目录

LED 点阵屏学习攻略

在经历了将近一个学期断断续续的点阵屏学习后，最后终于在 AVR 平台下完成了 128*32 点阵屏的无闪烁显示。现把整个学习过程总结如下：

无论是 51 单片机还是 AVR 单片机，点阵屏的显示原理是一样的，所以首先从 51 讲起。

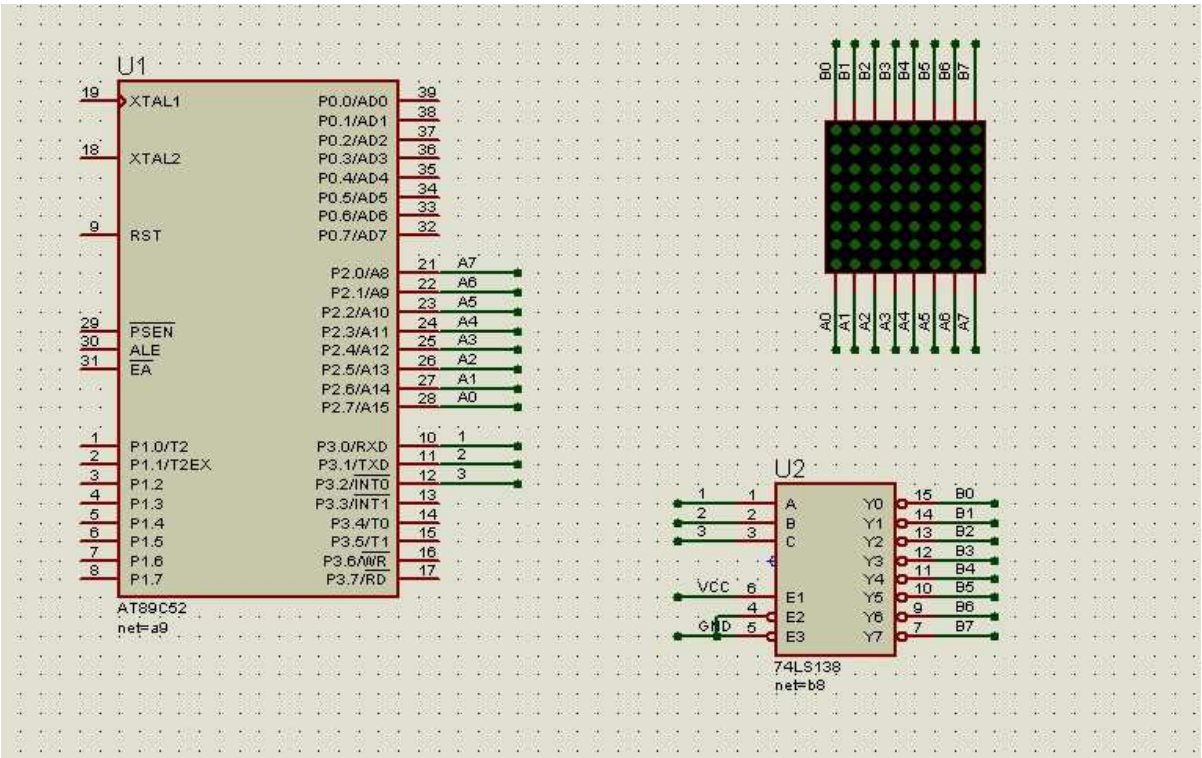
说明：以下所有试验如无特殊说明均在 Keil uVision3 + Proteus 6.9 SP5 下仿真完成。

一．基于 51 的点阵屏显示：

（1）点亮第一个 8*8 点阵:

1.首先在 Proteus 下选择我们需要的元件，AT89C52、74LS138、MATRIX-8*8-GREEN(在这里使用绿色的点阵)。在 Proteus 6.9 中 8*8 的点阵总共有四种颜色，分别为 MATRIX-8*8-GREEN，MATRIX-8*8-BLUE，MATRIX-8*8-ORANGE ，MATRIX-8*8-RED。在这里请大家牢记：红色的为上列选下行选；其它颜色的为上行选下列选！而所有的点阵都是高电平选中列，低电平选中行！也就是说如果某一个点所处的行信号为低，列信号为高，则该点被点亮！此结论是我们编程的基础。

2.在选择完以上三个元件后，我们开始布线，具体如下图：

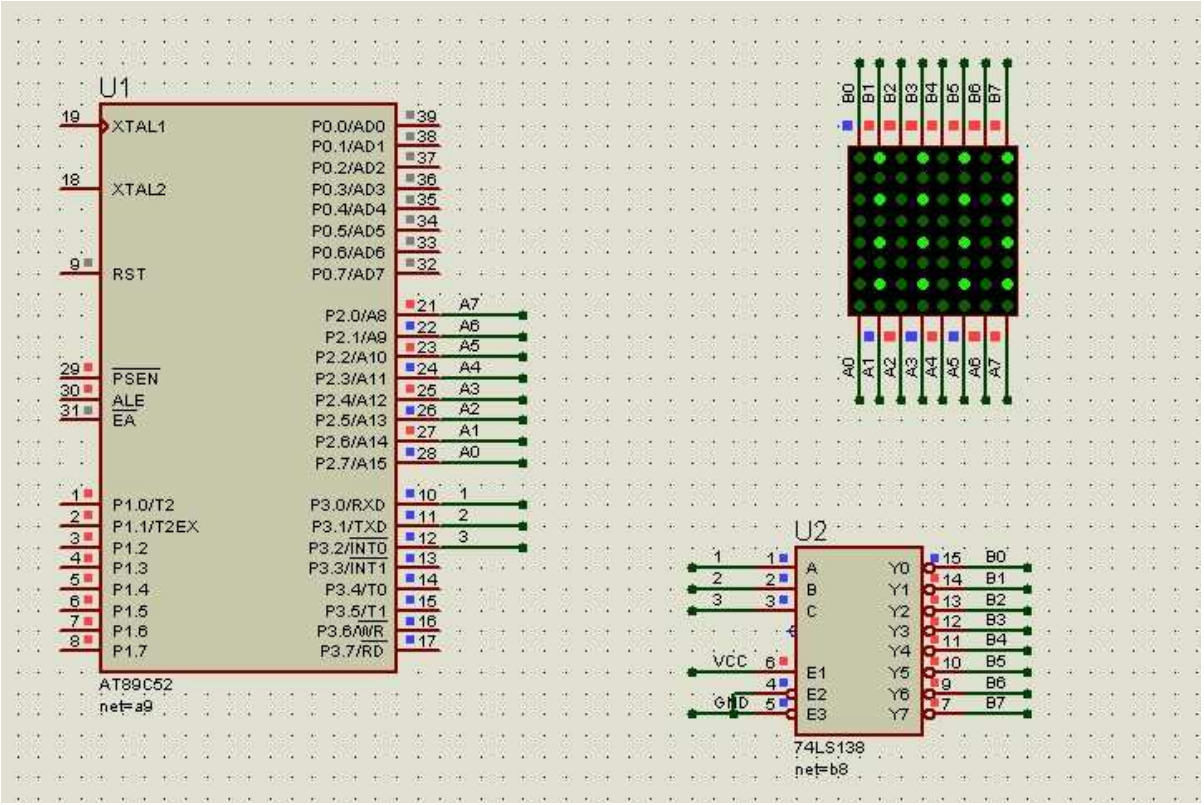


这里 P2 是列选，P3 连接 38 译码器后作为行选。

选择 38 译码器的原因：38 译码器每次可输出相应一个 I/O 口的低电平，正好与点阵屏的低电平选中行相对，并且节省了 I/O 口，大大方便了我们的编程和以后的扩展。

3.下面让我们把它点亮，先看一个简单的程序：

（将奇数行偶数列的点点亮，效果如下图）



下面是源代码：

```
/************8*8LED 点阵屏显示*****************/

#include<reg52.h>


void delay(int z) //延时函数

{
    int x,y;
    for(x=0;x<z;x++)
        for(y=0;y<110;y++);
}

void main()
{
    while(1)
    {
        P3=0;               //行选，选择第一行

        P2=0x55;          //列选，即该行显示的数据

        delay(5);         //延时
```

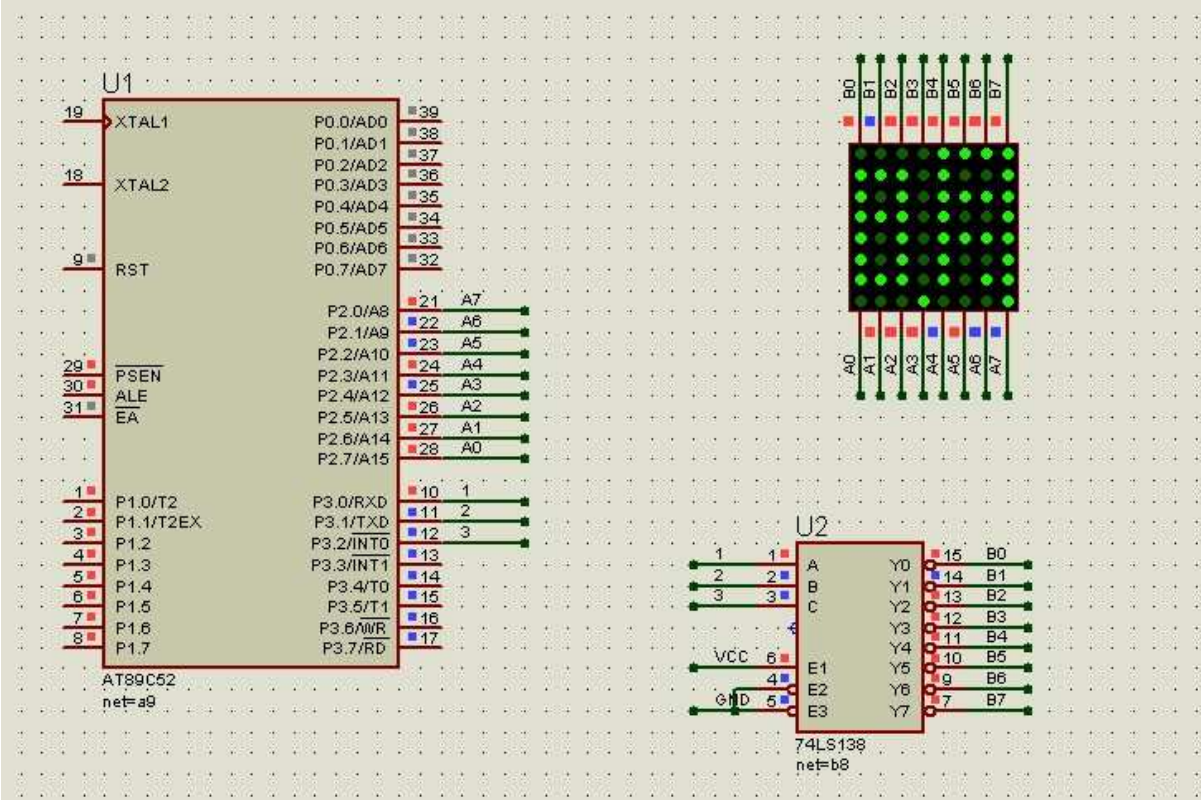
```
        /*****下同*****/

            P3=2;           //第三行

            P2=0x55;
            delay(5);


            P3=4;           //第五行

            P2=0x55;
            delay(5);


            P3=6;            //第七行

            P2=0x55;
            delay(5);
        }
    }
```

上面的程序实现了将此 8*8 点阵的奇数行偶数列的点点亮的功能。重点让我们看 while 循环内，首先是行选 P3=0，此时 38 译码器的输入端为 000，则输出端为 01111111，即 B0 端为低电平，此时选中了点阵屏的第一行，接着列选我们给 P2 口赋 0x55,即 01010101，此时又选中了偶数列，紧接着延时。然后分别对第三、五、七行进行相同的列选。这样就点亮了此点阵屏奇数行偶数列交叉的点。

完成这个程序，我们会发现其实点阵屏的原理是如此简单，和数码管的动态显示非常相似，只不过换了一种方式而已。

4.完成了上面的点亮过程，下面我们让这个 8*8 的点阵屏显示一个汉字 :“明”

先看效果图：



源代码如下：

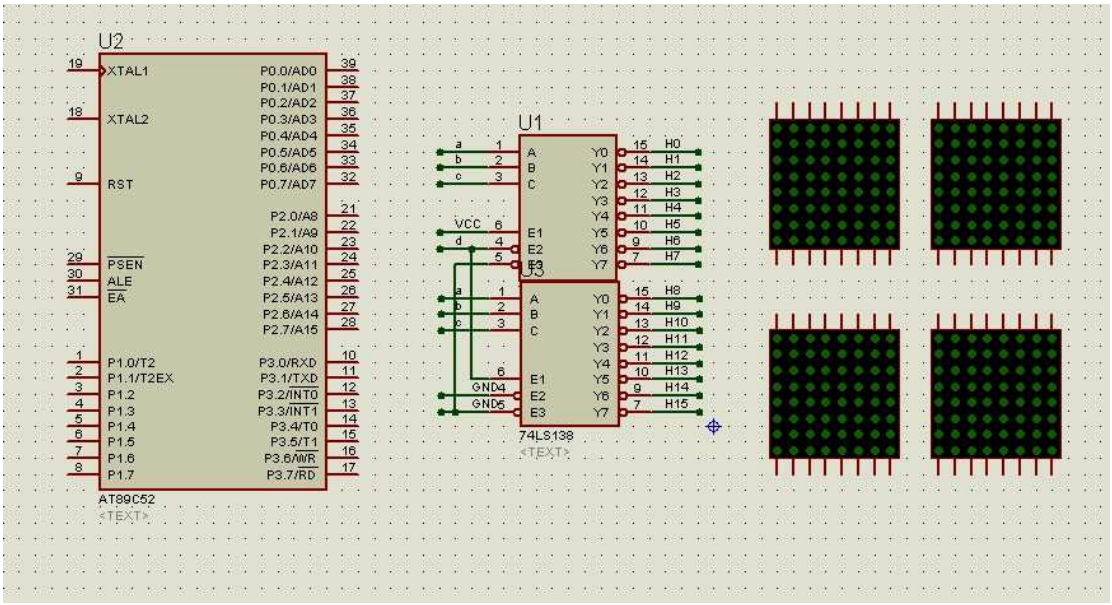
```
/************8*8LED 点阵屏显示*****************/

#include<reg52.h>


char code table[]={0x0f,0xe9,0xaf,0xe9,0xaf,0xa9,0xeb,0x11}; //"明"   字编码


void delay(int z) //延时函数
{
    int x,y;
    for(x=0;x<z;x++)
        for(y=0;y<110;y++);
}

void main()
{
    int num;

    while(1)     //循环显示

    {
```

```
            for(num=0;num<8;num++)          //8 行扫描 P3 行选，P2 列选
          {
            P3=num;                 //行选
            P2=table[num];      //列选
            delay(5);               //延时
          }
      }
}
```

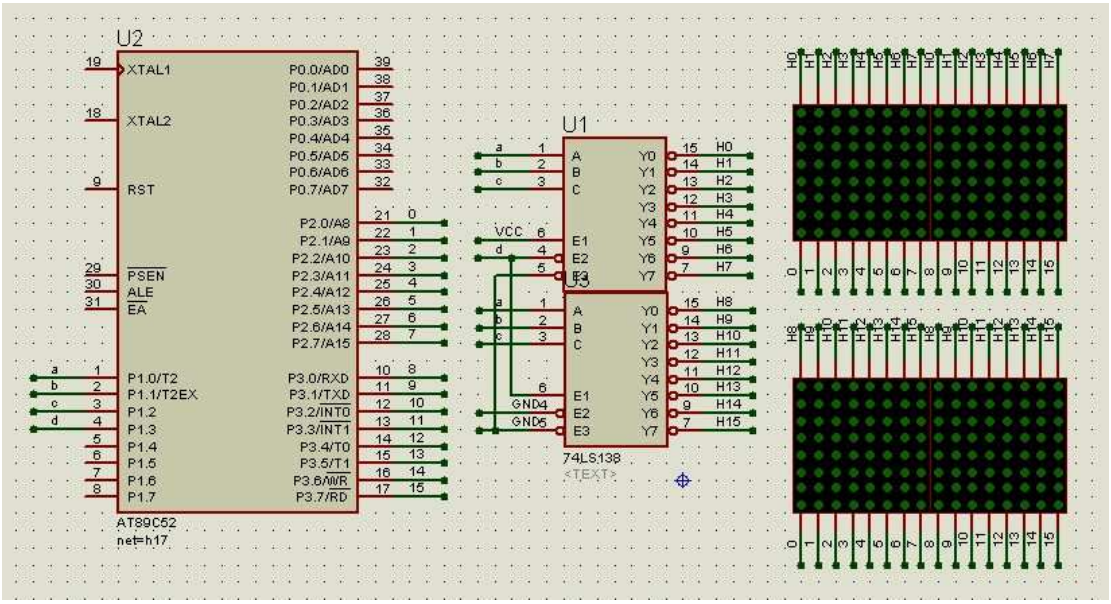
因为要显示一个汉字，这里我们使用了一个数组 table[ ]来存储该字的编码，重点还是来看 while 循环，首先在 for 循环内完成对 8*8 点阵屏的 8 行依次扫描。我们来分析第一行的情况即 num=0 的时候，首先 P3=0，选中第一行，然后 P2=table[0]，即 P2 等于 table 数组中第一个数据 0x0f，则此时就点亮了第一行相应的点。接着延时，其他行同理。这样我们就完成了一个最简单汉字的显示。

（2）16*16 点阵的显示原理

1.虽然完成了上面 8*8 点阵的显示，但是由于点的数量太少以至于它的显示效果并不是很理想，事实上现在大部分点阵的汉字都是 16*16 显示的，下面让我们来学习 16*16 点阵的显示。和上面一样我们先选择元件：AT89C52，74LS138,，MATRIX-8*8-GREEN，因为要显示 16*16 的汉字，我们就不能再使用一个 38 译码器进行行选了，这里我们用两个 38 译码器组合成一个 4 选 16 的译码器（当然也可以使用 74159）。而 MATRIX-8*8-GREEN 点阵需要 4 个。完成后如下图：



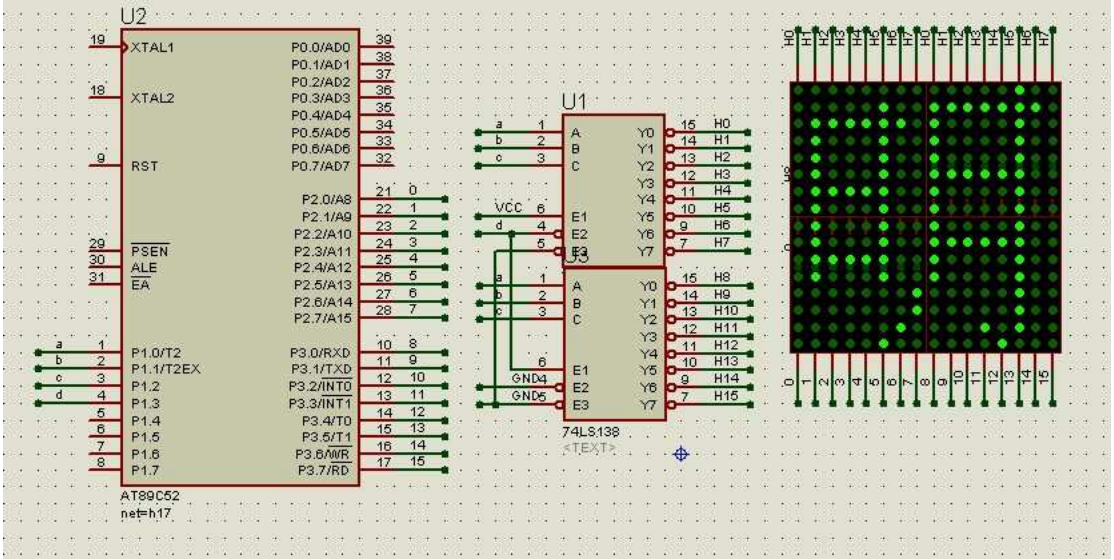
2.先来看看 4 选 16 的译码器是如何工作的，这里有 4 个输入端 a、b、c、d，16 个输出端 H0~H15 ,如上图连线后即可完成类似于 38 译码器一样的工作。只不过扩展到了 16 行选。关于连线的原理这里不再赘述，只要明白 38 译码器的原理这个可以轻松理解。接着完成全部布线。如下图所示：



3.连好线后，P1 作为行选，P2、P3 一起作为列选。现在 16*16 的点阵被分成两块并不完整的部分，我们可以整体移动（包括点阵屏、连线以及连接点，）来方便我们观察显示的效果（最好同时去掉仿真中电平的指示灯）。接着我们来看一个程序，还是让此点阵屏显示

一个汉字“明”。

先看效果图：



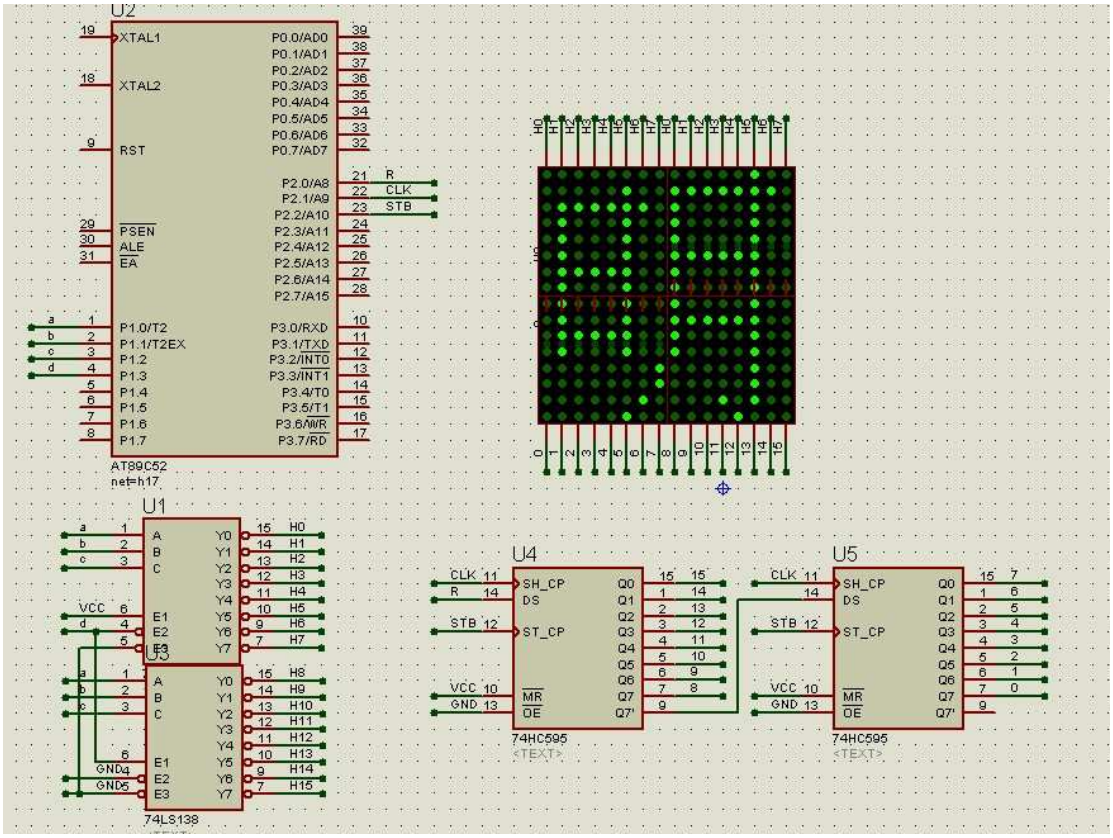
源代码如下：

```
/************16*16LED 点阵屏显示*****************/

#include<reg52.h>

char code table[]={0x00,0x20,0x20,0x7F,0x7E,0x21,0x22,0x21,
                   0x22,0x21,0x22,0x3F,0x3E,0x21,0x22,0x21,
                   0x22,0x21,0x22,0x3F,0x3E,0x21,0x22,0x21,

                   0x80,0x20,0x80,0x20,0x40,0x28,0x20,0x10}; //“明”

void delay(int z)
{
    int x,y;
    for(x=0;x<z;x++)
        for(y=0;y<110;y++);
}

void main()
{
    int num;
    while(1)
    {
        for(num=0;num<16;num++)
        {
          P1=num;                  //行选
```

```
                P2=table[2*num];      //列选
                P3=table[2*num+1];   //列选
                delay(2);
            }
        }
    }
```

4..先来看这次使用的 table 数组，因为是 16*16 的点阵，所以总共有 32 个数据，其中第 1、2 个数据用于第一行的显示，第 2、3 个数据用于第二行的显示，以此类推，总共 16 行。然后还是来看 while 循环内，同样 for 循环依次扫描 16 行，以第一行为例，即 num=0 时，首先 P1=0，选中第一行，P2=table[0]、P3=table[1]送出列选数据，即第一行要显示的两个字节的数据。其他行同理。这样很轻松的我们就完成了 16*16 点阵的显示。程序虽然完成了，但是回过头来看一看就会发现，我们在这里使用了 P2 与 P3 口一起来做列选，浪费了大量的 I/O/资源，而且现在点阵屏的大小还只有 16*16，如果想要扩展的更大，已经没有足够的 I/O 口可用了。所以一定要想出更好的办法进行列选。

5.为了解决上面提到的问题，我们来学习一个新的元件：74HC595。它实质上是一个串行移位寄存器，能够实现**"串入并出"**的功能，关于它的使用我们还是用上一个列子来讲解，先来看看它的实现，如图：



可以看到这里我们仅使用了三个 I/O 口就完成了列选数据的发送。主要来看 74HC595 是如何实现“串入并出”的，这里我们使用了两个 595 进行了级联，即第二个 595 的数据输入端连接了第一个 595 的级联输出口 Q7’。也就是说，我们只需要从第一个 595 的输入端串行输入数据，便可以实现把数据送入第二个 595 的功能。而且 595 的数量可以进行无限的级联，而不管有多少个 595，我们只需要一个数据输入端就可以，这样就大大节省了 I/O 资源。对于 595 的具体使用还是来看程序。

源代码如下：

```
/************16*16LED 点阵屏显示*****************/
#include<reg52.h>

sbit R=P2^0;      //数据输入端口

sbit CLK=P2^1;        // 时钟信号

sbit STB=P2^2;        // 锁存端
```

```
char code table[]={0x00,0x20,0x20,0x7F,0x7E,0x21,0x22,0x21,
                   0x22,0x21,0x22,0x3F,0x3E,0x21,0x22,0x21,
                   0x22,0x21,0x22,0x3F,0x3E,0x21,0x22,0x21,
                   0x80,0x20,0x80,0x20,0x40,0x28,0x20,0x10}; //“明”
void delay(int z)
{
    int x,y;
    for(x=0;x<z;x++)
        for(y=0;y<110;y++);
}

void WriteByte(char dat)            //写一个字节的数据
{
    char i;
    for(i=0;i<8;i++)            //循环 8 次把编码传给锁存器
        {
          dat=dat>>1;  //右移一位，取出该字节的最低位
          R=CY;               //将该字节的最低位传给 R
          CLK=0;            //将数据移入 595，上升沿
          CLK=1;
        }
}

void main()
{
    int num;
    while(1)
    {
         for(num=0;num<16;num++)
        {
          WriteByte(table[2*num]);     //送出一个字节
          WriteByte(table[2*num+1]);
          P1=num;         //行选
          STB=1;          //输出锁存器中的数据，下降沿
          STB=0;
```

```
            delay(2);
        }
    }
}
```

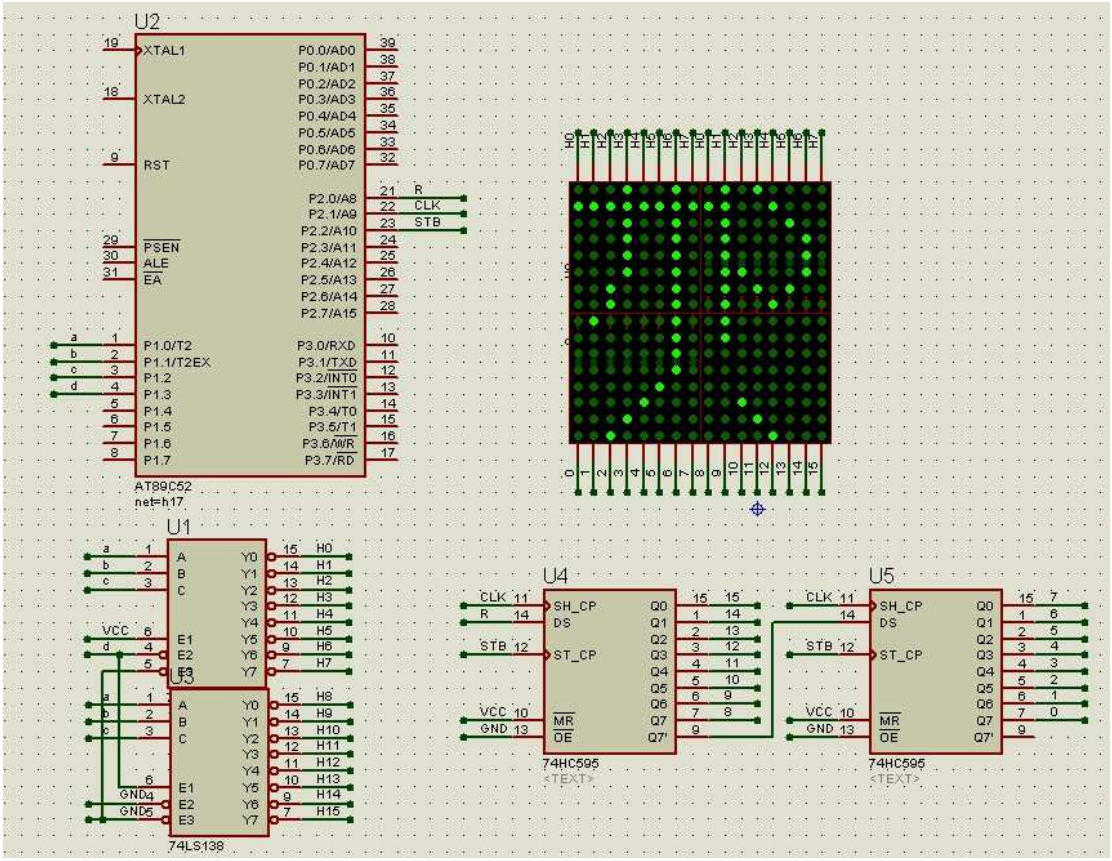
先来看不同之处，这里我们首先位定义了 R、CLK、STB，分别对应于 74HC595 的 DS、SH_CP、ST_CP 用以实现串行数据输入、数据移位以及并行数据输出。然后来看 WriteByte(char dat) 函数，该函数实现了串行向 595 中输入一个字节数据的功能。来看 for 循环，首先 dat=dat>>1，把要输入的数据右移一位，这样最低位便进入移位寄存器 CY 中，紧接着我们让 R=CY，把该位传给 595 的输入端，CLK 一个上升沿的跳变就实现了把该位数据移入 595 的功能。8 次循环便可以将一个字节的数据送出。重点还是看 while 循环内，同样也是 16 行的扫描，然后就是 WriteByte(table[2*num]) 等同于上面的 P2=table[2*num] ,WriteByte(table[2*num+1]) 等同于 P3=table[2*num+1]，完成列选，接着行选，然后有一个 STB 的下降沿的跳变，这个变化能够实现并行输出移位寄存器中的数据。这样就完成了整个过程。

（3）16*16 点阵的移位控制

点阵的移位一般有上、下、左、右的移动，这里我们重点讲上移和左移，其它同理。

1. 点阵的上移：

点阵的上移相对来说很简单，看效果图如下：



源代码 :(该程序实现了循环上移显示“邢台”)

```
/************16*16LED 点阵屏显示*****************/

#include<reg52.h>


sbit R=P2^0;      //数据输入端口

sbit CLK=P2^1;        //  时钟信号

sbit STB=P2^2;        //  锁存端

char code table[]={

/*--  文字:  邢  --*/

/*--  宋体 12;  此字体下对应的点阵为 : 宽 x 高=16x16    --*/

        0x00,0x00,0xFE,0x3E,0x48,0x22,0x48,0x22,
        0x48,0x12,0x48,0x12,0x48,0x0A,0xFF,0x13,
        0x48,0x22,0x48,0x42,0x48,0x42,0x48,0x46,
        0x44,0x2A,0x44,0x12,0x42,0x02,0x40,0x02,

/*--  文字:  台  --*/
```

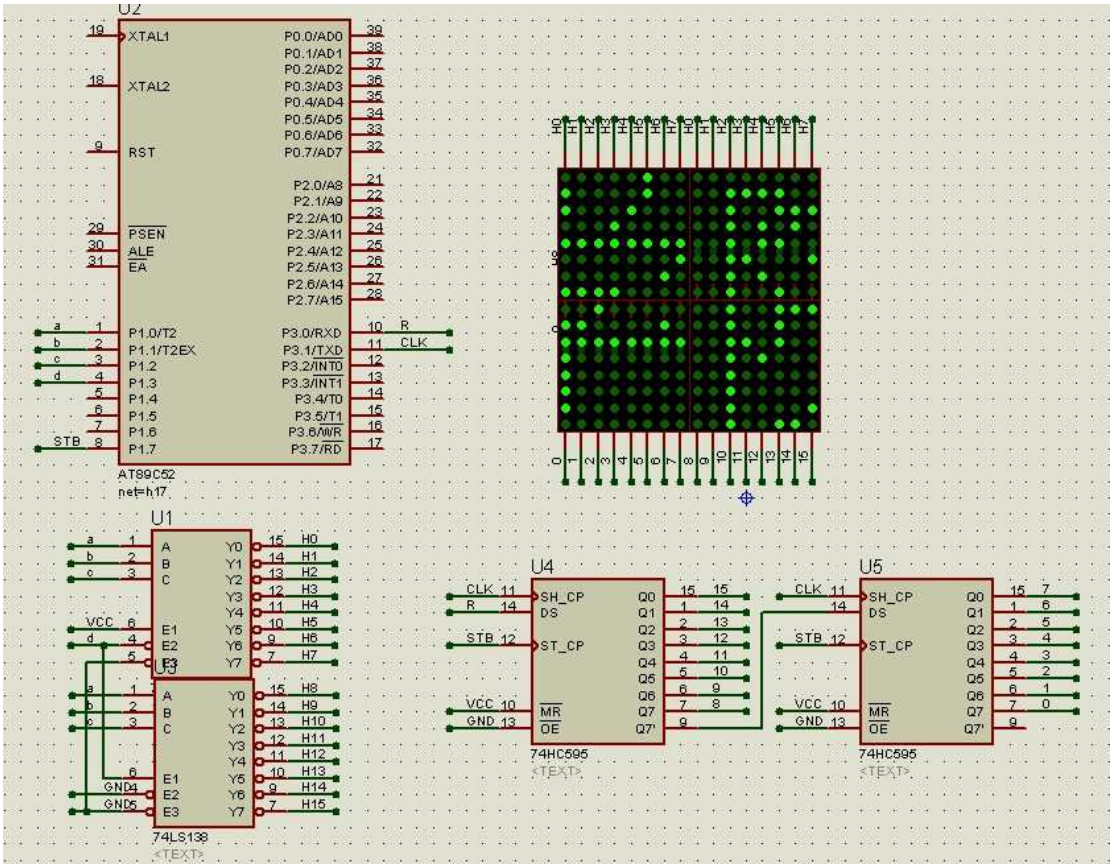
```
/*--  宋体 12;  此字体下对应的点阵为：宽 x 高=16x16    --*/
    0x40,0x00,0x40,0x00,0x20,0x00,0x10,0x04,
    0x08,0x08,0x04,0x10,0xFE,0x3F,0x00,0x20,
    0x00,0x08,0xF8,0x1F,0x08,0x08,0x08,0x08,
    0x08,0x08,0x08,0x08,0xF8,0x0F,0x08,0x08,
};

void delay(int z)
{
    int x,y;
    for(x=0;x<z;x++)
        for(y=0;y<110;y++);
}

void WriteByte(char dat)            //写一个字节的数据
{
    char i;
    for(i=0;i<8;i++)            //循环 8 次把编码传给锁存器
        {
          dat=dat>>1;  //右移一位，取出该字节的最低位
          R=CY;                //将该字节的最低位传给 R
          CLK=0;            //将数据送出，上升沿
          CLK=1;
        }
}

void main()
{
    int num,move,speed;
    while(1)
    {
        if(++speed>8)     //移动速度控制
        {
            speed=0;
            move++;       //移位
            if(move>16)   //是否完成移位一个汉字
```

```
                move=0;    //从头开始
            }

            for(num=0;num<16;num++)
           {
              WriteByte(table[2*num+move*2]);      //送出一个字节
              WriteByte(table[2*num+1+move*2]);
              P1=num;          //行选

              STB=1;          //输出锁存器中的数据，下降沿
              STB=0;
              delay(2);
           }
      }
}
```

可以看到这个程序和静态显示的程序没有太大的差距，主要就是加入了一个 move 变量来控制移动，WriteByte(table[2*num+move*2])中当 move 变量变化的时候更改了写入 595 中的数据，正好实现了移动显示的效果。而 speed 变量的 if 判断语句能够控制移动速度的大小。下面重点讲左移。

2. 点阵的左移：

因为点阵的数据最终是一个一个字节的并行送出的，所以要实现点阵的左移，我们就需要考虑如何才能够动态的更改每一个发送字节的数据，而汉字的每一个字节的编码是固定的，这里我们可以使用一个数据缓冲区来完成点阵的左移。重点说一下点阵左移中关键的一步操作 temp=(BUFF[s]>>tempyid) | (BUFF[s+1]<<(8-tempyid))。这里 temp 作为要发送的一个字节数据，它由数据缓冲区中的数据组合而成，并且动态的变化，大致来说就是首先第一个字节的数据右移 tempyid 位，第二个字节的数据左移 8-tempyid 位，两者相或后组成一个字节新的数据，只要我们一直不断地移位、相或、发送，就能实现左移的效果。不太好理解，先来看实例（循环左移显示“邢台学院”），效果图如下：



见源代码：

```
#include <AT89x51.H>
#define uchar unsigned char
#define uint unsigned int

uchar yid,h;                    //YID 为移动计数器，H 为行段计数器

uint zimuo;                     //字模计数器

uchar code hanzi[];             //汉字字模

uchar BUFF[4];                  //缓存

void in_data(void);             //调整数据

void rxd_data(void);            //发送数据

void sbuf_out();                //16 段扫描


uchar code table[]={//篇幅有限，省略编码};
```

```
void main(void)
{
 uchar i,d=10;
 yid=0;
 zimuo=0;
 while(1)
  {
    while(yid<16)                          //数据移位。
    {
         for(i=0;i<d;i++)                  //移动速度
          {
          sbuf_out();
          }
          yid++;                  //移动一步
    }
   yid=0;
   zimuo=zimuo+32;                  //后移一个字，
   if(zimuo>=96)                  //到最后从头开始，有字数决定
   zimuo=0;
  }
}
/*********************************/
void sbuf_out()
        {
          for(h=0;h<16;h++)     //16 行扫描
               {
               in_data();                  //调整数据
               rxd_data();                //串口发送数据
               P1=0x7f;                   //关闭显示。
               P1_7=1;                    //锁存为高，595 锁存信号
               P1=h;                      //送行选
               }
```

```
            }

/*****************************************************/
void in_data(void)
{
 char s;
     for(s=1;s>=0;s--)                //h 为向后先择字节计数器，zimuoo 为向后选字计数器
     {
      BUFF[2*s+1]=table[zimuo+1+32*s+2*h]; //把第一个字模的第一个字节放入 BUFF0
                                           //中,第二个字模的第一个字节放入 BUFF2 中
      BUFF[2*s]=table[zimuo+32*s+2*h];     //  把第一个字模的第二个字节放入 BUFF1 中,
                                           //第二个字模的第二个字节放入 BUFF3 中
     }
}

/*********************************************************/
void rxd_data(void)                       //串行发送数据
{
  char s;
  uchar inc,tempyid,temp;
  if(yid<8)
    inc=0;
  else
    inc=1;
  for(s=0+inc;s<2+inc;s++)                //发送 2 字节数据
    {
      if(yid<8)
      tempyid=yid;
      else
      tempyid=yid-8;
     temp=(BUFF[s]>>tempyid)|(BUFF[s+1]<<(8-tempyid));//h1 左移 tempyid 位后和 h2 右移
     8-tempyid 相或，取出移位后的数据

    SBUF=temp;//把 BUFF 中的字节从大到小移位相或后发送输出。
```

```
        while(!TI);    //注：这里使用了串口，串口数据的发送为最低位在前。
        TI=0;          //等待发送中断
        }
}
```

首先来看定义的数据缓冲区 BUFF[ ]，这里一开始将会存储第一个汉字与第二个汉字的第一行的编码，该缓冲区动态的存储点阵屏每一行要发送的数据，注意这里 BUFF 的大小为 4 个字节，比 16*16 点阵屏要显示的汉字多了一个汉字行的大小，这一点是必要的，这样我们才能实现利用该缓冲区进行左移控制，接着来看 in_data(void)函数，利用该函数，我们实现了动态的修改缓冲区中的数据，这里不再详述过程，重点看程序的注释即可。然后看 rxd_data(void)函数，该函数的作用正是利用串口串行发送数据，也就是上面提到的移位、相或然后发送，关于在移位过程中的具体实现细节以及如何协调的进行数据发送，首先来看 inc 变量，该变量决定了从 BUFF 缓冲区中的第一个还是第二个数据开始读取，当移位开始后，在移完一个字节的数据之前我们都从 BUFF 数据缓冲区中的第一个字节开始读取，当移完一个字节后，inc 变成 1，这时我们从 BUFF 数据缓冲区中的第二个字节开始读取，于此同时后一个字节总是在和前一个字节的数据进行移位相或，达到慢慢向前推进的效果，这里有一个临界点，就是当移位满 16 位后，即一个汉字移出点阵屏后，这时候我们就需要将数据缓冲区中的数据进行更新，即后移一个字，这时数据缓冲区中的数据就变成了第二个汉字和第三个汉字的第一行汉字的编码，以此类推。下面来看 sbuf_out()函数，该函数实现了 16 行的扫描，最后来看 while 循环内，这时主函数内已经很简单了，首先在 while(yid<16)内，有控制移动速度的 for 循环，即显示几次静态的画面移动一步，而 zimuo 变量为移位过程中汉字的选择变量，它每 32 位的变化，正好是一个 16*16 汉字的编码个数。这样就完成了整个点阵左移的控制（这里使用了串口实现点阵的左移，当然我们也可以不用串口，关于非串口实现的左移后面介绍），它的过程比较复杂，需反复思考。

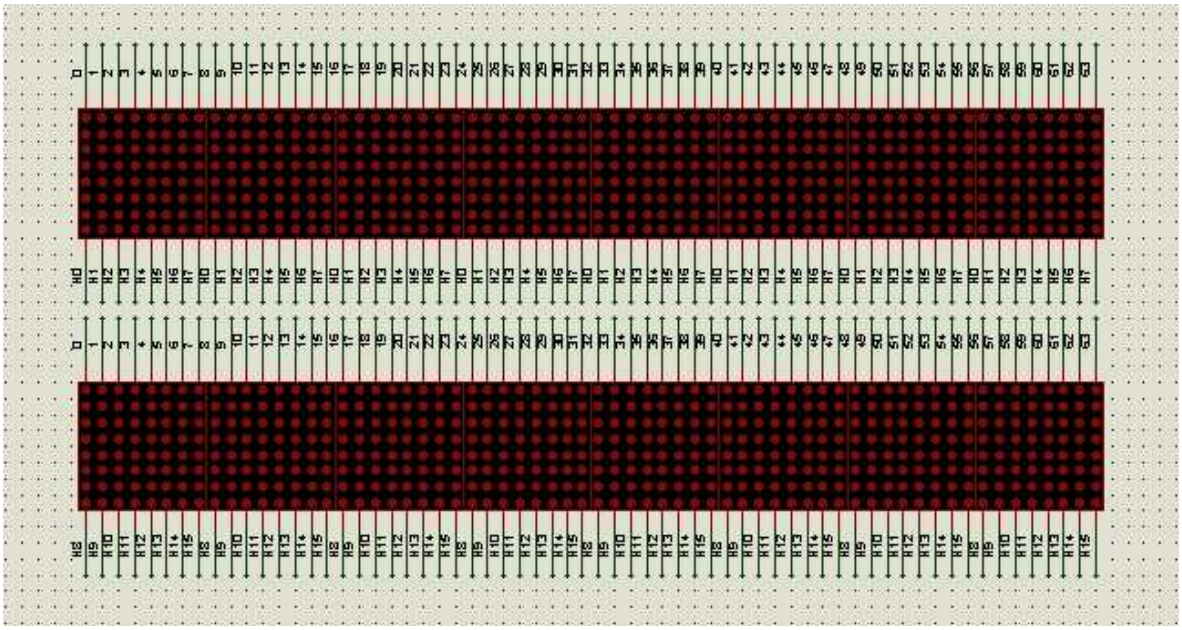
（4）128*32 点阵扩展显示

1. 128*32 点阵的静态显示

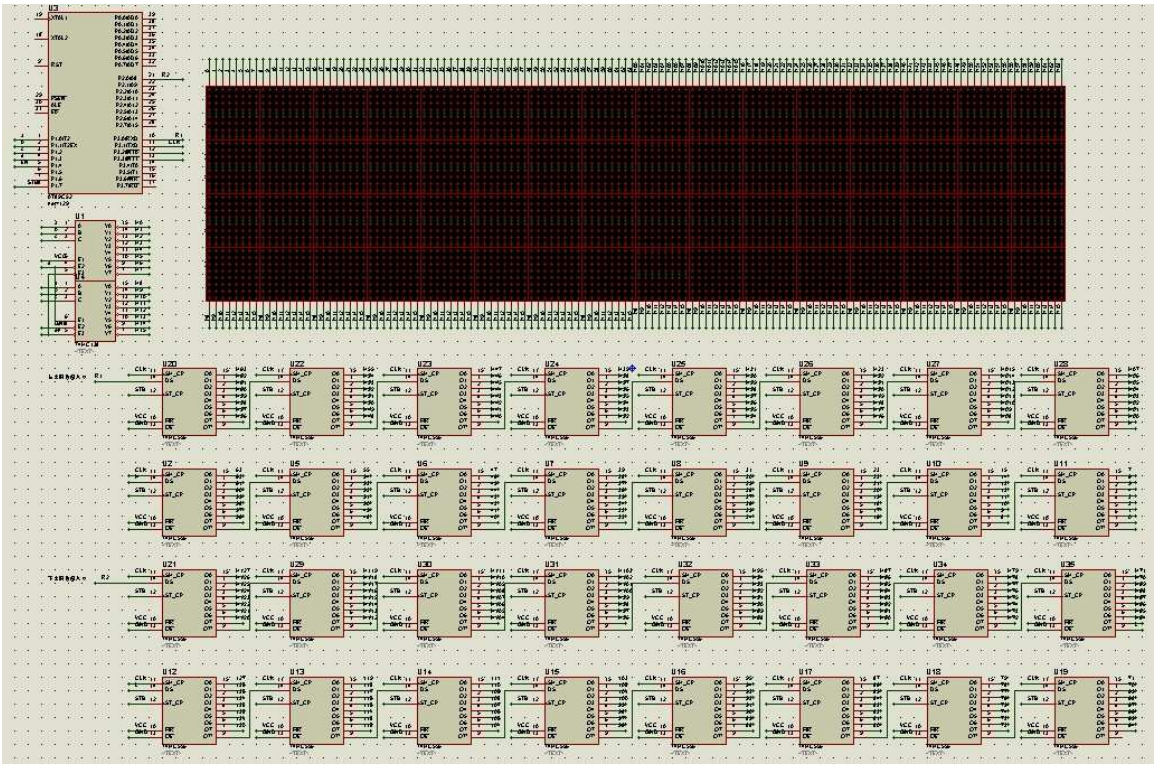
完成了 16*16 点阵的静态与移动显示之后，就已经算是掌握了点阵屏显示的主要部分，以后不管想要操纵什么样的点阵屏，只要把握上面的原理，都能按照我们的想法进行显示。所以接下来的讲解不会再有上面那么详细。

下面来看 128*32 点阵是如何显示的。

这里的布线有点繁琐，首先来看一下 64*16 点阵的布线,如下图：

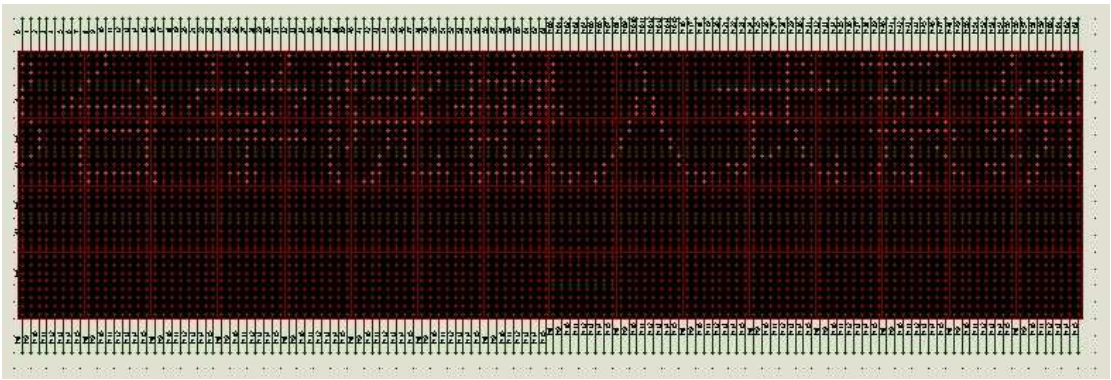


前面已经提到过，在该仿真环境下红色点阵为上列选下行选。理解了 64*16 的点阵布线，我们来看一下 128*32 的仿真图：



这是一个完整的 128*32 的点阵屏，只是在上面小屏的基础上级联了更多的 595，因为只有一个 4 选 16 的译码器，而该点阵有 32 行，这里我们使用两个数据输入端，分别对应点阵屏的上、下半屏。下面以操作上半屏为例（下半屏同理）。

看效果图：



见源代码：

```
#include <AT89x51.H>
#define uchar unsigned char
#define uint unsigned int

uchar yid,h;                    //YID 为移动计数器，H 为行段计数器
```

```
uint zimuo;                    //字模计数器

uchar code hanzi[];            //汉字字模

uchar BUFF[18];                //缓存

void in_data(void);            //调整数据

void rxd_data(void);           //发送数据

void sbuf_out();               //16 段扫描


uchar code table[]={//篇幅有限，省略编码};


void main(void)
{
 uchar i,d=10;
 yid=0;
 zimuo=0;
 while(1)
  {
    while(yid<16)                        //数据移位。
    {
          for(i=0;i<d;i++)               //移动速度
           {
           sbuf_out();
           }
           yid++;                  //移动一步
    }
   yid=0;
   zimuo=zimuo+32;                  //后移一个字，
   if(zimuo>=480)                  //到最后从头开始，有字数决定
   zimuo=0;
  }
}
/*********************************/
void sbuf_out()
```

```
        {
          for(h=0;h<16;h++)     //16 行扫描
              {
              in_data();                  //调整数据
              rxd_data();                 //串口发送数据
              P1=0x7f;                    //关闭显示。
              P1_7=1;                     //锁存为高，595 锁存信号
              P1=h;                       //送行选
              }
          }
/*******************************************************/
void in_data(void)
{
 char s;
     for(s=8;s>=0;s--)              //h 为向后先择字节计数器，zimuoo 为向后选字计数器
     {
      BUFF[2*s+1]=table[zimuo+1+32*s+2*h]; //把第一个字模的第一个字节放入 BUFF0
                                           //中,第二个字模的第一个字节放入 BUFF2 中
      BUFF[2*s]=table[zimuo+32*s+2*h];     //  把第一个字模的第二个字节放入 BUFF1 中,
                                           //第二个字模的第二个字节放入 BUFF3 中
     }
}
/*******************************************************/
void rxd_data(void)                       //串行发送数据
{
  char s;
  uchar inc,tempyid,temp;
  if(yid<8)
     inc=0;
```

```
  else
    inc=1;
  for(s=0+inc;s<8+inc;s++)              //发送 8 字节数据
    {
      if(yid<8)
      tempyid=yid;
      else
      tempyid=yid-8;
      temp=(BUFF[s]>>tempyid)|(BUFF[s+1]<<(8-tempyid));//h1 左移 tempyid 位后和 h2 右移
      8-tempyid 相或，取出移位后的数据

    SBUF=temp;//把 BUFF 中的字节从大到小移位相或后发送输出。
    while(!TI);   //注：这里使用了串口，串口数据的发送为最低位在前。
    TI=0;          //等待发送中断
    }
}
```

该程序同上面的左移程序大同小略，只有几处不同。它同样实现了上半屏循环左移显示一系列汉字的功能。对该程序，就不再详细讲解。

二．基于 AVR 的点阵屏显示

（1）静态显示

上面已经讲完了基于 51 的点阵屏显示，相信大家已经掌握了点阵屏显示的原理，下面仍然依据该原理，我们使用 AVR 单片机来进行控制。

下面的程序均在 ICCAVR Version 6.31A 环境下编写并均在点阵实验板上测试通过。

首先来看一下 128*32 点阵屏两行静态显示的程序，源代码如下：

```
#include<iom16.h>
#include<macros.h>
```

```
#define uchar unsigned char

#define uint unsigned int

#include "delay.h"

#include "code.h"

#pragma data: data

#define screen_size      8                  //半屏显示汉字个数:8   32*128


uchar BUFF_1[screen_size*2+2];          //缓存

uchar BUFF_2[screen_size*2+2];          //缓存

uchar disrow;                           //disrow  为 16 行变量


uchar temp_up,temp_down;

uchar Move_up,Move_down;

uchar temp_up,temp_down;

uint   zimo_up,zimo_down;


#define       HC595_data1_H()       PORTB |= BIT(0)

#define       HC595_data1_L()       PORTB &=~BIT(0)

#define       HC595_data2_H()       PORTB |= BIT(1)

#define       HC595_data2_L()       PORTB &=~BIT(1)

#define       HC595_clk_H           PORTB |= BIT(2)

#define       HC595_clk_L           PORTB &=~BIT(2)
```

```
#define        HC595_lock_H            PORTB |= BIT(3)

#define        HC595_lock_L            PORTB &=~BIT(3)

/*******************************************************

*                  函数说明：595 发送一个字节数据                  *

*******************************************************/

void HC595_send_2byte(uchar byte1,uchar byte2)

{

 uchar i;


 HC595_lock_L;

 for (i=0x01;i!=0;i=i<<1)

  {

    if(byte1&i)

        HC595_data1_L();

    else

        HC595_data1_H();

    if(byte2&i)

        HC595_data2_L();

    else

        HC595_data2_H();


    HC595_clk_H;
```

```
    HC595_clk_L;


    }
}

/*********************************************************
函数名:void Move_Up(const uchar *p,uint f)
功能:上半屏缓存数据  左移
输入:
输出:
*********************************************************/
void Move_Up(const uchar *p,uint f)
{
  signed char s;
      for(s=screen_size;s>=0;s--)
      {
        BUFF_1[2*s]=p[f+32*s+2*disrow];
        BUFF_1[2*s+1]=p[f+1+32*s+2*disrow];
      }
}

/*********************************************************
```

函数名:void Move_Down(const uchar *p,uint f)

功能:下半屏缓存数据 左移

输入:

输出:

```
/*************************************************************/
void Move_Down(const uchar *p,uint f)
{
    signed char s;
     for(s=screen_size;s>=0;s--)
     {
       BUFF_2[2*s]=p[f+32*s+2*disrow];
       BUFF_2[2*s+1]=p[f+1+32*s+2*disrow];
     }
}
```

```
/*************************************************************
```

函数名:void display(void)

功能:显示刷新

输入:

输出:

```
/*************************************************************/
void display(void)
```

```
{
    uchar i = Move_up;
    uchar j = Move_down;
    uchar s;
    uchar inc,tempyid,temp1,temp2;

    if(i<8)
      inc=0;
    else
      inc=1;
    for(s=0+inc;s<screen_size*2+inc;s++)            //发送 16 字节数据
      {
        if(i<8)
          tempyid=i;
        else
          tempyid=i-8;
        temp1=((BUFF_1[s]>>tempyid)|(BUFF_1[s+1]<<(8-tempyid)));
        temp2=((BUFF_2[s]>>tempyid)|(BUFF_2[s+1]<<(8-tempyid)));
          HC595_send_2byte(temp1,temp2);      //发送数据
       }
}
void port_init(void)
```

```
{
  PORTA = 0x00;
  DDRA   = 0xFF;

  PORTB = 0x00;
  DDRB   = 0xFF;

  PORTC = 0x00; //m103 output only
  DDRC   = 0xFF;

  PORTD = 0x00;
  DDRD   = 0xFF;
}

//call this routine to initialize all peripherals
void init_devices(void)
{
  //stop errant interrupts until set up
  CLI(); //disable all interrupts
  port_init();

  MCUCR = 0x00;
```

```
 GICR   = 0x00;
 TIMSK = 0x00; //timer interrupt sources
 SEI(); //re-enable interrupts
 //all peripherals are now initialized
}

void main(void)
{
 init_devices();
 //insert your functional code here...

 while(1)                                    //重复循环显示
  {
      for(disrow=0;disrow<16;disrow++)
          {

            Move_Up(table1,0);          //上屏显示
            Move_Down(table2,0);        //下屏显示

            display();                  //刷新显示数据

            PORTA=disrow;                   //输出行信号
```

```
            HC595_lock_H;              //锁存为高，595 锁存信号
            HC595_lock_L;
           // delay_us(200);
            }
    }
}
```

该程序同上面的左移程序的核心内容是一样的，但是这里并没有移位也没有使用串口，而是在调整好数据缓冲区的内容后，直接通过 I/O 口发送的。这里的 HC595_send_2byte 函数实现了同时发送上下半屏两个字节数据的功能，大大化简了程序。其他基本和上面的程序一样。

（2）移位控制

接着来看移位是如何实现的，完成了上面的静态显示，移位对于我们来说就已经非常简单了，只需要在上面程序的基础上进行如下的修改：

首先加入一个新的函数来控制移位：

```
void Speed_up(speed)
{

        if(temp_up++>=speed)              //控制上屏移动显示速度
       {
          temp_up=0 ;
         if(Move_up++>=15)
          {
           Move_up=0;                  //16 列移位计数器清零
```

```
            zimo_up=zimo_up+32;          //取字模计数器加 32，准备下一个字
            if(zimo_up>=(hanzi_size_S*32)) //字模是否到最后来。
                zimo_up=0;               //从头在来。
            }
        }
}
```

主函数修改如下:

```
void main(void)
{
 init_devices();
 //insert your functional code here...
 while(1)                                //重复循环显示
  {

     for(disrow=0;disrow<16;disrow++)
         {

            Move_Up(table1,zimo_up);          //上屏移动显示 OURAVR
            Move_Down(table2,zimo_up);

            display();                        //刷新显示数据
```

```
            PORTA=disrow;                    //输出行信号


            HC595_lock_H;              //锁存为高，595 锁存信号
            HC595_lock_L;


           // delay_us(200);
            }
          Speed_up(8);    //移动速度控制
    }
}
```

这样就完成了点阵整屏左移的控制，有了上面的基础，我们会发现这里并没有什么难度了。讲到这里我们已经基本上介绍完了点阵屏的基本控制，当然点阵屏还有更多的内容，这里只是利用了行扫描实现了最简单的上、下、左、右的移动，我们还可以利用点的扫描去实现更多的特效，有兴趣的可以自己去探索。