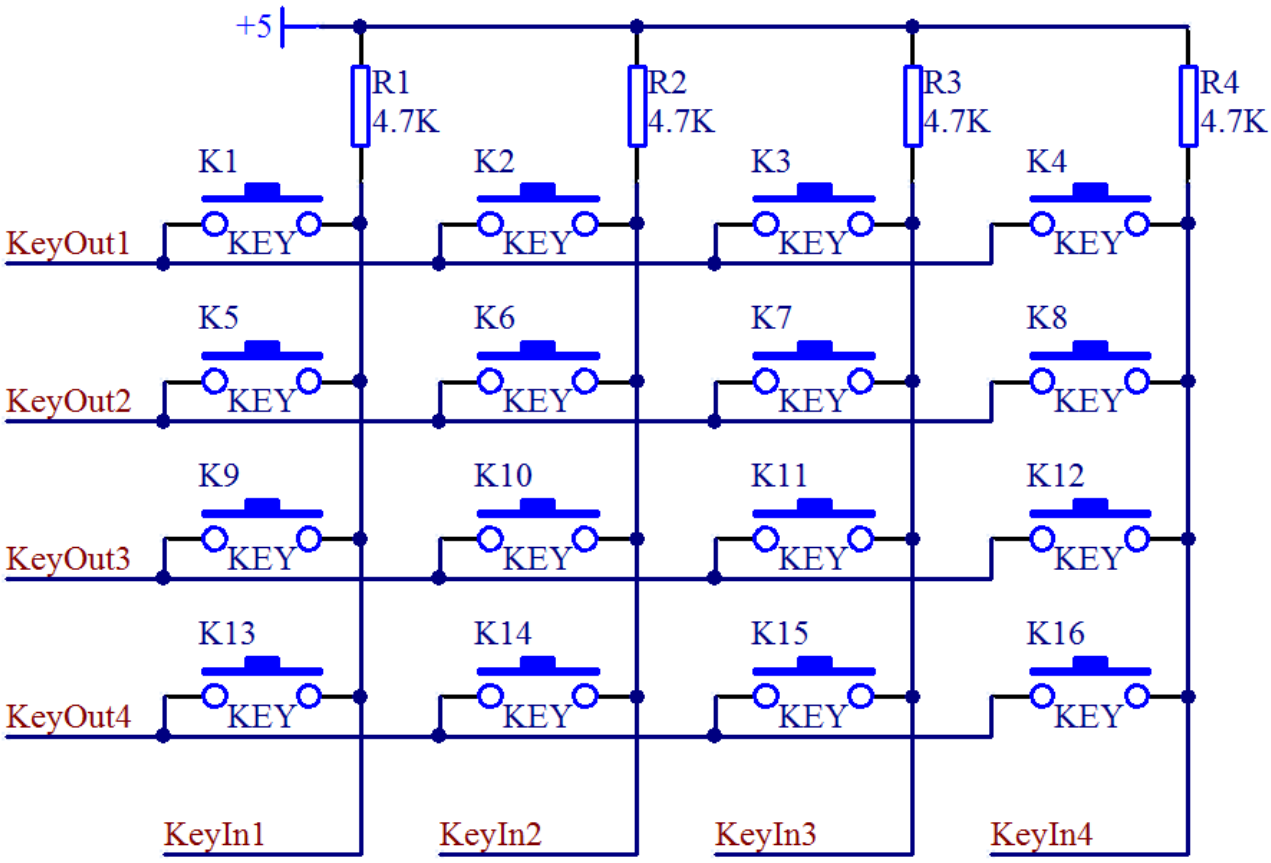

图8-8 矩阵按键原理图

如果独立按键理解了，矩阵按键也不难理解，那么我们一起来分析一下。图8-8中，一共有4组按键，我们只看其中一组，如图8-9所示。大家认真看一下，如果 KeyOut1 输出一个低电平，KeyOut1 就相当于是 GND，是否相当于4个独立按键呢。当然这时候 KeyOut2、KeyOut3、KeyOut4 都必须输出高电平，它们都输出高电平才能保证与它们相连的三路按键不会对这一路产生干扰，大家可以对照两张原理图分析一下。

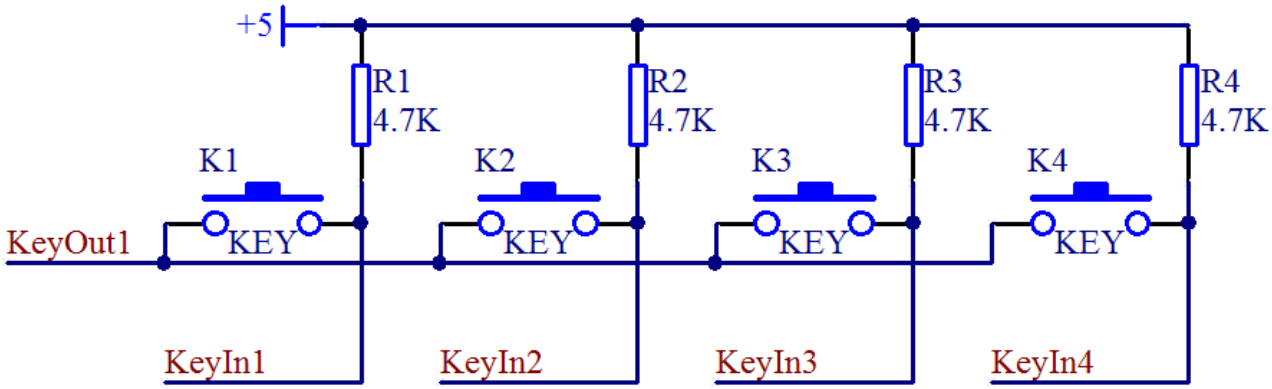

图8-9 矩阵按键变独立按键示意图

8.5 单片机独立按键扫描程序

原理搞清楚了，那么下面我们就先编写一个独立按键的程序，把最基本的功能验证一下。

```
 #include <reg52.h>


sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;
sbit LED9 = P0^7;
sbit LED8 = P0^6;
sbit LED7 = P0^5;
sbit LED6 = P0^4;
sbit KEY1 = P2^4;
sbit KEY2 = P2^5;
sbit KEY3 = P2^6;
sbit KEY4 = P2^7;

void main(){
    ENLED = 0;  //选择独立 LED 进行显示
    ADDR3 = 1;
    ADDR2 = 1;
    ADDR1 = 1;
    ADDR0 = 0;
    P2 = 0xF7;  //P2.3 置0，即 KeyOut1 输出低电平

    while (1){
        //将按键扫描引脚的值传递到 LED 上
        LED9 = KEY1;  //按下时为0，对应的 LED 点亮
        LED8 = KEY2;
        LED7 = KEY3;
        LED6 = KEY4;
    }
}
```

本程序固定在 KeyOut1 上输出低电平，而 KeyOut2～4 保持高电平，就相当于是把矩阵按键的第一行，即 K1～K4 作为4个独立按键来处理，然后把这4个按键的状态直接送给 LED9～6 这4个 LED 小灯，那么当按键按下时，对应按键的输入引脚是0，对应小灯控制信号也是0，于是灯就亮了，这说明上述关于按键检测的理论都是可实现的。

绝大多数情况下，按键是不会一直按住的，所以我们通常检测按键的动作并不是检测一个固定的电平值，而是检测电平值的变化，即按键在按下和弹起这两种状态之间的变化，只要发生了这种变化就说明现在按键产生动作了。

程序上，我们可以把每次扫描到的按键状态都保存起来，当一次按键状态扫描进来的时候，与前一次的状态做比较，如果发现这两次按键状态不一致，就说明按键产生动作了。当上一次的状态是未按下而现在是按下，此时按键的动作就是“按下”；当上一次的状态是按下而现在是未按下，此时按键的动作就是“弹起”。显然，每次按键动作都会包含一次“按下”和一次“弹起”，我们可以任选其一来执行程序，或者两个都用，以执行不同的程序也是可以的。下面就用程序来实现这个功能，程序只取按键 K4 为例。

```
 #include <reg52.h>

sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;
sbit KEY1 = P2^4;
sbit KEY2 = P2^5;
sbit KEY3 = P2^6;
sbit KEY4 = P2^7;

unsigned char code LedChar[] = {  //数码管显示字符转换表
    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
    0x80, 0x90, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E
};

void main(){
    bit backup = 1;  //定义一个位变量，保存前一次扫描的按键值
    unsigned char cnt = 0;  //定义一个计数变量，记录按键按下的次数
    ENLED = 0;  //选择数码管 DS1 进行显示
    ADDR3 = 1;
    ADDR2 = 0;
    ADDR1 = 0;
    ADDR0 = 0;
    P2 = 0xF7;  //P2.3 置0，即 KeyOut1 输出低电平
    P0 = LedChar[cnt];  //显示按键次数初值

    while (1){
        //当前值与前次值不相等说明此时按键有动作
        if (KEY4 != backup){
            //如果前次值为0，则说明当前是由0变1，即按键弹起
            if (backup == 0){
```

```
                cnt++;  //按键次数+1
                //只用1个数码管显示，所以加到10就清零重新开始
                if (cnt >= 10){
                   cnt = 0;
                }
                P0 = LedChar[cnt]; //计数值显示到数码管上
            }
            backup = KEY4;  //更新备份为当前值，以备进行下次比较
        }
    }
}
```

先来介绍出现在程序中的一个新知识点，就是变量类型——bit，这个在标准 C 语言里边是没有的。51单片机有一种特殊的变量类型就是 bit 型。比如 unsigned char 型是定义了一个无符号的8位的数据，它占用一个字节(Byte)的内存，而 bit 型是1位数据，只占用1个位(bit)的内存，用法和标准 C 中其他的基本数据类型是一致的。它的优点就是节省内存空间，8个 bit 型变量才相当于1个 char 型变量所占用的空间。虽然它只有0和1两个值，但也已经可以表示很多东西了，比如：按键的按下和弹起、LED 灯的亮和灭、三极管的导通与关断等等，联想一下已经学过的内容，它是不是能用最小的内存代价来完成很多工作呢？

在这个程序中，我们以 K4 为例，按一次按键，就会产生“按下”和“弹起”两个动态的动作，我们选择在“弹起”时对数码管进行加1操作。理论是如此，大家可以在板子上用 K4 按键做做实验试试，多按几次，是不是会发生这样一种现象：有的时候我明明只按了一下按键，但数字却加了不止1，而是2或者更多？但是我们的程序并没有任何逻辑上的错误，这是怎么回事呢？于是我们就得来说说按键抖动和消抖的问题了。

8.6 单片机按键消抖程序

通常按键所用的开关都是机械弹性开关，当机械触点断开、闭合时，由于机械触点的弹性作用，一个按键开关在闭合时不会马上就稳定的接通，在断开时也不会一下子彻底断开，而是在闭合和断开的瞬间伴随了一连串的抖动，如图8-10所示。

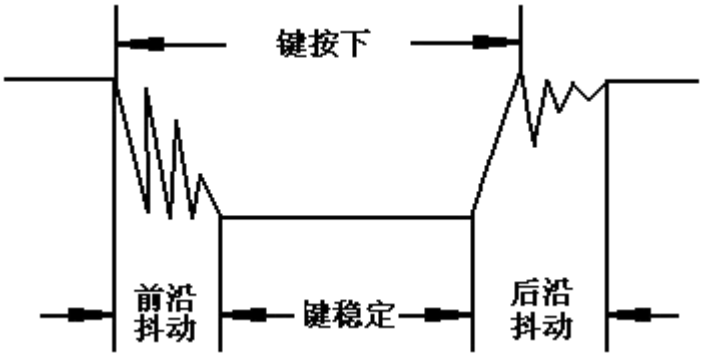

图8-10 按键抖动状态图

按键稳定闭合时间长短是由操作人员决定的，通常都会在 100 ms 以上，刻意快速按的话能达到 40-50 ms 左右，很难再低了。抖动时间是由按键的机械特性决定的，一般都会在 10 ms 以内，为了确保程序对按键的一次闭合或者一次断开只响应一次，必须进行按键的消抖处理。当检测到按键状态变化时，不是立即去响应动作，而是先等待闭合或断开稳定后再进行处理。按键消抖可分为硬件消抖和软件消抖。

硬件消抖就是在按键上并联一个电容，如图8-11所示，利用电容的充放电特性来对抖动过程中产生的电压毛刺进行平滑处理，从而实现消抖。但实际应用中，这种方式的效果往往不是很好，而且还增加了成本和电路复杂度，所以实际中使用的并不多。

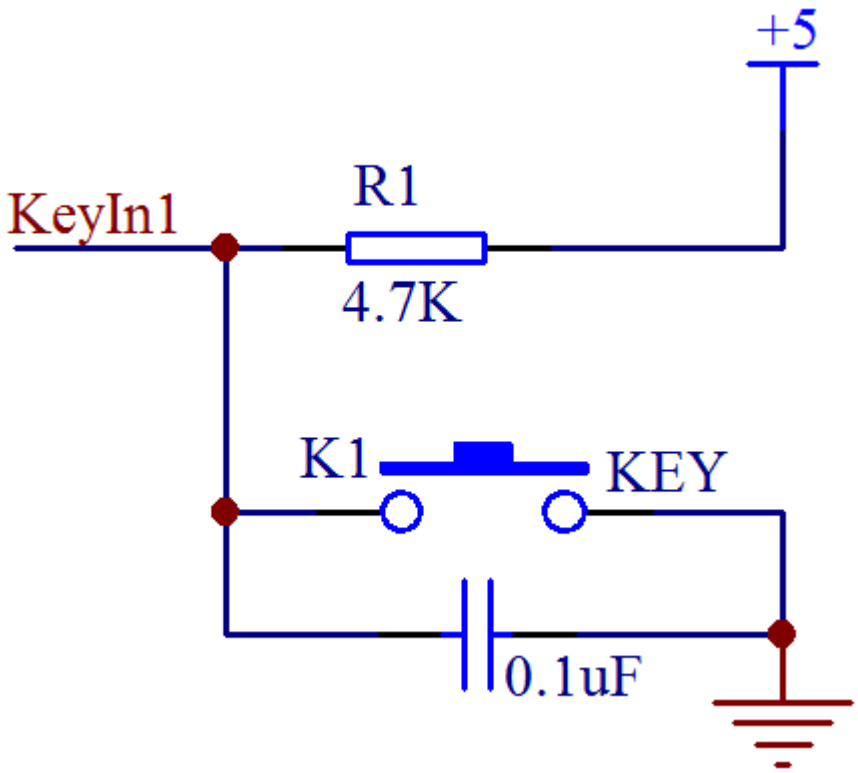

图8-11 硬件电容消抖

在绝大多数情况下，我们是用软件即程序来实现消抖的。最简单的消抖原理，就是当检测到按键状态变化后，先等待一个 10 ms 左右的延时时间，让抖动消失后再进行一次按键状态检测，如果与刚才检测到的状态相同，就可以确认按键已经稳定的动作了。将上一个的程序稍加改动，得到新的带消抖功能的程序如下。

```
 #include <reg52.h>


sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;
sbit KEY1 = P2^4;
sbit KEY2 = P2^5;
sbit KEY3 = P2^6;
sbit KEY4 = P2^7;

unsigned char code LedChar[] = { //数码管显示字符转换表
    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
    0x80, 0x90, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E
};

void delay();
```

```
void main(){
    bit keybuf = 1; //按键值暂存，临时保存按键的扫描值
    bit backup = 1; //按键值备份，保存前一次的扫描值
    unsigned char cnt = 0; //按键计数，记录按键按下的次数

    ENLED = 0;  //选择数码管 DS1 进行显示
    ADDR3 = 1;
    ADDR2 = 0;
    ADDR1 = 0;
    ADDR0 = 0;
    P2 = 0xF7;  //P2.3 置0，即 KeyOut1 输出低电平
    P0 = LedChar[cnt];  //显示按键次数初值

    while (1){
        keybuf = KEY4;  //把当前扫描值暂存
        if (keybuf != backup){  //当前值与前次值不相等说明此时按键有动作
            delay();  //延时大约 10 ms
            if (keybuf == KEY4){  //判断扫描值有没有发生改变，即按键抖动
                if (backup == 0){  //如果前次值为0，则说明当前是弹起动作
                    cnt++;  //按键次数+1
                    //只用1个数码管显示，所以加到10就清零重新开始
                    if (cnt >= 10){
                       cnt = 0;
                    }
                    P0 = LedChar[cnt]; //计数值显示到数码管上
                }
                backup = keybuf; //更新备份为当前值，以备进行下次比较
            }
        }
    }
}
/* 软件延时函数，延时约 10 ms */
void delay(){
    unsigned int i = 1000;
    while (i--);
}
```

大家把这个程序下载到板子上再进行试验试试，按一下按键而数字加了多次的问题是不是就这样解决了？把问题解决掉的感觉是不是很爽呢？

这个程序用了一个简单的算法实现了按键的消抖。作为这种很简单的演示程序，我们可以这样来写，但是实际做项目开发的时候，程序量往往很大，各种状态值也很多， while(1)这个主循环要不停的扫描各种状态值是否有发生变化，及时的进行任务调度，如果程序中间加了这种 delay 延时操作后，很可能某一事件发生了，但是我们程序还在进行 delay 延时操作中，当这个事件发生完了，程序还在 delay 操作中，当我们 delay 完事再去检查的

时候，已经晚了，已经检测不到那个事件了。为了避免这种情况的发生，我们要尽量缩短 while(1)循环一次所用的时间，而需要进行长时间延时的操作，必须想其它的办法来处理。

那么消抖操作所需要的延时该怎么处理呢？其实除了这种简单的延时，我们还有更优异的方法来处理按键抖动问题。举个例子：我们启用一个定时中断，每 2 ms 进一次中断，扫描一次按键状态并且存储起来，连续扫描8次后，看看这连续8次的按键状态是否是一致的。8次按键的时间大概是 16 ms，这 16 ms 内如果按键状态一直保持一致，那就可以确定现在按键处于稳定的阶段，而非处于抖动的阶段，如图8-12。

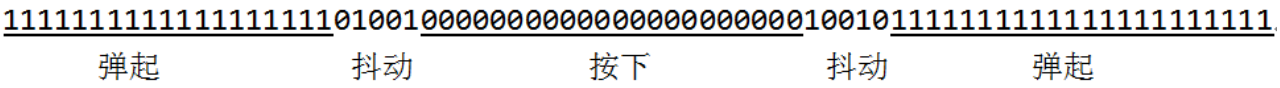

图8-12 按键连续扫描判断

假如左边时间是起始0时刻，每经过 2 ms 左移一次，每移动一次，判断当前连续的8次按键状态是不是全1或者全0，如果是全1则判定为弹起，如果是全0则判定为按下，如果0和1交错，就认为是抖动，不做任何判定。想一下，这样是不是比简单的延时更加可靠？

利用这种方法，就可以避免通过延时消抖占用单片机执行时间，而是转化成了一种按键状态判定而非按键过程判定，我们只对当前按键的连续 16 ms 的8次状态进行判断，而不再关心它在这 16 ms 内都做了什么事情，那么下面就按照这种思路用程序实现出来，同样只以 K4 为例。

```
#include <reg52.h>


sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;
sbit KEY1 = P2^4;
sbit KEY2 = P2^5;
sbit KEY3 = P2^6;
sbit KEY4 = P2^7;

unsigned char code LedChar[] = {  //数码管显示字符转换表
    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
    0x80, 0x90, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E
};
bit KeySta = 1;  //当前按键状态

void main(){
    bit backup = 1;  //按键值备份，保存前一次的扫描值
    unsigned char cnt = 0;  //按键计数，记录按键按下的次数
```

```
    EA = 1;  //使能总中断
    ENLED = 0;  //选择数码管 DS1 进行显示
    ADDR3 = 1;
    ADDR2 = 0;
    ADDR1 = 0;
    ADDR0 = 0;
    TMOD = 0x01; //设置 T0 为模式1
    TH0 = 0xF8; //为 T0 赋初值 0xF8CD，定时 2 ms
    TL0 = 0xCD;
    ET0 = 1;  //使能 T0 中断
    TR0 = 1;  //启动 T0
    P2 = 0xF7;  //P2.3 置 0，即 KeyOut1 输出低电平
    P0 = LedChar[cnt];  //显示按键次数初值

    while (1){
        if (KeySta != backup){  //当前值与前次值不相等说明此时按键有动作
            if (backup == 0){  //如果前次值为0，则说明当前是弹起动作
                cnt++;  //按键次数+1
                if (cnt >= 10){  //只用1个数码管显示，所以加到10就清零重新开始
                    cnt = 0;
                }
                P0 = LedChar[cnt]; //计数值显示到数码管上
            }
            //更新备份为当前值，以备进行下次比较
            backup = KeySta;
        }
    }
}
/* T0 中断服务函数，用于按键状态的扫描并消抖 */
void InterruptTimer0() interrupt 1{
    //扫描缓冲区，保存一段时间内的扫描值
    static unsigned char keybuf = 0xFF;

    TH0 = 0xF8; //重新加载初值
    TL0 = 0xCD;
    //缓冲区左移一位，并将当前扫描值移入最低位
    keybuf = (keybuf<<1) | KEY4;
    //连续8次扫描值都为0，即 16 ms 内都只检测到按下状态时，可认为按键已按下
    if (keybuf == 0x00){
        KeySta = 0;
    //连续8次扫描值都为1，即 16 ms 内都只检测到弹起状态时，可认为按键已弹起
    }else if (keybuf == 0xFF){
        KeySta = 1;
    }
    else{
```

```
        //其它情况则说明按键状态尚未稳定，则不对 KeySta 变量值进行更新
    }
}
```

这个算法是我们在实际工程中经常使用按键所总结的一个比较好的方法，介绍给大家，今后都可以用这种方法消抖了。当然，按键消抖也还有其它的方法，程序实现更是多种多样，大家也可以再多考虑下其它的算法，拓展下思路。

8.7 单片机矩阵按键的扫描

我们讲独立按键扫描的时候，大家已经简单认识了矩阵按键是什么样子了。矩阵按键相当于4组每组各4个独立按键，一共是16个按键。那我们如何区分这些按键呢？想一下我们生活所在的地球，要想确定我们所在的位置，就要借助经纬线，而矩阵按键就是通过行线和列线来确定哪个按键被按下的。那么在程序中我们又如何进行这项操作呢？

前边讲过，按键按下通常都会保持 100 ms 以上，如果在按键扫描中断中，我们每次让矩阵按键的一个 KeyOut 输出低电平，其它三个输出高电平，判断当前所有 KeyIn 的状态，下次中断时再让下一个 KeyOut 输出低电平，其它三个输出高电平，再次判断所有 KeyIn，通过快速的中断不停的循环进行判断，就可以最终确定哪个按键按下了，这个原理是不是跟数码管动态扫描有点类似？数码管我们在动态赋值，而按键这里我们在动态读取状态。至于扫描间隔时间和消抖时间，因为现在有4个 KeyOut 输出，要中断4次才能完成一次全部按键的扫描，显然再采用 2 ms 中断判断8次扫描值的方式时间就太长了（2*4*=64 ms），那么我们就改用 1 ms 中断判断4次采样值，这样消抖时间还是 16 ms(1*4*4)。下面就用程序实现出来，程序循环扫描板子上的 K1～K16 这16个矩阵按键，分离出按键动作并在按键按下时把当前按键的编号显示在一位数码管上（用 0~F 表示，显示值=按键编号-1）。

```
#include <reg52.h>


sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;
sbit KEY_IN_1 = P2^4;
sbit KEY_IN_2 = P2^5;
sbit KEY_IN_3 = P2^6;
sbit KEY_IN_4 = P2^7;
sbit KEY_OUT_1 = P2^3;
sbit KEY_OUT_2 = P2^2;
sbit KEY_OUT_3 = P2^1;
sbit KEY_OUT_4 = P2^0;

unsigned char code LedChar[] = { //数码管显示字符转换表
    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
    0x80, 0x90, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E
};
unsigned char KeySta[4][4] = { //全部矩阵按键的当前状态
    {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}
```

```
};

void main(){
    unsigned char i, j;
    unsigned char backup[4][4] = { //按键值备份，保存前一次的值
        {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}
    };

    EA = 1;  //使能总中断
    ENLED = 0;  //选择数码管 DS1 进行显示
    ADDR3 = 1;
    ADDR2 = 0;
    ADDR1 = 0;
    ADDR0 = 0;
    TMOD = 0x01;  //设置 T0 为模式1
    TH0 = 0xFC;  //为 T0 赋初值 0xFC67，定时 1 ms
    TL0 = 0x67;
    ET0 = 1;  //使能 T0 中断
    TR0 = 1;  //启动 T0
    P0 = LedChar[0];  //默认显示0

    while (1){
        for (i=0; i<4; i++){ //循环检测4*4的矩阵按键
            for (j=0; j<4; j++){
                if (backup[i][j] != KeySta[i][j]){ //检测按键动作
                    if (backup[i][j] != 0){ //按键按下时执行动作
                        P0 = LedChar[i*4+j];  //将编号显示到数码管
                    }
                    backup[i][j] = KeySta[i][j]; //更新前一次的备份值
                }
            }
        }
    }
}
/* T0 中断服务函数，扫描矩阵按键状态并消抖 */
void InterruptTimer0() interrupt 1{
    unsigned char i;
    static unsigned char keyout = 0; //矩阵按键扫描输出索引
    static unsigned char keybuf[4][4] = { //矩阵按键扫描缓冲区
        {0xFF, 0xFF, 0xFF, 0xFF}, {0xFF, 0xFF, 0xFF, 0xFF},
        {0xFF, 0xFF, 0xFF, 0xFF}, {0xFF, 0xFF, 0xFF, 0xFF}
    };

    TH0 = 0xFC; //重新加载初值
    TL0 = 0x67;
```

```
    //将一行的4个按键值移入缓冲区
    keybuf[keyout][0] = (keybuf[keyout][0] << 1) | KEY_IN_1;
    keybuf[keyout][1] = (keybuf[keyout][1] << 1) | KEY_IN_2;
    keybuf[keyout][2] = (keybuf[keyout][2] << 1) | KEY_IN_3;
    keybuf[keyout][3] = (keybuf[keyout][3] << 1) | KEY_IN_4;
    //消抖后更新按键状态
    for (i=0; i<4; i++){ //每行4个按键，所以循环4次
        //连续4次扫描值为0，即 4*4 ms 内都是按下状态时，可认为按键已稳定的按下
        if ((keybuf[keyout][i] & 0x0F) == 0x00){
            KeySta[keyout][i] = 0;
            //连续4次扫描值为1，即 4*4 ms 内都是弹起状态时，可认为按键已稳定的弹起
        }else if ((keybuf[keyout][i] & 0x0F) == 0x0F){
            KeySta[keyout][i] = 1;
        }
    }
    //执行下一次的扫描输出
    keyout++;  //输出索引递增
    keyout = keyout & 0x03; //索引值加到4即归零

    //根据索引，释放当前输出引脚，拉低下次的输出引脚
    switch (keyout){
        case 0: KEY_OUT_4 = 1; KEY_OUT_1 = 0; break;
        case 1: KEY_OUT_1 = 1; KEY_OUT_2 = 0; break;
        case 2: KEY_OUT_2 = 1; KEY_OUT_3 = 0; break;
        case 3: KEY_OUT_3 = 1; KEY_OUT_4 = 0; break;
        default: break;
    }
}
```

这个程序完成了矩阵按键的扫描、消抖、动作分离的全部内容，希望大家认真研究一下，彻底掌握矩阵按键的原理和应用方法。在程序中还有两点值得说明一下。

首先，可能你已经发现了，中断函数中扫描 KeyIn 输入和切换 KeyOut 输出的顺序与前面提到的顺序不同，程序中我首先对所有的 KeyIn 输入做了扫描、消抖，然后才切换到了下一次的 KeyOut 输出，也就是说我们中断每次扫描的实际是上一次输出选择的那行按键，这是为什么呢？因为任何信号从输出到稳定都需要一个时间，有时它足够快而有时却不够快，这取决于具体的电路设计，我们这里的输入输出顺序的颠倒就是为了让输出信号有足够的时间（一次中断间隔）来稳定，并有足够的时间来完成它对输入的影响，当你的按键电路中还有硬件电容消抖时，这样处理就是绝对必要的了，虽然这样使得程序理解起来有点绕，但它的适应性是最好的，换个说法就是，这段程序足够“健壮”，足以应对各种恶劣情况。

其次，是一点小小的编程技巧。注意看 keyout = keyout & 0x03;这一行，在这里我是要让 keyout 在0～3之间变化，加到4就自动归零，按照常规你可以用前面讲过的 if 语句轻松实现，但是你现在看一下这样程序是不是同样可以做到这一点呢？因为0、1、2、3这四个数值正好占用2个二进制的位，所以我们把一个字节的高6位一直清

零的话，这个字节的值自然就是一种到4归零的效果了。看一下，这样一句代码比 if 语句要更为简洁吧，而效果完全一样。

8.8 单片机简易加法计算器程序

学到这里，我们已经掌握了一种显示设备和一种输入设备的使用，那么是不是可以来做点综合性的实验了。好吧，那我们就来做一个简易的加法计算器，用程序实现从板子上标有0～9数字的按键输入相应数字，该数字要实时显示到数码管上，用标有向上箭头的按键代替加号，按下加号后可以再输入一串数字，然后回车键计算加法结果，并同时显示到数码管上。虽然这远不是一个完善的计算器程序，但作为初学者也足够你研究一阵子了。

首先，本程序相对于之前的例程要复杂得多，需要完成的工作也多得多，所以我们把各个子功能都做成独立的函数，以使程序便于编写和维护。大家分析程序的时候就从主函数和中断函数入手，随着程序的流程进行就可以了。大家可以体会体会划分函数的好处，想想如果还是只有主函数和中断函数来实现的话程序会是什么样子。

其次，大家可以看到我们再把矩阵按键扫描分离出动作以后，并没有直接使用行列数所组成的数值作为分支判断执行动作的依据，而是把抽象的行列数转换为了一种叫做标准键盘键码（就是电脑键盘的编码）的数据，然后用得到的这个数据作为下一步分支判断执行动作的依据，为什么多此一举呢？有两层含义：第一，尽量让自己设计的东西（包括硬件和软件）向已有的行业规范或标准看齐，这样有助于别人理解认可你的设计，也有助于你的设计与别人的设计相对接，毕竟标准就是为此而生的嘛。第二，有助于程序的层次化而方便维护与移植，比如我们现在用的按键是*44的，但如果后续又增加了一行成了*45的，那么由行列数组成的编号可能就变了，我们就要在程序的各个分支中查找修改，稍不留神就会出错，而采用这种转换后，我们则只需要维护 KeyCodeMap 这样一个数组表格就行了，看上去就像是把程序的底层驱动与应用层的功能实现函数分离开了，应用层不用关心底层的实现细节，底层改变后也无需在应用层中做相应修改，两层程序之间是一种标准化的接口。这就是程序的层次化，而层次化是构建复杂系统的必备条件，那么现在就先通过简单的示例来学习一下吧。

作为初学者针对这种程序的学习方式是，先从头到尾读一到三遍，边读边理解，然后边抄边理解，彻底理解透彻后，自己尝试独立写出来。完全采用记忆模式来学习这种例程，一两个例程你可能感觉不到什么提高，当这种例程背过上百八十个的时候，厚积薄发的感觉就来了。同时，在抄读的过程中也要注意学习编程规范，这些可都是无形的财富，可以为你日后的研发工作加分的哦。

```
#include <reg52.h>


sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;
sbit KEY_IN_1 = P2^4;
sbit KEY_IN_2 = P2^5;
sbit KEY_IN_3 = P2^6;
```

```
sbit KEY_IN_4 = P2^7;
sbit KEY_OUT_1 = P2^3;
sbit KEY_OUT_2 = P2^2;
sbit KEY_OUT_3 = P2^1;
sbit KEY_OUT_4 = P2^0;

unsigned char code LedChar[] = { //数码管显示字符转换表
    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
    0x80, 0x90, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E
};
unsigned char LedBuff[6] = { //数码管显示缓冲区
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
};
unsigned char code KeyCodeMap[4][4] = { //矩阵按键编号到标准键盘键码的映射表
    { 0x31, 0x32, 0x33, 0x26 }, //数字键1、数字键2、数字键3、向上键
    { 0x34, 0x35, 0x36, 0x25 }, //数字键4、数字键5、数字键6、向左键
    { 0x37, 0x38, 0x39, 0x28 }, //数字键7、数字键8、数字键9、向下键
    { 0x30, 0x1B, 0x0D, 0x27 } //数字键0、ESC 键、 回车键、 向右键
};
unsigned char KeySta[4][4] = { //全部矩阵按键的当前状态
    {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}
};

void KeyDriver();
void main(){
    EA = 1;  //使能总中断
    ENLED = 0;  //选择数码管进行显示
    ADDR3 = 1;
    TMOD = 0x01; //设置 T0 为模式1
    TH0 = 0xFC; //为 T0 赋初值 0xFC67，定时 1 ms
    TL0 = 0x67;
    ET0 = 1; //使能 T0 中断
    TR0 = 1; //启动 T0
    LedBuff[0] = LedChar[0]; //上电显示0

    while (1){
        KeyDriver();  //调用按键驱动函数
    }
}
/* 将一个无符号长整型的数字显示到数码管上，num-待显示数字 */
void ShowNumber(unsigned long num){
    signed char i;
    unsigned char buf[6];
    //把长整型数转换为6位十进制的数组
    for (i=0; i<6; i++){
```

```
        buf[i] = num % 10;
        num = num / 10;
    }
    //从最高位起，遇到0转换为空格，遇到非0则退出循环
    for (i=5; i>=1; i--){
        if (buf[i] == 0){
            LedBuff[i] = 0xFF;
        }else{
            break;
        }
    }
    for ( ; i>=0; i--){ //剩余低位都如实转换为数码管显示字符
        LedBuff[i] = LedChar[buf[i]];
    }
}
/* 按键动作函数，根据键码执行相应的操作，keycode-按键键码 */
void KeyAction(unsigned char keycode){
    static unsigned long result = 0; //用于保存运算结果
    static unsigned long addend = 0; //用于保存输入的加数

    if ((keycode>=0x30) && (keycode<=0x39)){ //输入0-9的数字
        //整体十进制左移，新数字进入个位
        addend = (addend*10)+(keycode-0x30);
        ShowNumber(addend); //运算结果显示到数码管
        //向上键用作加号，执行加法或连加运算
    }else if (keycode == 0x26){
        result += addend;  //进行加法运算
        addend = 0;
        ShowNumber(result);  //运算结果显示到数码管
        //回车键，执行加法运算(实际效果与加号相同)
    }else if (keycode == 0x0D){
        result += addend;  //进行加法运算
        addend = 0;
        ShowNumber(result); //运算结果显示到数码管
    }else if (keycode == 0x1B){ //Esc 键，清零结果
        addend = 0;
        result = 0;
        ShowNumber(addend); //清零后的加数显示到数码管
    }
}
/* 按键驱动函数，检测按键动作，调度相应动作函数，需在主循环中调用 */
void KeyDriver(){
    unsigned char i, j;
    static unsigned char backup[4][4] = { //按键值备份，保存前一次的值
        {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}
```

```
    };

    for (i=0; i<4; i++){ //循环检测4*4的矩阵按键
        for (j=0; j<4; j++){
            if (backup[i][j] != KeySta[i][j]){ //检测按键动作
                if (backup[i][j] != 0){ //按键按下时执行动作
                    KeyAction(KeyCodeMap[i][j]); //调用按键动作函数
                }
                backup[i][j] = KeySta[i][j];//刷新前一次的备份值
            }
        }
    }
}
/* 按键扫描函数，需在定时中断中调用，推荐调用间隔 1 ms */
void KeyScan(){
    unsigned char i;
    //矩阵按键扫描输出索引
    static unsigned char keyout = 0;

    static unsigned char keybuf[4][4] = { //矩阵按键扫描缓冲区
        {0xFF, 0xFF, 0xFF, 0xFF}, {0xFF, 0xFF, 0xFF, 0xFF},
        {0xFF, 0xFF, 0xFF, 0xFF}, {0xFF, 0xFF, 0xFF, 0xFF}
    };
    //将一行的4个按键值移入缓冲区
    keybuf[keyout][0] = (keybuf[keyout][0] << 1) | KEY_IN_1;
    keybuf[keyout][1] = (keybuf[keyout][1] << 1) | KEY_IN_2;
    keybuf[keyout][2] = (keybuf[keyout][2] << 1) | KEY_IN_3;
    keybuf[keyout][3] = (keybuf[keyout][3] << 1) | KEY_IN_4;
    //消抖后更新按键状态
    //每行4个按键，所以循环4次
    for (i=0; i<4; i++){
        //连续4次扫描值为0，即 4*4 ms 内都是按下状态时，可认为按键已稳定的按下
        if ((keybuf[keyout][i] & 0x0F) == 0x00){
            KeySta[keyout][i] = 0;
        //连续4次扫描值为1，即 4*4 ms 内都是弹起状态时，可认为按键已稳定的弹起
        }else if ((keybuf[keyout][i] & 0x0F) == 0x0F){
            KeySta[keyout][i] = 1;
        }
    }
    //执行下一次的扫描输出
    keyout++; //输出索引递增
    keyout = keyout & 0x03; //索引值加到4即归零
    //根据索引，释放当前输出引脚，拉低下次的输出引脚
    switch (keyout){
        case 0: KEY_OUT_4 = 1; KEY_OUT_1 = 0; break;
```

```
        case 1: KEY_OUT_1 = 1; KEY_OUT_2 = 0; break;
        case 2: KEY_OUT_2 = 1; KEY_OUT_3 = 0; break;
        case 3: KEY_OUT_3 = 1; KEY_OUT_4 = 0; break;
        default: break;
    }
}
/* 数码管动态扫描刷新函数，需在定时中断中调用 */
void LedScan(){
    static unsigned char i = 0; //动态扫描的索引
    P0 = 0xFF; //显示消隐

    switch (i){
        case 0: ADDR2=0; ADDR1=0; ADDR0=0; i++; P0=LedBuff[0]; break;
        case 1: ADDR2=0; ADDR1=0; ADDR0=1; i++; P0=LedBuff[1]; break;
        case 2: ADDR2=0; ADDR1=1; ADDR0=0; i++; P0=LedBuff[2]; break;
        case 3: ADDR2=0; ADDR1=1; ADDR0=1; i++; P0=LedBuff[3]; break;
        case 4: ADDR2=1; ADDR1=0; ADDR0=0; i++; P0=LedBuff[4]; break;
        case 5: ADDR2=1; ADDR1=0; ADDR0=1; i=0; P0=LedBuff[5]; break;
        default: break;
    }
}
/* T0 中断服务函数，用于数码管显示扫描与按键扫描 */
void InterruptTimer0() interrupt 1{
    TH0 = 0xFC; //重新加载初值
    TL0 = 0x67;
    LedScan(); //调用数码管显示扫描函数
    KeyScan(); //调用按键扫描函数
}
```

3

9. 单片机中的步进电机与蜂鸣器

对于技术的学习，希望大家一定要有足够的耐性和韧性。如果你决定从事单片机这门技术，那就一定要坚持学习下去，不能半途而废，当你坚持学习一段时间后你会发现自己慢慢会喜欢这些玩意，对这些东西有了浓厚的兴趣和感情，那你离成功就不远了。学到第九课了，鼓励鼓励自己，再加把劲哦！在本章中我们首先来了解单片机 IO 的一些细节内容，然后在此基础上再学习两种常用设备的使用方法——步进电机和蜂鸣器。

9.1 单片机 IO 口的结构

上节课我们提到了单片机 IO 口的其中一种“准双向 IO”的内部结构，实际上我们的单片机 IO 口还有另外三种状态，分别是开漏、推挽、高阻态，我们通过图9-1来分析下另外这三种状态。

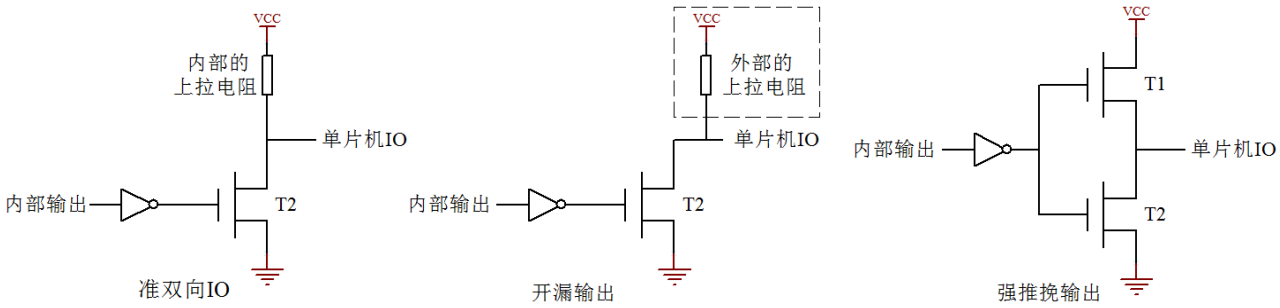

图9-1 单片机 IO 结构示意图

前边我们简单介绍“准双向 IO”的时候，我们是用三极管来说明的，出于严谨的态度，我们这里按照实际情况用 MOS 管画图示意。实际上三极管是靠电流导通的，而 MOS 管是靠电压导通的，具体缘由和它们的内部构造有关系，在这里我们暂且不必关心，如果今后有必要了解可以直接查找模拟电子书或者百度相关资料进行细致学习。在单片机 IO 口状态这一块内容上，我们可以把 MOS 管当三极管来理解。在图9-1中，T1 相当于一个 PNP 三极管，T2 相当于一个 NPN 三极管。

其中准双向 IO 口原理已经讲过了，开漏输出和准双向 IO 的唯一区别，就是开漏输出把内部的上拉电阻去掉了。开漏输出如果要输出高电平时，T2 关断，IO 电平要靠外部的上拉电阻才能拉成高电平，如果没有外部上拉电阻 IO 电平就是一个不确定态。标准51单片机的 P0 口默认就是开漏输出，如果要用的时候外部需要加上拉电阻。而强推挽输出就是有比较强的驱动能力，如图9-1中第三张小图，当内部输出一个高电平时，通过 MOS 管直接输出电流，没有电阻的限流，电流输出能力也比较大；如果内部输出一个低电平，那反向电流也可以很大，强推挽的一个特点就是驱动能力强。

单片机 IO 还有一种状态叫高阻态。通常我们用来做输入引脚的时候，可以将 IO 口设置成高阻态，高阻态引脚本身如果悬空，用万用表测量的时候可能是高可能是低，它的状态完全取决于外部输入信号的电平，高阻态引脚对 GND 的等效电阻很大（理论上相当于无穷大，但实际上总是有限值而非无穷大），所以称之为高阻。

这就是单片机的 IO 口的四种状态，在我们51单片机的学习过程中，主要应用的是准双向 IO 口，随着我们学习的深入，其它状态也会有接触，在这里介绍给大家学习一下。

9.2 单片机上下拉电阻

前边似乎我们很多次提到了上拉电阻，下拉电阻，具体到底什么样的电阻算是上下拉电阻，上下拉电阻都有何作用呢？

上拉电阻就是将不确定的信号通过一个电阻拉到高电平，同时此电阻也起到一个限流作用，下拉就是下拉到低电平。

比如我们的 IO 设置为开漏输出高电平或者是高阻态时，默认的电平就是不确定的，外部经一个电阻接到 VCC，也就是上拉电阻，那么相应的引脚就是高电平；经一个电阻到 GND，也就是下拉电阻，那么相应的引脚就是一个低电平。

上拉电阻应用很多，都可以起到什么作用呢？我们现在主要先了解最常用的以下4点： 1. OC 门要输出高电平，必须外部加上拉电阻才能正常使用，其实 OC 门就相当于单片机 IO 的开漏输出，其原理可参照图9-1中的开漏电路。 2. 加大普通 IO 口的驱动能力。标准51单片机的内部 IO 口的上拉电阻，一般都是在几十 K 欧，比如 STC89C52 内部是 20 K 的上拉电阻，所以最大输出电流是 250 uA，因此外部加个上拉电阻，可以形成和内部上拉电阻的并联结构，增大高电平时电流的输出能力。 3. 在电平转换电路中，比如我们前边讲的 5 V 转 12 V 的电路中，上拉电阻其实起到的是限流电阻的作用，可以回顾一下图3-8。 4. 单片机中未使用的引脚，比如总线引脚，引脚悬空时，容易受到电磁干扰而处于紊乱状态，虽然不会对程序造成什么影响，但通常会增加单片机的功耗，加上一个对 VCC 的上拉电阻或者一个对 GND 的下拉电阻后，可以有效的抵抗电磁干扰。

那么我们在进行电路设计的时候，又该如何选择合适的上下拉电阻的阻值呢？ 1. 从降低功耗的方面考虑应当足够大，因为电阻越大，电流越小。 2. 从确保足够的引脚驱动能力考虑应当足够小，电阻小了，电流才能大。 3. 在开漏输出时，过大的上拉电阻会导致信号上升沿变缓。我们来解释一下：实际电平的变化都是需要时间的，虽然很小，但永远都达不到零，而开漏输出时上拉电阻的大小就直接影响了这个上升过程所需要的时间，如图9-2所示。想一下，如果电阻很大，而信号频率又很快的话，最终将导致信号还没等上升到高电平就又变为低了，于是信号就无法正确传送了。

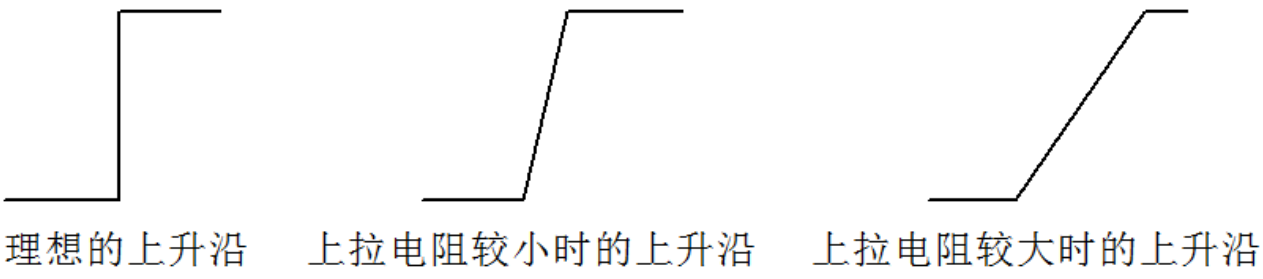

图9-2 上拉电阻阻值对波形的影响

综合考虑各种情况，我们常用的上下拉电阻值大多选取在 1 K 到 10 K 之间，具体到底多大通常要根据实际需求来选，通常情况下在标准范围内就可以了，不一定是一个固定的值。

9.3 电机的分类

电机的分类方式有很多，从用途角度可划分为驱动类电机和控制类电机。直流电机属于驱动类电机，这种电机是将电能转换成机械能，主要应用在电钻、小车轮子、电风扇、洗衣机等设备上。步进电机属于控制类电机，它是将脉冲信号转换成一个转动角度的电机，在非超载的情况下，电机的转速、停止的位置只取决于脉冲信号的频率和脉冲数，主要应用在自动化仪表、机器人、自动生产流水线、空调扇叶转动等设备。

步进电机又分为反应式、永磁式和混合式三种： - 反应式步进电机：结构简单成本低，但是动态性能差、效率低、发热大、可靠性难以保证，所以现在基本已经被淘汰了。 - 永磁式步进电机：动态性能好、输出力矩较大，但误差相对来说大一些，因其价格低而广泛应用于消费性产品。 - 混合式步进电机：综合了反应式和永磁式的优点，力矩大、动态性能好、步距角小，精度高，但是结构相对来说复杂，价格也相对高，主要应用于工业。

我们本章内容主要讲解 28BYJ-48 这款步进电机，先介绍型号中包含的具体含义： - 28：步进电机的有效最大外径是28毫米 - B：表示是步进电机 - Y：表示是永磁式 - J：表示是减速型 - 48：表示四相八拍

9.4 28BYJ-48 步进电机原理

28BYJ-48 是4相永磁式减速步进电机，其外观如图9-3所示：

图9-3 步进电机外观

我们先来解释“4相永磁式”的概念，28BYJ-48 的内部结构示意图9-4所示。先看里圈，它上面有6个齿，分别标注为0～5，这个叫做转子，顾名思义，它是要转动的，转子的每个齿上都带有永久的磁性，是一块永磁体，这就是“永磁式”的概念。再看外圈，这个就是定子，它是保持不动的，实际上它是跟电机的外壳固定在一起的，它上面有8个齿，而每个齿上都缠上了一个线圈绕组，正对着的2个齿上的绕组又是串联在一起的，也就是说正对着的2个绕组总是会同时导通或关断的，如此就形成了4相，在图中分别标注为 A-B-C-D，这就是“4相”的概念。

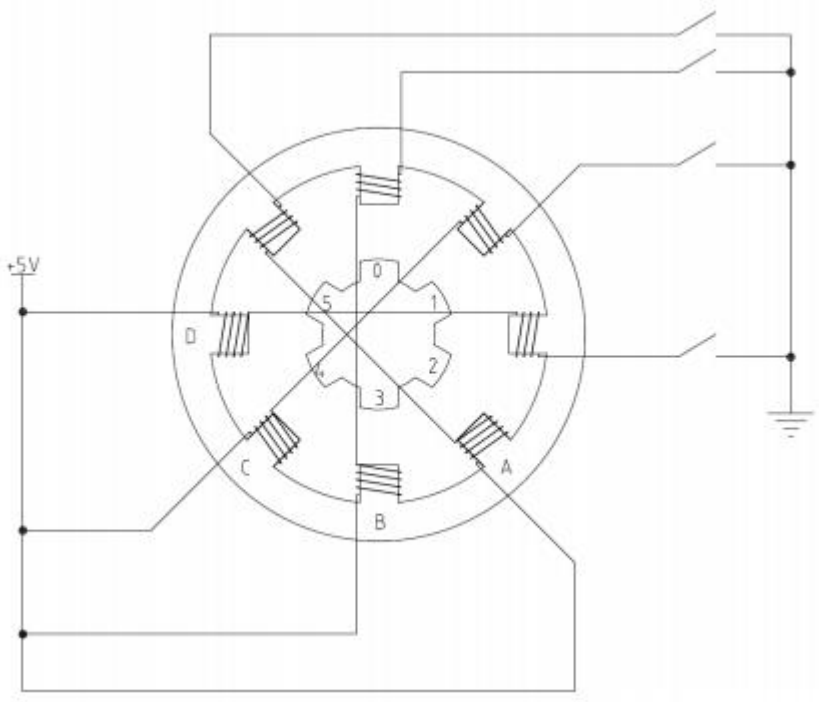

图9-4 步进电机内部结构示意图

现在我们分析一下它的工作原理： 假定电机的起始状态就如图9-4所示，逆时针方向转动，起始时是 B 相绕组的开关闭合，B 相绕组导通，那么导通电流就会在正上和正下两个定子齿上产生磁性，这两个定子齿上的磁性就会对转子上的0和3号齿产生最强的吸引力，就会如图所示的那样，转子的0号齿在正上、3号齿在正下而处于平衡状态；此时我们会发现，转子的1号齿与右上的定子齿也就是 C 相的一个绕组呈现一个很小的夹角，2号齿与右边的定子齿也就是 D 相绕组呈现一个稍微大一点的夹角，很明显这个夹角是1号齿和 C 绕组夹角的2倍，同理，左侧的情况也是一样的。

接下来，我们把 B 相绕组断开，而使 C 相绕组导通，那么很明显，右上的定子齿将对转子1号齿产生最大的吸引力，而左下的定子齿将对转子4号齿，产生最大的吸引力，在这个吸引力的作用下，转子1、4号齿将对齐到右上和左下的定子齿上而保持平衡，如此，转子就转过了起始状态时1号齿和 C 相绕组那个夹角的角度。

再接下来，断开 C 相绕组，导通 D 相绕组，过程与上述的情况完全相同，最终将使转子2、5号齿与定子 D 相绕组对齐，转子又转过了上述同样的角度。

那么很明显，当 A 相绕组再次导通，即完成一个 B-C-D-A 的四节拍操作后，转子的0、3号齿将由原来的对齐到上下2个定子齿，而变为了对齐到左上和右下的两个定子齿上，即转子转过了一个定子齿的角度。依此类推，再来一个四节拍，转子就将再转过一个齿的角度，8个四节拍以后转子将转过完整的一圈，而其中单个节拍使转子转过的角度就很容易计算出来了，即360度/(8*4)=11.25度，这个值就叫做步进角度。而上述这种工作模式就是步进电机的单四拍模式——单相绕组通电四节拍。

我们再来讲解一种具有更优性能的工作模式，那就是在单四拍的每两个节拍之间再插入一个双绕组导通的中间节拍，组成八拍模式。比如，在从 B 相导通到 C 项导通的过程中，假如一个 B 相和 C 相同时导通的节拍，这个时候，由于 B、C 两个绕组的定子齿对它们附近的转子齿同时产生相同的吸引力，这将导致这两个转子齿的中心线对比到 B、C 两个绕组的中心线上，也就是新插入的这个节拍使转子转过了上述单四拍模式中步进角度的一半，即5.625度。这样一来，就使转动精度增加了一倍，而转子转动一圈则需要8*8=64拍了。另外，新增加的这个中间节拍，还会在原来单四拍的两个节拍引力之间又加了一把引力，从而可以大大增加电机的整体扭力输出，使电机更“有劲”了。

除了上述的单四拍和八拍的工作模式外，还有一个双四拍的工作模式——双绕组通电四节拍。其实就是把八拍模式中的两个绕组同时通电的那四拍单独拿出来，而舍弃掉单绕组通电的那四拍而已。其步进角度同单四拍是一样的，但由于它是两个绕组同时导通，所以扭矩会比单四拍模式大，在此就不做过多解释了。

八拍模式是这类4相步进电机的最佳工作模式，能最大限度的发挥电机的各项性能，也是绝大多数实际工程中所选择的模式，因此我们就重点来讲解如何用单片机程序来控制电机按八拍模式工作。

9.5 让 28BYJ-48 步进电机转起来

再重新看一下上面的步进电机外观图和内部结构图：步进电机一共有5根引线，其中红色的是公共端，连接到 5 V 电源，接下来的橙、黄、粉、蓝就对应了 A、B、C、D 相；那么如果要导通 A 相绕组，就只需将橙色线接地即可，B 相则黄色接地，依此类推；再根据上述单四拍和八拍工作过程的讲解，可以得出下面的绕组控制顺序表，如表9-1所示：

表 9-1 八拍模式绕组控制顺序表

	1	2	3	4	5	6	7	8
P1-红	VCC	VCC	VCC	VCC	VCC	VCC	VCC	VCC
P2-橙	GND	GND						GND
P3-黄		GND	GND	GND				
P4-粉				GND	GND	GND		
P5-蓝						GND	GND	GND

我们板子上控制步进电机部分是和板子上的显示控制的 74HC138 译码器部分复用的 P1.0～P1.3，关于跳线我们在第3章已经讲过了，通过调整跳线帽的位置可以让 P1.0～P1.3控制步进电机的四个绕组，如图9-5所示。

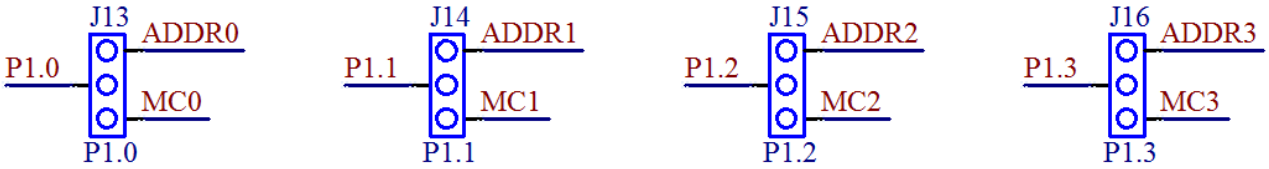

图9-5 显示译码与步进电机的选择跳线

如果要使用电机的话，需要把4个跳线帽都调到跳线组的左侧（开发板上的实际位置），即左侧针和中间针连通（对应原理图中的中间和下边的针），就可以使用 P1.0 到 P1.3 控制步进电机了，如要再使用显示部分的话，就要再换回到右侧了。那如果大家既想让显示部分正常工作，又想让电机工作该怎么办呢？跳线帽保持在右侧，用杜邦线把步进电机的控制引脚（即左侧的排针）连接到其它的暂不使用的单片机 IO 上即可。

再来看一下我们步进电机的原理图，步进电机的控制电路如图9-6所示。

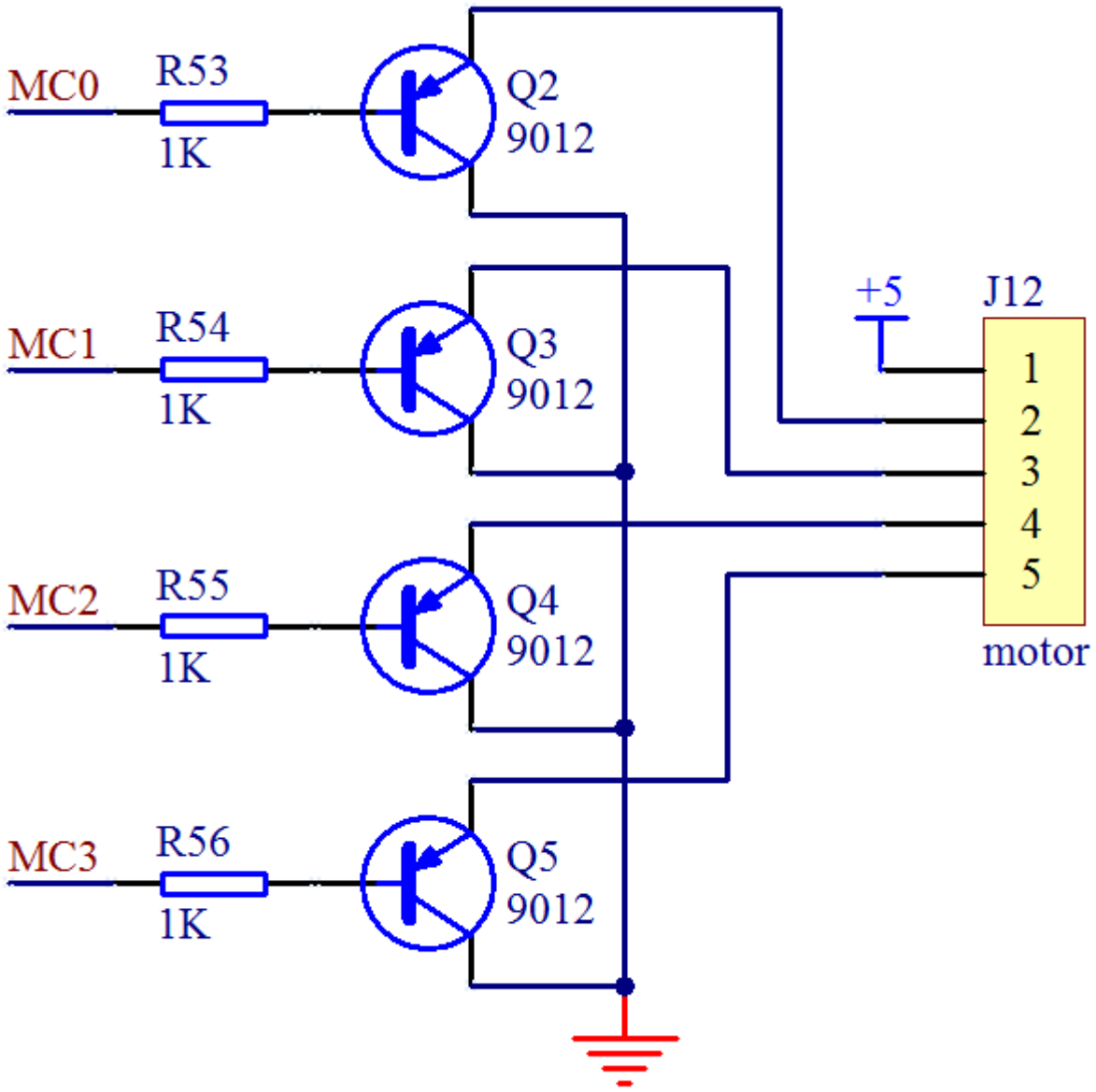

图9-6 步进电机控制电路

诚然，单片机的 IO 口可以直接输出 0 V 和 5 V 的电压，但是电流驱动能力，也就是带载能力非常有限，所以我们在每相的控制线上都增加一个三极管来提高驱动能力。由图中可以看出，若要使 A 相导通，则必须是 Q2 导通，此时 A 相也就是橙色线就相当于接地了，于是A 相绕组导通，此时单片机 P1 口低4位应输出 0b1110，即 0xE；如要 A、B 相同时导通，那么就是 Q2、Q3 导通，P1 口低4位应输出 0b1100，即 0xC，依此类推，我们可以得到下面的八拍节拍的 IO 控制代码数组：

```
    unsigned char code BeatCode[8] = { 0xE, 0xC, 0xD, 0x9, 0xB, 0x3, 0x7, 0x6 };
```

到这里，似乎所有的逻辑问题都解决了，循环将这个数组内的值送到 P1 口就行了。但是，只要再深入想一下就会发现还有个问题：多长时间送一次数据，也就是说一个节拍要持续多长时间合适呢？是随意的吗？当然不是

了，这个时间是由步进电机的启动频率决定的。启动频率，就是步进电机在空载情况下能够正常启动的最高脉冲频率，如果脉冲频率高于该值，电机就不能正常启动。表9-2就是由厂家提供的步进电机参数表，我们来看一下。

表9-2 28BYJ-48 步进电机参数表

供电电压	相数	相电阻Ω	步进角度	减速比	启动频率P.P.S	转矩g.cm	噪声dB	绝缘介电强度
5V	4	50±10%	5.625/64	1:64	≥550	≥300	≤35	600VAC

表中给出的参数是≥550，单位是 P.P.S，即每秒脉冲数，这里的意思就是说：电机保证在你每秒给出550个步进脉冲的情况下，可以正常启动。那么换算成单节拍持续时间就是 1 s/550=1.8 ms，为了让电机能够启动，我们控制节拍刷新时间大于 1.8 ms 就可以了。有了这个参数，我们就可以动手写出最简单的电机转动程序了，如下：

```
 #include <reg52.h>


unsigned char code BeatCode[8] = { //步进电机节拍对应的 IO 控制代码
    0xE, 0xC, 0xD, 0x9, 0xB, 0x3, 0x7, 0x6
};
void delay();

void main(){
    unsigned char tmp;  //定义一个临时变量
    unsigned char index = 0;  //定义节拍输出索引
    while (1){
        tmp = P1; //用 tmp 把 P1 口当前值暂存
        tmp = tmp & 0xF0; //用&操作清零低4位
        //用|操作把节拍代码写到低4位
        tmp = tmp | BeatCode[index];
        //把低4位的节拍代码和高4位的原值送回 P1
        P1 = tmp;
        index++;  //节拍输出索引递增
        index = index & 0x07;  //用&操作实现到8归零
        delay();  //延时 2 ms，即 2 ms 执行一拍
    }
}
/* 软件延时函数，延时约 2ms */
void delay(){
    unsigned int i = 200;
    while (i--);
}
```

把程序编译下载到板子上试试吧！看看电机转了没有？要记得换跳线哦！

9.6 28BYJ-48 步进电机转动精度与深入分析

转是转了，但是不是感觉有点不太对劲呢？太慢了？别急，咱们继续。根据本章开头讲解的原理，八拍模式时，步进电机转过一圈是需要64个节拍，而我们程序中是每个节拍持续 2 ms，那么转一圈就应该是 128 ms，即1秒钟转7圈多，可怎么看上去它好像是7秒多才转了一圈呢？

那么，是时候来了解“永磁式减速步进电机”中这个“减速”的概念了。图9-7是这个 28BYJ-48 步进电机的拆解图，从图中可以看到，位于最中心的那个白色小齿轮才是步进电机的转子输出，64个节拍只是让这个小齿轮转了一圈，然后它带动那个浅蓝色的大齿轮，这就是一级减速。大家看一下右上方的白色齿轮的结构，除电机转子和最终输出轴外的3个传动齿轮都是这样的结构，由一层多齿和一层少齿构成，而每一个齿轮都用自己的少齿层去驱动下一个齿轮的多齿层，这样每2个齿轮都构成一级减速，一共就有了4级减速，那么总的减速比是多少呢？即转子要转多少圈最终输出轴才转一圈呢？

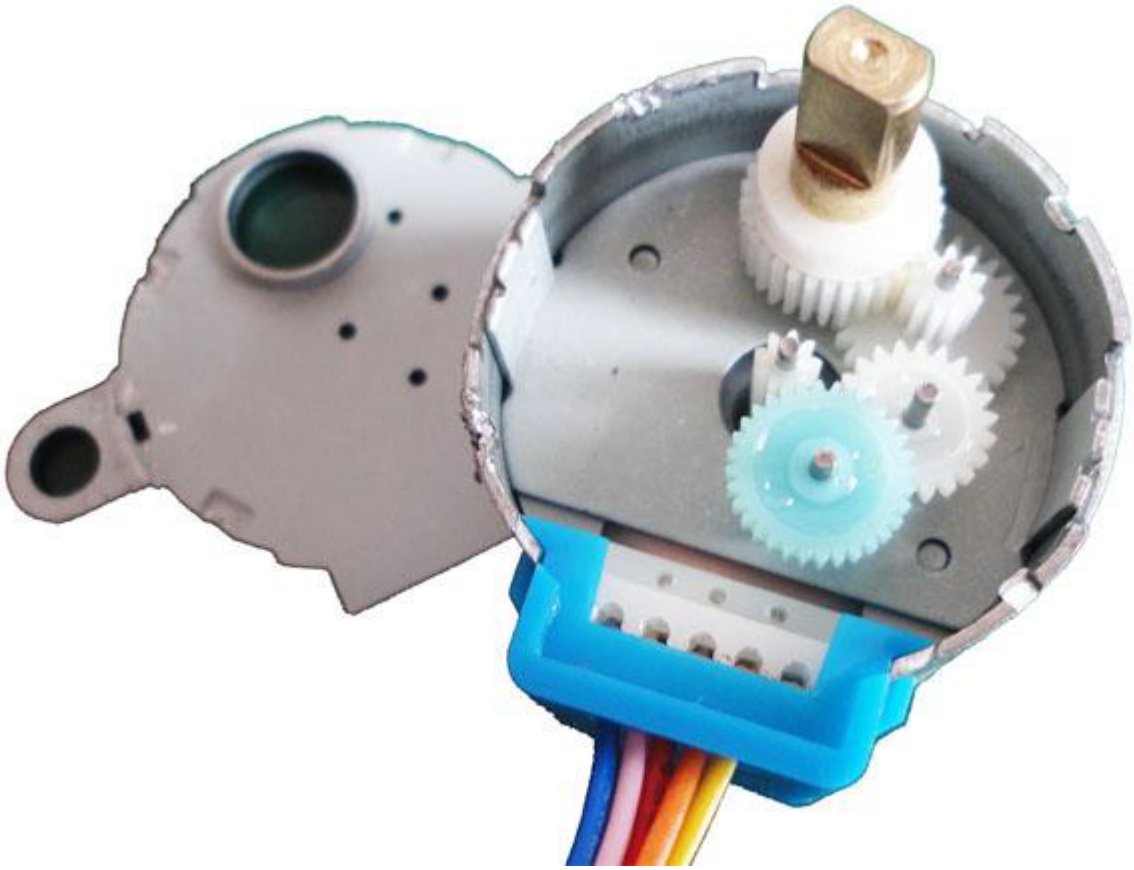

图9-7 步进电机内部齿轮示意图

回头看一下电机参数表中的减速比这个参数吧——1:64，转子转64圈，最终输出轴才会转一圈，也就是需要64*64=4096个节拍输出轴才转过一圈，2 ms*4096=8192 ms，8秒多才转一圈呢，是不是跟刚才的实验结果正好吻合

了？4096个节拍转动一圈，那么一个节拍转动的角度——步进角度就是360/4096，看一下表中的步进角度参数5.625/64，算一下就知道这两个值是相等的，一切都已吻合了。

关于基本的控制原理本该到这里就全部结束了，但是，我们希望大家都能培养一种“实践是检验真理的唯一标准”的思维方式！回想一下，步进电机最大的特点是什么？精确控制转动量！那么我们是不是应该检验一下它到底是不是能精确呢？精确到什么程度呢？怎么来检验呢？让它转过90度，然后量一下准不准？也行，但是如果它只差了1度甚至不到1度，你能准确测量出来吗？在没有精密仪器的情况很难。我们还是让它多转几个整圈，看看它最后停下的位置还是不是原来的位置。对应的，我们把程序修改一下，以方便控制电机转过任意的圈数。

```
 #include <reg52.h>

void TurnMotor(unsigned long angle);
void main(){
    TurnMotor(360*25);  //360度*25，即25圈
    while (1);
}
/* 软件延时函数，延时约 2 ms */
void delay(){
    unsigned int i = 200;
    while (i--);
}
/* 步进电机转动函数，angle-需转过的角度 */
void TurnMotor(unsigned long angle){
    unsigned char tmp;  //临时变量
    unsigned char index = 0;  //节拍输出索引
    unsigned long beats = 0;  //所需节拍总数
    //步进电机节拍对应的 IO 控制代码
    unsigned char code BeatCode[8] = {
        0xE, 0xC, 0xD, 0x9, 0xB, 0x3, 0x7, 0x6
    };
    //计算需要的节拍总数，4096拍对应一圈
    beats = (angle*4096) / 360;
    //判断 beats 不为0时执行循环，然后自减1
    while (beats--){
        tmp = P1;  //用 tmp 把 P1 口当前值暂存
        tmp = tmp & 0xF0;  //用&操作清零低4位
        tmp = tmp | BeatCode[index]; //用|操作把节拍代码写到低4位
        P1 = tmp;  //把低4位的节拍代码和高4位的原值送回 P1
        index++;  //节拍输出索引递增
        index = index & 0x07;  //用&操作实现到8归零
        delay();  //延时 2 ms，即 2 ms 执行一拍
    }
    P1 = P1 | 0x0F; //关闭电机所有的相
}
```

上述程序中，我们先编写了一个控制电机转过指定角度的函数，这个角度值由函数的形式参数给出，然后在主函数中就可以方便的通过更改调用时的实际参数来控制电机转过任意的角度了。我们用了360*25，也就是25圈，当然你也可以随意改为其它的值，看看是什么结果。我们的程序会执行25*8=200秒的时间，先记下输出轴的初始位置，然后上电并耐心等它执行完毕，看一下，是不是„„有误差？怎么回事，哪儿出问题了，不是说能精确控制转动量吗？

这个问题其实是出在了减速比上，再来看一下，厂家给出的减速比是1:64，不管是哪个厂家生产的电机，只要型号是 28BYJ-48，其标称的减速比就都是1:64。但实际上呢？经过我们的拆解计算发现：真实准确的减速比并不是这个值1:64，而是1:63.684！得出这个数据的方法也很简单，实际数一下每个齿轮的齿数，然后将各级减速比相乘，就可以得出结果了，实测的减速比为(32/9)*(22/11)*(26/9)*(31/10)≈63.684，从而得出实际误差为0.004 9，即约为百分之0.5，转100圈就会差出半圈，那么我们刚才转了25圈，是不是就差了八分之一圈了，也就是45度，看一下刚才的误差是45度吧。那么按照1:63.684 的实际减速比，可以得出转过一圈所需要节拍数是64*63.68 4≈4076。那么就把上面程序中电机驱动函数里的4096改成4076再试一下吧。是不是看不出丝毫的误差了？但实际上误差还是存在的，因为上面的计算结果都是约等得出的，实际误差大约是0.000056，即万分之0.56，转一万圈才会差出半圈，已经可以忽略不计了。

那么厂家的参数为什么会有误差呢？难道厂家不知道吗？要解释这个问题，我们得回到实际应用中，步进电机最通常的目的是控制目标转过一定的角度，通常都是在360度以内的，而这个 28BYJ-48 最初的设计目的是用来控制空调的扇叶的，扇叶的活动范围是不会超过180度的，所以在这种应用场合下，厂商给出一个近似的整数减速比1:64已经足够精确了，这也是合情合理的。然而，正如我们的程序那样，我们不一定是要用它来驱动空调扇叶，我们可以让它转动很多圈来干别的，这个时候就需要更为精确的数据了，这也是我们希望同学们都能了解并掌握的，就是说我们要能自己“设计”系统并解决其中发现的问题，而不要被所谓的“现成的方案”限制住思路。

9.7 28BYJ-48 步进电机控制程序基础

解决了精度问题，让我们再次回到我们的电机控制程序上吧。上面给出的两个例程都不是实用的程序，为什么？因为程序中存在大段的延时，而在延时的时候是什么其它的事都干不了的，想想第二个程序，整整200秒什么别的事都干不了，这在实际的控制系统中是绝对不允许的。那么怎么改造一下呢？当然还是用定时中断来完成了，既然每个节拍持续时间是 2 ms，那我们直接用定时器定时 2 ms 来刷新节拍就行了。改造后的程序如下：

```
 #include <reg52.h>

unsigned long beats = 0; //电机转动节拍总数
void StartMotor(unsigned long angle);
void main(){
    EA = 1;  //使能总中断
    TMOD = 0x01; //设置 T0 为模式1
    TH0 = 0xF8; //为 T0 赋初值 0xF8CD，定时 2 ms
    TL0 = 0xCD;
    ET0 = 1; //使能 T0 中断
    TR0 = 1; //启动 T0
    StartMotor(360*2+180); //控制电机转动2圈半
    while (1);
}
/* 步进电机启动函数，angle-需转过的角度 */
void StartMotor(unsigned long angle){
    //在计算前关闭中断，完成后再打开，以避免中断打断计算过程而造成错误
    EA = 0;
    beats = (angle * 4076) / 360; //实测为4076拍转动一圈
    EA = 1;
}
/* T0 中断服务函数，用于驱动步进电机旋转 */
void InterruptTimer0() interrupt 1{
    unsigned char tmp;  //临时变量
    static unsigned char index = 0; //节拍输出索引
    unsigned char code BeatCode[8] = { //步进电机节拍对应的 IO 控制代码
        0xE, 0xC, 0xD, 0x9, 0xB, 0x3, 0x7, 0x6
    };

    TH0 = 0xF8; //重新加载初值
    TL0 = 0xCD;
    //节拍数不为 0 则产生一个驱动节拍
    if (beats != 0){
        tmp = P1; //用 tmp 把 P1 口当前值暂存
        tmp = tmp & 0xF0; //用&操作清零低4位
```

```
        //用|操作把节拍代码写到低4位
        tmp = tmp | BeatCode[index];
        //把低4位的节拍代码和高4位的原值送回 P1
        P1 = tmp;
        index++; //节拍输出索引递增
        index = index & 0x07; //用&操作实现到8归零
        beats--; //总节拍数-1
    }else{ //节拍数为0则关闭电机所有的相
        P1 = P1 | 0x0F;
    }
}
```

程序还是比较简单的，电机转动的启动函数 StartMotor 只负责计算一个需要的总节拍数 beats，然后在中断函数内检测这个变量，不为0时就执行节拍操作，同时将其减1，直到减到0为止。

这里，我们要特别说明一下的是 StartMotor 函数中对 EA 的两次操作。我们可以看到对 beats 的赋值计算语句是夹在 EA=0;EA=1;这两行语句中间的，也就是说这行赋值计算语句在执行前先关闭了中断，而等它执行完后，才又重新打开了中断。在它执行过程中单片机是不会响应中断的，即中断函数 InterruptTimer0 不会被执行，即使这时候定时器溢出了，中断发生了，也只能等待 EA 重新置1后，才能得到响应，中断函数 InterruptTimer0 才会被执行。

那么为什么要这么做呢？我们来想一下：在本书开始我们就曾提到，我们所使用的 STC89C52 单片机是8位单片机，这个8位的概念就是说单片机操作数据时都是按8位即按1个字节进行的，那么要操作多个字节（不论是读还是写）就必须分多次进行了。而我们程序中定义的 beats 这个变量是 unsigned long 型，它要占用4个字节，那么对它的赋值最少也要分4次才能完成了。我们想象一下，假如在完成了其中第一个字节的赋值后，恰好中断发生了，InterruptTimer0 函数得到执行，而这个函数内可能会对 beats 进行减1的操作，减法就有可能发生借位，借位就会改变其它的字节，但因为此时其它的字节还没有被赋入新值，于是错误就会发生了，减1所得到的结果就不是预期的值了！所以要避免这种错误的发生就得先暂时关闭中断，等赋值完成后再打开中断。而如果我们使用的是 char 或 bit 型变量的话，因为它们都是在 CPU 的一次操作中就完成的，所以即使不关中断，也不会发生错误。问题分析清楚了，如何取舍还得根据实际情况来，遇上这类问题的时候多多考虑考虑吧。

9.8 实用的 28BYJ-48 步进电机控制程序

上面我们虽然完成了用中断控制电机转动的程序，但实际上这个程序还是没多少实用价值的，我们不能每次想让它转动的时候都上下电啊，是吧。还有就是它不但能正转还得能反转啊，也就是说不但能转过去，还得能转回来呀。好吧，我们就来做一个实例程序吧，结合第8章的按键程序，我们设计这样一个功能程序：按数字键1～9，控制电机转过1～9圈；配合上下键改变转动方向，按向上键后正向转1～9圈，向下键则反向转1～9圈；左键固定正转90度，右键固定反转90；Esc 键终止转动。通过这个程序，我们也可以进一步体会到如何用按键来控制程序完成复杂的功能，以及控制和执行模块之间如何协调工作，而你的编程水平也可以在这样的实践练习中得到锻炼和提升。

```
 #include <reg52.h>


sbit KEY_IN_1 = P2^4;
sbit KEY_IN_2 = P2^5;
sbit KEY_IN_3 = P2^6;
sbit KEY_IN_4 = P2^7;
sbit KEY_OUT_1 = P2^3;
sbit KEY_OUT_2 = P2^2;
sbit KEY_OUT_3 = P2^1;
sbit KEY_OUT_4 = P2^0;

unsigned char code KeyCodeMap[4][4] = { //矩阵按键编号到标准键盘键码的映射表
    { 0x31, 0x32, 0x33, 0x26 }, //数字键1、数字键2、数字键3、向上键
    { 0x34, 0x35, 0x36, 0x25 }, //数字键4、数字键5、数字键6、向左键
    { 0x37, 0x38, 0x39, 0x28 }, //数字键7、数字键8、数字键9、向下键
    { 0x30, 0x1B, 0x0D, 0x27 } //数字键0、ESC 键、 回车键、 向右键
};
unsigned char KeySta[4][4] = { //全部矩阵按键的当前状态
    {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}
};
signed long beats = 0; //电机转动节拍总数
void KeyDriver();

void main(){
    EA = 1; //使能总中断
    TMOD = 0x01; //设置 T0 为模式1
    TH0 = 0xFC; //为 T0 赋初值 0xFC67，定时 1 ms
    TL0 = 0x67;
    ET0 = 1; //使能 T0 中断
    TR0 = 1; //启动 T0
```

```
    while (1){
        KeyDriver(); //调用按键驱动函数
    }
}
/* 步进电机启动函数，angle-需转过的角度 */
void StartMotor(signed long angle){
    //在计算前关闭中断，完成后再打开，以避免中断打断计算过程而造成错误
    EA = 0;
    beats = (angle * 4076) / 360; //实测为4076拍转动一圈
    EA = 1;
}
/* 步进电机停止函数 */
void StopMotor(){
    EA = 0;
    beats = 0;
    EA = 1;
}
/* 按键动作函数，根据键码执行相应的操作，keycode-按键键码 */
void KeyAction(unsigned char keycode){
    static bit dirMotor = 0; //电机转动方向
    //控制电机转动 1-9 圈
    if ((keycode>=0x30) && (keycode<=0x39)){
        if (dirMotor == 0){
            StartMotor(360*(keycode-0x30));
        }else{
            StartMotor(-360*(keycode-0x30));
        }
    }else if (keycode == 0x26){ //向上键，控制转动方向为正转
        dirMotor = 0;
    }else if (keycode == 0x28){ //向下键，控制转动方向为反转
        dirMotor = 1;
    }else if (keycode == 0x25){ //向左键，固定正转90度
        StartMotor(90);
    }else if (keycode == 0x27){ //向右键，固定反转90度
        StartMotor(-90);
    }else if (keycode == 0x1B){ //Esc 键，停止转动
        StopMotor();
    }
}
/* 按键驱动函数，检测按键动作，调度相应动作函数，需在主循环中调用 */
void KeyDriver(){
    unsigned char i, j;
    static unsigned char backup[4][4] = { //按键值备份，保存前一次的值
        {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}
```

```
    };

    for (i=0; i<4; i++){ //循环检测4*4的矩阵按键
        for (j=0; j<4; j++){
            if (backup[i][j] != KeySta[i][j]){ //检测按键动作
                if (backup[i][j] != 0){ //按键按下时执行动作
                    KeyAction(KeyCodeMap[i][j]); //调用按键动作函数
                }
                backup[i][j] = KeySta[i][j]; //刷新前一次的备份值
            }
        }
    }
}
/* 按键扫描函数，需在定时中断中调用，推荐调用间隔 1 ms */
void KeyScan(){
    unsigned char i;
    static unsigned char keyout = 0; //矩阵按键扫描输出索引

    static unsigned char keybuf[4][4] = { //矩阵按键扫描缓冲区
        {0xFF, 0xFF, 0xFF, 0xFF}, {0xFF, 0xFF, 0xFF, 0xFF},
        {0xFF, 0xFF, 0xFF, 0xFF}, {0xFF, 0xFF, 0xFF, 0xFF}
    };

    //将一行的4个按键值移入缓冲区
    keybuf[keyout][0] = (keybuf[keyout][0] << 1) | KEY_IN_1;
    keybuf[keyout][1] = (keybuf[keyout][1] << 1) | KEY_IN_2;
    keybuf[keyout][2] = (keybuf[keyout][2] << 1) | KEY_IN_3;
    keybuf[keyout][3] = (keybuf[keyout][3] << 1) | KEY_IN_4;
    //消抖后更新按键状态
    for (i=0; i<4; i++){ //每行4个按键，所以循环4次
        if ((keybuf[keyout][i] & 0x0F) == 0x00){
            //连续4次扫描值为0，即 4*4 ms 内都是按下状态时，可认为按键已稳定的按下
            KeySta[keyout][i] = 0;
        }else if ((keybuf[keyout][i] & 0x0F) == 0x0F){
            //连续4次扫描值为1，即 4*4 ms 内都是弹起状态时，可认为按键已稳定的弹起
            KeySta[keyout][i] = 1;
        }
    }
    //执行下一次的扫描输出
    keyout++; //输出索引递增
    keyout = keyout & 0x03; //索引值加到4即归零
    //根据索引，释放当前输出引脚，拉低下次的输出引脚
    switch (keyout){
        case 0: KEY_OUT_4 = 1; KEY_OUT_1 = 0; break;
        case 1: KEY_OUT_1 = 1; KEY_OUT_2 = 0; break;
```

```
        case 2: KEY_OUT_2 = 1; KEY_OUT_3 = 0; break;
        case 3: KEY_OUT_3 = 1; KEY_OUT_4 = 0; break;
        default: break;
    }
}
/* 电机转动控制函数 */
void TurnMotor(){
    unsigned char tmp; //临时变量
    static unsigned char index = 0; //节拍输出索引
    unsigned char code BeatCode[8] = { //步进电机节拍对应的 IO 控制代码
        0xE, 0xC, 0xD, 0x9, 0xB, 0x3, 0x7, 0x6
    };

    if (beats != 0){ //节拍数不为0则产生一个驱动节拍
        if (beats > 0){ //节拍数大于0时正转
            index++; //正转时节拍输出索引递增
            index = index & 0x07; //用&操作实现到8归零
            beats--; //正转时节拍计数递减
            }else{ //节拍数小于0时反转
            index--; //反转时节拍输出索引递减
            index = index & 0x07; //用&操作同样可以实现到-1时归7
            beats++; //反转时节拍计数递增
        }
        tmp = P1; //用 tmp 把 P1 口当前值暂存
        tmp = tmp & 0xF0; //用&操作清零低4位
        tmp = tmp | BeatCode[index]; //用|操作把节拍代码写到低4位
        P1 = tmp; //把低4位的节拍代码和高4位的原值送回 P1
    }else{ //节拍数为0则关闭电机所有的相
        P1 = P1 | 0x0F;
    }
}
/* T0 中断服务函数，用于按键扫描与电机转动控制 */
void InterruptTimer0() interrupt 1{
    static bit div = 0;
    TH0 = 0xFC; //重新加载初值
    TL0 = 0x67;
    KeyScan(); //执行按键扫描
    //用一个静态 bit 变量实现二分频，即 2 ms 定时，用于控制电机
    div = ~div;
    if (div == 1){
        TurnMotor();
    }
}
```

这个程序是第8章和本章知识的一个综合——用按键控制步进电机转动。程序中有这么几点值得注意，我们分述如下： - 针对电机要完成正转和反转两个不同的操作，我们并没有使用正转启动函数和反转启动函数这么两个函数来完成，也没有在启动函数定义的时候增加一个形式参数来指明其方向。我们这里的启动函数 void StartMotor(signed long angle)与单向正转时的启动函数唯一的区别就是把形式参数 angle 的类型从 unsigned long 改为了 signed long，我们用有符号数固有的正负特性来区分正转与反转，正数表示正转 angle 度，负数就表示反转 angle 度，这样处理是不是很简洁又很明了呢？而你对有符号数和无符号数的区别用法是不是也更有体会了？ - 针对终止电机转动的操作，我们定义了一个单独的 StopMotor 函数来完成，尽管这个函数非常简单，尽管它也只在 Esc 按键分支内被调用了，但我们仍然把它单独提出来作为了一个函数。而这种做法就是基于这样一条编程原则：尽可能用单独的函数来完成硬件的某种操作，当一个硬件包含多个操作时，把这些操作函数组织在一起，形成一个对上层的统一接口。这样的层次化处理，会使得整个程序条理清晰，既有利于程序的调试维护，又有利于功能的扩充。 - 中断函数中要处理按键扫描和电机驱动两件事情，而为了避免中断函数过于复杂，我们就又分出了按键扫描和电机驱动两个函数（这也同样符合上述2的编程原则），而中断函数的逻辑就变得简洁而清晰了。这里还有个矛盾，就是按键扫描我们选择的定时时间是 1 ms，而本章之前的实例中电机节拍持续时间都是 2 ms；很显然，用 1 ms 的定时可以定出 2 ms 的间隔，而用 2 ms 的定时却得不到准确的 1 ms 间隔；所以我们的做法就是，定时器依然定时 1 ms，然后用一个 bit 变量做标志，每 1 ms 改变一次它的值，而我们只选择值为1的时候执行一次动作，这样就是 2 ms 的间隔了；如果我要 3 ms、4 ms„„呢，把 bit 改为 char 或 int 型，然后对它们递增，判断到哪个值该归零，就可以了。这就是在硬件定时器的基础上实现准确的软件定时，其实类似的操作我们在讲数码管的时候也用过了，回想一下吧。

9.9 单片机蜂鸣器控制程序和驱动电路

蜂鸣器从结构区分分为压电式蜂鸣器和电磁式蜂鸣器。压电式为压电陶瓷片发音，电流比较小一些，电磁式蜂鸣器为线圈通电震动发音，体积比较小。

按照驱动方式分为有源蜂鸣器和无源蜂鸣器。这里的有源和无源不是指电源，而是振荡源。有源蜂鸣器内部带了振荡源，如图9-8所示中，给了 BUZZ 引脚一个低电平，蜂鸣器就会直接响。而无源蜂鸣器内部是不带振荡源的，要让他响必须给 500 Hz～4.5 KHz 之间的脉冲频率信号来驱动它才会响。有源蜂鸣器往往比无源蜂鸣器贵一些，因为里边多了振荡电路，驱动发音也简单，靠电平就可以驱动，而无源蜂鸣器价格比较便宜，此外无源蜂鸣器声音频率可以控制，而音阶与频率又有确定的对应关系，因此就可以做出来“do re mi fa sol la si”的效果，可以用它制作出简单的音乐曲目，比如生日歌、两只老虎等等。

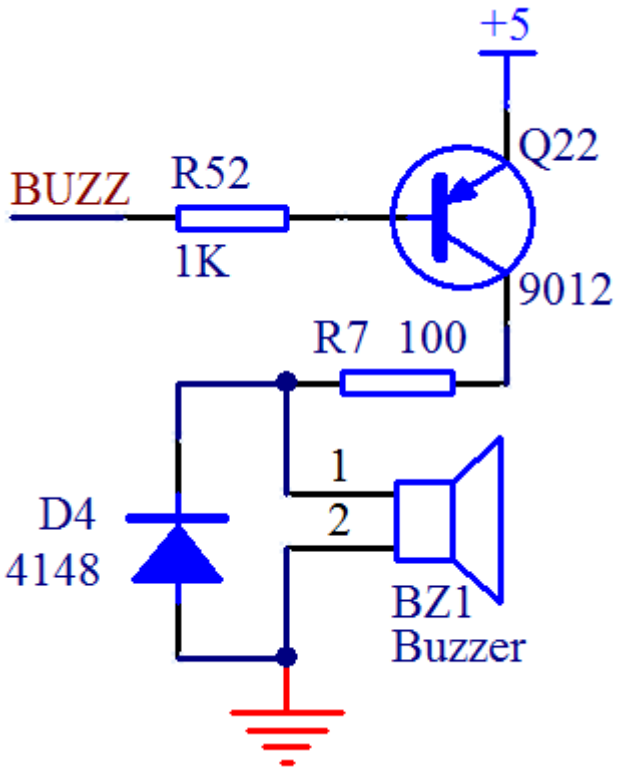

图 9-8 蜂鸣器电路原理图

我们来看一下图9-8的电路，蜂鸣器电流依然相对较大，因此需要用三极管驱动，并且加了一个100欧的电阻作为限流电阻。此外还加了一个 D4 二极管，这个二极管叫做续流二极管。我们的蜂鸣器是感性器件，当三极管导通给蜂鸣器供电时，就会有导通电流流过蜂鸣器。而我们知道，电感的一个特点就是电流不能突变，导通时电流是逐渐加大的，这点没有问题，但当关断时，经“电源-三极管-蜂鸣器-地”这条回路就截断了，过不了任何电流了，那么储存的电流往哪儿去呢，就是经过这个 D4 和蜂鸣器自身的环路来消耗掉了，从而就避免了关断时由于电感电流造成的反向冲击。接续关断时的电流，这就是续流二极管名称的由来。

蜂鸣器经常用于电脑、打印机、万用表这些设备上做提示音，提示音一般也很简单，就是简单发出个声音就行，我们用程序简单做了个 4 KHZ 频率下的发声和 1 KHZ 频率下的发声程序，同学们可以自己研究下程序，比较下实际效果。

```
 #include <reg52.h>

sbit BUZZ = P1^6; //蜂鸣器控制引脚
unsigned char T0RH = 0; //T0 重载值的高字节
unsigned char T0RL = 0; //T0 重载值的低字节
void OpenBuzz(unsigned int frequ);
void StopBuzz();

void main(){
    unsigned int i;
    TMOD = 0x01; //配置 T0 工作在模式1，但先不启动
    EA = 1;
    while (1){ //使能全局中断
        OpenBuzz(4000); //以 4 KHz 的频率启动蜂鸣器
        for (i=0; i<40000; i++);
        StopBuzz(); //停止蜂鸣器
        for (i=0; i<40000; i++);
        OpenBuzz(1000); //以 1 KHz 的频率启动蜂鸣器
        for (i=0; i<40000; i++);
        StopBuzz(); //停止蜂鸣器
        for (i=0; i<40000; i++);
    }
}
/* 蜂鸣器启动函数，frequ-工作频率 */
void OpenBuzz(unsigned int frequ){
    unsigned int reload;//计算所需的定时器重载值
    reload = 65536 - (11059200/12)/(frequ*2); //由给定频率计算定时器重载值
    T0RH = (unsigned char)(reload >> 8); //16 位重载值分解为高低两个字节
    T0RL = (unsigned char)reload;
    TH0 = 0xFF; //设定一个接近溢出的初值，以使定时器马上投入工作
    TL0 = 0xFE;
    ET0 = 1; //使能 T0 中断
    TR0 = 1; //启动 T0
}
/* 蜂鸣器停止函数 */
void StopBuzz(){
    ET0 = 0; //禁用 T0 中断
    TR0 = 0; //停止 T0
}
/* T0 中断服务函数，用于控制蜂鸣器发声 */
```

```
void InterruptTimer0() interrupt 1{
    TH0 = TORH; //重新加载重载值
    TL0 = TORL;
    BUZZ = ~BUZZ; //反转蜂鸣器控制电平
}
```

另外用蜂鸣器来输出音乐，仅仅是好玩而已，应用很少，里边包含了音阶、乐谱的相关内容，程序也有一点复杂，所以就不详细给大家去讲解了。仅提供一个可以播放《两只老虎》的程序，大家可以下载到板子上玩玩，满足一下好奇心。

```
 #include <reg52.h>

sbit BUZZ = P1^6; //蜂鸣器控制引脚
unsigned int code NoteFrequ[] = { //中音1-7 和高音1-7 对应频率列表
    523, 587, 659, 698, 784, 880, 988, //中音1-7
    1047, 1175, 1319, 1397, 1568, 1760, 1976 //高音1-7
};
unsigned int code NoteReload[] = { //中音1-7 和高音1-7 对应的定时器重载值
    65536 - (11059200/12) / (523*2), //中音1
    65536 - (11059200/12) / (587*2), //2
    65536 - (11059200/12) / (659*2), //3
    65536 - (11059200/12) / (698*2), //4
    65536 - (11059200/12) / (784*2), //5
    65536 - (11059200/12) / (880*2), //6
    65536 - (11059200/12) / (988*2), //7
    65536 - (11059200/12) / (1047*2), //高音1
    65536 - (11059200/12) / (1175*2), //2
    65536 - (11059200/12) / (1319*2), //3
    65536 - (11059200/12) / (1397*2), //4
    65536 - (11059200/12) / (1568*2), //5
    65536 - (11059200/12) / (1760*2), //6
    65536 - (11059200/12) / (1976*2), //7
};
bit enable = 1; //蜂鸣器发声使能标志
bit tmrflag = 0; //定时器中断完成标志
unsigned char TORH = 0xFF; //T0 重载值的高字节
unsigned char TORL = 0x00; //T0 重载值的低字节
void PlayTwoTiger();

void main(){
    unsigned int i;
    EA = 1; //使能全局中断
    TMOD = 0x01; //配置 T0 工作在模式1
    TH0 = TORH;
    TL0 = TORL;
```

```
    ET0 = 1; //使能 T0 中断
    TR0 = 1; //启动 T0

    while (1){
        PlayTwoTiger(); //播放乐曲--两支老虎
        for (i=0; i<40000; i++); //停止一段时间
    }
}
/* 两支老虎乐曲播放函数 */
void PlayTwoTiger(){
    unsigned char beat; //当前节拍索引
    unsigned char note; //当前节拍对应的音符
    unsigned int time = 0; //当前节拍计时
    unsigned int beatTime = 0; //当前节拍总时间
    unsigned int soundTime = 0; //当前节拍需发声时间
    //两只老虎音符表
    unsigned char code TwoTigerNote[] = {
    1,   2,   3, 1,    1,   2,   3, 1,     3, 4, 5,   3, 4, 5,
    5,6, 5,4, 3, 1,    5,6, 5,4, 3, 1,     1, 5, 1,   1, 5, 1,

    };
    //两只老虎节拍表，4 表示一拍，1 就是 1/4 拍，8 就是 2 拍
    unsigned char code TwoTigerBeat[] = {
    4,   4,   4, 4,   4,   4,   4, 4,     4, 4, 8,   4, 4, 8,
    3,1, 3,1, 4, 4,   3,1, 3,1, 4, 4,     4, 4, 8,   4, 4, 8,
    };

    //用节拍索引作为循环变量
    for (beat=0; beat<sizeof(TwoTigerNote); ){
        while (!tmrflag); //每次定时器中断完成后，检测并处理节拍
        tmrflag = 0;
        if (time == 0){ //当前节拍播完则启动一个新节拍
            note = TwoTigerNote[beat] - 1;
            T0RH = NoteReload[note] >> 8;
            T0RL = NoteReload[note];
            //计算节拍总时间，右移2位相当于除4，移位代替除法可以加快执行速度
            beatTime = (TwoTigerBeat[beat] * NoteFrequ[note]) >> 2;
            //计算发声时间，为总时间的 0.75，移位原理同上
            soundTime = beatTime - (beatTime >> 2);
            enable = 1; //指示蜂鸣器开始发声
            time++;
        }else{ //当前节拍未播完则处理当前节拍
            //当前持续时间到达节拍总时间时归零，
            //并递增节拍索引，以准备启动新节拍
            if (time >= beatTime){
```

```
                time = 0;
                beat++;
            }else{ //当前持续时间未达到总时间时，
                time++; //累加时间计数
                //到达发声时间后，指示关闭蜂鸣器，
                //插入 0.25*总时间的静音间隔，
                if (time == soundTime){
                    enable = 0; //用以区分连续的两个节拍
                }
            }
        }
    }
}
/* T0 中断服务函数，用于控制蜂鸣器发声 */
void InterruptTimer0() interrupt 1{
    TH0 = T0RH; //重新加载重载值
    TL0 = T0RL;
    tmrflag = 1;
    if (enable){ //使能时反转蜂鸣器控制电平
        BUZZ = ~BUZZ;
    }else{ //未使能时关闭蜂鸣器
        BUZZ = 1;
    }
}
```

4

10. 单片机实例练习与经验积累

本章内容主要通过一些实践例程，来提高大家对编程的熟练度，并且帮助大家进行一些算法和技巧上的积累。虽然是练习为主，但也涉及到了不少软硬件知识的学习，比如数据类型转换、中断响应延迟、位操作技巧、以及 PWM 的知识等。同学们在学习本章内容的时候，还是那句话，一定要能够达到不看教材，能够独立把程序做出来的效果，那样才能基本上掌握相关知识点和内容。

10.1 单片机数字秒表程序

不同数据类型间的相互转换

在 C 语言中，不同数据类型之间是可以混合运算的。当表达式中的数据类型不一致时，首先转换为同一种类型，然后再进行计算。C 语言有两种方法实现类型转换，一是自动类型转换，另外一种是强制类型转换。这块内容是比较繁杂的，因此我们根据常用的编程应用来讲部分相关内容。

当不同数据类型之间混合运算的时候，不同类型的数据首先会转换为同一类型，转换的主要原则是：短字节的数据向长字节数据转换。比如：

```
unsigned char a;
unsigned int b;
unsigned int c;
c = a *b;
```

在运算的过程中，程序会自动全部按照 unsigned int 型来计算。比如 a=10，b=200，c 的结果就是2000。那当a=100，b=700，那 c 是70000吗？新手最容易犯这种错误，大家要注意每个变量类型的取值范围，c 的数据类型是 unsigned int 型，取值范围是 0~65535，而70000超过 65535了，其结果会溢出，最终 c 的结果是(70000 - 65536) = 4464。

那要想让 c 正常获得70000这个结果，需要把 c 定义成一个 unsigned long 型。我们如果写成：

```
unsigned char a=100;
unsigned int b=700;
unsigned long c=0;
c = a*b;
```

有做过实验的同学，会发现这个 c 的结果还是4464，这个是个什么情况呢？

大家注意，C 语言不同类型运算的时候数值会转换同一类型运算，但是每一步运算都会进行识别判断，不会进行一个总的分析判断。比如我们这段代码中 a 和 b 相乘的时候，是按照 unsigned int 类型运算的，运算的结果也是 unsigned int 类型的4464，只是最终把 unsigned int 类型 4464赋值给了一个 unsigned long 型的变量而已。我们在运算的时候如何避免这类问题的产生呢？可以采用强制类型转换的方法。

在一个变量前边加上一个数据类型名，并且这个类型名用小括号括起来，就表示把这个变量强制转换成括号里的类型。如 c = (unsigned long)a * b;由于强制类型转换运算符优先级高于*，所以这个地方的运算是先把 a 转换成一个 unsigned long 型的变量，而后与 b 相乘，根据 C 语言的规则 b 会自动转换成一个 unsigned long 型的变量，而后运算完毕结果也是一个 unsigned long 型的，最终赋值给了 c。

不同类型变量之间的相互赋值，短字节类型变量向长字节类型变量赋值时，其值保持不变，比如：

```
unsigned char a=100;
unsigned int b=700;
b=a;
```

那么最终 b 的值就是100了。但是如果我们的程序是

```
unsigned char a=100;
unsigned int b=700;
a=b;
```

那么 a 的值仅仅是取了 b 的低8位，我们首先要把700变成一个16位的二进制数据，然后取它的低8位出来，也就是188，这就是长字节类型给短字节类型赋值的结果，会从长字节类型的低位开始截取刚好等于短字节类型长度的位，然后赋给短字节类型。

在51单片机里边，有一种特殊情况，就是 bit 类型的变量，这个 bit 类型的强制类型转换，是不符合上边讲的这个原则的，比如：

```
bit a=0;
unsigned char b;
a=(bit)b;
```

这个地方要特别注意，使用 bit 做强制类型转换，不是取 b 的最低位，而是它会判断 b 这个变量是0还是非0的值，如果 b 是0，那么 a 的结果就是0，如果 b 是任意非0的其它值，那么 a 的结果都是1。

定时时间精准性调整

在6.5.2章节有一个数码管秒表显示程序，那个程序是1秒数码管加1，但是细心的同学做了实验后，经过长时间运行会发现，和我们实际的时间有了较大误差了，那如何去调整这种误差呢？要解决问题，先找到问题是什么原因造成的。

先对我们前面讲过的中断内容做一个较深层次的补充。还是讲解中断的那个场景，当我们在看电视的时候，突然发生了水开的中断，我们必须去提水的时候，第一，我们从电视跟前跑到厨房需要一定的时间，第二，因为我们看的电视是智能数字电视，因此在去提水之前我们可以使用遥控器将我们的电视进行暂停操作，方便回来后继续从刚才的剧情往下进行。

那么暂停电视，跑到厨房提水，这一点点时间是很短的，在实际生活中可以忽略不计，但是在单片机秒表程序中，误差是会累计的，每1秒钟都差了几个微妙，时间一久，造成的累计误差就不可小觑了。

单片机系统里，硬件进入中断需要一定的时间，大概是几个机器周期，还要进行原始数据保护，就是把进中断之前程序运行的一些变量先保存起来，专业术语叫做中断压栈，进入中断后，重新给定时器 TH 和 TL 赋值，也需要几个机器周期，这样下来就会消耗一定的时间，我们得把这些时间补偿回来。

方法一，使用软件 debug 进行补偿。

我们在前边讲过使用 debug 来观察程序运行时间，那我们可以把我们2次进入中断的时间间隔观察出来，看看和我们实际定时的时间相差了几个机器周期，然后在进行定时器初值赋值的时候，进行一个调整。我们用的是 11.0592 M 的晶振，发现差了几个机器周期，就把定时器初值加上几个机器周期，这样就相当于进行了一个补偿。

方法二，使用累计误差计算出来。

有的时候，除了程序本身存在的误差外，硬件精度也可能会影响到时钟的精度，比如晶振，会随着温度变化出现温漂现象，就是实际值和标称值要差一点。那么我们还可以采取累计误差的方法来提高精度。比如我们可以让时钟运行半个小时或者一个小时，看看最终时间差了几秒，然后算算一共进了多少次定时器中断，把这差的几秒平均分配到每次的定时器中断中，就可以实现时钟的调整。

大家要明白，这个世界上本就没有绝对的精确，我们只能在一定程度上提高精确度，但是永远都不会使误差为零，如果在这个基础上还感觉精度不够的话，不要着急，后边我们会专门讲时钟芯片的，通常时钟芯片计时的精度比单片机的精度要高一些。

字节操作修改位的技巧

这里再介绍个编程小技巧，在编程时，有的情况下需要改变一个字节中的某一位或者几位，但是又不想改变其它位原有的值，该如何操作呢？

比如我们学定时器的时候遇到一个寄存器 TCON，这个寄存器是可以进行位操作的，可以直接写 TR0=1;TR0 是 TCON 的一个位，因为这个寄存器是允许位操作，这样写是没有任何问题的。还有一个寄存器 TMOD，这个寄存器是不支持位操作的，那如果我们要使用 T0 的模式1，我们希望达到的效果是 TMOD 的低4位是 0b0001，但如果我们直接写成 TMOD =0x01 的话，实际上已经同时操作到了高4位，即属于 T1 的部分，设置成了 0b0000，如果 T1 定时器没有用到的话，那我们随便怎么样都行，但是如果程序中既用到了 T0，又用到了 T1，那我们设置 T0 的同时已经干扰到了 T1 的模式配置，这是我们不希望看到的结果。

在这种情况下，就可以用我们前边学过的“&”和“|”运算了。对于二进制位操作来说，不管该位原来的值是0还是1，它跟0进行&运算，得到的结果都是0，而跟1进行&运算，将保持原来的值不变；不管该位原来的值是0还是1，它跟1进行|运算，得到的结果都是1，而跟0进行|运算，将保持原来的值不变。

利用上述这个规律，我们就可以着手解决刚才的问题了。如果我们现在要设置 TMOD 使定时器0工作在模式1下，又不干扰定时器1的配置，我们可以进行这样的操作：TMOD =TMOD & 0xF0; TMOD = TMOD | 0x01;第一步与 0xF0 做&运算后，TMOD 的高4位不变，低4位清零，变成了 0bxxxx0000；然后再进行第二步与 0x01 进行|运算，那么高7位均不变，最低位变成1了，这样就完成了只将低4位的值修改位 0b0001，而高4位保持原值不变的任务，即只设置了 T0 而不影响 T1。熟练掌握并灵活运用这个方法，会给你以后的编程带来便利。

另外，在 C 语言中，a &= b;等价于 a = a&b;同理，a |= b;等价于 a = a|b;那么刚才的一段代码就可以写成 TMOD &= 0xF0;TMOD |= 0x01 这样的简写形式。这种写法可以一定程度上简化代码，是 C 语言常用的一种编程风格。

数码管扫描函数算法改进

在学习数码管动态扫描的时候，为了方便大家理解，我们程序写的细致一些，给大家引入了 switch 的用法，随着编程能力与领悟能力的增强，对于 74HC138 这种非常有规律的数字器件，我们在编程上也可以改进一下逻辑算法，让程序变的更简洁。这种逻辑算法，通常不是靠学一下可以全部掌握的，而是通过不断的编写程序以及研究他人程序的过程中一点点积累起来的，从今天开始，大家就要开始积累吧。

前边动态扫描刷新函数我们是这么写的：

```
P0 = 0xFF;
switch (i){
    case 0: ADDR2=0; ADDR1=0; ADDR0=0; i++; P0=LedBuff[0]; break;
    case 1: ADDR2=0; ADDR1=0; ADDR0=1; i++; P0=LedBuff[1]; break;
    case 2: ADDR2=0; ADDR1=1; ADDR0=0; i++; P0=LedBuff[2]; break;
    case 3: ADDR2=0; ADDR1=1; ADDR0=1; i++; P0=LedBuff[3]; break;
    case 4: ADDR2=1; ADDR1=0; ADDR0=0; i++; P0=LedBuff[4]; break;
    case 5: ADDR2=1; ADDR1=0; ADDR0=1; i=0; P0=LedBuff[5]; break;
    default: break;
}
```

我们来分析每一个 case 分支，它们的结构是相同的，即改变 ADDR2～0、改变索引 i、取数据写入 P0，只要把 case 后的常量与 ADDR2～0 和 LedBuff 的下标对比，就可以发现它们其实是相等的，那么我们可以直接把常量值（实际上就是 i 在改变前的值）赋值给它们即可，而不必写上6遍。还剩下一个 i 的操作，它进行了5次相同的++与一次归0操作，那么很明显用++和 if 判断就可以替代这些操作。下面就是我们据此改进后的代码：

```
P0 = 0xFF;
P1 = (P1 & 0xF8) | i;
P0 = LedBuff[i];
if (i < 5){
    i++;
```

```
}else{
    i = 0;
}
```

大家看一下，P1 = (P1 & 0xF8) | i;这行代码就利用了上面讲到的&和|运算来将 i 的低3位直接赋值到 P1 口的低3位上，而 P0 的赋值也只需要一行代码，i 的处理也很简单。这样写成的代码是不是要简洁的多，也巧妙的多，而功能与前面的 switch 是一样的，同样可以完美实现动态显示刷新的功能。

秒表程序

做了一个秒表程序给同学们做参考，程序中涉及到的知识点我们都讲过了，包括了定时器、数码管、中断、按键等多个知识点。多知识点同时应用到一个程序中的小综合，因此需要大家完全消化掉。此程序是一个“真正的”并且“实用的”秒表程序，第一它有足够的分辨率，保留到小数点后两位，即每 10ms 计一次数，第二它也足够精确，因为我们补偿了定时器中断延时造成的误差，如果你愿意，它完全可以为用来测量你的百米成绩。这种小综合也是将来做大项目程序的基础，因此还是老规矩，大家边抄边理解，理解透彻后独立写出来就算此关通过。

```
 #include <reg52.h>

sbit ADDR3 = P1^3;
sbit ENLED = P1^4;
sbit KEY1 = P2^4;
sbit KEY2 = P2^5;
sbit KEY3 = P2^6;
sbit KEY4 = P2^7;
unsigned char code LedChar[] = { //数码管显示字符转换表
    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
    0x80, 0x90, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E
};
unsigned char LedBuff[6] = { //数码管显示缓冲区
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
};
unsigned char KeySta[4] = { //按键当前状态
    1, 1, 1, 1
};

bit StopwatchRunning = 0; //秒表运行标志
bit StopwatchRefresh = 1; //秒表计数刷新标志
unsigned char DecimalPart = 0; //秒表的小数部分
unsigned int IntegerPart = 0; //秒表的整数部分
unsigned char T0RH = 0; //T0 重载值的高字节
unsigned char T0RL = 0; //T0 重载值的低字节
```

```
void ConfigTimer0(unsigned int ms);
void StopwatchDisplay();
void KeyDriver();

void main(){
    EA = 1;  //开总中断
    ENLED = 0;  //使能选择数码管
    ADDR3 = 1;
    P2 = 0xFE; //P2.0 置0，选择第4行按键作为独立按键
    ConfigTimer0(2); //配置 T0 定时 2 ms
    while (1){
        if (StopwatchRefresh){ //需要刷新秒表示数时调用显示函数
            StopwatchRefresh = 0;
            StopwatchDisplay();
        }
        KeyDriver(); //调用按键驱动函数
    }
}
/* 配置并启动 T0，ms-T0 定时时间 */
void ConfigTimer0(unsigned int ms){
    unsigned long tmp; //临时变量
    tmp = 11059200 / 12; //定时器计数频率
    tmp = (tmp * ms) / 1000; //计算所需的计数值
    tmp = 65536 - tmp;  //计算定时器重载值
    tmp = tmp + 18; //补偿中断响应延时造成的误差
    T0RH = (unsigned char)(tmp>>8); //定时器重载值拆分为高低字节
    T0RL = (unsigned char)tmp;
    TMOD &= 0xF0; //清零 T0 的控制位
    TMOD |= 0x01; //配置 T0 为模式1
    TH0 = T0RH; //加载 T0 重载值
    TL0 = T0RL;
    ET0 = 1; //使能 T0 中断
    TR0 = 1; //启动 T0
}
/* 秒表计数显示函数 */
void StopwatchDisplay(){
    signed char i;
    unsigned char buf[4]; //数据转换的缓冲区
    //小数部分转换到低 2 位
    LedBuff[0] = LedChar[DecimalPart%10];
    LedBuff[1] = LedChar[DecimalPart/10];
    //整数部分转换到高 4 位
    buf[0] = IntegerPart%10;
    buf[1] = (IntegerPart/10)%10;
    buf[2] = (IntegerPart/100)%10;
```

```
    buf[3] = (IntegerPart/1000)%10;
    for (i=3; i>=1; i--){ //整数部分高位的0转换为空字符
        if (buf[i] == 0){
            LedBuff[i+2] = 0xFF;
        }else{
            break;
        }
    }
    for ( ; i>=0; i--){ //有效数字位转换为显示字符
        LedBuff[i+2] = LedChar[buf[i]];
    }
    LedBuff[2] &= 0x7F; //点亮小数点
}
/* 秒表启停函数 */
void StopwatchAction(){
    if (StopwatchRunning){ //已启动则停止
        StopwatchRunning = 0;
    }else{  //未启动则启动
        StopwatchRunning = 1;
    }
}
/* 秒表复位函数 */
void StopwatchReset(){
    StopwatchRunning = 0; //停止秒表
    DecimalPart = 0; //清零计数值
    IntegerPart = 0;
    StopwatchRefresh = 1; //置刷新标志
}
/* 按键驱动函数，检测按键动作，调度相应动作函数，需在主循环中调用 */
void KeyDriver(){
    unsigned char i;
    static unsigned char backup[4] = {1,1,1,1};

    for (i=0; i<4; i++){ //循环检测4个按键
        if (backup[i] != KeySta[i]){ //检测按键动作
            if (backup[i] != 0){ //按键按下时执行动作
                if (i == 1){ //Esc 键复位秒表
                    StopwatchReset();
                }else if (i == 2){//回车键启停秒表
                    StopwatchAction();
                }
            }
            backup[i] = KeySta[i]; //刷新前一次的备份值
        }
    }
```

```
}
/* 按键扫描函数，需在定时中断中调用 */
void KeyScan(){
    unsigned char i;
    static unsigned char keybuf[4] = { //按键扫描缓冲区
        0xFF, 0xFF, 0xFF, 0xFF
    };

    //按键值移入缓冲区
    keybuf[0] = (keybuf[0] << 1) | KEY1;
    keybuf[1] = (keybuf[1] << 1) | KEY2;
    keybuf[2] = (keybuf[2] << 1) | KEY3;
    keybuf[3] = (keybuf[3] << 1) | KEY4;
    //消抖后更新按键状态
    for (i=0; i<4; i++){
        if (keybuf[i] == 0x00){
            //连续8次扫描值为 0，即 16 ms 内都是按下状态时，可认为按键已稳定的按下
            KeySta[i] = 0;
        }else if (keybuf[i] == 0xFF){
            //连续8次扫描值为 1，即 16 ms 内都是弹起状态时，可认为按键已稳定的弹起
            KeySta[i] = 1;
        }
    }
}
/* 数码管动态扫描刷新函数，需在定时中断中调用 */
void LedScan(){
    static unsigned char i = 0; //动态扫描索引
    P0 = 0xFF; //关闭所有段选位，显示消隐
    P1 = (P1 & 0xF8) | i; //位选索引值赋值到 P1 口低3位
    P0 = LedBuff[i]; //缓冲区中索引位置的数据送到 P0 口
    if (i < 5){ //索引递增循环，遍历整个缓冲区
        i++;
    }else{
        i = 0;
    }
}
/* 秒表计数函数，每隔 10 ms 调用一次进行秒表计数累加 */
void StopwatchCount(){
    if (StopwatchRunning){ //当处于运行状态时递增计数值
        DecimalPart++; //小数部分+1
        if (DecimalPart >= 100){ //小数部分计到100时进位到整数部分
            DecimalPart = 0;
            IntegerPart++; //整数部分+1
            if (IntegerPart >= 10000){ //整数部分计到10000时归零
                IntegerPart = 0;
```

```
            }
        }
        StopwatchRefresh = 1; //设置秒表计数刷新标志
    }
}
/* T0 中断服务函数，完成数码管、按键扫描与秒表计数 */
void InterruptTimer0() interrupt 1{
    static unsigned char tmr10ms = 0;
    TH0 = T0RH; //重新加载重载值
    TL0 = T0RL;
    LedScan(); //数码管扫描显示
    KeyScan(); //按键扫描
    //定时 10ms 进行一次秒表计数
    tmr10ms++;
    if (tmr10ms >= 5){
        tmr10ms = 0;
        StopwatchCount(); //调用秒表计数函数
    }
}
```

关于这个程序有两点值得提一下：首先是定时器配置函数，虽然这样在程序里通过计算得出初值（重载值）增加了些许代码，但它换来的是便利性和编程效率，因为只要你完成这个函数，之后所有需要用定时器定时 x 毫秒的场合，你都可以直接把函数拿过去，用所需要的毫秒数作为实参调用它即可，不需要在用计算器埋头算一通了，是不是很值呢。其次是我们没有使用矩阵按键的程序，而是只用矩阵按键的第4行作为独立按键来使用，因为秒表只需要2个键就够了，这里是想告诉大家，处理问题要灵活，千万不能墨守成规，能用简单方法解决的问题，就不要选择复杂的方案。

10.2 单片机中 PWM 的原理与控制程序

PWM 在单片机中的应用是非常广泛的，它的基本原理很简单，但往往应用于不同场合上意义也不完全一样，这里我先把基本概念和基本原理给大家介绍一下，后边遇到用的时候起码知道是个什么东西。

PWM 是 Pulse Width Modulation 的缩写，它的中文名字是脉冲宽度调制，一种说法是它利用微处理器的数字输出来对模拟电路进行控制的一种有效的技术，其实就是使用数字信号达到一个模拟信号的效果。这是个什么概念呢？我们一步步来介绍。

首先从它的名字来看，脉冲宽度调制，就是改变脉冲宽度来实现不同的效果。我们先来看三组不同的脉冲信号，如图10-1所示。

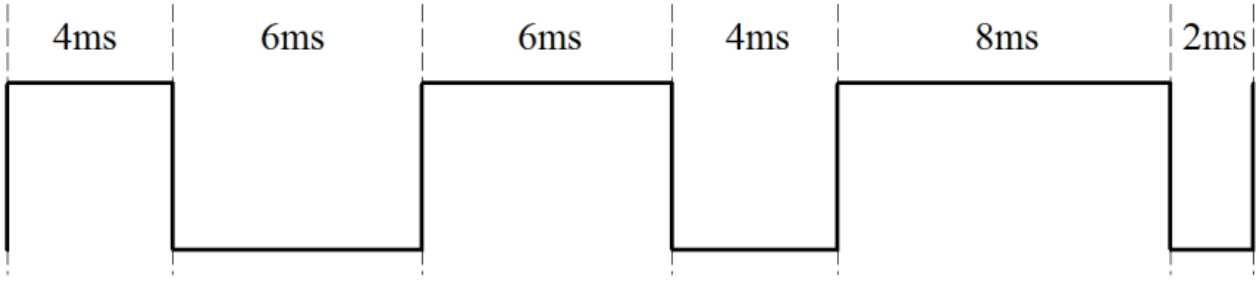

图10-1 PWM 波形

这是一个周期是 10 ms，即频率是 100 Hz 的波形，但是每个周期内，高低电平脉冲宽度各不相同，这就是 PWM 的本质。在这里大家要记住一个概念，叫做“占空比”。占空比是指高电平的时间占整个周期的比例。比如第一部分波形的占空比是40%，第二部分波形占空比是60%，第三部分波形占空比是80%，这就是 PWM 的解释。

那为何它能对模拟电路进行控制呢？大家想一想，我们数字电路里，只有0和1两种状态，比如我们第2章学会的点亮 LED 小灯那个程序，当我们写一个 LED = 0;小灯就会长亮，当我们写一个 LED = 1;小灯就会灭掉。当我们让小灯亮和灭间隔运行的时候，小灯是闪烁。

如果我们把这个间隔不断的减小，减小到我们的肉眼分辨不出来，也就是 100 Hz 以上的频率，这个时候小灯表现出来的现象就是既保持亮的状态，但亮度又没有 LED = 0;时的亮度高。那我们不断改变时间参数，让 LED = 0;的时间大于或者小于 LED = 1;的时间，会发现亮度都不一样，这就是模拟电路的感觉了，不再是纯粹的0和1，还有亮度不断变化。大家会发现，如果我们用 100 Hz 的信号，如图10-1所示，假如高电平熄灭小灯，低电平点亮小灯的话，第一部分波形熄灭 4 ms，点亮 6 ms，亮度最高，第二部分熄灭 6 ms，点亮 4 ms，亮度次之，第三部分熄灭 8 ms，点亮 2 ms，亮度最低。那么用程序验证一下我们的理论，我们用定时器 T0 定时改变 P0.0 的输出来实现 PWM，与纯定时不同的是，这里我们每周期内都要重载两次定时器初值，即用两个不同的初值来控制高低电平的不同持续时间。为了使亮度的变化更加明显，程序中使用的占空比差距更大。

```
 #include <reg52.h>

sbit PWMOUT = P0^0;
sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;

unsigned char HighRH = 0; //高电平重载值的高字节
unsigned char HighRL = 0; //高电平重载值的低字节
unsigned char LowRH = 0; //低电平重载值的高字节
unsigned char LowRL = 0; //低电平重载值的低字节

void ConfigPWM(unsigned int fr, unsigned char dc);
void ClosePWM();

void main(){
    unsigned int i;
    EA = 1; //开总中断
    ENLED = 0; //使能独立 LED
    ADDR3 = 1;
    ADDR2 = 1;
    ADDR1 = 1;
    ADDR0 = 0;

    while (1){
        ConfigPWM(100, 10); //频率 100 Hz，占空比 10%
        for (i=0; i<40000; i++);
        ClosePWM();
        ConfigPWM(100, 40); //频率 100 Hz，占空比 40%
        for (i=0; i<40000; i++);
        ClosePWM();
        ConfigPWM(100, 90); //频率 100 Hz，占空比 90%
        for (i=0; i<40000; i++);
        ClosePWM(); //关闭 PWM，相当于占空比100%
        for (i=0; i<40000; i++);
    }
}
/* 配置并启动 PWM，fr-频率，dc-占空比 */
void ConfigPWM(unsigned int fr, unsigned char dc){
    unsigned int high, low;
    unsigned long tmp;

    tmp = (11059200/12) / fr; //计算一个周期所需的计数值
```

```
    high = (tmp*dc) / 100; //计算高电平所需的计数值
    low = tmp - high; //计算低电平所需的计数值
    high = 65536 - high + 12; //计算高电平的重载值并补偿中断延时
    low = 65536 - low + 12;//计算低电平的重载值并补偿中断延时

    HighRH = (unsigned char)(high>>8); //高电平重载值拆分为高低字节
    HighRL = (unsigned char)high;
    LowRH = (unsigned char)(low>>8); //低电平重载值拆分为高低字节
    LowRL = (unsigned char)low;

    TMOD &= 0xF0; //清零 T0 的控制位
    TMOD |= 0x01; //配置 T0 为模式1
    TH0 = HighRH; //加载 T0 重载值
    TL0 = HighRL;
    ET0 = 1; //使能 T0 中断
    TR0 = 1; //启动 T0
    PWMOUT = 1; //输出高电平
}
/* 关闭 PWM */
void ClosePWM(){
    TR0 = 0; //停止定时器
    ET0 = 0; //禁止中断
    PWMOUT = 1; //输出高电平
}
/* T0 中断服务函数，产生 PWM 输出 */
void InterruptTimer0() interrupt 1{
    if (PWMOUT == 1){ //当前输出为高电平时，装载低电平值并输出低电平
        TH0 = LowRH;
        TL0 = LowRL;
        PWMOUT = 0;
    }else{ //当前输出为低电平时，装载高电平值并输出高电平
        TH0 = HighRH;
        TL0 = HighRL;
        PWMOUT = 1;
    }
}
```

需要提醒大家的是，由于标准51单片机中没有专门的 PWM 模块，所以我们用定时器加中断的方式来产生 PWM，而现在有很多的单片机都会集成硬件的 PWM 模块，这种情况下需要我们做的就仅仅是计算一下周期计数值和占空比计数值然后配置到相关的 SFR 中即可，既使程序得到了简化又确保了 PWM 的输出品质（因为消除了中断延时的影响）。

大家编译下载程序后，会发现小灯从最亮到灭一共4个亮度等级。如果我们让亮度等级更多，并且让亮度等级连续起来，会产生一个小灯渐变的效果，与呼吸有点类似，所以我们习惯上称之为呼吸灯，程序代码如下，这个程序

用了2个定时器2个中断，这是我们第一次这样用，大家可以学习一下。我们来试试这个程序，试完了大家一定要能自己把程序写出来，切记。

```
 #include <reg52.h>

sbit PWMOUT = P0^0;
sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;

unsigned long PeriodCnt = 0; //PWM 周期计数值
unsigned char HighRH = 0; //高电平重载值的高字节
unsigned char HighRL = 0; //高电平重载值的低字节
unsigned char LowRH = 0; //低电平重载值的高字节
unsigned char LowRL = 0; //低电平重载值的低字节
unsigned char T1RH = 0; //T1 重载值的高字节
unsigned char T1RL = 0; //T1 重载值的低字节

void ConfigTimer1(unsigned int ms);
void ConfigPWM(unsigned int fr, unsigned char dc);

void main(){
    EA = 1; //开总中断
    ENLED = 0; //使能独立 LED
    ADDR3 = 1;
    ADDR2 = 1;
    ADDR1 = 1;
    ADDR0 = 0;

    ConfigPWM(100, 10); //配置并启动 PWM
    ConfigTimer1(50); //用 T1 定时调整占空比
    while (1);
}
/* 配置并启动 T1，ms-定时时间 */
void ConfigTimer1(unsigned int ms){
    unsigned long tmp; //临时变量
    tmp = 11059200 / 12; //定时器计数频率
    tmp = (tmp * ms) / 1000; //计算所需的计数值
    tmp = 65536 - tmp; //计算定时器重载值
    tmp = tmp + 12; //补偿中断响应延时造成的误差
    T1RH = (unsigned char)(tmp>>8); //定时器重载值拆分为高低字节
    T1RL = (unsigned char)tmp;
    TMOD &= 0x0F; //清零 T1 的控制位
```

```
    TMOD |= 0x10; //配置 T1 为模式1
    TH1 = T1RH; //加载 T1 重载值
    TL1 = T1RL;
    ET1 = 1; //使能 T1 中断
    TR1 = 1; //启动 T1
}
/* 配置并启动 PWM，fr-频率，dc-占空比 */
void ConfigPWM(unsigned int fr, unsigned char dc){
    unsigned int high, low;
    PeriodCnt = (11059200/12) / fr; //计算一个周期所需的计数值
    high = (PeriodCnt*dc) / 100; //计算高电平所需的计数值
    low = PeriodCnt - high; //计算低电平所需的计数值
    high = 65536 - high + 12; //计算高电平的定时器重载值并补偿中断延时
    low = 65536 - low + 12; //计算低电平的定时器重载值并补偿中断延时
    HighRH = (unsigned char)(high>>8); //高电平重载值拆分为高低字节
    HighRL = (unsigned char)high;
    LowRH = (unsigned char)(low>>8); //低电平重载值拆分为高低字节
    LowRL = (unsigned char)low;

    TMOD &= 0xF0; //清零 T0 的控制位
    TMOD |= 0x01; //配置 T0 为模式1
    TH0 = HighRH; //加载 T0 重载值
    TL0 = HighRL;
    ET0 = 1; //使能 T0 中断
    TR0 = 1; //启动 T0
    PWMOUT = 1; //输出高电平
}
/* 占空比调整函数，频率不变只调整占空比 */
void AdjustDutyCycle(unsigned char dc){
    unsigned int high, low;
    high = (PeriodCnt*dc) / 100; //计算高电平所需的计数值
    low = PeriodCnt - high; //计算低电平所需的计数值
    high = 65536 - high + 12; //计算高电平的定时器重载值并补偿中断延时
    low = 65536 - low + 12; //计算低电平的定时器重载值并补偿中断延时

    HighRH = (unsigned char)(high>>8); //高电平重载值拆分为高低字节
    HighRL = (unsigned char)high;
    LowRH = (unsigned char)(low>>8); //低电平重载值拆分为高低字节
    LowRL = (unsigned char)low;
}
/* T0 中断服务函数，产生 PWM 输出 */
void InterruptTimer0() interrupt 1{
    if (PWMOUT == 1){ //当前输出为高电平时，装载低电平值并输出低电平
        TH0 = LowRH;
        TL0 = LowRL;
```

```
        PWMOUT = 0;
    }else{ //当前输出为低电平时，装载高电平值并输出高电平
        TH0 = HighRH;
        TL0 = HighRL;
        PWMOUT = 1;
    }
}
/* T1 中断服务函数，定时动态调整占空比 */
void InterruptTimer1() interrupt 3{
    static bit dir = 0;
    static unsigned char index = 0;
    unsigned char code table[13] = { //占空比调整表
        5, 18, 30, 41, 51, 60, 68, 75, 81, 86, 90, 93, 95
    };

    TH1 = T1RH; //重新加载 T1 重载值
    TL1 = T1RL;
    AdjustDutyCycle(table[index]); //调整 PWM 的占空比
    if (dir == 0){ //逐步增大占空比
        index++;
        if (index >= 12){
            dir = 1;
        }
    }else{ //逐步减小占空比
        index--;
        if (index == 0){
            dir = 0;
        }
    }
}
```

呼吸灯效果做出来后，利用这个基本原理，其它各种效果的灯光闪烁都应该可以做出来，大家看到的 KTV 里边那绚丽的灯光闪烁，其实就是采用的 PWM 技术控制的。

10.3 单片机交通灯控制程序和设计原理

同学们在学习技术的时候，一定要多动脑筋，遇到问题后，三思而后问。有些时候你考虑的和真理就差一点点了，没有坚持下去，别人告诉你后才恍然大悟。这样得到的结论，可以让你学到知识，但是却培养不了你的逻辑思维能力。不是不能问，而是要在认真思考的基础上再发问。

有同学有疑问，板子上只有8个流水灯，那如果我要做很多个流水灯一起花样显示怎么办呢？那我们在讲课的时候其实都提到过了，板子上是有8个流水灯，还有6个数码管，还有1个点阵 LED，一个数码管相当于8个小灯，一个点阵相当于64个小灯，那如果全部算上的话，我们板子上实际共接了8+6*8+64=120个小灯，你如果单独只接小灯，花样灯就做出来了。

还有同学问，板子上流水灯和数码管可以一起工作吗？如何一起工作呢？我们刚说了，一个数码管是8个小灯，但是大家反过来想一想，8个流水灯不也就是相当于一个数码管吗。那板子上6个数码管我们可以让他们同时亮，7个数码管就不会了吗？当然了，思考的习惯是要慢慢培养的，想不到的同学继续努力，每天前进一小步，坚持一段时间后回头看看，就会发现你学会了很多。

我们做了一个交通灯的程序给大家做学习参考。因为板子资源有限，所以我把左边 LED8 和 LED9 一起亮作为绿灯，把中间 LED5 和 LED6 一起亮作为黄灯，把右边 LED2 和 LED3 一起亮作为红灯，用数码管的低2位做倒计时，让 LED 和数码管同时参与工作。程序并不复杂，也没有什么新知识点，大家完全可以自己分析了，然后下载编译试试看吧。

```
 #include <reg52.h>

sbit ADDR3 = P1^3;
sbit ENLED = P1^4;
unsigned char code LedChar[] = { //数码管显示字符转换表
    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
    0x80, 0x90, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E
};
unsigned char LedBuff[7] = { //数码管+独立 LED 显示缓冲区
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
};
bit flag1s = 1; //1 秒定时标志
unsigned char T0RH = 0; //T0 重载值的高字节
unsigned char T0RL = 0; //T0 重载值的低字节

void ConfigTimer0(unsigned int ms);
void TrafficLight();

void main(){
```

```
    EA = 1; //开总中断
    ENLED = 0; //使能数码管和 LED
    ADDR3 = 1;
    ConfigTimer0(1); //配置 T0 定时 1 ms

    while (1){
        if (flag1s){ //每秒执行一次交通灯刷新
            flag1s = 0;
            TrafficLight();
        }
    }
}
/* 配置并启动 T0，ms-T0 定时时间 */
void ConfigTimer0(unsigned int ms){
    unsigned long tmp; //临时变量
    tmp = 11059200 / 12; //定时器计数频率
    tmp = (tmp * ms) / 1000; //计算所需的计数值
    tmp = 65536 - tmp; //计算定时器重载值
    tmp = tmp + 13; //补偿中断响应延时造成的误差
    T0RH = (unsigned char)(tmp>>8); //定时器重载值拆分为高低字节
    T0RL = (unsigned char)tmp;

    TMOD &= 0xF0; //清零 T0 的控制位
    TMOD |= 0x01; //配置 T0 为模式1
    TH0 = T0RH; //加载 T0 重载值
    TL0 = T0RL;
    ET0 = 1; //使能 T0 中断
    TR0 = 1; //启动 T0
}
/* 交通灯显示刷新函数 */
void TrafficLight(){
static unsigned char color = 2; //颜色索引：0-绿色/1-黄色/2-红色
static unsigned char timer = 0; //倒计时定时器

if (timer == 0){ //倒计时到0时，切换交通灯
    switch (color){ //LED8/9 代表绿灯，LED5/6 代表黄灯，LED2/3 代表红灯
        case 0: //切换到黄色，亮3秒
            color = 1;
            timer = 2;
            LedBuff[6] = 0xE7;
            break;

        case 1: //切换到红色，亮30秒
            color = 2;
            timer = 29;
```

```
            LedBuff[6] = 0xFC;
            break;

        case 2: //切换到绿色，亮40秒
            color = 0;
            timer = 39;
            LedBuff[6] = 0x3F;
            break;
            default:
            break;
    }
}else{ //倒计时未到0时，递减其计数值
    timer--;
}
    LedBuff[0] = LedChar[timer%10]; //倒计时数值个位显示
    LedBuff[1] = LedChar[timer/10]; //倒计时数值十位显示
}
/* LED 动态扫描刷新函数，需在定时中断中调用 */
void LedScan(){
    static unsigned char i = 0; //动态扫描索引
    P0 = 0xFF; //关闭所有段选位，显示消隐
    P1 = (P1 & 0xF8) | i; //位选索引值赋值到 P1 口低3位
    P0 = LedBuff[i]; //缓冲区中索引位置的数据送到 P0 口
    if (i < 6){ //索引递增循环，遍历整个缓冲区
        i++;
    }else{
        i = 0;
    }
}
/* T0 中断服务函数，完成 LED 扫描和秒定时 */
void InterruptTimer0() interrupt 1{
    static unsigned int tmr1s = 0; //1秒定时器
    TH0 = T0RH; //重新加载重载值
    TL0 = T0RL;
    LedScan(); //LED 扫描显示
    tmr1s++; //1秒定时的处理

    if (tmr1s >= 1000){
        tmr1s = 0;
        flag1s = 1; //设置秒定时标志
    }
}
```

10.4 51单片机 RAM 区域的划分

前边介绍单片机资源的时候，我们提到过 STC89C52 共有512字节的 RAM，是用来保存数据的，比如我们定义的变量都是直接存在 RAM 里边的。但是单片机的这512字节的 RAM 在地位上并不都是平等的，而是分块的，块与块之间在物理结构和用法上都是有区别的，因此我们在使用的时候，也要注意一些问题。

51单片机的 RAM 分为两个部分，一块是片内 RAM，一块是片外 RAM。标准51的片内 RAM 地址从 0x00H～0x7F 共128个字节，而现在我们用的51系列的单片机都是带扩展片内 RAM 的，即 RAM 是从 0x00～0xFF 共256个字节。片外 RAM 最大可以扩展到 0x0000～0xFFFF 共 64 K 字节。这里有一点大家要明白，片内 RAM 和片外 RAM 的地址不是连起来的，片内是从 0x00 开始，片外也是从 0x0000 开始的。还有一点，片内和片外这两个名词来自于早期的51单片机，分别指在芯片内部和芯片外部，但现在几乎所有的51单片机芯片内部都是集成了片外 RAM 的，而真正的芯片外扩展则很少用到了，虽然它还叫片外 RAM，但实际上它现在也是在单片机芯片内部的，我们的 STC89C52 就是这样。以下是几个 Keil C51 语言中的关键字，代表了 RAM 不同区域的划分，大家先记一下。data：片内 RAM 从 0x00~0x7F idata：片内 RAM 从 0x00~0xFF pdata：片外 RAM 从 0x00~0xFF xdata：片外 RAM 从 0x0000~0xFFFF

大家可以看出来，data 是 idata 的一部分，pdata 是 xdata 的一部分。为什么还这样去区分呢？因为 RAM 分块的访问方式主要和汇编指令有关，因此这块内容大家了解一下即可，只需要记住如何访问速度更快就行了。

我们定义一个变量 a，可以这样：unsigned char data a=0，而我们前边定义变量时都没有加 data 这个关键字，是因为在 Keil 默认设置下，data 是可以省略的，即什么都不加的时候变量就是定义到 data 区域中的。data 区域 RAM 的访问在汇编语言中用的是直接寻址，执行速度是最快的。如果你定义成 idata，不仅仅可以访问 data 区域，还可以访问 0x80H~0xFF 的范围，但加了 idata 关键字后，访问的时候51单片机用的是通用寄存器间接寻址，速度较 data 会慢一些，而且我们平时大多数情况下不太希望访问到 0x80H~0xFF，因为这块通常用于中断与函数调用的堆栈，所以在绝大多数情况下，我们使用内部 RAM 的时候，只用 data 就可以了。

对于外部 RAM 来说，使用 pdata 定义的变量存到了外部 RAM 的 0x00~0xFF 的地址范围内，这块地址的访问和 idata 类似，都是用通用寄存器间接寻址，而如果你定义成 xdata，可以访问的范围更广泛，从0到 64 K 的地址都可以访问到，但是它需要使用2个字节寄存器 DPTRH 和 DPTRL 来进行间接寻址，速度是最慢的。

我们的 STC89C52 共有512字节的 RAM，分为256字节的片内 RAM 和256字节的片外 RAM。一般情况下，我们是使用 data 区域，data 不够用了，我们就用 xdata，如果希望程序执行效率尽量高一点，就使用 pdata 关键字来定义。其它型号有更大的 RAM 的51系列单片机，如果要使用更大的 RAM，就必须得用 xdata 来访问了。

10.5 单片机长短按键的应用

在单片机系统中应用按键的时候，如果只需要按下一次按键加1或减1，那用第8章学到的知识就可以完成了，但如果想连续加很多数字的时候，要一次次按下这个按键确实有点不方便，这时我们会希望一直按住按键，数字就自动持续增加或减小，这就是所谓的长短按键应用。

当检测到一个按键产生按下动作后，马上执行一次相应的操作，同时在程序里记录按键按下的持续时间，该时间超过1秒后（主要是为了区别短按和长按这两个动作，因短按的时间通常都达到几百 ms），每隔 200 ms（如果你需要更快那就用更短的时间，反之亦然）就自动再执行一次该按键对应的操作，这就是一个典型的长按键效果。

对此，我们做了一个模拟定时炸弹效果的实例，提供给大家作为参考。打开开关后，数码管显示数字0，按向上的按键数字加1，按向下的按键数字减1，长按向上按键1秒后，数字会持续增加，长按向下按键1秒后，数字会持续减小。设定好数字后，按下回车按键，时间就会进行倒计时，当倒计时到0的时候，用蜂鸣器和板子上的8个 LED 小灯做炸弹效果，蜂鸣器持续响，LED 小灯全亮。

```
 #include <reg52.h>


sbit BUZZ = P1^6;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;
sbit KEY_IN_1 = P2^4;
sbit KEY_IN_2 = P2^5;
sbit KEY_IN_3 = P2^6;
sbit KEY_IN_4 = P2^7;
sbit KEY_OUT_1 = P2^3;
sbit KEY_OUT_2 = P2^2;
sbit KEY_OUT_3 = P2^1;
sbit KEY_OUT_4 = P2^0;

unsigned char code LedChar[] = { //数码管显示字符转换表
    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
    0x80, 0x90, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E
};
unsigned char LedBuff[7] = { //数码管+独立 LED 显示缓冲区
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
};
unsigned char code KeyCodeMap[4][4] = { //矩阵按键编号到标准键盘键码的映射表
    { 0x31, 0x32, 0x33, 0x26 }, //数字键1、数字键2、数字键3、向上键
    { 0x34, 0x35, 0x36, 0x25 }, //数字键4、数字键5、数字键6、向左键
    { 0x37, 0x38, 0x39, 0x28 }, //数字键7、数字键8、数字键9、向下键
```

```
    { 0x30, 0x1B, 0x0D, 0x27 } //数字键0、ESC 键、 回车键、 向右键
}
unsigned char KeySta[4][4] = { //全部矩阵按键的当前状态
    {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}
};
unsigned long pdata KeyDownTime[4][4] = { //每个按键按下的持续时间，单位 ms
    {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}
};

bit enBuzz = 0; //蜂鸣器使能标志
bit flag1s = 0; //1秒定时标志
bit flagStart = 0; //倒计时启动标志
unsigned char T0RH = 0; //T0 重载值的高字节
unsigned char T0RL = 0; //T0 重载值的低字节
unsigned int CountDown = 0; //倒计时计数器

void ConfigTimer0(unsigned int ms);
void ShowNumber(unsigned long num);
void KeyDriver();

void main(){
    EA = 1; //使能总中断
    ENLED = 0; //选择数码管和独立 LED
    ADDR3 = 1;
    ConfigTimer0(1); //配置 T0 定时 1ms
    ShowNumber(0); //上电显示0

    while (1){
        KeyDriver(); //调用按键驱动函数
        if (flagStart && flag1s){ //倒计时启动且1秒定时到达时，处理倒计时
            flag1s = 0;
            if (CountDown > 0){ //倒计时未到0时，计数器递减
                CountDown--;
                ShowNumber(CountDown); //刷新倒计时数显示
                if (CountDown == 0){ //减到0时，执行声光报警
                    enBuzz = 1;
                    //启动蜂鸣器发声
                    LedBuff[6] = 0x00; //点亮独立 LED
                }
            }
        }
    }
}
/* 配置并启动 T0，ms-T0 定时时间 */
void ConfigTimer0(unsigned int ms){
```

```
    unsigned long tmp; //临时变量
    tmp = 11059200 / 12; //定时器计数频率
    tmp = (tmp * ms) / 1000; //计算所需的计数值
    tmp = 65536 - tmp; //计算定时器重载值
    tmp = tmp + 28; //补偿中断响应延时造成的误差
    T0RH = (unsigned char)(tmp>>8); //定时器重载值拆分为高低字节
    T0RL = (unsigned char)tmp;
    TMOD &= 0xF0; //清零 T0 的控制位
    TMOD |= 0x01; //配置 T0 为模式1
    TH0 = T0RH; //加载 T0 重载值
    TL0 = T0RL;
    ET0 = 1; //使能 T0 中断
    TR0 = 1; //启动 T0
}
    /* 将一个无符号长整型的数字显示到数码管上，num-待显示数字 */
void ShowNumber(unsigned long num){
    signed char i;
    unsigned char buf[6];
    for (i=0; i<6; i++){ //把长整型数转换为6位十进制的数组
        buf[i] = num % 10;
        num = num / 10;
    }
    for (i=5; i>=1; i--){ //从最高位起，遇到0转换为空格，遇到非0则退出循环
        if (buf[i] == 0){
            LedBuff[i] = 0xFF;
        }else{
            break;
        }
    }

    for ( ; i>=0; i--){ //剩余低位都如实转换为数码管显示字符
        LedBuff[i] = LedChar[buf[i]];
    }
}
/* 按键动作函数，根据键码执行相应的操作，keycode-按键键码 */
void KeyAction(unsigned char keycode){ //按键动作函数，根据键码执行相应动作
    if (keycode == 0x26){ //向上键，倒计时设定值递增
        if (CountDown < 9999){ //最大计时9999秒
            CountDown++;
            ShowNumber(CountDown);
        }
    }else if (keycode == 0x28){ //向下键，倒计时设定值递减
        if (CountDown > 1){ //最小计时1秒
            CountDown--;
            ShowNumber(CountDown);
```

```
        }
    }else if (keycode == 0x0D){ //回车键，启动倒计时
        flagStart = 1; //启动倒计时
    }else if (keycode == 0x1B){ //Esc 键，取消倒计时
        enBuzz = 0; //关闭蜂鸣器
        LedBuff[6] = 0xFF; //关闭独立 LED
        flagStart = 0; //停止倒计时
        CountDown = 0; //倒计时数归零
        ShowNumber(CountDown);
    }
}
/* 按键驱动函数，检测按键动作，调度相应动作函数，需在主循环中调用 */
void KeyDriver(){
    unsigned char i, j;
    static unsigned char pdata backup[4][4] = { //按键值备份，保存前一次的值
        {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}, {1, 1, 1, 1}
    };
    static unsigned long pdata TimeThr[4][4] = { //快速输入执行的时间阈值
        {1000, 1000, 1000, 1000}, {1000, 1000, 1000, 1000},
        {1000, 1000, 1000, 1000}, {1000, 1000, 1000, 1000}
    };

    for (i=0; i<4; i++){ //循环扫描 4*4 的矩阵按键
        for (j=0; j<4; j++){
            if (backup[i][j] != KeySta[i][j]){ //检测按键动作
                if (backup[i][j] != 0){ //按键按下时执行动作
                    KeyAction(KeyCodeMap[i][j]); //调用按键动作函数
                }
            }
            backup[i][j] = KeySta[i][j];
            if (KeyDownTime[i][j] > 0){
                if (KeyDownTime[i][j] >= TimeThr[i][j]){ //达到阈值时执行一次动作
                    KeyAction(KeyCodeMap[i][j]); //调用按键动作函数
                    TimeThr[i][j] += 200; //时间阈值增加 200 ms，以准备下次执行
                }
            }else{ //按键弹起时复位阈值时间
                TimeThr[i][j] = 1000; //恢复 1 s 的初始阈值时间
            }
        }
    }
}
/* 按键扫描函数，需在定时中断中调用 */
void KeyScan(){
    unsigned char i;
    static unsigned char keyout = 0; //矩阵按键扫描输出索引
```

```
    static unsigned char keybuf[4][4] = { //矩阵按键扫描缓冲区
        {0xFF, 0xFF, 0xFF, 0xFF}, {0xFF, 0xFF, 0xFF, 0xFF},
        {0xFF, 0xFF, 0xFF, 0xFF}, {0xFF, 0xFF, 0xFF, 0xFF}
    };

    //将一行的4个按键值移入缓冲区
    keybuf[keyout][0] = (keybuf[keyout][0] << 1) | KEY_IN_1;
    keybuf[keyout][1] = (keybuf[keyout][1] << 1) | KEY_IN_2;
    keybuf[keyout][2] = (keybuf[keyout][2] << 1) | KEY_IN_3;
    keybuf[keyout][3] = (keybuf[keyout][3] << 1) | KEY_IN_4;
    //消抖后更新按键状态
    for (i=0; i<4; i++){ //每行4个按键，所以循环4次
        //连续 4 次扫描值为 0，即 4*4 ms 内都是按下状态时，可认为按键已稳定的按下
        if ((keybuf[keyout][i] & 0x0F) == 0x00){
            KeySta[keyout][i] = 0;
            KeyDownTime[keyout][i] += 4; //按下的持续时间累加
        //连续4次扫描值为1，即 4*4 ms 内都是弹起状态时，可认为按键已稳定的弹起
        }else if ((keybuf[keyout][i] & 0x0F) == 0x0F){
            KeySta[keyout][i] = 1;
        }
    }
    KeyDownTime[keyout][i] = 0; //按下的持续时间清零
    //执行下一次的扫描输出
    keyout++; //输出索引递增
    keyout &= 0x03; //索引值加到4即归零
    switch (keyout){ //根据索引，释放当前输出引脚，拉低下次的输出引脚
        case 0: KEY_OUT_4 = 1; KEY_OUT_1 = 0; break;
        case 1: KEY_OUT_1 = 1; KEY_OUT_2 = 0; break;
        case 2: KEY_OUT_2 = 1; KEY_OUT_3 = 0; break;
        case 3: KEY_OUT_3 = 1; KEY_OUT_4 = 0; break;
        default: break;
    }
}
/* LED 动态扫描刷新函数，需在定时中断中调用 */
void LedScan(){
    static unsigned char i = 0; //动态扫描索引
    P0 = 0xFF; //关闭所有段选位，显示消隐
    P1 = (P1 & 0xF8) | i; //位选索引值赋值到 P1 口低3位
    P0 = LedBuff[i]; //缓冲区中索引位置的数据送到 P0 口
    if (i < 6){ //索引递增循环，遍历整个缓冲区
        i++;
    }else{
        i = 0;
    }
```

```
}
/* T0 中断服务函数，完成数码管、按键扫描与秒定时 */
void InterruptTimer0() interrupt 1{
    static unsigned int tmr1s = 0; //1秒定时器
    TH0 = T0RH; //重新加载重载值
    TL0 = T0RL;
    if (enBuzz){ //蜂鸣器发声处理
        BUZZ = ~BUZZ; //驱动蜂鸣器发声
    }else{
        BUZZ = 1; //关闭蜂鸣器
    }
    LedScan(); //LED 扫描显示
    KeyScan(); //按键扫描
    if (flagStart){ //倒计时启动时处理1秒定时
        tmr1s++;
        if (tmr1s >= 1000){
            tmr1s = 0;
            flag1s = 1;
        }
    }else{ //倒计时未启动时1秒定时器始终归零
        tmr1s = 0;
    }
}
```

长按键功能实现的重点有两个：第一，是在原来的矩阵按键扫描函数 KeyScan 内，当检测到按键按下后，持续的对一个时间变量进行累加，其目的是用这个时间变量来记录按键按下的时间；第二，是在按键驱动函数 KeyDriver 里，除了原来的检测到按键按下这个动作时执行按键动作函数 KeyAction 外，还监测表示按键按下时间的变量，根据它的值来完成长按时的连续快速按键动作功能。

5

11. UART 串口通信

通信，按照传统的理解就是信息的传输与交换。对于单片机来说，通信则与传感器、存储芯片、外围控制芯片等技术紧密结合，成为整个单片机系统的“神经中枢”。没有通信，单片机所实现的功能仅仅局限于单片机本身，就无法通过其它设备获得有用信息，也无法将自己产生的信息告诉其它设备。如果单片机通信没处理好的话，它和外围器件的合作程度就受到限制，最终整个系统也无法完成强大的功能，由此可见单片机通信技术的重要性。UART（Universal Asynchronous Receiver/Transmitter，即通用异步收发器）串行通信是单片机最常用的一种通信技术，通常用于单片机和电脑之间以及单片机和单片机之间的通信。

11.1 单片机串行通信介绍

通信按照基本类型可以分为并行通信和串行通信。并行通信时数据的各个位同时传送，可以实现字节为单位通信，但是通信线多占用资源多，成本高。比如我们前边用到的 P0 = 0xFE;一次给 P0 的8个 IO 口分别赋值，同时进行信号输出，类似于有8个车道同时可以过去8辆车一样，这种形式就是并行的，我们习惯上还称 P0、P1、P2 和 P3 为51单片机的4组并行总线。

而串行通信，就如同一条车道，一次只能一辆车过去，如果一个 0xFE 这样一个字节的数据要传输过去的话，假如低位在前高位在后的话，那发送方式就是0-1-1-1-1-1-1-1-1，一位一位的发送出去的，要发送8次才能发送完一个字节。

STC89C52 有两个引脚是专门用来做 UART 串行通信的，一个是 P3.0 一个是 P3.1，它们还分别有另外的名字叫做 RXD 和 TXD，由它们组成的通信接口就叫做串行接口，简称串口。用两个单片机进行 UART 串口通信，基本的演示图如图11-1所示。

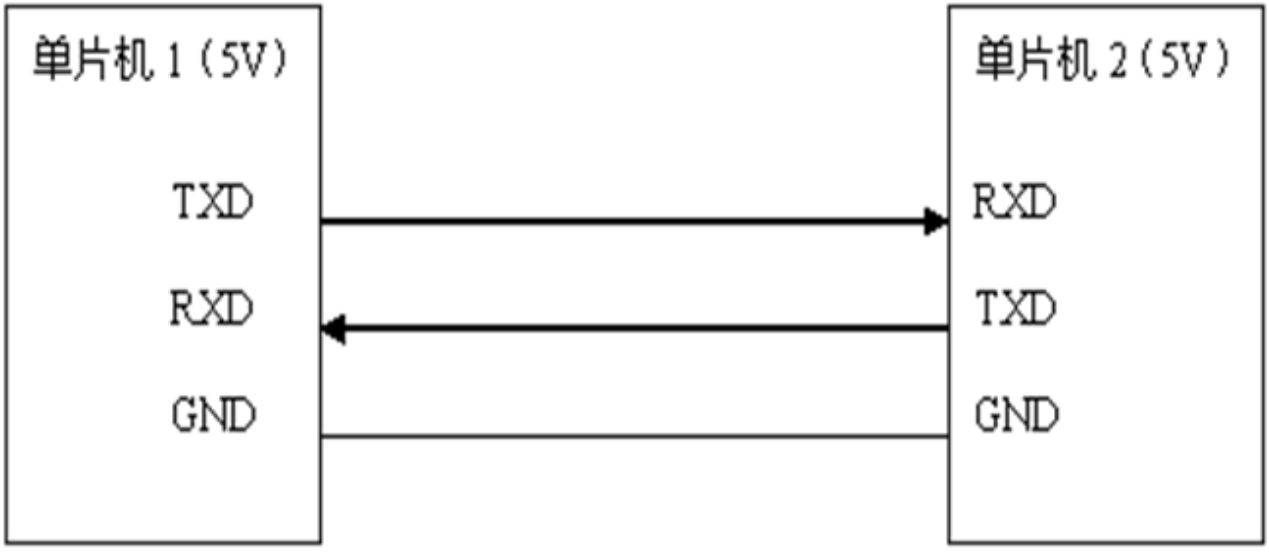

图11-1 单片机之间 UART 通信示意图

图中，GND 表示单片机系统电源的参考地，TXD 是串行发送引脚，RXD 是串行接收引脚。两个单片机之间要通信，首先电源基准得一样，所以我们要把两个单片机的 GND 相互连接起来，然后单片机1的 TXD 引脚接到单片机2的 RXD 引脚上，即此路为单片机1发送而单片机2接收的通道，单片机1的 RXD 引脚接到单片机2的 TXD 引脚上，即此路为单片机2发送而单片机1接收的通道。这个示意图就体现了两个单片机相互收发信息的过程。

当单片机1想给单片机2发送数据时，比如发送一个 0xE4 这个数据，用二进制形式表示就是 0b11100100，在 UART 通信过程中，是低位先发，高位后发的原则，那么就让 TXD 首先拉低电平，持续一段时间，发送一位0，然后继续拉低，再持续一段时间，又发送了一位0，然后拉高电平，持续一段时间，发了一位1,,,,一直到把8位二进制数

字 0b11100100 全部发送完毕。这里就涉及到了一个问题，就是持续的这“一段时间”到底是多久？由此便引入了通信中的一个重要概念——波特率，也叫做比特率。

波特率就是发送二进制数据位的速率，习惯上用 baud 表示，即我们发送一位二进制数据的持续时间=1/baud。在通信之前，单片机1和单片机2首先都要明确的约定好它们之间的通信波特率，必须保持一致，收发双方才能正常实现通信，这一点大家一定要记清楚。

约定好速度后，我们还要考虑第二个问题，数据什么时候是起始，什么时候是结束呢？

不管是提前接收还是延迟接收，数据都会接收错误。在 UART 通信的时候，一个字节是8位，规定当没有通信信号发生时，通信线路保持高电平，当要发送数据之前，先发一位0表示起始位，然后发送8位数据位，数据位是先低后高的顺序，数据位发完后再发一位1表示停止位。这样本来要发送一个字节的8位数据，而实际上我们一共发送了10位，多出来的两位其中一位起始位，一位停止位。而接收方呢，原本一直保持的高电平，一旦检测到了一位低电平，那就知道了要开始准备接收数据了，接收到8位数据位后，然后检测到停止位，再准备下一个数据的接收。我们图示看一下，如图11-2所示。

图11-2 串口数据发送示意图

图11-2串口数据发送示意图，实际上是一个时域示意图，就是信号随着时间变化的对应关系。比如在单片机的发送引脚上，左边的是先发生的，右边的是后发生的，数据位的切换时间就是波特率分之一秒，如果能够理解时域的概念，后边很多通信的时序图就很容易理解了。

11.2 RS232 通信接口

在我们的台式电脑上，一般都会有一个9针的串行接口，这个串行接口叫做 RS232 接口，它和 UART 通信有关联，但是由于现在笔记本电脑都不带这种9针串口了，所以和单片机通信越来越趋向于使用 USB 虚拟的串口，因此这一节的内容作为了解内容，大家知道有这么回事就行了。

我们先来认识一下这个标准串口，在物理结构上分为9针的和9孔的，习惯上我们也称之为公头和母头，如图11-3所示。

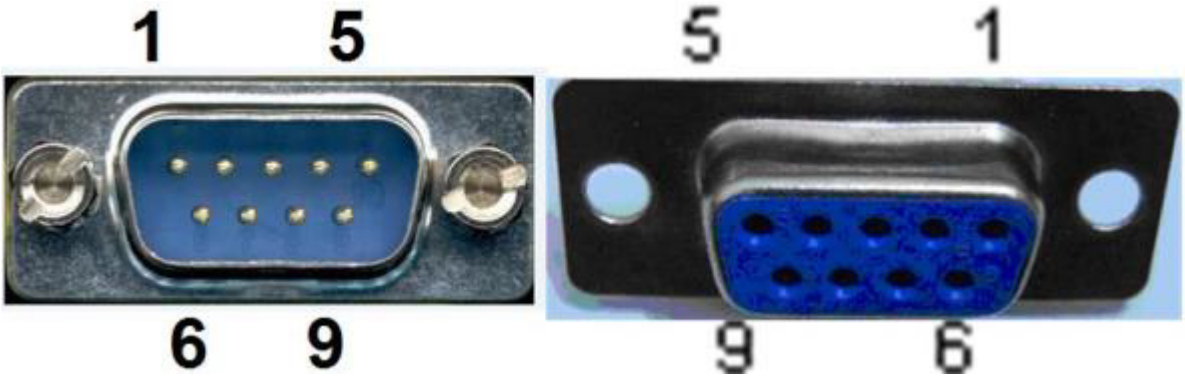

图11-3 RS232 通信接口

RS232 接口一共有9个引脚，分别定义是：1、载波检测 DCD；2、接收数据 RXD；3、发送数据 TXD；4、数据终端准备好 DTR；5、信号地线 SG；6、数据准备好 DSR；7、请求发送 RTS；8、清除发送 CTS；9、振铃提示 RI。我们要让这个串口和我们单片机进行通信，我们只需要关心其中的2脚 RXD、3脚 TXD 和5脚 GND 即可。

虽然这三个引脚的名字和我们单片机上的串口名字一样，但是却不能直接和单片机对连通信，这是为什么呢？随着我们了解的内容越来越多，我们得慢慢知道，不是所有的电路都是 5 V 代表高电平而 0 V 代表低电平的。对于 RS232 标准来说，它是个反逻辑，也叫做负逻辑。为何叫负逻辑？它的 TXD 和 RXD 的电压，-3 V～-15 V 电压代表是1，+3 V～+15 V 电压代表是0。低电平代表的是1，而高电平代表的是 0，所以称之为负逻辑。因此电脑的9针 RS232 串口是不能和单片机直接连接的，需要用一个电平转换芯片 MAX232 来完成，如图11-4所示。

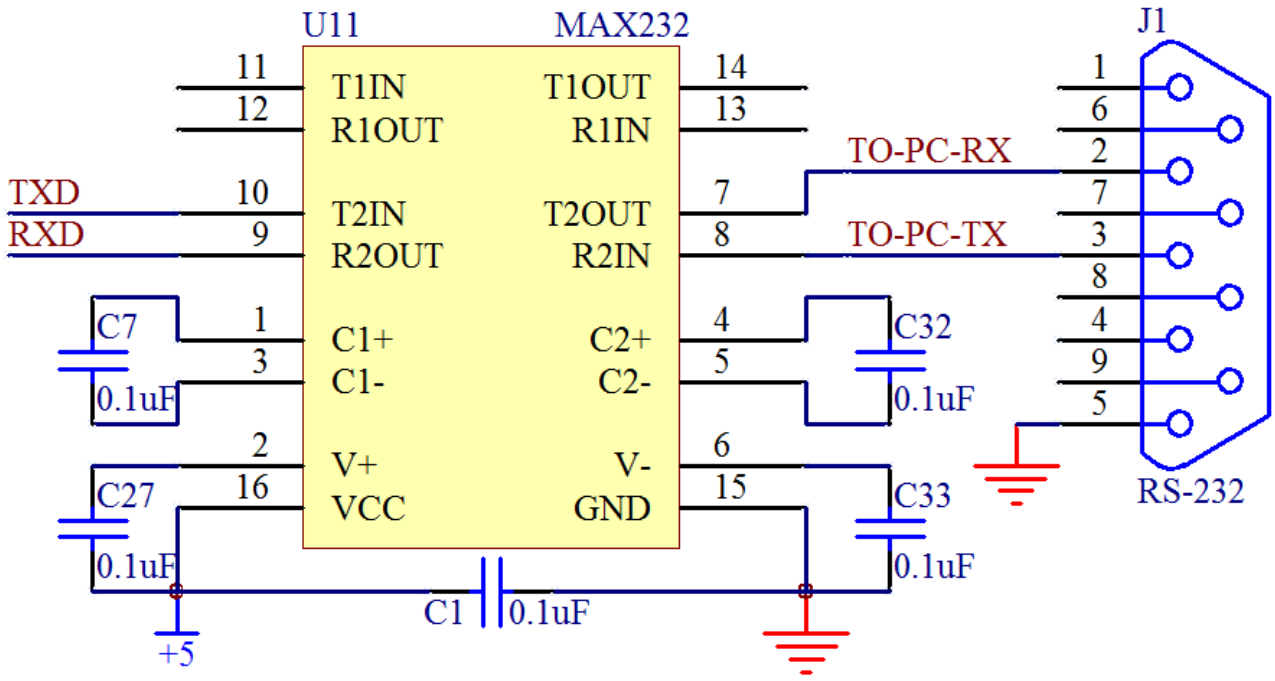

图11-4 MAX232 转接图

这个芯片就可以实现把标准 RS232 串口电平转换成我们单片机能够识别和承受的 UART 0 V/5 V 电平。从这里大家似乎慢慢有点明白了，其实 RS232 串口和 UART 串口，它们的协议类型是一样的，只是电平标准不同而已，而 MAX232 这个芯片起到的就是中间人的作用，它把 UART 电平转换成 RS232 电平，也把 RS232 电平转换成 UART 电平，从而实现标准 RS232 接口和单片机 UART 之间的通信连接。

11.3 USB 转串口通信

随着技术的发展，工业上还有 RS232 串口通信的大量使用，但是商业技术的应用上，已经慢慢的使用 USB 转 UART 技术取代了 RS232 串口，绝大多数笔记本电脑已经没有串口这个东西了，那我们要实现单片机和电脑之间的通信该怎么办呢？

我们只需要在电路上添加一个 USB 转串口芯片，就可以成功实现 USB 通信协议和标准 UART 串行通信协议的转换，在我们的开发板上，我们使用的是 CH340T 这个芯片，如图11-5所示。

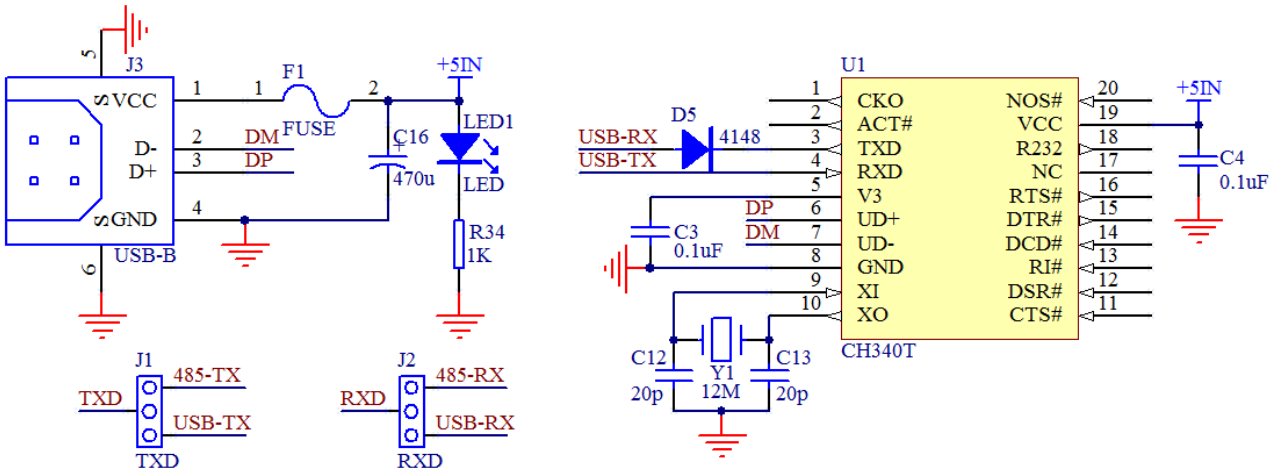

图11-5 USB 转串口电路

图中左下方 J1 和 J2 是两个跳线的组合，大家可以在我们板子左下方的位置找到，我们需要用跳线帽把中间和下边的针短接在一起。右侧的 CH340T 这个电路很简单，把电源、晶振接好后，6脚和7脚的 DP 和 DM 分别接 USB 口的2个数据引脚上去，3脚和4脚通过跳线接到了我们单片机的 TXD 和 RXD 上去。

CH340T 的电路里3脚位置加了个4148的二极管，是一个小技巧。因为 STC89C52 这个单片机下载程序时需要冷启动，就是先点下载后上电，上电瞬间单片机会先检测需要不需要下载程序。虽然单片机的 VCC 是由开关来控制，但是由于 CH340T 的3脚是输出引脚，如果没有此二极管，开关后级单片机在断电的情况下，CH340T 的3脚和单片机的 P3.0（即 RXD）引脚连在一起，有电流会通过这个引脚流入后级电路并且给后级的电容充电，造成后级有一定幅度的电压，这个电压值虽然只有两三伏左右，但是可能会影响到正常的冷启动。加了二极管后，一方面不影响通信，另外一个方面还可以消除这种不良影响。这个地方可以暂时作为了解，大家如果自己做这类电路，可以参考一下。

11.4 单片机 IO 口模拟 UART 串口通信

为了让大家充分理解 UART 串口通信的原理，我们先把 P3.0 和 P3.1 当做 IO 口来进行模拟实际串口通信的过程，原理搞懂后，我们再使用寄存器配置实现串口通信过程。

对于 UART 串口波特率，常用的值是300、600、1200、2400、4800、9600、14400、19200、28800、38400、57600、115200等速率。IO 口模拟 UART 串行通信程序是一个简单的演示程序，我们使用串口调试助手下发一个数据，数据加1后，再自动返回。

串口调试助手，这里我们直接使用 STC-ISP 软件自带的串口调试助手，先把串口调试助手的使用给大家说一下，如图11-6所示。第一步要选择串口助手菜单，第二步选择十六进制显示，第三步选择十六进制发送，第四步选择 COM 口，这个 COM 口要和自己电脑设备管理器里的那个 COM 口一致，波特率按我们程序设定好的选择，我们程序中让一个数据位持续时间是1/9600秒，那这个地方选择波特率就是选9600，校验位选 N，数据位8，停止位1。

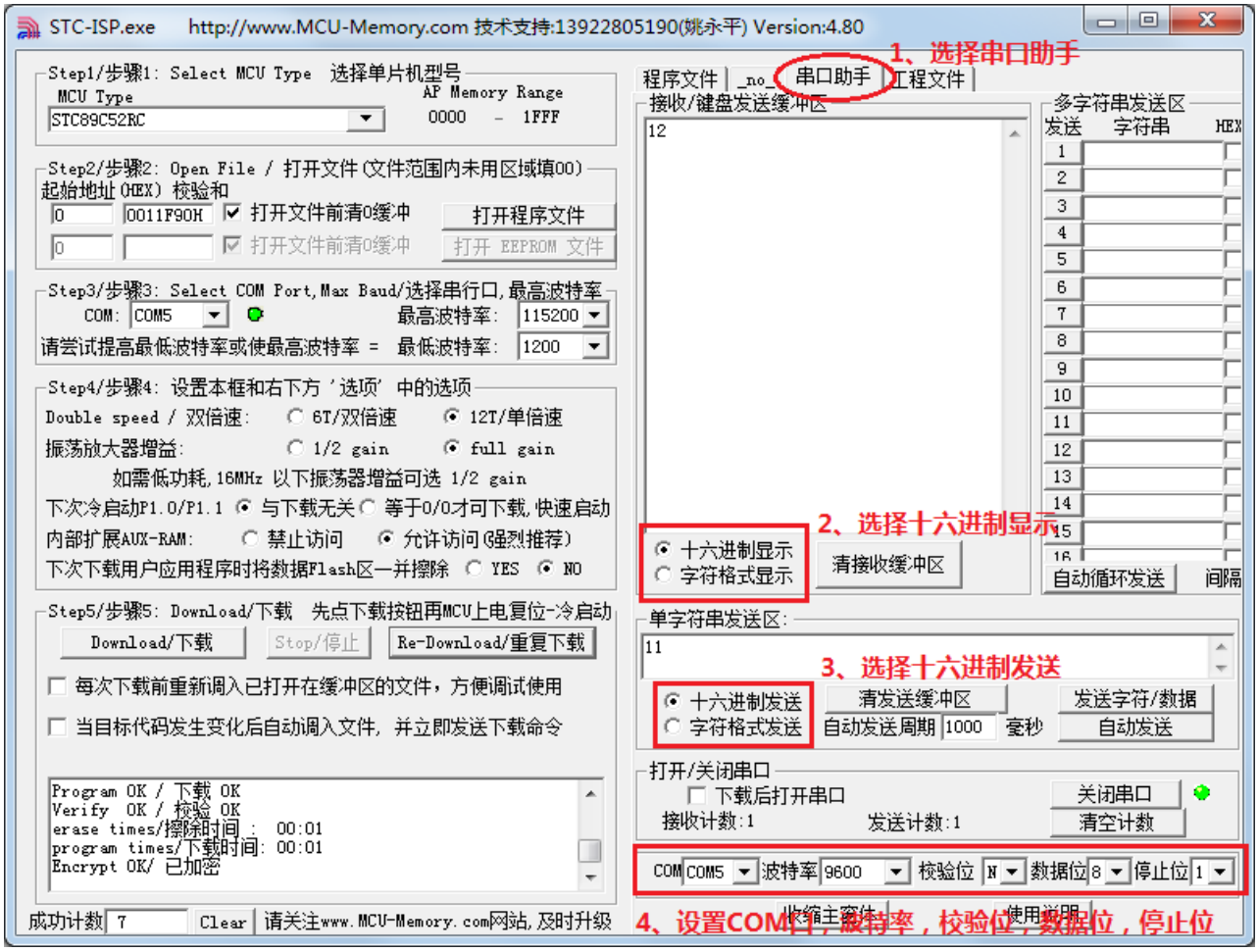

图11-6 串口调试助手示意图

串口调试助手的实质就是利用电脑上的 UART 通信接口，发送数据给我们的单片机，也可以把我们的单片机发送的数据接收到这个调试助手界面上。

因为初次接触通信方面的技术，所以我把后面的 IO 模拟串口通信程序进行一下解释，大家可以边看我的解释边看程序，把底层原理先彻底弄懂。

变量定义部分就不用说了，直接看 main 主函数。首先是对通信的波特率的设定，在这里我们配置的波特率是9600，那么串口调试助手也得是9600。配置波特率的时候，我们用的是定时器 T0 的模式2。模式2中，不再是 TH0 代表高8位，TL0 代表低8位了，而只有 TL0 在进行计数，当 TL0 溢出后，不仅仅会让 TF0 变1，而且还会将 TH0 中的内容重新自动装到 TL0 中。这样有一个好处，就是我们可以把想要的定时器初值提前存在 TH0 中，当 TL0 溢出后，TH0 自动把初值就重新送入 TL0 了，全自动的，不需要程序中再给 TL0 重新赋值了，配置方式很简单，大家可以自己看下程序并且计算一下初值。

波特率设置好以后，打开中断，然后等待接收串口调试助手下发的数据。接收数据的时候，首先要进行低电平检测 while (PIN_RXD)，若没有低电平则说明没有数据，一旦检测到低电平，就进入启动接收函数 StartRXD()。接收函数最开始启动半个波特率周期，初学可能这里不是很明白。大家回头看一下我们的图11-2里边的串口数据示意图，如果在数据位电平变化的时候去读取，因为时序上的误差以及信号稳定性的问题很容易读错数据，所以我们希望在信号最稳定的时候去读数据。除了信号变化的那个沿的位置外，其它位置都很稳定，那么我们现在就约定在信号中间位置去读取电平状态，这样能够保证我们读的一定是正确的。

一旦读到了起始信号，我们就把当前状态设定成接收状态，并且打开定时器中断，第一次是半个周期进入中断后，对起始位进行二次判断一下，确认一下起始位是低电平，而不是一个干扰信号。以后每经过1/9600秒进入一次中断，并且把这个引脚的状态读到 RxdBuf 里边。等待接收完毕之后，我们再把这个 RxdBuf 加1，再通过 TXD 引脚发送出去，同样需要先发一位起始位，然后发8个数据位，再发结束位，发送完毕后，程序运行到 while (PIN_RXD)，等待第二轮信号接收的开始。

```
#include <reg52.h>

sbit PIN_RXD = P3^0; //接收引脚定义
sbit PIN_TXD = P3^1; //发送引脚定义
bit RxdOrTxd = 0; //指示当前状态为接收还是发送
bit RxdEnd = 0; //接收结束标志
bit TxdEnd = 0; //发送结束标志
unsigned char RxdBuf = 0; //接收缓冲器
unsigned char TxdBuf = 0; //发送缓冲器
void ConfigUART(unsigned int baud);
void StartTXD(unsigned char dat);
void StartRXD();

void main(){
    EA = 1; //开总中断
```

```
    ConfigUART(9600);
    while (1){ //配置波特率为9600
        while (PIN_RXD); //等待接收引脚出现低电平，即起始位
        StartRXD(); //启动接收
        while (!RxdEnd); //等待接收完成
        StartTXD(RxdBuf+1); //接收到的数据+1后，发送回去
        while (!TxdEnd); //等待发送完成
    }
}
/* 串口配置函数，baud-通信波特率 */
void ConfigUART(unsigned int baud){
    TMOD &= 0xF0; //清零 T0 的控制位
    TMOD |= 0x02; //配置 T0 为模式2
    TH0 = 256 - (11059200/12)/baud; //计算 T0 重载值
}
/* 启动串行接收 */
void StartRXD(){
    TL0 = 256 - ((256-TH0)>>1); //接收启动时的 T0 定时为半个波特率周期
    ET0 = 1; //使能 T0 中断
    TR0 = 1; //启动 T0
    RxdEnd = 0; //清零接收结束标志
    RxdOrTxd = 0; //设置当前状态为接收
}
/* 启动串行发送，dat-待发送字节数据 */
void StartTXD(unsigned char dat){
    TxdBuf = dat; //待发送数据保存到发送缓冲器
    TL0 = TH0; //T0 计数初值为重载值
    ET0 = 1; //使能 T0 中断
    TR0 = 1; //启动 T0
    PIN_TXD = 0; //发送起始位
    TxdEnd = 0; //清零发送结束标志
    RxdOrTxd = 1; //设置当前状态为发送
}
/* T0 中断服务函数，处理串行发送和接收 */
void InterruptTimer0() interrupt 1{
    static unsigned char cnt = 0; //位接收或发送计数
    if (RxdOrTxd){ //串行发送处理
        cnt++;
        if (cnt <= 8){ //低位在先依次发送 8bit 数据位
            PIN_TXD = TxdBuf & 0x01;
            TxdBuf >>= 1;
        }else if (cnt == 9){ //发送停止位
            PIN_TXD = 1;
        }else{ //发送结束
            cnt = 0; //复位 bit 计数器
```

```
            TR0 = 0; //关闭 T0
            TxdEnd = 1; //置发送结束标志
        }
    }else{ //串行接收处理
        if (cnt == 0){ //处理起始位
            if (!PIN_RXD){ //起始位为0时，清零接收缓冲器，准备接收数据位
                RxdBuf = 0;
                cnt++;
            }
        }else{ //起始位不为0时，中止接收
            TR0 = 0; //关闭 T0
        }else if (cnt <= 8){ //处理8位数据位
            RxdBuf >>= 1; //低位在先，所以将之前接收的位向右移
            //接收脚为1时，缓冲器最高位置1，
            //而为0时不处理即仍保持移位后的0
            if (PIN_RXD){
                RxdBuf |= 0x80;
            }
            cnt++;
        }else{ //停止位处理
            cnt = 0; //复位 bit 计数器
            TR0 = 0; //关闭 T0
            if (PIN_RXD){ //停止位为1时，方能认为数据有效
                RxdEnd = 1; //置接收结束标志
            }
        }
    }
}
```

11.5 UART 串口通信的基本应用

通信的三种基本类型

常用的通信从传输方向上可以分为单工通信、半双工通信、全双工通信三类。

单工通信就是指只允许一方向另外一方传送信息，而另一方不能回传信息。比如电视遥控器、收音机广播等，都是单工通信技术。

半双工通信是指数据可以在双方之间相互传播，但是同一时刻只能其中一方发给另外一方，比如我们的对讲机就是典型的半双工。

全双工通信就发送数据的同时也能够接收数据，两者同步进行，就如同我们的电话一样，我们说话的同时也可以听到对方的声音。

UART 模块介绍

IO 口模拟串口通信，让大家了解了串口通信的本质，但是我们的单片机程序却需要不停的检测扫描单片机 IO 口收到的数据，大量占用了单片机的运行时间。这时候就会有聪明人想了，其实我们并不是很关心通信的过程，我们只需要一个通信的结果，最终得到接收到的数据就行了。这样我们可以在单片机内部做一个硬件模块，让它自动接收数据，接收完了，通知我们一下就可以了，我们的51单片机内部就存在这样一个 UART 模块，要正确使用它，当然还得先把对应的特殊功能寄存器配置好。

51单片机的 UART 串口的结构由串行口控制寄存器 SCON、发送和接收电路三部分构成，先来了解一下串口控制寄存器 SCON。如表11-1表11-2所示。

表11-1 SCON——串行控制寄存器的位分配（地址 0x98、可位寻址）

位	7	6	5	4	3	2	1	0
符号	SM0	SM1	SM2	REN	TB8	RB8	TI	RI
复位值	0	0	0	0	0	0	0	0

表11-2 SCON——串行控制寄存器的位描述

位	符号	描述
7	SM0	这两位共同决定了串口通信的模式 0～模式 3 共 4 种模式。我们最常用的就是模式 1，也就是 SM0=0，SM1=1，下边我们重点就讲模式 1，其它模式从略。
6	SM1	
5	SM2	多机通信控制位（极少用），模式 1 直接清零。
4	REN	使能串行接收。由软件置位使能接收，软件清零则禁止接收。
3	TB8	模式 2 和 3 中要发送的第 9 位数据（很少用）。
2	RB8	模式 2 和 3 中接收到的第 9 位数据（很少用），模式 1 用来接收停止位。
1	TI	发送中断标志位，当发送电路发送到停止位的中间位置时，TI 由硬件置 1，必须通过软件清零。
0	RI	接收中断标志位，当接收电路接收到停止位的中间位置时，RI 由硬件置 1，必须通过软件清零。

前边学了那么多寄存器的配置，相信 SCON 这个地方，对于大多数同学来说已经不是难点了，应该能看懂并且可以自己配置了。对于串口的四种模式，模式1是最常用的，就是我们前边提到的1位起始位，8位数据位和1位停止位。下面我们就详细介绍模式1的工作细节和使用方法，至于其它3种模式与此也是大同小异，真正遇到需要使用的时候大家再去查阅相关资料就行了。

在我们使用 IO 口模拟串口通信的时候，串口的波特率是使用定时器 T0 的中断体现出来的。在硬件串口模块中，有一个专门的波特率发生器用来控制发送和接收数据的速度。对于STC89C52 单片机来讲，这个波特率发生器只能由定时器 T1 或定时器 T2 产生，而不能由定时器 T0 产生，这和我们模拟的通信是完全不同的概念。

如果用定时器2，需要配置额外的寄存器，默认是使用定时器1的，我们本章内容主要就使用定时器 T1 作为波特率发生器来讲解，方式1下的波特率发生器必须使用定时器 T1 的模式2，也就是自动重装载模式，定时器的重载值计算公式为：

```
TH1 = TL1 = 256 - 晶振值/12 /2/16 /波特率
```

和波特率有关的还有一个寄存器，是一个电源管理寄存器 PCON，他的最高位可以把波特率提高一倍，也就是如果写 PCON |= 0x80 以后，计算公式就成了：

```
TH1 = TL1 = 256 - 晶振值/12 /16 /波特率
```

公式中数字的含义这里解释一下，256是8位定时器的溢出值，也就是 TL1 的溢出值，晶振值在我们的开发板上就是11059200，12是说1个机器周期等于12个时钟周期，值得关注的是这个16，我们来重点说明。在 IO 口模拟串口通信接收数据的时候，采集的是这一位数据的中间位置，而实际上串口模块比我们模拟的要复杂和精确一些。他

采取的方式是把一位信号采集16次，其中第7、8、9次取出来，这三次中其中两次如果是高电平，那么就认定这一位数据是1，如果两次是低电平，那么就认定这一位是0，这样一旦受到意外干扰读错一次数据，也依然可以保证最终数据的正确性。

了解了串口采集模式，在这里要给大家留一个思考题。“晶振值/12/2/16/波特率”这个地方计算的时候，出现不能除尽，或者出现小数怎么办，允许出现多大的偏差？把这部分理解了，也就理解了我们的晶振为何使用 11.0592 M 了。

串口通信的发送和接收电路在物理上有2个名字相同的 SBUF 寄存器，它们的地址也都是 0x99，但是一个用来做发送缓冲，一个用来做接收缓冲。意思就是说，有2个房间，两个房间的门牌号是一样的，其中一个只出人不进人，另外一个只进人不出人，这样的话，我们就可以实现 UART 的全双工通信，相互之间不会产生干扰。但是在逻辑上呢，我们每次只操作 SBUF，单片机会自动根据对它执行的是“读”还是“写”操作来选择是接收 SBUF 还是发送 SBUF，后边通过程序，我们就会彻底了解这个问题。

UART 串口程序

一般情况下，我们编写串口通信程序的基本步骤如下所示： 1. 配置串口为模式1。 2. 配置定时器 T1 为模式2，即自动重装模式。 3. 根据波特率计算 TH1 和 TL1 的初值，如果有需要可以使用 PCON 进行波特率加倍。 4. 打开定时器控制寄存器 TR1，让定时器跑起来。

这里还要特别注意一下，就是在使用 T1 做波特率发生器的时候，千万不要再使能 T1 的中断了。

我们先来看一下由 IO 口模拟串口通信直接改为使用硬件 UART 模块时的程序代码，看看程序是不是简单了很多，因为大部分的工作硬件模块都替我们做了。程序功能和 IO 口模拟的是完全一样的。

```
 #include <reg52.h>

void ConfigUART(unsigned int baud);
void main(){
    ConfigUART(9600); //配置波特率为 9600
    while (1){
        while (!RI); //等待接收完成
        RI = 0; //清零接收中断标志位
        SBUF = SBUF + 1; //接收到的数据+1 后，发送回去
        while (!TI); //等待发送完成
        TI = 0; //清零发送中断标志位
    }
}
/* 串口配置函数，baud-通信波特率 */
void ConfigUART(unsigned int baud){
    SCON = 0x50; //配置串口为模式 1
```

```
    TMOD &= 0x0F; //清零 T1 的控制位
    TMOD |= 0x20; //配置 T1 为模式2
    TH1 = 256 - (11059200/12/32)/baud; //计算 T1 重载值
    TL1 = TH1; //初值等于重载值
    ET1 = 0; //禁止 T1 中断
    TR1 = 1; //启动 T1
}
```

当然了，这个程序还是用在主循环里等待接收中断标志位和发送中断标志位的方法来编写的，而实际工程开发中，当然就不能这么干了，我们也只是为了用直观的对比来告诉同学们硬件模块可以大大简化程序代码，那么实际使用串口的时候就用到串口中断了，来看一下用中断实现的程序。请注意一点，因为接收和发送触发的是同一个串口中断，所以在串口中断函数中就必须先判断是哪种中断，然后再作出相应的处理。

```
 #include <reg52.h>

void ConfigUART(unsigned int baud);
void main(){
    EA = 1; //使能总中断
    ConfigUART(9600); //配置波特率为 9600
    while (1);
}
/* 串口配置函数，baud-通信波特率 */
void ConfigUART(unsigned int baud){
    SCON = 0x50; //配置串口为模式1
    TMOD &= 0x0F; //清零 T1 的控制位
    TMOD |= 0x20; //配置 T1 为模式2
    TH1 = 256 - (11059200/12/32)/baud; //计算 T1 重载值
    TL1 = TH1; //初值等于重载值
    ET1 = 0; //禁止 T1 中断
    ES = 1; //使能串口中断
    TR1 = 1; //启动 T1
}
/* UART 中断服务函数 */
void InterruptUART() interrupt 4{
    if (RI){ //接收到字节
        RI = 0; //手动清零接收中断标志位
        SBUF = SBUF + 1; //接收的数据+1 后发回，左边是发送 SBUF，右边是接收 SBUF
    }
    if (TI){ //字节发送完毕
        TI = 0; //手动清零发送中断标志位
    }
}
```

大家可以试验一下，看看是不是和前边用 IO 口模拟通信实现的效果一致，而主循环却完全空出来了，我们就可以随意添加其它功能代码进去。

单片机通信实例与 ASCII 码

我们学习串口通信主要是要实现单片机和电脑之间的信息交互，可以用电脑控制单片机的一些信息，可以把单片机的一些信息状况发给电脑上的软件。下面我们就做一个简单的例程，实现单片机串口调试助手发送的数据，在我们开发板上的数码管上显示出来。

```
 #include <reg52.h>


sbit ADDR3 = P1^3;
sbit ENLED = P1^4;

unsigned char code LedChar[] = { //数码管显示字符转换表
    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
    0x80, 0x90, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E
};
unsigned char LedBuff[7] = { //数码管+独立 LED 显示缓冲区
    0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
};
unsigned char T0RH = 0; //T0 重载值的高字节
unsigned char T0RL = 0; //T0 重载值的低字节
unsigned char RxdByte = 0; //串口接收到的字节
void ConfigTimer0(unsigned int ms);
void ConfigUART(unsigned int baud);
void main(){
    EA = 1; //使能总中断
    ENLED = 0; //选择数码管和独立 LED
    ADDR3 = 1;
    ConfigTimer0(1); //配置 T0 定时 1 ms
    ConfigUART(9600); //配置波特率为9600
    while (1){ //将接收字节在数码管上以十六进制形式显示出来
        LedBuff[0] = LedChar[RxdByte & 0x0F];
        LedBuff[1] = LedChar[RxdByte >> 4];
    }
}
/* 配置并启动 T0，ms-T0 定时时间 */
void ConfigTimer0(unsigned int ms){
    unsigned long tmp; //临时变量
    tmp = 11059200 / 12; //定时器计数频率
    tmp = (tmp * ms) / 1000; //计算所需的计数值
    tmp = 65536 - tmp; //计算定时器重载值
    tmp = tmp + 13; //补偿中断响应延时造成的误差
```

```
    T0RH = (unsigned char)(tmp>>8); //定时器重载值拆分为高低字节
    T0RL = (unsigned char)tmp;
    TMOD &= 0xF0; //清零 T0 的控制位
    TMOD |= 0x01; //配置 T0 为模式1
    TH0 = T0RH; //加载 T0 重载值
    TL0 = T0RL;
    ET0 = 1; //使能 T0 中断
    TR0 = 1; //启动 T0
}
/* 串口配置函数，baud-通信波特率 */
void ConfigUART(unsigned int baud){
    SCON = 0x50; //配置串口为模式1
    TMOD &= 0x0F; //清零 T1 的控制位
    TMOD |= 0x20; //配置 T1 为模式2
    TH1 = 256 - (11059200/12/32)/baud; //计算 T1 重载值
    TL1 = TH1; //初值等于重载值
    ET1 = 0; //禁止 T1 中断
    ES = 1; //使能串口中断
    TR1 = 1; //启动 T1
}
/* LED 动态扫描刷新函数，需在定时中断中调用 */
void LedScan(){
    static unsigned char i = 0; //动态扫描索引
    P0 = 0xFF; //关闭所有段选位，显示消隐
    P1 = (P1 & 0xF8) | i; //位选索引值赋值到 P1 口低3位
    P0 = LedBuff[i]; //缓冲区中索引位置的数据送到 P0 口
    if (i < 6){ //索引递增循环，遍历整个缓冲区
        i++;
    }else{
        i = 0;
    }
}
/* T0 中断服务函数，完成 LED 扫描 */
void InterruptTimer0() interrupt 1{
    TH0 = T0RH; //重新加载重载值
    TL0 = T0RL;
    LedScan(); //LED 扫描显示
}
/* UART 中断服务函数 */
void InterruptUART() interrupt 4{
    if (RI){ //接收到字节
        RI = 0; //手动清零接收中断标志位
        RxdByte = SBUF; //接收到的数据保存到接收字节变量中
        //接收到的数据又直接发回，叫作-"echo"，
        //用以提示用户输入的信息是否已正确接收
```

```
        SBUF = RxdByte;
    }
    if (TI){ //字节发送完毕
        TI = 0; //手动清零发送中断标志位
    }
}
```

大家在做这个实验的时候，有个小问题要注意一下。因为 STC89C52 下载程序是使用了 UART 串口下载，下载完程序后，程序运行起来了，可是下载软件最后还会通过串口发送一些额外的数据，所以程序刚下载进去不是显示00，而可能是其他数据。大家只要把电源开关关闭，重新打开一次就好了。

细心的同学可能会发现，在串口调试助手发送选项和接收选项处，还有个“字符格式发送”和“字符格式显示”，这是什么意思呢？

先抛开我们使用的汉字不谈，那么我们常用的字符就包含了0~9的数字、A~Z/a~z 的字母、还有各种标点符号等。那么在单片机系统里面我们怎么来表示它们呢？ ASCII 码（American Standard Code for Information Interchange，即美国信息互换标准代码）可以完成这个使命：我们知道，在单片机中一个字节的数据可以有0～255共256个值，我们取其中的0～127共128个值赋予了它另外一层涵义，即让它们分别来代表一个常用字符，其具体的对应关系如表11-3所示。

表11-3 ASCII 码字符表

ASCII	控制	ASCII	字符	ASCII	字符	ASCII	字符
000	NUL字符	032	(space)	064值	@	096	?
001	SOH	033	!	065	A	097	a
002	STX	034	″	066	B	098	b
003	ETX	035	#	067	C	099	c
004	EOT	036	$	068	D	100	d
005	END	037	%	069	E	101	e
006	ACK	038	&	070	F	102	f
007	BEL	039	’	071	G	103	g
008	BS	040	(	072	H	104	h
009	HT	041	)	073	I	105	i
010	LF	042	*	074	J	106	j
011	VT	043	+	075	K	107	k
012	FF	044	,	076	L	108	l
013	CR	045	-	077	M	109	m
014	SO	046	.	078	N	110	n
015	SI	047	/	079	O	111	o
016	DLE	048	0	080	P	112	p

ASCII	控制	ASCII	字符	ASCII	字符	ASCII	字符
017	DC1	049	1	081	Q	113	q
018	DC2	050	2	082	R	114	r
019	DC3	051	3	083	S	115	s
020	DC4	052	4	084	T	116	t
021	NAK	053	5	085	U	117	u
022	SYN	054	6	086	V	118	v
023	ETB	055	7	087	W	119	w
024	CAN	056	8	088	X	120	x
025	EM	057	9	089	Y	121	y
026	SUB	058	:	090	Z	122	z
027	ESC	059	;	091	[	123	{
028	FS	060	<	092	|124	¦	
029	GS	061	=	093	]	125	}
030	RS	062	>	094	^	126	~
031	US	063	?	095	_	127	DEL

这样我们就在常用字符和字节数据之间建立了一一对应的关系，那么现在一个字节就既可以代表一个整数又可以代表一个字符了，但它本质上只是一个字节的数据，而我们赋予了它不同的涵义，什么时候赋予它哪种涵义就看编程者的意图了。ASCII 码在单片机系统中应用非常广泛，我们后续的课程也会经常使用到它，下面我们来对它做一个直观的认识，同学们一定要深刻理解其本质。

对照上述表格，我们就可以实现字符和数字之间的转换了，比如还是这个程序，我们发送的时候改成字符格式发送，接收还是用十六进制接收，这样接收和数码管好做一下对比。

我们用字符格式发送一个小写的 a，返回一个十六进制的 0x61，数码管上显示的也是61，ASCII 码表里字符 a 对应十进制是97，等于十六进制的 0x61；我们再用字符格式发送一个数字1，返回一个十六进制的 0x31，数码管上显示的也是31，ASCII 表里字符1对应的十进制是49，等于十六进制的 0x31。这下大家就该清楚了：所谓的十六进制发送和十六进制接收，都是按字节数据的真实值进行的；而字符格式发送和字符格式接收，是按 ASCII 码表中字符形式进行的，但它实际上最终传输的还是一个字节数据。这个表格，当然不需要大家去记住，理解它，用的时候过来查就行了。

通信的学习，不像前边控制部分那么直观了，通信部分我们的程序只能获得一个结果，而其过程我们却无法直接看到，所以慢慢的可能大家就会知道有示波器和逻辑分析仪这类测量仪器。如果学校实验室或者公司里有示波器或者逻辑分析仪这类仪器，可以拿过来抓一下串口波形，直观的了解一下。如果暂时还没有这些仪器，先知道这么回事，有条件再说。因为工具类设备有的比较昂贵，有条件可以尽量使用学校或者公司的。在这里我用一款简易的逻辑分析仪把串口通信的波形抓出来给大家看一下，大家了解一下即可，如图11-7所示。

图11-7 逻辑分析仪串口数据示意图

分析仪和示波器的作用，就是把通信过程的波形抓出来进行分析。先大概说一下波形的意思。波形左边是低位，右边是高位，上边这个波形是电脑发送给单片机的，下边这个波形是单片机回发给电脑的。以上边的波形为例，左边第一位是起始位0，从低位到高位依次是10001100，顺序倒一下，就是数据 0x31，也就是 ASCII 码表里的‘1’。大家可以注意到分析仪在每个数据位都给标了一个白色的点，表示是数据，起始位和无数据的时候都没有这个白点。时间标 T1 和 T2 的差值在右边显示出来是 0.102 ms，大概是9600分之一，稍微有点偏差，在容许范围内即可。通过图11-7，我们可以清晰的了解了串口通信的收发的详细过程。

那我们这里再来了解一下，如果我们使用串口调试助手，用字符格式直接发送一个“12”，我们在我们的数码管上应该显示什么呢？串口调试助手应该返回什么呢？经过试验发现，我们数码管显示的是32，而串口调试助手返回十六进制显示的是31、32两个数据，如图11-8所示。

图11-8 串口调试助手数据显示

我们再用逻辑分析仪把这个数据抓出来看一下，如图11-9所示。

图11-9 逻辑分析仪抓取数据

对于 ASCII 码表来说，数字本身是字符而非数据，所以如果发送“12”的话，实际上是是分别发送了“1”和“2”两个字符，单片机呢，先收到第一个字符“1”，在数码管上会显示出31这个对应数字，但是马上

就又收到了“2”这个字符，数码管瞬间从31变成了32，而我们视觉上呢，是没有办法发现这种快速变化的，所以我们感觉数码管直接显示的是32。

6

12. C 语言指针基础与1602液晶的初步认识

我们在上 C 语言课的时候，学到指针，每一位教 C 语言的老师都会告诉我们一句：指针是 C 语言的灵魂。由此可见，指针是否学会是判断一个人是否真正学会 C 语言的重要指标之一，但是很多同学只知道其重要性，却没学会其灵活性。

简单的程序，100来行代码，不需要指针我们也可以轻松搞定，但是当代码写到几千上万行甚至更多的时候，利用指针就可以直接而快速的处理内存中的各种数据结构中的数据，特别是数组、字符串和内存的动态分配等，它为函数之间各类数据传递提供了简洁便利的方法。说了这么多作用估计大家没用过指针也体会不到，但这里就是表达这样一个意思，指针很重要，必须要学会、学好。

指针相对其它知识点来说比较难讲，主要在于例子不好举。简单的程序用指针去做会把简单的程序搞复杂，复杂的程序用指针去写牵扯的知识太多可能又不好理解。从一个角度讲，没学会指针就等于没学会 C 语言，所以再难也不是我们学不好的理由。这节课我就从我对指针的理解尽可能的把指针形象的介绍给大家，帮大家啃下这块硬骨头，同学们学习这节课内容也要打起十二分精神，集中注意力认真去学，争取拿下指针。

12.1 C 语言变量的地址

要研究指针，我们得先来深入理解内存地址这个概念。打个比方：整个内存就相当于一个拥有很多房间的大楼，每个房间都有房间号，比如从101、102、103一直到 NNN，我们可以说这些房间号就是房间的地址。相对应的内存中的每个单元也都有自己的编号，比如从 0x00、0x01、0x02 一直到 0xNN，我们同样可以说这些编号就是内存单元的地址。房间里可以住人，对应的内存单元里就可以“住进”变量了：假如一位名字叫 A 的人住在101房间，我们可以说 A 的住址就是101，或者101就是 A 的住址；对应的，假如一个名为 x 的变量住在编号为 0x00 的这个内存单元中，那么我们可以说变量 x 的内存地址就是 0x00，或者 0x00 就是变量 x 的地址。

基本的内存单元是字节，英文单词为 Byte，我们所使用的 STC89C52 单片机共有512字节的 RAM，就是我们所谓的内存，但它分为内部256字节和外部256字节，我们仅以内部的256字节为例，很明显其地址的编号从0开始就是0x00～0xFF。我们用 C 语言定义的各种变量就存在 0x00～0xFF 的地址范围内，而不同类型的变量会占用不同数量的内存单元，即字节，可以结合前面讲过的 C 语言变量类型深入理解。假如现在定义了

```
unsigned char a = 1;
unsigned char b = 2;
unsigned int c = 3;
unsigned long d = 4;
```

这样4个变量，我们把这4个变量分别放到内存中，就会是表12-1中所列的样子，我们先来大概了解一下他们的存储方式。

表12-1 变量存储方式

内存地址	存储的数据
……	……
0x07	d
0x06	d
0x05	d
0x04	d
0x03	c
0x02	c
0x01	b
0x00	a

变量 a、b 和 c 和 d 之间的变量类型不同，因此在内存中所占的存储单元也不一样，a 和 b 都占一个字节，c 占了2个字节，而 d 占了4个字节。那么，a 的地址就是 0x00，b 的地址就是 0x01，c 的地址就是 0x02，d 的

地址就是 0x04，它们的地址的表达方式可以写成：&a，&b，&c，&d。这样就代表了相应变量的地址，C 语言中变量前加一个&表示取这个变量的地址，&在这里就叫做“取址符”。

讲到这里，有一点延伸内容，大家可以了解下：比如变量 c 是 unsigned int 类型的，占了2个字节，存储在了 0x02 和 0x03 这两个内存地址上，那么 0x02 是它的低字节还是高字节呢？

这个问题由所用的 C 编译器与单片机架构共同决定，单片机类型不同就有可能不同，大家知道这么回事即可。比如：在我们使用的 Keil+51 单片机的环境下，0x02 存的是高字节，0x03 存的是低字节。这是编译底层实现上的细节问题，并不影响上层的应用，如下这两种情况在应用上丝毫不受这个细节的影响：强制类型转换——b = (unsigned char) c，那么 b 的值一定是 c 的低字节；取地址——&c，则得到的一定是 0x02，这都是 C 语言本身所决定的规则，不因单片机编译器的不同而有所改变。

实际生活中，我们要寻找一个人有两种方式，一种方式是通过它的名字来找人，还有第二种方式就是通过它的住宅地址来找人。我们在派出所的户籍管理系统的信息输入方框内，输入小明的家庭住址，系统会自动指向小明的相关信息，输入小刚的家庭住址，系统会自动指向小刚的相关信息。这个供我们输入地址的方框，在户籍管理系统叫做“地址输入框”。

那么，在 C 语言中，我们要访问一个变量，同样有两种方式：一种是通过变量名来访问，另一种自然就是通过变量的地址来访问了。在 C 语言中，地址就等同于指针，变量的地址就是变量的指针。我们要把地址送到上边那个所谓的“地址输入框”内，这个“地址输入框”既可以输入 x 的指针，又可以输入 y 的指针，所以相当于一个特殊的变量——保存指针的变量，因此称之为指针变量，简称为指针，而通常我们说的指针就是指指针变量。

地址输入框输入谁的地址，指向的就是这个人的信息，而给指针变量输入哪个普通变量的地址，它自然就指向了这个变量的内容，通常的说法就是指针指向了该变量。

12.2 C 语言指针变量的声明

在 C 语言中，变量的地址往往都是编译系统自动分配的，对我们用户来说，我们是不知道某个变量的具体地址的。所以我们定义一个指针变量 p，把普通变量 a 的地址直接送给指针变量 p 就是 p = &a;这样的写法。

对于指针变量 p 的定义和初始化，一般有两种方式，这两种方式，初学者很容易混淆，因此这个地方没别的方法，就是死记硬背，记住即可。

方法1：定义时直接进行初始化赋值。

```
unsigned char a;
unsigned char *p = &a;
```

方法2：定义后再进行赋值。

```
unsigned char a;
unsigned char *p;
p = &a;
```

大家仔细看会看出来这两种写法的区别，它们都是正确的。我们在定义的指针变量前边加了个，这个p 就代表了这个 p 是个指针变量，不是个普通的变量，它是专门用来存放变量地址的。此外，我们定义*p 的时候，用了 unsigned char 来定义，这里表示的是这个指针指向的变量类型是 unsigned char 型的。

指针变量似乎比较好理解，大家也能很容易就听明白。但是为什么很多人弄不明白指针呢？因为在 C 语言中，有一些运算和定义，他们是有区别的，很多同学就是没弄明白它们的区别，指针就始终学不好。这里我要重点强调两个区别，只要把这两个区别弄明白了，起码指针变量这部分就不是问题了。这两个重点现在大家死记硬背，直接记住即可，靠理解有可能混淆概念。

第一个重要区别：指针变量 p 和普通变量 a 的区别。

我们定义一个变量 a，同时也可以给变量 a 赋值 a = 1，也可以赋值 a = 2。

我们定义一个指针变量 p，另外还定义了一个普通变量 a= 1，普通变量 b=2，那么这个指针变量可以指向 a 的地址，也可以指向 b 的地址，可以写成 p = &a，也可以写成 p = &b，但就是不能写成 p = 1 或者 p = 2 或者 p = a，这三种表达方式都是错的。

因此这个地方，不要看到定义*p 的时候前边有个 unsigned char 型，就错误的赋值 p=1，这个只是说明 p 指向的变量是这个 unsigned char 类型的，而 p 本身，是指针变量，不可以给它赋值普通的值或者变量，后边我们会直接把指针变量称之为指针，大家要注意一下这个小细节。

前边这个区别似乎比较好理解，还有第二个重要区别，一定要记清楚。

第二个重要区别：定义指针变量 *p 和取值运算 *p 的区别。

“*”这个符号，在我们的 C 语言有三个用法，第一个用法很简单，乘法操作就是用这个符号，这里就不讲了。

第二个用法，是定义指针变量的时候用的，比如 unsigned char *p*，*这个地方使用“”*代表的意思是 p 是一个指针变量，而非普通的变量。

还有第三种用法，就是取值运算，和定义指针变量是完全两码事，比如：

```
unsigned char  a = 1;
unsigned char  b = 2;
unsigned char  *p;
p = &a;
b = *p;
```

这样两步运算完了之后，b 的值就成了1了。在这段代码中，&a 表示取 a 这个变量的地址，把这个地址送给 p 之后，再用 *p 运算表示的是取指针变量 p 指向的地址的变量的值，又把这个值送给了 b，最终的结果相当于 b=a。同样是 *p，放在定义的位置就是定义指针变量，放在执行代码中就是取值运算。

这两个重要区别，大家可以反复阅读三四遍，把这两个重要区别弄明白，指针的大门就顺利的踏进去一只脚了。至于详细的用法，我们后边用得多了就会慢慢熟悉起来了。

12.3 C 语言指针的简单示例

前边我们提到了，指针的意义往往在小程序里是体现不出来的，对于简单程序来说，有时候用了指针，反而可能比没用指针还麻烦，但是为了让大家巩固一下指针的用法，我还是写了个使用指针的流水灯程序，目的是让大家从简单程序开始了解指针，当程序复杂的时候不至于手足无措。

```
 #include <reg52.h>

sbit ADDR0 = P1^0;
sbit ADDR1 = P1^1;
sbit ADDR2 = P1^2;
sbit ADDR3 = P1^3;
sbit ENLED = P1^4;
void ShiftLeft(unsigned char *p);

void main(){
    unsigned int i;
    unsigned char buf = 0x01;

    ENLED = 0; //使能选择独立 LED
    ADDR3 = 1;
    ADDR2 = 1;
    ADDR1 = 1;
    ADDR0 = 0;

    while (1){
        P0 = ~buf; //缓冲值取反送到 P0 口
        for (i=0; i<20000; i++); //延时
        ShiftLeft(&buf); //缓冲值左移一位
        if (buf == 0){ //如移位后为0则重赋初值
            buf = 0x01;
        }
    }
}
/* 将指针变量 p 指向的字节左移一位 */
void ShiftLeft(unsigned char *p){
    *p = *p << 1; //利用指针变量可以向函数外输出运算结果
}
```

这是一个使用指针实现流水灯的例子，纯粹是为了讲指针而写这样一段程序，程序中传递的是 buf 的地址，把这个地址直接传递给函数 ShiftLeft 的形参指针变量 p，也就是 p 指向了 buf。对比之前的函数调用，大家是否

看明白，如果是普通变量传递，只能单向的，也就是说，主函数传递给子函数的值，子函数只能使用却不能改变。而现在我们传递的是指针，不仅仅子函数可以使用 buf 里边的值，而且还可以对 buf 里边的值进行修改。

此外再强调一句，只要是 *p 前边带了变量类型如 unsigned char，就是表示定义了一个指针变量 p，而执行代码中的 *p，是指 p 所指向的内容。

通过理论的学习和这样一个例程，我想大家对指针应该有概念了，至于它的灵活应用，需要我们在后边的程序中慢慢去体会，理论上就不再过多赘述了。

12.4 C 语言指向数组元素的指针

指向数组元素的指针和运算法则

所谓指向数组元素的指针，其本质还是变量的指针。因为数组中的每个元素，其实都可以直接看成是一个变量，所以指向数组元素的指针，也就是变量的指针。

指向数组元素的指针不难，但很常用。我们用程序来解释会比较直观一些。

```
unsigned char number[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
unsigned char *p;
```

如果我们写 p = &number[0];那么指针 p 就指向了 number 的第0号元素，也就是把number[0]的地址赋值给了p，同理，如果写 p = &number[1];p 就指向了数组 number 的第1号元素。p = &number[x];其中 x 的取值范围是0～9，就表示 p 指向了数组 number 的第 x 号元素。指针本身，也可以进行几种简单的运算，这几种运算对于数组元素的指针来说应用最多。 1. 比较运算。比较的前提是两个指针指向同种类型的对象，比如两个指针变量 p 和 q 它们指向了具有同种数据类型的数组，那它们可以进行 <，>，>=，<=，==等关系运算。如果 p==q 为真的话，表示这两个指针指向的是同一个元素。 2. 指针和整数可以直接进行加减运算。比如还是上边我们那个指针 p 和数组 number，如果 p = &number[0]，那么 p+1 就指向了 number[1]，p+9 就指向了 number[9]。当然了，如果 p = &number[9]，p-9 也就指向了 number[0]。 3. 两个指针变量在一定条件下可以进行减法运算。如 p = &number[0]; q = &number[9];那么 q-p 的结果就是9。但是这个地方大家要特别注意，这个9代表的是元素的个数，而不是真正的地址差值。如果我们的 number 的变量类型是 unsigned int 型，占2个字节，q-p 的结果依然是9，因为它代表的是数组元素的个数。

在数组元素指针这里还有一种情况，就是数组名字其实就代表了数组元素的首地址，也就是说：

```
p = &number[0];
p = number;
```

这两种表达方式是等价的，因此以下几种表达形式和内容需要大家格外注意一下。

根据指针的运算规则，p+x 代表的是 number[x]的地址，那么 number+x 代表的也是number[x]的地址。或者说，它们指向的都是 number 数组的第 x 号元素。

(p+x)和(number+x)都表示 number[x]。

指向数组元素的指针也可以表示成数组的形式，也就是说，允许指针变量带下标，即 p[i]和 *(p+i)是等价的。但是为了避免混淆与规范起见，这里我们建议大家不要写成前者，而一律采用后者的写法。但如果看到别人那么写，也知道是怎么回事即可。

二维数组元素的指针和一维数组类似，需要介绍的内容不多。假如现在一个指针变量 p 和一个二维数组 number[3][4]，它的地址的表达方式也就是 p=&number[0][0]，有一个地方要注意，既然数组名代表了数组元素的首地址，那么也就是说 p 和 number 都是指数组的首地址。对二维数组来说，number[0]，number[1]，number[2]都可以看成是一维数组的数组名字，所以 number[0]等价于 &number[0][0]， number[1]等价于 &number[1][0]，number[2]等价于 &number[2][0]。加减运算和一维数组是类似的，不再详述。

指向数组元素指针的实例

在 C 语言里边，sizeof()可以用来获取括号内的对象所占用的内存字节数，虽然它写作函数的形式，但它并不是一个函数，而是 C 语言的一个关键字，sizeof()整体在程序代码中就相当于一个常量，也就是说这个获取操作是在程序编译的时候进行的，而不是在程序运行的时候进行。这是一个实际编程中很有用的关键字，灵活运用它可以为程序带来更好的可读性、易维护性和可移植性，在后续的例程学习中将会慢慢有所体会的。

sizeof()括号中可以是变量名，也可以是变量类型名，其结果是等效的。而其更大的用处是与数组名搭配使用，这样可以获取整个数组占用的字节数，就不用自己动手计算了，可以避免错误，而如果日后改变了数组的维数时，也不需要再到执行代码中逐个修改，便于程序的维护和移植。

下面我们提供了一个简单的串口演示例程，可以体验一下指针和 sizeof()的用法。例程首先接收上位机下发的命令，根据命令值分别把不同数组的数据回发给上位机，程序还用到了指针的自增运算，也就是+1 运算，大家可以认真考虑一下指针 ptrTxd 在串口发送的过程中的指向是如何变化的。在上位机串口调试助手中分别下发 1、2、3、4，就会得到不同的数组回发，注意这里都用十六进制发送和十六进制显示。

此外，这个程序还应用到一个小技巧，大家要学会使用。我们前边讲了串口发送中断标志位 TI 是硬件置位，软件清零的。通常来讲，我们想一次发送多个数据的时候，就需要把第一个字节写入 SBUF，然后再等待发送中断，在后续中断中再发送剩余的数据，这样我们的数据发送过程就被拆分到了两个地方——主循环内和中断服务函数内，无疑就使得程序结构变得零散了。这个时候，为了使程序结构尽量紧凑，在启动发送的时候，不是向 SBUF 中写入第一个待发的字节，而是直接让 TI=1，注意，这时候会马上进入串口中断，因为中断标志位置1了，但是串口线上并没有发送任何数据，于是，我们所有的数据发送都可以在中断中进行，而不用再分为两部分了。大家可以在程序中体会一下这个技巧的好处。

```
#include <reg52.h>

bit cmdArrived = 0; //命令到达标志，即接收到上位机下发的命令
unsigned char cmdIndex = 0; //命令索引，即与上位机约定好的数组编号
unsigned char cntTxd = 0; //串口发送计数器
unsigned char *ptrTxd; //串口发送指针

unsigned char array1[1] = {1};
unsigned char array2[2] = {1,2};
```

```
unsigned char array3[4] = {1,2,3,4};
unsigned char array4[8] = {1,2,3,4,5,6,7,8};

void ConfigUART(unsigned int baud);

void main(){
    EA = 1; //开总中断
    ConfigUART(9600); //配置波特率为9600

    while (1){
        if (cmdArrived){
            cmdArrived = 0;
            switch (cmdIndex){
                case 1:
                    ptrTxd = array1; //数组1的首地址赋值给发送指针
                    cntTxd = sizeof(array1); //数组1的长度赋值给发送计数器
                    TI = 1; //手动方式启动发送中断，处理数据发送
                    break;
                case 2:
                    ptrTxd = array2;
                    cntTxd = sizeof(array2);
                    TI = 1;
                    break;
                case 3:
                    ptrTxd = array3;
                    cntTxd = sizeof(array3);
                    TI = 1;
                    break;
                case 4:
                    ptrTxd = array4;
                    cntTxd = sizeof(array4);
                    TI = 1;
                    break;
                default:
                    break;
            }
        }
    }
}
/* 串口配置函数，baud-通信波特率 */
void ConfigUART(unsigned int baud){
    SCON = 0x50; //配置串口为模式1
    TMOD &= 0x0F; //清零 T1 的控制位
    TMOD |= 0x20; //配置 T1 为模式2
    TH1 = 256 - (11059200/12/32)/baud; //计算 T1 重载值
```

```
    TL1 = TH1; //初值等于重载值
    ET1 = 0; //禁止 T1 中断
    ES = 1; //使能串口中断
    TR1 = 1; //启动 T1
}
/* UART 中断服务函数 */
void InterruptUART() interrupt 4{
    if (RI){ //接收到字节
        RI = 0; //清零接收中断标志位
        cmdIndex = SBUF; //接收到的数据保存到命令索引中
        cmdArrived = 1;//设置命令到达标志
    }
    if (TI){ //字节发送完毕
        TI = 0; //清零发送中断标志位
        if (cntTxd > 0){ //有待发送数据时，继续发送后续字节
            SBUF = *ptrTxd; //发出指针指向的数据
            cntTxd--; //发送计数器递减
            ptrTxd++; //发送指针递增
        }
    }
}
```

12.5 C 语言字符数组和字符指针

常量和符号常量

在程序运行过程中，其值不能被改变的量称之为常量。常量分为不同的类型，有整型常量如1、2、3、100；浮点型常量3.14、0.56、-4.8；字符型常量„a?、„b?、„0?；字符串常量“a”、“abc”、“1234”、“1234abcd”等。

细心的同学会发现，整型和浮点型常量我们直接写的数字，而字符型常量用单引号来表示一个字符，用双引号来表示一个字符串，尤其大家要注意„a?和“a”是不一样的，这个等会我们要详细介绍。

常量一般有两种表现形式： - 直接常量：直接以值的形式表示的常量称之为直接常量。上述举例这些都是直接常量，直接写出来了。 - 符号常量：用标识符命名的常量称之为符号常量，就是为上面的直接常量再取一个名字。使用符号常量一是方便理解，提高程序可读性，更重要的是方便程序的后续维护，习惯上符号常量我们都用大写字母和下划线来命名。

比如，我们可以把3.14取名为 PI（即π）。再比如，我们上节课的串口程序，我们用的波特率是9600，如果用符号常量来进行提前声明的话，那我们要修改成其它速率的话，就不用在程序中找9600修改了，直接修改声明处就可以了，两种方法举例说明。用 const 声明。比如我们在程序开始位置定义一个符号常量 BAUD。

定义形式是：

```
const  类型  符号常量名字=常量值;
```

如

```
const unsigned int BAUD = 9600;  /*注意结尾有个分号*/
```

我们就可以在程序中直接把9600改成 BAUD，这样我们如果要改波特率的话，直接在程序开头位置改一下这个值就可以了。用预处理命令 #define 来完成，预处理命令我们先来认识 #define。

定义形式是：

```
#define  符号常量名  常量值
```

如

```
#define  BAUD  9600  /*注意结尾没有分号*/
```

这样定义以后，只要在程序中出现 BAUD 的话，意思就是完全替代了后边的9600这个数字。

不知大家是否记得，我们之前定义数码管真值表的时候，用了一个 code 关键字。

```
unsigned char code LedChar[] = {  //数码管显示字符转换表
    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8,
    0x80, 0x90, 0x88, 0x83, 0xC6, 0xA1, 0x86, 0x8E
};
```

我们当时说加了 code 之后，这个真值表的数据只能被使用，不能被改变，如果我们直接写 LedChar[0] = 1;这样就错了。实际上 code 这个关键字是51单片机特有的，如果是其它类型的单片机我们只需要写成 const unsigned char LedChar[]={}就可以了，自动保存到 FLASH 里，而51单片机只用 const 而不加 code 的话，这个数组会保存到 RAM 中，而不会保存到 FLASH 中，鉴于此，在51这个体系下，const 反倒变得不那么重要了，它的作用被 code 取代了，这里大家知道这么回事即可。

我们来对各种类型的常量做进一步说明。

整型常量和浮点型常量就没多少可说的了，之前我们应用的都很熟练了，整型直接写数字就是十进制如128，前边 0x 开头的表示是十六进制 0x80，浮点型直接写带小数点的数据就可以了。

字符型常量是由一对单引号括起来的单个字符。它分为两种形式，一种是普通字符，一种是转义字符。

普通字符就是那些我们可以直接书写直接看到的有形的字符，比如阿拉伯数字0～9，英文字符 A～z，以及标点符号等。它们都是 ASCII 码表中的字符，而它们在单片机中都占用一个字节的空间，其值就是对应的 ASCII 码值。比如„a?的值是97，„A?的值是65，„0?的值是48，如果定义一个变量 unsigned char a = „a?，那么变量 a 的值就是97。

除了上述这些字符之外，还有一些特殊字符，它们一些是无形的，像回车符、换行符这些都是看不到的，还有一些像?\”这类字符它们已经有特殊用途了，想象一下如果写 ’’’觉得编译器会怎么去解释呢。针对这些特殊符号，为了可以让它们正常进入到我们的程序代码中，C 语言就规定了转义字符，它是以反斜杠()开头的特定字符序列，让它们来表示这些特殊字符，比如我们用 \n 来代表换行。我们用一个简单表格来说明一下常用的转义字符的意思，如表12-2所示。

表 12-2 常用转义字符及含义

字符形式	含义
\n	换行
\t	横向跳格（相当于 Tab）
\v	竖向跳格
\b	退格
\r	光标移到行首
\	反斜杠字符„\?
\?	单引号字符

字符形式	含义
\”	双引号字符
\f	走纸换页
\0	空值

表格不需要大家记住，用到了，过来查就可以了。

字符串常量是用双引号括起来的字符序列，一般我们都称之字符串。如“a”、“1234”、“welcome to www.kingst.org”等都是字符串常量。字符串常量在内存中按顺序逐个存储字符串中的字符的 ASCII 码值，并且特别注意，最后还有一个字符„\0?，„\0?字符的 ASCII 码值是0，它是字符串结束标志，在写字符串的时候，这个„\0?是隐藏的，我们看不到，但是实际却是存在的。所以“a”就比„a?多了一个 „\0?，“a”的就占了2个字节，而 „a?只占一个字节。

还有一个地方要注意， 就是字符串中的空格， 也是一个字符，比如 “welcome to www.kingst.org ”一共占了26个字节的空间。其中21个字母，2个„.?，2个 „ ?（空格字符）以及一个„\0?。

字符和字符串数组实例

为了对比字符串、字符数组、常量数组的区别，我们写个了简单的演示程序，定义了4个数组分别是：

```
unsigned char array1[] = "1-Hello!\r\n";
unsigned char array2[] = {'2', '-', 'H', 'e', 'l', 'l', 'o', '!', '\r', '\n'};
unsigned char array3[] = {51, 45, 72, 101, 108, 108, 111, 33, 13, 10};
unsigned char array4[] = "4-Hello!\r\n";
```

在串口调试助手下，发送十六进制的1、2、3、4，使用字符形式显示的话，会分别往电脑上送这4个数组中对应的那个数组。我们只是在起始位置做了区分，其它均没有区别。大家可以比较一下效果。

此外还要说明一点，数组1和数组4，数组1我们是发完整的字符串，而数组4我们仅仅发送数组中的字符，没有发结束符号。串口调试助手用字符形式显示是没有区别的，但是大家如果改用十六进制显示，大家会发现数组1比数组4多了一个字节„ \0 ?的 ASCII 值00。

```
 #include <reg52.h>

bit cmdArrived = 0; //命令到达标志，即接收到上位机下发的命令
unsigned char cmdIndex = 0; //命令索引，即与上位机约定好的数组编号
unsigned char cntTxd = 0; //串口发送计数器
unsigned char *ptrTxd; //串口发送指针

unsigned char array1[] = "1-Hello!\r\n";
unsigned char array2[] = {'2', '-', 'H', 'e', 'l', 'l', 'o', '!', '\r', '\n'};
unsigned char array3[] = {51, 45, 72, 101, 108, 108, 111, 33, 13, 10};
```

```
unsigned char array4[] = "4-Hello!\r\n";

void ConfigUART(unsigned int baud);
void main(){
    EA = 1; //开总中断
    ConfigUART(9600); //配置波特率为 9600

    while (1){
        if (cmdArrived){
            cmdArrived = 0;
            switch (cmdIndex){
                case 1:
                    ptrTxd = array1; //数组1的首地址赋值给发送指针
                    cntTxd = sizeof(array1); //数组1的长度赋值给发送计数器
                    TI = 1; //手动方式启动发送中断，处理数据发送
                    break;
                case 2:
                    ptrTxd = array2;
                    cntTxd = sizeof(array2);
                    TI = 1;
                    break;
                case 3:
                    ptrTxd = array3;
                    cntTxd = sizeof(array3);
                    TI = 1;
                    break;
                case 4:
                    ptrTxd = array4;
                    cntTxd = sizeof(array4) - 1; //字符串实际长度为数组长度减1
                    TI = 1;
                    break;
                default:
                    break;
            }
        }
    }
}
/* 串口配置函数，baud-通信波特率 */
void ConfigUART(unsigned int baud){
    SCON = 0x50; //配置串口为模式1
    TMOD &= 0x0F; //清零 T1 的控制位
    TMOD |= 0x20; //配置 T1 为模式2
    TH1 = 256 - (11059200/12/32)/baud; //计算 T1 重载值
    TL1 = TH1; //初值等于重载值
    ET1 = 0; //禁止 T1 中断
```

```
    ES = 1; //使能串口中断
    TR1 = 1; //启动 T1
}
/* UART 中断服务函数 */
void InterruptUART() interrupt 4{
    if (RI){ //接收到字节
        RI = 0; //清零接收中断标志位
        cmdIndex = SBUF; //接收到的数据保存到命令索引中
        cmdArrived = 1; //设置命令到达标志
    }
    if (TI){ //字节发送完毕
        TI = 0; //清零发送中断标志位
        if (cntTxd > 0){ //有待发送数据时，继续发送后续字节
            SBUF = *ptrTxd; //发出指针指向的数据
            cntTxd--; //发送计数器递减
            ptrTxd++; //发送指针递增
        }
    }
}
```

12.6 1602 液晶介绍(电路和引脚图)

前边我们讲的流水灯、数码管、LED 点阵这三种都是 LED 设备，这节课我们来学习一下 LCD 显示设备—— 1602 液晶。那个大大的，平时第一行显示16个小黑块，第二行什么都不显示的东西就是 1602 液晶，是不是早就注意到它了呢？

大家学习这些电子器件，头脑中要逐渐形成一种意识，不管是我们的单片机，还是 74HC138，甚至三极管等等，都是有数据手册的。不管是设计电路还是编写程序，器件的数据手册是我们最好的参考资料，那我们今天来学习 1602，首先就要看它的数据手册。手册大家可以在网上找到，这里我讲的时候只挑手册的重点讲。

首先我们来看一个主要技术参数表格，如表12-3所示。

表12-3 1602 液晶主要技术参数

显示容量	16 x 2个字符
芯片工作电压	4.5~5.5 V
工作电流	2.0 mA(5.0 V)
模块最佳工作电压	5.0 V
字符尺寸	2.95 x 4.35 mm (宽乘高)

1602 液晶，从它的名字我们就可以看出它的显示容量，就是可以显示2行，每行16个字符的液晶。它的工作电压是 4.5 V～5.5 V，对于这点我们设计电路的时候，直接按照 5 V 系统设计，但是保证我们的 5 V 系统最低不能低于 4.5 V。在 5 V 工作电压下测量它的工作电流是 2 mA，大家注意，这个 2 mA 仅仅是指液晶，而它的黄绿背光都是用 LED 做的，所以功耗不会太小的，一二十毫安还是有的。

1602 液晶一共16个引脚，每个引脚的功能，我们都可以在它的数据手册上获得。而这些基本的信息，在我们设计电路和编写代码之前，必须先看明白，如表12-4所示。

表12-4 1602 液晶引脚功能

编号	符号	引脚说明	编号	符号	引脚说明
1	VSS	电源地	9	D2	Data I/O
2	VDD	电源正极	10	D3	Data I/O
3	VL	液晶显示偏压信号	11	D4	Data I/O
4	RS	数据/命令选择端(H/L)	12	D5	Data I/O
5	R/W	读/写选择端(H/L)	13	D6	Data I/O
6	E	使能信号	14	D7	Data I/O
7	D0	Data I/O	15	BLA	背光源正极
8	D1	Data I/O	16	BLK	背光源负极

液晶的电源1脚2脚以及背光电源15脚16脚，不用多说，正常接就可以了。

3脚叫做液晶显示偏压信号，大家注意到小黑块没有，当我们要显示一个字符的时候，有的黑点显示，有的黑点就不能显示，这样就可以实现我们想要的字符了。我们这个3脚就是用来调整显示的黑点和不显示的之间的对比度，调整好了对比度，就可以让我们的显示更加清晰一些。在进行电路设计实验的时候，通常的办法是在这个引脚上接个电位器，也就是我们初中学过的滑动变阻器。通过调整电位器的分压值，来调整3脚的电压。而当产品批量生产的时候，我们可以把我们调整好的这个值直接用简单电路来实现，就如同在我们板子上，我们直接使用的是一个18欧的下拉电阻，市面上有的 1602 的下拉电阻大概1到 1.5 K 也是比较合适的值。

4脚是数据命令选择端。对于液晶，有时候我们要发送一些命令，让它实现我们想要的一些状态，有时候我们要发给它一些数据，让它显示出来，液晶就通过这个引脚来判断接收到的是命令还是数据，这个引脚我们接到了 ADDR0 上，通过跳线帽和 P1.0 连接在一起。大家注意学会读手册，看到这个引脚描述里：数据/命令选择端，而后跟了括号(H/L)，他的意思就是当这个引脚是 H(High)高电平的时候，是数据，当这个引脚是 L(Low)低电平的时候，是命令。

5脚和4脚用法类似，功能是读写选择端。我们既可以写给液晶数据或者命令，也可以读取液晶内部的数据或状态，就是控制这个引脚。因为液晶本身内部有 RAM，实际上我们送给液晶的命令或者数据，液晶需要先保存在缓存里，然后再写到内部的寄存器或者 RAM 中，这个就需要一定的时间。所以我们进行读写操作之前，首先要读一下液晶当前状态，是不是在“忙”，如果不忙，我们可以读写数据，如果在“忙”，我们就需要等待液晶忙完了，再进行操作。读状态是常用的，不过读液晶数据我接触的场合没怎么用过，大家了解这个功能即可。这个引脚我们接到了 ADDR1 上，通过跳线帽和 P1.1 连接在一起。

6脚是使能信号，很关键，液晶的读写命令和数据，都要靠它才能正常读写，我们后边详细讲这个引脚怎么用。这个引脚我们通过跳线帽接到了 ENLCD 上，这个位置的跳线是为了和另外一个 12864 液晶的切换使用而设计的。

7到14引脚就是8个数据引脚了，我们就是通过这8个引脚读写数据和命令的。我们统一接到了 P0 口上。来看一下开发板上的 1602 接口的原理图，如图12-1所示。

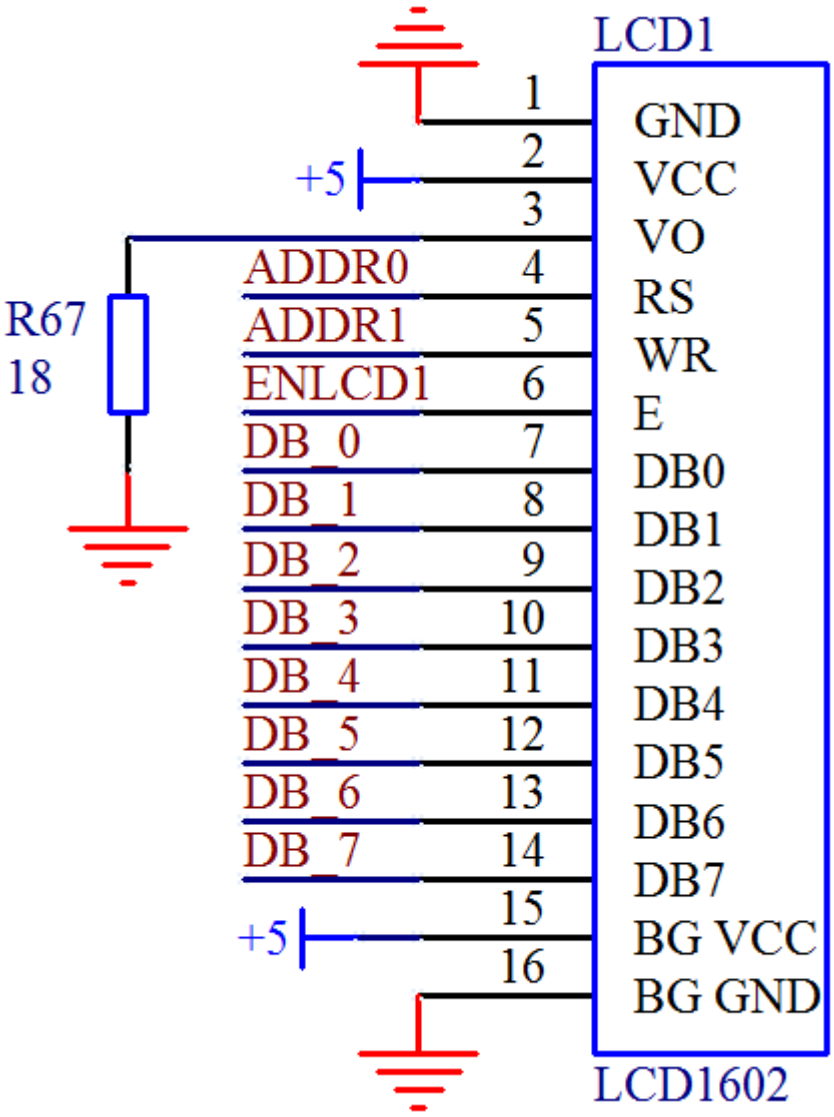

图12-1 1602 液晶接口原理图

12.7 1602 液晶的读写时序介绍

1602 液晶内部带了80个字节的显示 RAM，用来存储我们发送的数据，它的结构如图12-2所示。

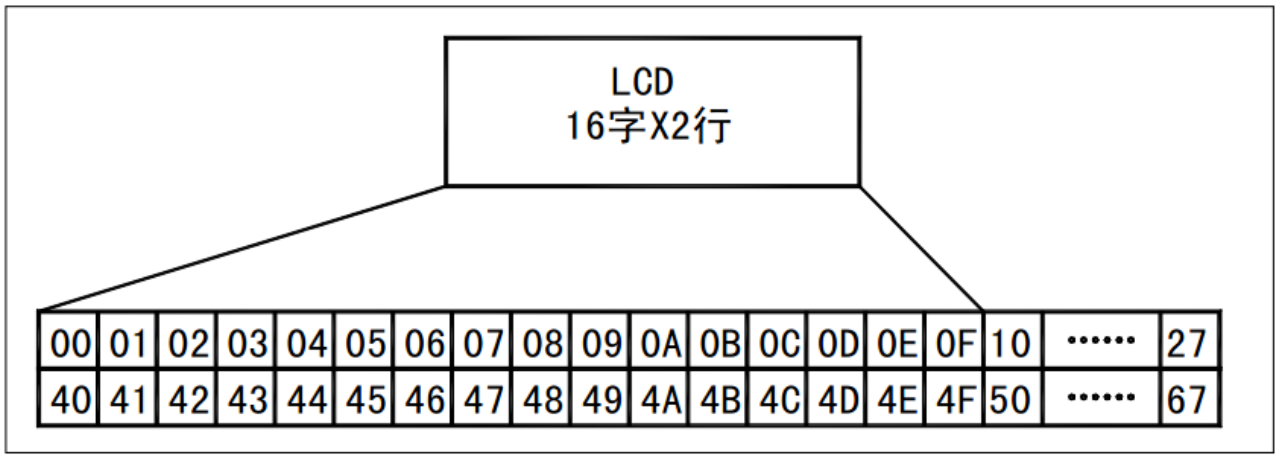

图12-2 1602 内部 RAM 结构

第一行的地址是 0x00H 到 0x27，第二行的地址从 0x40 到 0x67，其中第一行 0x00 到 0x0F 是与液晶上第一行16个字符显示位置相对应的，第二行 0x40 到 0x4F 是与第二行16个字符显示位置相对应的。而每行都多出来一部分，是为了显示移动字幕设置的。1602 字符液晶是显示字符的，因此它跟 ASCII 字符表是对应的。比如我们给 0x00 这个地址写一个„a?，也就是十进制的97，液晶的最左上方的那个小块就会显示一个字母 a。此外，我们本章学过指针了，液晶内部有个数据指针，它指向哪里，我们写的那个数据就会送到相应的那个地址里。

液晶有一个状态字字节，我们通过读取这个状态字的内容，就可以知道 1602 液晶的一些内部情况，如表12-5所示。

表12-5 1602 液晶状态字

bit0~bit6	当前数据的指针的值
bit7	读写操作使能| 1：禁止 0：允许

这个状态字节有8个位，最高位表示了当前液晶是不是“忙”，如果这个位是1表示液晶正“忙”，禁止我们读写数据或者命令，如果是0，则可以进行读写。而低7位就表示了当前数据地址指针的位置。

1602 的基本操作时序，一共有4个，这些大家都不需要记住，但是都需要理解，因为我们现在不是为了应付考试，所以不需要你把手册背熟，但是你写程序的时候，打开手册要能看懂如何操作，还要再提醒一句，单片机读外部状态前，必须先保证自己是高电平哦。

我们这里要做 1602 液晶的程序，因此先把用到的总线接口做一个统一声明：

```
 #define LCD1602_DB P0

sbit LCD1602_RS = P1^0;
sbit LCD1602_RW = P1^1;
sbit LCD1602_E = P1^5;
```

1、读状态：RS=L，R/W=H，E=H。这是个很简单的逻辑，就是说，我们直接写

```
LCD1602_DB = 0xFF;
LCD1602_RS = 0;
LCD1602_RW = 1;
LCD1602_E = 1;
sta = LCD1602_DB;
```

这样就把当前液晶的状态字读到了 sta 这个变量中，我们可以通过判断 sta 最高位的值来了解当前液晶是否处于“忙”状态，也可以得知当前数据的指针位置。两个问题，一是如果我们当前读到的状态是“不忙”，那么我们程序可以进行读写操作，如果当前状态是“忙”，那么我们还得继续等待重新判断液晶的状态；问题二，大家可以看我们的原理图，流水灯、数码管、点阵、1602 液晶都用到了 P0 口总线，我们读完了液晶状态继续保持 LCD1602_E 是高电平的话，1602 液晶会继续输出它的状态值，输出的这个值会占据了 P0 总线，干扰到流水灯数码管等其它外设，所以我们读完了状态，通常要把这个引脚拉低来释放总线，这里我们用了一个 do...while 循环语句来实现。

```
LCD1602_DB = 0xFF;
LCD1602_RS = 0;
LCD1602_RW = 1;
do {
    LCD1602_E = 1;
    sta = LCD1602_DB; //读取状态字
    LCD1602_E = 0;  //读完撤销使能，防止液晶输出数据干扰 P0 总线
} while (sta & 0x80);
//bit7 等于1表示液晶正忙，重复检测直到其等于0为止
```

2、读数据：RS=H，R/W=L，E=H。这个逻辑也很简单，但是读数据不常用，大家了解一下就可以了，这里就不详细解释了。

3、写指令：RS=L，R/W=L，D0~D7=指令码，E=高脉冲。

这个在逻辑上没什么难的，只是 E=高脉冲这个问题要解释一下。这个指令一共有4条语句，其中前三条语句顺序无所谓，但是 E=高脉冲这一句很关键。实际上流程是这样的：因为我们现在是写数据，所以我们首先要保证我们的 E 引脚是低电平状态，而前三句不管我们怎么写，1602 液晶只要没有接收到 E 引脚的使能控制，它都不会来读总线上的信号的。当通过前三句准备好数据之后，E 使能引脚从低电平到高电平变化，然后 E 使能引脚再从高电平到低电平出现一个下降沿，1602 液晶内部一旦检测到这个下降沿后，并且检测到 RS=L，R/W=L，就马上来读

取 D0~D7 的数据，完成单片机写 1602 指令过程。归纳总结我们写了个 E=高脉冲，意思就是：E 使能引脚先从低拉高，再从高拉低，形成一个高脉冲。

4、写数据：RS=H，R/W=L，D0~D7=数据，E=高脉冲。

写数据和写指令是类似的，就是把 RS 改成 H，把总线改成数据即可。

此外要顺便提一句，这里用的 1602 液晶所使用的接口时序是摩托罗拉公司所创立的 6800 时序，还有另外一种时序是 Intel 公司的 8080 时序，也有部分液晶模块采用，只是相对来说比较少见，大家知道这么回事即可。

这里还要说明一个问题，就是从这4个时序大家可以看出来，1602 液晶的使能引脚 E，高电平的时候是有效，低电平的时候是无效，前面也提到了高电平时会影响 P0 口，因此正常情况下，如果我们没有使用液晶的话，那么程序开始写一句 LCD1602_E=0，就可以避免 1602 干扰到其它外设。我们之前的程序没有加这句，是因为我们板子在这个引脚上加了一个 15 K 的下拉电阻，这个下拉电阻就可以保证这个引脚上电后默认是低电平，如图12-3所示。

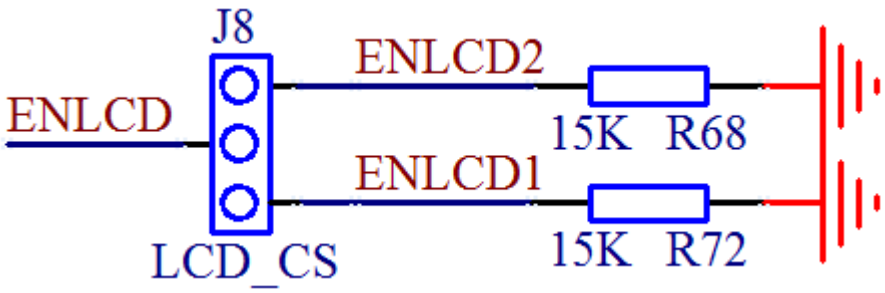

图12-3 液晶使能引脚的下拉电阻

如果不加这个下拉电阻，刚开始讲点亮 LED 小灯的时候，我们就得写一句：LCD1602_E=0，可能很多初学者容易弄不明白，所以我们才加了这样一个电路。但是在实际开发过程中，就不必要这样了。如果这是个实际产品，能用软件去处理的，我们就不会用硬件去实现，所以大家在做实际产品的时候，这块电路可以直接去掉，只需要在程序开头多加一条语句即可。

12.8 1602 液晶指令介绍

与单片机寄存器的用法类似，1602 液晶在使用的时候，我们首先要进行初始的功能配置，1602 液晶有以下几个指令需要了解。

1) 显示模式设置 写指令 0x38，设置 16x2 显示，5x7 点阵，8位数据接口。这条指令对我们这个液晶来说是固定的，必须写 0x38，大家仔细看会发现我们的液晶实际上内部点阵是 5x8 的，还有一些 1602 液晶还兼容串行通信，用2个 IO 口即可，但是速度慢，我们这个液晶就是固定的 0x38 模式。

2) 显示开/关以及光标设置指令 这里有2条指令，第一条指令，一个字节中8位，其中高5位是固定的 0b0000 1，低3位我们分别用 DCB 从高到低表示，D=1 表示开显示，D=0 表示关显示；C=1 表示显示光标，C=0 表示不显示光标；B=1 表示光标闪烁，B=0 表示光标不闪烁。

第二条指令，高6位是固定的 0b000001，低2位我们分别用 NS 从高到低表示，其中 N=1 表示读或者写一个字符后，指针自动加1，光标自动加1，N=0 表示读或者写一个字符后指针自动减1，光标自动减1；S=1 表示写一个字符后，整屏显示左移(N=1)或右移(N=0)，以达到光标不移动而屏幕移动的效果，如同我们的计算器输入一样的效果，而 S=0 表示写一个字符后，整屏显示不移动。

3) 清屏指令 固定的，写入 0x01 表示显示清屏，其中包含了数据指针清零，所有的显示清零。写入 0x02 则仅仅是数据指针清零，显示不清零。

4) RAM 地址设置指令 该指令码的最高位为1，低7位为 RAM 的地址，RAM 地址与液晶上字符的关系如上图12-2所示。通常，我们在读写数据之前都要先设置好地址，然后再进行数据的读写操作。

12.9 1602 液晶简单显示程序

1602 液晶手册提供了一个初始化过程，由于不检测“忙”位，所以程序比较复杂，而我们总结了一个更加简易方便的过程提供给大家，手册上描述的那个，大家仅仅作为了解就可以了，下面我把程序写出来大家看下，我们的初始化只用了4条语句，没有像手册介绍的那么繁琐。

```
 #include <reg52.h>

 #define LCD1602_DB P0

sbit LCD1602_RS = P1^0;
sbit LCD1602_RW = P1^1;
sbit LCD1602_E = P1^5;

void InitLcd1602();
void LcdShowStr(unsigned char x, unsigned char y, unsigned char *str);

void main(){
    unsigned char str[] = "Kingst Studio";
    InitLcd1602();
    LcdShowStr(2, 0, str);
    LcdShowStr(0, 1, "Welcome to KST51");
    while (1);
}
    /* 等待液晶准备好 */
void LcdWaitReady(){
    unsigned char sta;
    LCD1602_DB = 0xFF;
    LCD1602_RS = 0;
    LCD1602_RW = 1;
    do {
        LCD1602_E = 1;
        sta = LCD1602_DB; //读取状态字
        LCD1602_E = 0;
    }while (sta & 0x80); //bit7 等于1表示液晶正忙，重复检测直到其等于0为止
}
/* 向 LCD1602 液晶写入一字节命令，cmd-待写入命令值 */
void LcdWriteCmd(unsigned char cmd){
    LcdWaitReady();
    LCD1602_RS = 0;
    LCD1602_RW = 0;
    LCD1602_DB = cmd;
```

```
    LCD1602_E = 1;
    LCD1602_E = 0;
}
/* 向 LCD1602 液晶写入一字节数据，dat-待写入数据值 */
void LcdWriteDat(unsigned char dat){
    LcdWaitReady();
    LCD1602_RS = 1;
    LCD1602_RW = 0;
    LCD1602_DB = dat;
    LCD1602_E = 1;
    LCD1602_E = 0;
}
/* 设置显示 RAM 起始地址，亦即光标位置，(x,y)-对应屏幕上的字符坐标 */
void LcdSetCursor(unsigned char x, unsigned char y){
    unsigned char addr;
    if (y == 0){ //由输入的屏幕坐标计算显示 RAM 的地址
        addr = 0x00 + x; //第一行字符地址从 0x00 起始
    }else{
        addr = 0x40 + x; //第二行字符地址从 0x40 起始
    }
    LcdWriteCmd(addr | 0x80); //设置 RAM 地址
}
/* 在液晶上显示字符串，(x,y)-对应屏幕上的起始坐标，str-字符串指针 */
void LcdShowStr(unsigned char x, unsigned char y, unsigned char *str){
    LcdSetCursor(x, y); //设置起始地址
    while (*str != '\0'){ //连续写入字符串数据，直到检测到结束符
        LcdWriteDat(*str++); //先取 str 指向的数据，然后 str 自加1
    }
}
/* 初始化 1602 液晶 */
void InitLcd1602(){
    LcdWriteCmd(0x38); //16*2 显示，5*7 点阵，8位数据接口
    LcdWriteCmd(0x0C); //显示器开，光标关闭
    LcdWriteCmd(0x06); //文字不动，地址自动+1
    LcdWriteCmd(0x01); //清屏
}
```

程序中有详细的注释，结合本节前面的讲解，大家自己分析下，掌握 1602 液晶的基本操作函数。LcdWriteDat(*str++)这行语句中对指针 str 的操作大家一定要理解透彻，先把 str 指向的数据取出来用，然后 str 再加1以指向下一个数据，这是非常常用的一种简写方式。另外关于本程序还有几点值得提一下： - 首先，我们把程序所有的功能都使用函数模块化了，这样非常有利于程序的维护，不管要写一个什么样的功能，只要调用相应的函数就可以了，大家注意学习这种编程方法。 - 其次，我们使用液晶的习惯，也是用数学上的(x,y)坐标来进行屏幕定位，但与数学坐标系不同的是，液晶的左上角的坐标是 x=0，y=0，往右边是 x+ 偏移，下边是 y+ 偏移。 - 第三，第一次接触多个参数传递的函数，而且还带了指针类型的参数，所以多留心熟悉一下。 - 第四，读写数据

和指令程序，每次都必须进行“忙”判断。 - 第五，领略一下指针在这个地方的巧妙用法，你可以尝试不用指针改写程序试试，感受一下指针的优势。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/mcu-tutorial-two/