讲座专家：EEWORLD 专家——老练
工作单位：南京华岳电子（www.njhuayue.com）
更多讲座内容，详见（http://bbs.eeworld.com.cn/thread-66169-1-1.html ）

第一讲 单片机 IO 口的使用

单片机的 IO 口控制是单片机初学者最为关心的问题，如何快速学会使用 IO 是初学者最为困难的地方。

众多的教科书上面介绍了很多 IO 的原理，这些长篇大论让很多初学者看起来难以理解，同时也会止步于单片机门外。我们现在所要学习的使用 IO 就是很简单的使用就可以了，IO 无非就是 4 种状态，输出为高、输出为低、输入为高、输入为低。

我们只要把握这四个方面就可以了，先看看我们的单片机接口，单片机共有 32 个 io。分别为 P0、P1、P2、P3 口，P0 口如果当作 IO 来使用时，必须要使用上拉电阻，因为 51 单片机内部没有上拉这一功能。

在这个学习板上面，大家可以看到 40 个 io 对应的位置，并且在数码管下面就增加了上拉电阻，使得 io 可以实现。

现在我们以 P1 口接的 led 指示灯来说明输出高低电平的功能，当我们使用汇编程序：

```
        ORG 0000H
        LJMP MAIN
        ORG 0100H
MAIN:   MOV P1,#00H
        JMP   MAIN
        END
```

这样，P1 口的 led 就会变亮，因为 P1 口赋值#00h 以后，就是将 led 的所有 io 都设置为低电平了。

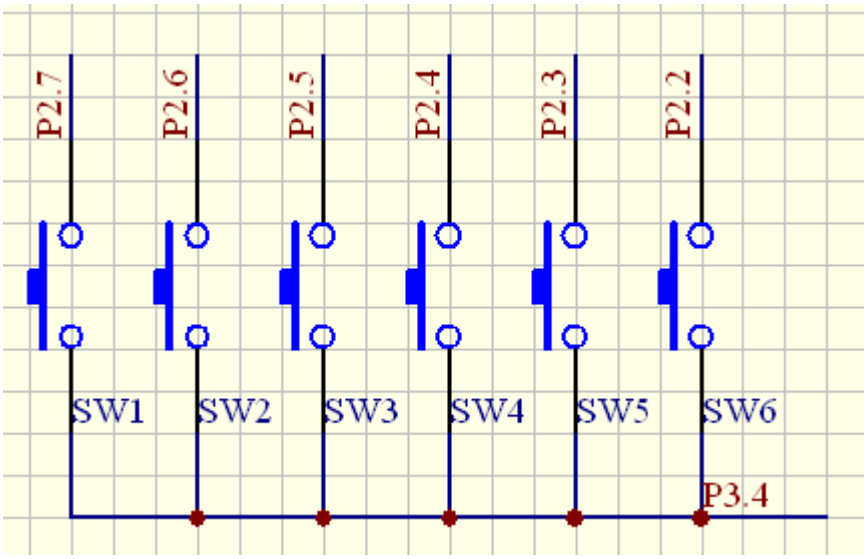
如果将 MOV P1,#00H 改成 MOV P1,#0FFH，那么 io 就设置成了高电平，led 就会被熄灭。当然 C51 中也比较简单。

```
#include <reg51.h>
Main()
{
 P1=#00H;
```

```
 While(1)
 {;}
}
```

这个同样的道理，如果 P1=#0FFH，就是熄灭 led。

当我们对键盘的程序如下图解释：



这个就是我们板子上面的键盘的接线图，当我们单独使用键盘的时候，我们要将 P3.4 置低电平，此时，我们也是根据前面 led 设置 io 的方式进行设置。

如下程序：

```
        ORG    0000H
        JMP     MAIN
        ORG    0100H
MAIN:   CLR    P3.4
        MOV    P1,#0FFH
        JNB     P2.7, MAIN
        MOV     P1,#00H
        JMP     MAIN
        END
```

C51 中：

```
      #include <reg51.h>
      Main()
      {
       P3^4=0;
       While(1)
       {
          P1=0x0FF;
          while(P2^7==0)
          {
            P1=0x00;
           }
        }
```

}

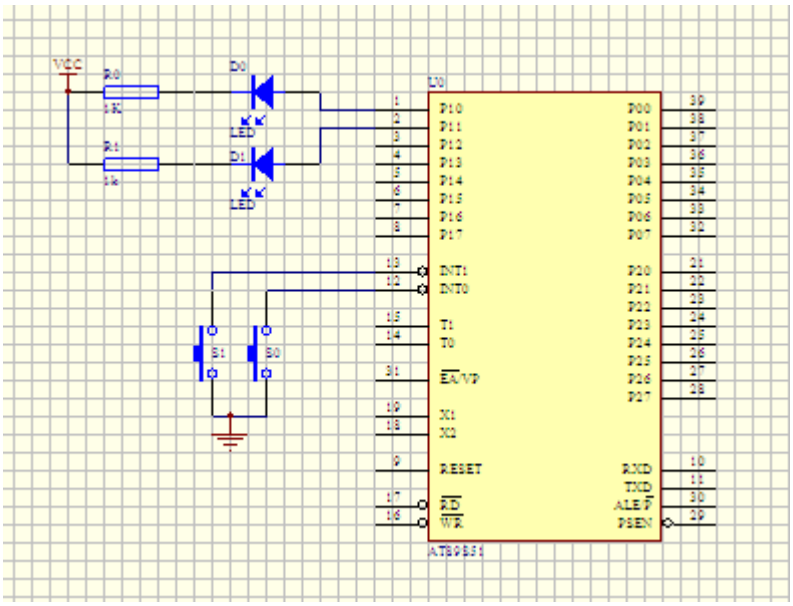
从这 2 个程序都可以看出 io 为低电平和高电平的效果。

第二讲 单片机外部中断使用

很多人都知道 51 单片机中中断的重要性，但是书中的长篇累牍让我们一下子理解有点很不适应。所以我们尽可能简化 51 单片机的中断，目前我们是要学会单片机中断的使用就可以了，而不是从原理基本说起。

在 51 单片机中有 2 个外部中断，这两个外部中断口处于低电平的时候开始触发中断信号，使得程序可以进入中断处理中断部分的程序。

我们还是老规矩，先看看硬件电路：



在图上的单片机原理图中，S0,S1 分别接的是单片机的外部中断 1 和外部中断 2，我们特意将两个发光二极管放在上面进行指示，以便我们可以看到中断运行的情况，此时我们可以设置中断程序的要求：

1、 指示灯平时没有中断按下的时候不进行工作，保持原有状态；

2、 当外部中断 0 响应的时候，我们就点亮 D0，让外部中断 0 响应的条件就是按下 S0 按键。

下面就是汇编程序：

```
        ORG     0000H
        LJMP    MAIN            ;主程序
        ORG     0003H
        LJMP    W_INT0          ;进入中断子程序
        ORG     0100H
MAIN:MOV SP, #50H               ;设置堆栈指针
        SETB    EA              ;CPU 所有中断开(IE 最高位 MSB)
        SETB    EX0             ;INT0 中断开
        CLR     IT0             ;INT0 低电平触发(为 1 则为下降沿触发)
        MOV     P1,     #0FFH
        JMP     $
```

```
W_INT0:CLR      P1.0
            RETI
          END
```

如下是 c51 的程序：

```
//---------------------------------------------------
#include <reg51.h>
//---------------------------------------------------
//重定义 I/0 引脚名称
sbit led1=P1^0;
//---------------------------------------------------
//固定函数声明
void int_0();            //外部中断 0
//---------------------------------------------------
void main(){
      P1=0X0FF;
      EA=1;                       // CPU 所有中断开(IE 最高位 MSB)
      EX0=1;                      // INT0 中断开
      IT0=0;                     // INT0 低电平触发(为 1 则为下降沿触发)
      while(1){
      {
       ;
      }
}
//-------------------------------------------------------------------
//INT0 中断    由 P3.2 引脚产生
void int_0() interrupt 0 using 0
{
    led0=0;
}
```

在上面的程序中，我们可以看出外部中断使用的方法，现在我们学会了使用中断 0，那么我们现在开始学使用外部中断 1 吧。

首先我们来设置程序的要求：

1、 外部中断 0 点亮 D0，关闭 D1

2、 外部中断 1 点亮 D1，关闭 D0

我们先看看汇编语言的程序：

```
;-------------------------------------------------------------------
            LED0        EQU        P1.0
            LED1        EQU        P1.1
            ORG          0000H
            LJMP         main
            ORG          0003H
            LJMP         W_INT0
            ORG          00013H
            LJMP         W_INT1
```

```
        ORG         0100H
main:
        SETB    EA          ;CPU 所有中断开(IE 最高位 MSB)
        SETB    EX0         ;INT0 中断开
        CLR     IT0         ;INT0 低电平触发(为 1 则为下降沿触发)
        SETB    EX1         ;INT1 中断开
        CLR     IT1         ;INT1 低电平触发(为 1 则为下降沿触发)
        MOV     P1,     #0FFH
        JMP     $

W_INT0:
        SETB    LED1            ;关闭 LED1
        CLR     LED0            ;点亮 LED0
        RETI
W_INT0:
        CLR     LED1            ;点亮 LED1
        SETB    LED0            ;关闭 LED0
        RETI

END
```

这是 c51 的程序：

```
//---------------------------------------------------
#include <reg51.h>
//---------------------------------------------------
//重定义 I/0 引脚名称
sbit LED0=P1^0;
sbit LED1=P1^1;
//---------------------------------------------------
//固定函数声明
void int_0();       //外部中断 0
void int_1();       //外部中断 1
//---------------------------------------------------
void main(){
    EA=1;                   // CPU 所有中断开(IE 最高位 MSB)

     EX0=1;                  // INT0 中断开
     IT0=0;                  // INT0 低电平触发(为 1 则为下降沿触发)

     EX1=1;                  // INT1 中断开
     IT1=0;                  // INT1 低电平触发(为 1 则为下降沿触发)

     while(1){
       {
```

```
            ;
          }
}
//---------------------------------------------------------------
void initial(){
      EA=1;                    // CPU 所有中断开(IE 最高位 MSB)

      EX0=1;                   // INT0 中断开
      IT0=0;                   // INT0 低电平触发(为 1 则为下降沿触发)

      EX1=1;                   // INT1 中断开
      IT1=0;                   // INT1 低电平触发(为 1 则为下降沿触发)

      return;
}
//---------------------------------------------------------------
//INT0 中断   由 P3.2 引脚产生
void int_0() interrupt 0 using 0
{
      Led1=1;                  //关闭 D1
      LED0=0;                  //点亮 D0
}
//---------------------------------------------------------------
//INT1 中断   由 P3.3 引脚产生
void int_1() interrupt 2 using 1
{
      LED0=1;                  //关闭 D0
      LED1=0;                  //点亮 D1
}
```

这样，通过程序和实践，我们就可以很轻易学会了外部中断的使用方法。

第三讲 定时器的使用

关于定时器中断，我们简单介绍一下原理就可以了，因为具体介绍已经有很多资料都已经提供了。

80C51 单片机内部设有两个 16 位的可编程定时器/计数器。可编程的意思是指其功能（如工作方式、定时时间、量程、启动方式等）均可由指令来确定和改变。在定时器/计数器中除了有两个 16 位的计数器之外，还有两个特殊功能寄存器(控制寄存器和方式寄存器)。我们可以看出，16 位的定时/计数器分别由两个 8 位专用寄存器组成，即：T0 由 TH0 和 TL0 构成；T1 由 TH1 和 TL1 构成。其访问地址依次为 8AH-8DH。每个寄存器均可单独访问。这些寄存器是用于存放定时或计数初值的。此外，其内部还有一个 8 位的定时器方式寄存器 TMOD 和一个 8 位的定时控制寄存器 TCON。这些寄存器之间是通过内部总线和控制逻辑电路连接起来的。TMOD 主要是用于选定定时器的工作方式； TCON 主要是用于控制定时器的启动

停止，此外 TCON 还可以保存 T0、T1 的溢出和中断标志。当定时器工作在计数方式时，外部事件通过引脚 T0 （P3.4）和 T1（P3.5）输入。定时计数器的原理：

当定时器/计数器为定时工作方式时，计数器的加 1 信号由振荡器的 12 分频信号产生，即每过一个机器周期，计数器加 1，直至计满溢出为止。显然，定时器的定时时间与系统的振荡频率有关。因一个机器周期等于 12 个振荡周期，所以计数频率 fcount=1/12osc。如果晶振为 12MHz，则计数周期为：
T=1/（12×106）Hz×1/12=1μs

这是最短的定时周期。若要延长定时时间，则需要改变定时器的初值，并要适当选择定时器的长度（如 8 位、13 位、16 位等）。

当定时器/计数器为计数工作方式时，通过引脚 T0 和 T1 对外部信号计数，外部脉冲的下降沿将触发计数。计数器在每个机器周期的 S5P2 期间采样引脚输入电 平。若一个机器周期采样值为 1，下一个机器周期采样值为 0，则计数器加 1。此后的机器周期 S3P1 期间，新的计数值装入计数器。所以检测一个由 1 至 0 的跳 变需要两个机器周期，故外部事年的最高计数频率为振荡频率的 1/24。例如，如果选用 12MHz 晶振，则最高计数频率为 0.5MHz。虽然对外部输入信号 的占空比无特殊要求，但为了确保某给定电平在变化前至少被采样一次，外部计数脉冲的高电平与低电平保持时间均需在一个机器周期以上。

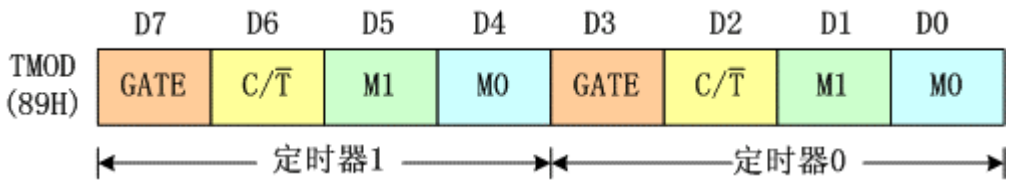
当 CPU 用软件给定时器设置了某种工作方式之后，定时器就会按设定的工作方式独立运行，不再占用 CPU 的操作时间，除非定时器计满溢出，才可能中断 CPU 当前操作。CPU 也可以重新设置定时器工作方式，以改变定时器的操作。由此可见，定时器是单片机中效率高而且工作灵活的部件。

综上所述，我们已知定时器/计数器是一种可编程部件，所以在定时器/计数器开始工作之前，CPU 必须将一些命令（称为控制字）写入定时/计数器。将控制字写入定时/计数器的过程叫定时器/计数器初始化。在初始化过程中，要将工作方式控制字写入方式寄存器，工作状态字（或相关位）写入控制寄存器，赋定时/计 数初值。下面我们就提出的控制字的格式及各位的主要功能与大家详细的讲解。

控制寄存器 定时器 / 计数器 T0 和 T1 有 2 个控制寄存器-TMOD 和 TCON，它们分别用来设置各个定时器 / 计数器的工作方式，选择定时或计数功能，控制启动运行，以及作为运行状态的标志等。其中，TCON 寄存器中另有 4 位用于中断系统。

TMOD 定时器/计数器方式寄存器

定时器方式控制寄存器 TMOD 在特殊功能寄存器中，字节地址为 89H，无位地址。TMOD 的格式如下图所示。



由图可见，TMOD 的高 4 位用于 T1，低 4 使用于 T0，4 种符号的含义如下：

GATE：门控制位。GATE 和软件控制位 TR、外部引脚信号 INT 的状态,共同控制定时器／计数器的打开或关闭。

C／T：定时器／计数器选择位。C/T＝1，为计数器方式；C／T＝0，为定时器方式。

M1M0：工作方式选择位，定时器／计数器的 4 种工作方式由 M1M0 设定。

M1M0	工作方式	功能描述
00	工作方式 0	13 位计数器
01	工作方式 1	16 位计数器
10	工作方式 2	自动再装入 8 位计数器
11	工作方式 3	定时器 0：分成两个 8 位计数器； 定时器 1：停止计数

定时器/计数器方式控制寄存器 TMOD 不能进行位寻址，只能用字节传送指令设置定时器工作方式，低半字节定义为定时器 0，高半字节定义为定时器 1。复位时，TMOD 所有位均为 0。

例：设定定时器 1 为定时工作方式，要求软件启动定时器 1 按方式 2 工作。定时器 0 为计数方式，要求由软件启动定时器 0，按方式 1 工作。

我们怎么来实现这个要求呢？

大家先看上面 TMOD 寄存器各位的分布图

第一个问题：控制定时器 1 工作在定时方式或计数方式是哪个位？通过前面的学习，我们已知道，C/T 位（D6）是定时或计数功能选择位，当 C/T=0 时定时/计数器就为定时工作方式。所以要使定时/计数器 1 工作在定时器方式就必需使 D6 为 0。

第二个问题：设定定时器 1 按方式 2 工作。上表中可以看出，要使定时/计数器 1 工作在方式 2，M0（D4） M1（D5）的值必须是 1 0。

第三个问题：设定定时器 0 为计数方式。与第一个问题一样，定时/计数器 0 的工作方式选择位也是 C/T（D2），当 C/T=1 时，就工作在计数器方式。

第四个问题：由软件启动定时器 0，前面已讲过，当门控位 GATE=0 时，定时/计数器的启停就由软件控制。

第五个问题：设定定时/计数器工作在方式 1，使定时/计数器 0 工作在方式 1，M0（D0）M1（D1）的值必须是 0 1。

从上面的分析我们可以知道，只要将 TMOD 的各位，按规定的要求设置好后，定时器/计灵敏器就会按我们预定的要求工作。我们分析的这个例子最后各位的情况如下：

D7 D6 D5 D4 D3 D2 D1 D0

0 0 1 0 0 1 0 1

二进制数 00100101=十六进制数 25H。所以执行 MOV TMOD,#25H 这条指令就可以实现上述要求。

TCON 定时器/计数器控制寄存器

TCON 在特殊功能寄存器中，字节地址为 88H，位地址(由低位到高位)为 88H 一 8FH，由于有位地址，十分便于进行位操作。

TCON 的作用是控制定时器的启、停，标志定时器溢出和中断情况。

TCON 的格式如下图所示。其中，TF1，TR1，TF0 和 TR0 位用于定时器 / 计数器；IE1，IT1，IE0 和 IT0 位用于中断系统。

	8FH	8EH	8DH	8CH	8BH	8AH	89H	88H
TCON (88H)	TF1	TR1	TF0	TF0	ID1	IT1	IE0	IT0

各位定义如下：

TF1：定时器 1 溢出标志位。当字时器 1 计满溢出时，由硬件使 TF1 置“1”，并且申请中断。进入中断服务程序后，由硬件自动清“0”，在查询方式下用软件清“0”。

TR1：定时器 1 运行控制位。由软件清“0”关闭定时器 1。当 GATE=1，且 INT1 为高电平时，TR1 置“1”启动定时器 1；当 GATE=0，TR1 置“1”启动定时器 1。

TF0：定时器 0 溢出标志。其功能及操作情况同 TF1。

TR0：定时器 0 运行控制位。其功能及操作情况同 TR1。

IE1：外部中断 1 请求标志。

IT1：外部中断 1 触发方式选择位。

IE0：外部中断 0 请求标志。

IT0：外部中断 0 触发方式选择位。

TCON 中低 4 位与中断有关，我们将在下节课讲中断时再给予讲解。由于 TCON 是可以位寻址的，因而如果只清溢出或启动定时器工作，可以用位操作命令。例 如：执行“CLR TF0”后则清定时器 0 的溢出；执行“SETB TR1”后可启动定时器 1 开始工作（当然前面还要设置方式定）。

定时器/计数器的初始化

由于定时器/计数器的功能是由软件编程确定的，所以一般在使用定时/计数器前都要对其进行初始化，使其按设定的功能工作。初始货的步骤一般如下：

1、确定工作方式（即对 TMOD 赋值）；

2、预置定时或计数的初值（可直接将初值写入 TH0、TL0 或 TH1、TL1）；

3、根据需要开放定时器/计数器的中断（直接对 IE 位赋值）；

4、启动定时器/计数器（若已规定用软件启动，则可把 TR0 或 TR1 置“1”；若已规定由外中断引脚电平启动，则需给外引脚步加启动电平。当实现了启动要求后，定时器即按规定的工作方式和初值开始计数或定时）。

下面介绍一下确定时时/计数器初值的具体方法。

因为在不同工作方式下计数器位数不同，因而最大计数值也不同。

现假设最大计数值为 M，那么各方式下的

最大值 M 值如下：

方式 0：M=213=8 192

方式 1：M=216=65 536

方式 2：M=28=256

方式 3：定时器 0 分成两个 8 位计数器，所以两个 M 均为 256。

因为定时器/计数器是作“加 1”计数，并在计数满溢出时产生中断，因此初值 X 可以这样计算：

X=M-计数值

下面举例说明初值的确定方法。

例 1、选择 T1 方式 0 用于定时，在 P1.1 输出周期为 1ms 方波，晶振 fosc=6MHz。

解：根据题意，只要使 P1.1 每隔 500us 取反一次即可得到 1ms 的方波，因而 T1 的定时时间为 500us，因定时时间不长，取方式 0 即可。则 M1 M0=0；因是定时器方式，所以 C/T=0；在此用软件启动 T1，所以 GATE=0。T0 不用，方式字可任意设置，只要不使其进入方式 3 即可，一般取 0， 故 TMOD=00H。系统复位后 TMOD 为 0，可不对 TMOD 重新清 0。

下面计算 500us 定时 T1 初始值：

机器周期 T=12/fosc=12/（6×106）Hz=2μs

设初值为 X，则：

（1013－X）×2×10-6s=500×10-6s

X=7942D=1111100000110B=1F06H

因为在作 13 位计数器用时，TL1 的高 3 位未用，应填写 0，TH1 占用高 8 位，所以 X 的实际填写应为：

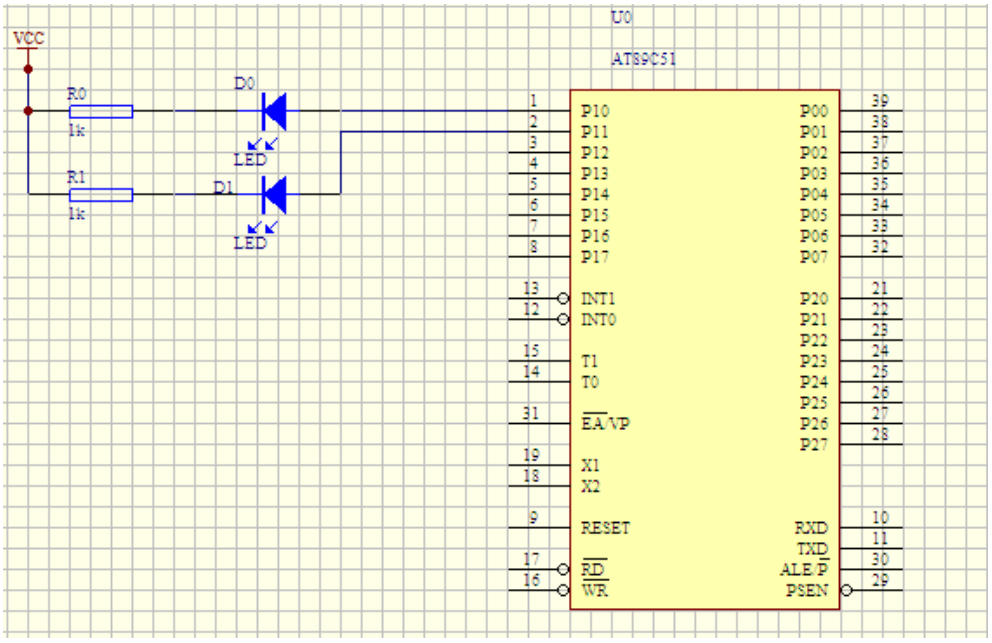
X=111100000000110B=F806H

结果：TH1=F8H，TL1=06H

源程序如下：

```
ORG 2000H

MOV TL1,#06H ;给 TL1 置初值

MOV TH1,#0F8H ;给 TH1 置初值

SETB TR1 ;启动 T1

LP1:JBC TF1,LP2 ;查询计数溢出否?

AJMP LP1

LP2:MOV TL1,#06H ;重新设置计数初值

MOV TH1,#0F8H

CPL P1.1 ;输出取反

AJMP LP1 ;重复循环
```

看了上面的介绍，我们现在应该会计算定时器的初值和设置相关的参数了，下面我们先看看硬件如何设计，在 protel99se 中，我们可以设计成如下 led 电路图：



我们在这里还是要设计自己的要求：
使用定时器 0，定时 1s，工作在方式 1，使用 16 位计数方式，让 D0 在 1s 后取反闪烁。
现在我们计算一下：在定时器中最大的延时只能做到 65 多 ms，无法做到 1s，那么我们只能通过寄存器进行计算，定时器为 50ms，累加到 1s，然后对 D0 进行取反操作。
看如下汇编程序：

```
        LED0    BIT     P1.0            ;灯的输出信号
        LED1    BIT     P1.1
        MSS     EQU     30H             ;20=1s
        MS1     EQU     31H
;*****************************************************
;主程序
;*****************************************************

        ORG     0000H
        NOP
        NOP
        LJMP    MAIN
        ORG     00BH                    ;计数器
        NOP
        NOP
        LJMP    SCAN    \
        ORG     01BH                    ;计数器
        NOP
        NOP
        LJMP    SCAN1
        ORG     0100H
MAIN:   MOV     P1,     #0FFH
```

```
        MOV     P3,     #0FFH
        MOV     SP,     #60H            ;设置堆栈
        MOV     MSS,    #00H
        MOV     TH0,    #3CH            ;50ms
        MOV     TL0,    #0B0H
        MOV     TH1,    #3CH            ;50ms
        MOV     TL1,    #0B0H
        MOV     TMOD,   #11H
        SETB    EA                      ;打开定时中断
        SETB    TR0
        SETB    ET0
        SETB    TR1
        SETB    ET1
        MOV     P1,     #0FFH
        AJMP    $
        NOP
;************************************************
;定时子程序
;************************************************
SCAN:   MOV     TH0,    #3CH
        MOV     TL0,    #0B0H
        PUSH    ACC
        INC     MSS
        MOV     A,      MSS
        CJNE    A,      #20,    SCA_E
        MOV     MSS,    #00
        CPL     LED0                    ;1S 到了对 LED0 取反
        AJMP    SCA_E
SCA_E:  POP     ACC
        RETI
SCAN1:  MOV     TH1,    #3CH
        MOV     TL1,    #0B0H
        PUSH    ACC
        INC     MS1
        MOV     A,      MS1
        CJNE    A,      #20,    SCA_E1
        MOV     MS1,    #00
        CPL     LED1                    ;1S 到了对 LED0 取反
        AJMP    SCA_E
SCA_E1: POP     ACC
        RETI

        END
```

如下为 c51 的程序：

```
//-------------------------------------------------
#include <reg52.h>
//-------------------------------------------------
//重定义 I/0 引脚名称
sbit LED0=P1^0;
sbit LED1=P1^1;
//-------------------------------------------------
//全局变量及位标志定义
unsigned char MSS;
unsigned char MS1;
//-------------------------------------------------
//固定函数声明
void timer_0();        //定时器中断 0
void timer_1();        //定时器中断 1
//-------------------------------------------------
void main(){
            TH0= 0x3C              ;50ms
            TL0=0x0B0;
            TH1= 0x3C              ;50ms
            TL1=0x0B0;
            EA=1;                                ;打开定时中断
            TR0=1;
            ET0=1;
            TR1=1;
            ET1=1;
            TMOD=0X11;
      while(1){
      {
      ;
      }
}

//-----------------------------------------------------------------
//定时器 0 中断
void timer_0() interrupt 1 using 2
{
      TH0= 0x3C              ;50ms
      TL0=0x0B0;
      MSS=MSS+1;
      If(MSS>=20)
      {
        LED0=!LED0;
      }
}
```

```
//-------------------------------------------------------------------
//定时器 1 中断
void timer_1() interrupt 3 using 3
{
      TH1= 0x3C              ;50ms
      TL1=0x0B0;
      MS1=MS1+1;
      If(MS1>=20)
      {
       LED1=!LED1;
      }

}
```

以上就是定时器的用法，有问题欢迎交流。

第四讲 串口中断知识讲解

MCS-51 单片机的串行口具有两条独立的数据线——发送端 TXD 和接收端 RXD，它允许数据同时往两个相反的方向传输。一般通信时发送数据由 TXD 端输出，接收数据由 RXD 端输入。MCS-51 单片机的串行口既可以用于网络通信，亦可实现串行异步通信，还可以用作同步移位寄存器。如果在串行口的输入输出引脚上加上电平转换器，就可方便地构成标准的 RS-232 接口。MCS-51 单片机的串行接口是一个全双工通信接口，它有两个物理上独立的接收、发送缓冲器 SBUF，可以同时发送和接收数据。但是发送缓冲器只能写入，不能读出；接收缓冲器只能读出，不能写入。两个缓冲器共用一个地址（99H）。

数据通信的基本概念

常用于数据通信的传输方式有单工、半双工、全双工和多工方式。

- 单工方式：数据仅按一个固定方向传送。因而这种传输方式的用途有限，常用于串行口的打印数据传输与简单系统间的数据采集。
- 半双工方式：数据可实现双向传送，但不能同时进行，实际的应用采用某种协议实现收/发开关转换。
- 全双工方式：允许双方同时进行数据双向传送,但一般全双工传输方式的线路和设备较复杂。
- 多工方式：以上三种传输方式都是用同一线路传输一种频率信号，为了充分地利用线路资源，可通过使用多路复用器或多路集线器，采用频分、时分或码分复用技术，即可实现在同一线路上资源共享功能。

根据同步方式，串行数据通信有两种形式，如图 5-5 所示。

- 异步通信。在这种通信方式中，接收器和发送器有各自的时钟，它们的工作是非同步的。异步通信用一帧来表示一个字符，其内容是一个起始位，紧接着是若干个数据位。
- 同步通信。同步通信格式中，发送器和接收器由同一个时钟源控制，在异步通信中，每传输一帧字符都必须加上起始位和停止位，占用了传输时间，若要求传送数据 量较大，速度就会慢得多。同步传输方式去掉了这些起始位和停止位，只在传输数据块时先送出一个同步头（字符）标志即可。
- 同步传输方式比异步传输方式速度快，这是它的优势。但同步传输方式也有其缺点，即它必须要用一个时钟来协调收发器的工作，所以它的设备也较复杂。

MCS-51 的串行口控制寄存器

在完成串行口初始化后，发送数据时，采用 MOV SBUF,A 指令，将要发送的数据写入 SBUF，则 CPU 自动启动和完成串行数据的输出；接收数据时，采用 MOV A,SBUF 指令，CPU 就自动将接收到的数据从 SBUF 中读出。

控制 MCS-51 单片机串行接口的控制寄存器有两个——特殊功能寄存器 SCON 和 PCON，用以设置串行端口的工作方式、接收/发送的运行状态、接收/发送数据的特征、数据传输率的大小，以及作为运行的中断标志等，其格式如下：

① 串行口控制寄存器 SCON。SCON 的字节地址是 98H，位地址（由低位到高位）分别是 98H 一 9FH。SCON 的格式如下：

SCON	D7	D6	D5	D4	D3	D2	D1	D0
	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0、SM1：串行口工作方式控制位。

00——方式 0；01——方式 1；

10——方式 2；11——方式 3。

SM2：仅用于方式 2 和方式 3 的多机通信控制位。

发送机 SM2＝1（要求程控设置）。

当为方式 2 或方式 3 时：

接收机 SM2＝1 时，若 RB8＝1，可引起串行接收中断；若 RB8＝0，不引起串行接收中断。SM2＝0 时，若 RB8＝1，可引起串行接收中断；若 RB8＝0，亦可引起串行接收中断。

- REN 串行接收允许位：0——禁止接收；1——允许接收。
- TB8：在方式 2、3 中，TB8 是发送机要发送的第 9 位数据。
- RB8：在方式 2、3 中，RB8 是接收机接收到的第 9 位数据，该数据正好来自发送机的 TB8。

- TI：发送中断标志位。发送前必须用软件清零，发送过程中 TI 保持零电平，发送完一帧数据后，由硬件自动置 1。如要再发送，必须用软件再清零。
- RI：接收中断标志位。接收前，必须用软件清零，接收过程中 RI 保持零电平，接收完一帧数据后，由片内硬件自动置 1。如要再接收，必须用软件再清零。

② 电源控制寄存器 PCON。PCON 的字节地址为 87H，无位地址，其格式如下：

PCON	D7	D6	D5	D4	D3	D2	D1	D0
位符号	SMOD	-	-	-	GF1	GF0	PD	IDL

PCON 是为在 CMOS 结构的 MCS-51 单片机上实现电源控制而附加的，对于 HMOS 结构的 MCS-51 系列单片机，除了第 7 位外，其余都是虚设的。与串行通信有关的也就是第 7 位，称作 SMOD，它的用处是使数据传输率加倍。

SMOD：数据传输率加倍位。在计算串行方式 1，2，3 的数据传输率时；0 表示不加倍；1 表示加倍。

其余有效位说明如下。

GF1、GF2：通用标志位。

PD：掉电控制位，0 表示正常方式，1 表示掉电方式。

IDL：空闲控制位，0 表示正常方式，1 表示空闲方式。

除了以上两个控制寄存器外，中断允许寄存器 IE 中的 ES 位也用来作为串行 I/O 中断允许位。当 ES＝1，允许 串行 I/O 中断；当 ES＝0，禁止串行 I/O 中断。中断优先级寄存器 IP 的 PS 位则用作串行 I/O 中断优先级控制位。当 PS=1，设定为高优先级；当 PS =0，设定为低优先级。

工作方式

MCS-51 单片机可以通过软件设置串行口控制寄存器 SCON 中 SM0（SCON.7）和 SM1（SCON.6）来指定串行口的 4 种工作方式。串行口操作模式选择如表 5-2 所示。

表 5-2 串行口操作模式选择表

SM0 SM1	模 式	功 能	波 特 率
0 0	0	同步移位寄存器	fOSC/12
0 1	1	8 位 UART	可变（T1 溢出率）

1 0	0	9 位 UART	fOSC/64 或 fOSC/32
1 1	1	9 位 UART	可变（T1 溢出率）

其中，f_{osc}是振荡器的频率，UART为通用异步接收和发送器的英文缩写。下面对这 4 种工作模式作进一步介绍。

1．方式 0

当设定 SM1、SM0 为 00 时，串行口工作于方式 0，它又叫同步移位寄存器输出方式。在方式 0 下，数据从 RXD（P3.0）端串行输出或输入，同步信号从 TXD（P3.1）端输出，发送或接收的数据为 8 位，低位在前，高位在后，没有起始位和停止位。数据传输 率固定为振荡器的频率 1/12，也就是每一机器周期传送一位数据。方式 0 可以外接移位寄存器，将串行口扩展为并行口，也可以外接同步输入/输出设备。

执行任何一条以 SBUF 为目的的寄存器指令，就开始发送。

2．方式 1

当设定 SM1、SM0 为 01 时，串行口工作于方式 1。方式 1 为数据传输率可变的 8 位异步通信方式，由 TXD 发 送，RXD 接收，一帧数据为 10 位，1 位起始位（低电平），8 位数据位（低位在前）和 1 位停止位（高电平）。数据传输率取决于定时器 1 或 2 的溢出速率 （1/溢出周期）和数据传输率是否加倍的选择位 SMOD。

对于有定时器/计数器 2 的单片机，当 T2CON 寄存器中 RCLK 和 TCLK 置位时，用定时器 2 作为接收和发送的数据传输率发生器，而 RCLK=TCLK=0 时，用定时器 1 作为接收和发送的数据传输率发生器。两者还可以交叉使用，即发送和接收采用不同的数据传输率。

类似于模式 0，发送过程是由执行任何一条以 SBUF 为目的的寄存器指令引起的。

3．方式 2

当设定 SM0、SM1 二位为 10 时，串行口工作于方式 2，此时串行口被定义为 9 位异步通信接口。采用这种方式 可接收或发送 11 位数据，以 11 位为一帧，比方式 1 增加了一个数据位，其余相同。第 9 个数据即 D8 位用作奇偶校验或地址/数据选择，可以通过软件来控制它，再加特殊功能寄存器 SCON 中的 SM2 位的配合，可使 MCS-51 单片机串行口适用于多机通信。发送时，第 9 位数据为 TB8，接收时，第 9 位数据送入 RB8。方式 2 的数据传输率固定，只有两种选择，为振荡率的 1/64 或 1/32 ，可由 PCON 的最高位选择。

4．方式 3

当设定 SM0、SM1 二位为 11 时，串行口工作于方式 3。方式 3 与方式 2 类似，唯一的区别是方式 3 的数据传输率是可变的。而帧格式与方式 2 一样为 11 位一帧。所以方式 3 也适合于多机通信。

数据传输率的确定

串行口每秒钟发送（或接收）的位数就是数据传输率。

对方式 0 来说，数据传输率已固定成 fosc/12，随着外部晶振的频率不同，数据传输率亦不相同。常用的 fosc 有 12MHz 和 6MHz，所以数据传输率相应为 1000×103 和 500×103bit/s。在此方式下，数据将自动地按固定的数据传输率发送/接收，完全不用设置。

对方式 2 而言，数据传输率的计算式为 2SMOD·fosc/64。当 SMOD＝0 时，数据传输率为 fm/64；当 SMOD＝1 时，数据传输率为 fosc/32。在此方式下，程控设置 SMOD 位的状态后，数据传输率就确定了，不需要再作其他设置。

对方式 1 和方式 3 来说，数据传输率和定时器 1 的溢出率有关，定时器 1 的溢出率为：

定时器 1 的溢出率=定时器 1 的溢出次数/秒

方式 1 和方式 3 的数据传输率计算式为：

2SMOD/32×T1 溢出率

根据 SMOD 状态位的不同，数据传输率有 T1/32 溢出率和 T1/16 溢出率两种。由于 T1 溢出率的设置是方便的，因而数据传输率的选择将十分灵活。

前已叙及，定时器 T1 有 4 种工作方式，为了得到其溢出率，而又不必进入中断服务程序，往往使 T1 设置在工作方式 2 的运行状态，也就是 8 位自动加入时间常数的方式。

表 5-3 所示常用数据传输率的设置方法。

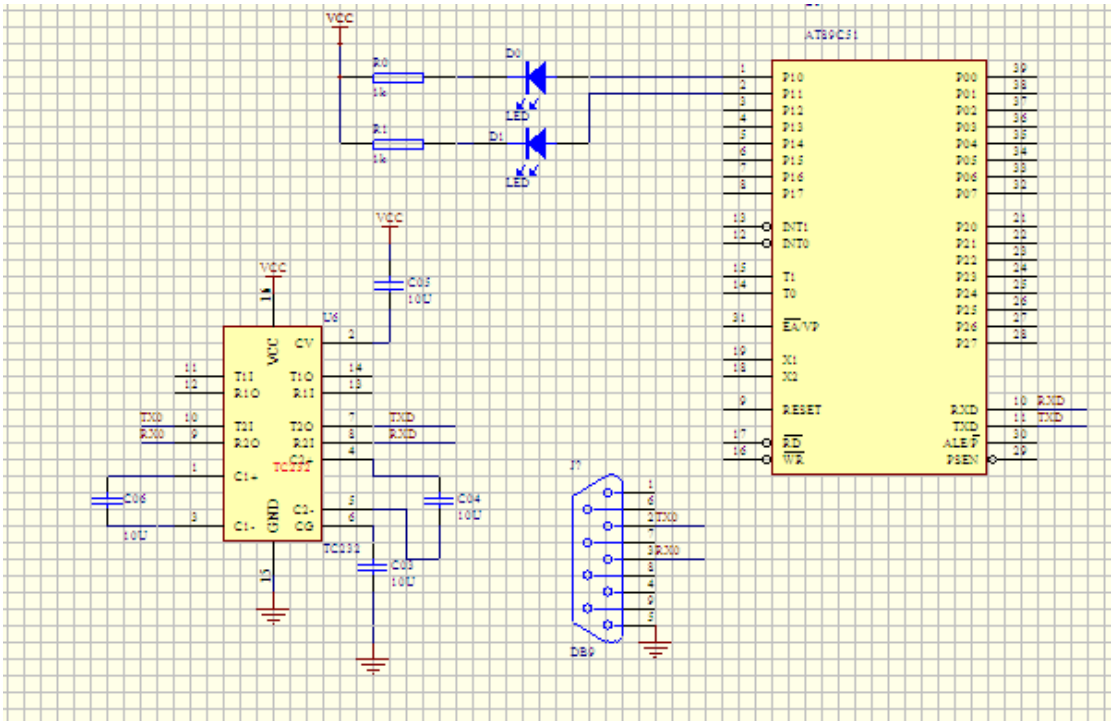
表 5-3 常用数据传输率设置方法

数据传输率/Hz	fOSC/MHz	SMOD	定时器 1		
			C/T	方式	重新装入值
方式 0 最大：1M	12	X	X	X	X
方式 2 最大：	12	1	X	X	X

375k 方式 1、3： 62.5k 19.2k 9.6k 4.8k 2.4k 1.2k 110	12 11.0592 11.0592 11.0592 11.0592 11.0592 12	1 1 0 0 0 0 0	0 0 0 0 0 0 0	2 2 2 2 2 2 1	FFH FDH FDH FAH F4H E8H 0FEEH

串行通信实例

我们先看看实例图形，我们先设计一个串口通讯的电路，一端和单片机连接，另一端和电脑连接。如下的 protel 所示：



此图中，使用的晶振是 11.0592mhz 的晶振，和电脑进行通讯，电脑上面

下面是汇编程序

```
        LED0    BIT    P1.0
        ORG       0000H
        LJMP     MAIN
        ORG      0023H
        LJMP     TRX
        ORG      0100H
MAIN:   MOV   SP,     $50H
        MOV   TMOD,   #21H
        MOV   SCON,   #80H
        MOV   TH1,    #0fdh          ;9600bit
        MOV   TL1,    #0fdh
        SETB  ES
        SETB REN
        SETB   EA
        MOV      SBUF,    #30H            ;发送数据第一位
        JNB      TI, $
        CLR      TI
        MOV      SBUF,    #31H            ;发送数据第一位
        JNB      TI, $
        CLR      TI
        MOV      SBUF,    #32H            ;发送数据第一位
        JNB      TI, $
        CLR      TI
        JMP      $
TRX:    PUSH        PSW             ;串行中断
```

```
        PUSH    ACC
        PUSH    DPL
        PUSH    DPH
        JNB  RI,ENDRX
        clr tb8
        CLR  RI
        MOV A,SBUF
        CJNE A,#30H,ENDRX
        SETB LED0
ENDRX:  POP  DPH
        POP  DPL
        POP  ACC
        POP  PSW
        RETI
END
```

这是一个单片机 C51 串口接收（中断）和发送例程，可以用来测试 51 单片机的中断接收和查询发送。

```
#include <reg51.h>
#include <string.h>
main()
{
    init_serial();                          //串行口初始化
```

```
    while (1)
        {
            send_char(0X30);      //向串口发送字符串
            send_char(0X31);      //向串口发送字符串
            send_char(0X32);      //向串口发送字符串
          while(1)
          {;}
        }
}
/* 串行口初始化 */
void init_serial( void )
{
    SCON = 0x50;                  //串行工作方式 1, 8 位异步通信方式
    TMOD |= 0x20;                 //定时器 1, 方式 2, 8 位自动重装
    PCON |= 0x80;                 //SMOD=1，表示数据传输率加倍
    TH1 = 0xFD;                   //数据传输率:9600 fosc=11.0592MHz
    IE |= 0x90;                   //允许串行中断
    TR1 = 1;                      //启动定时器 1
}
/* 向串口发送一个字符 */
void send_char( unsigned char x)
{
    SBUF=x;
    while (TI== 0 );
    TI= 0;
```

```
}
/* 串口接收中断函数 */
void serial () interrupt 4 using 3
{
     if (RI)
         {
         unsigned char x;
         RI = 0;
         x=SBUF;                  //接收字符
         if ( x= 0X30 )
             {
             LED0=0;
             }
         }
}
```

第五讲 单片机抗干扰

本章很多是收集整理的，我也经常参考这些，虽然不是经典名言，但是至少给了我们一个依据。

一、下面的一些系统要特别注意抗电磁干扰：

1、微控制器时钟频率特别高，总线周期特别快的系统。

2、系统含有大功率，大电流驱动电路，如产生火花的继电器，大电流开关等。

3、含微弱模拟信号电路以及高精度A/D变换电路的系统。

二、为增加系统的抗电磁干扰能力采取如下措施：

1、选用频率低的微控制器：

选用外时钟频率低的微控制器可以有效降低噪声和提高系统的抗干扰能力。同样频率的方波和正弦波，方波中的高频成份比正弦波多得多。虽然方波的高频成份的波的幅度，比基波小，但频率越高越容易发射出成为噪声源，微控制器产生的最有影响的高频噪声大约是时钟频率的 3 倍。

2、减小信号传输中的畸变

微控制器主要采用高速CMOS技术制造。信号输入端静态输入电流在 1mA左右，输入电容10PF左右，输入阻抗相当高，高速CMOS电路的输出端都有相当的带载能力，即相当大的输

出值，将一个门的输出端通过一段很长线引到输入阻抗相当高的输入端，反射问题就很严重，它会引起信号畸变，增加系统噪声。当Tpd＞Tr时，就成了一个传输线问题，必须考虑信号反射，阻抗匹配等问题。

信号在印制板上的延迟时间与引线的特性阻抗有关，即与印制线路板材料的介电常数有关。可以粗略地认为，信号在印制板引线的传输速度，约为光速的 1/3 到 1/2 之间。微控制器构成的系统中常用逻辑电话元件的Tr（标准延迟时间）为 3 到 18ns之间。

在印制线路板上，信号通过一个 7W的电阻和一段 25cm长的引线，线上延迟时间大致在4~20ns之间。也就是说，信号在印刷线路上的引线越短越好，最长不宜超过 25cm。而且过孔数目也应尽量少，最好不多于 2 个。

当信号的上升时间快于信号延迟时间，就要按照快电子学处理。此时要考虑传输线的阻抗匹配，对于一块印刷线路板上的集成块之间的信号传输，要避免出现Td＞Trd的情况，印刷线路板越大系统的速度就越不能太快。

用以下结论归纳印刷线路板设计的一个规则：

信号在印刷板上传输，其延迟时间不应大于所用器件的标称延迟时间。

3、减小信号线间的交叉干扰：

A点一个上升时间为Tr的阶跃信号通过引线AB传向B端。信号在AB线上的延迟时间是Td。在D点，由于A点信号的向前传输，到达B点后的信号反射和AB线的延迟，Td时间以后会感应出一个宽度为Tr的页脉冲信号。在C点，由于AB上信号的传输与反射，会感应出一个宽度为信号在AB线上的延迟时间的两倍，即 2Td的正脉冲信号。这就是信号间的交叉干扰。干扰信号的强度与C点信号的di/at有关，与线间距离有关。当两信号线不是很长时，AB上看到的实际是两个脉冲的迭加。

CMOS工艺制造的微控制由输入阻抗高，噪声高，噪声容限也很高，数字电路是迭加100~200mv噪声并不影响其工作。若图中AB线是一模拟信号，这种干扰就变为不能容忍。如印刷线路板为四层板，其中有一层是大面积的地，或双面板，信号线的反面是大面积的地时，这种信号间的交叉干扰就会变小。原因是，大面积的地减小了信号线的特性阻抗，信号在D端的反射大为减小。特性阻抗与信号线到地间的介质的介电常数的平方成反比，与介质厚度的自然对数成正比。若AB线为一模拟信号，要避免数字电路信号线CD对AB的干扰，AB线下方要有大面积的地，AB线到CD线的距离要大于AB线与地距离的 2~3 倍。可用局部屏蔽地，在有引结的一面引线左右两侧布以地线。

4、减小来自电源的噪声

电源在向系统提供能源的同时，也将其噪声加到所供电的电源上。电路中微控制器的复位线，中断线，以及其它一些控制线最容易受外界噪声的干扰。电网上的强干扰通过电源进入电路，即使电池供电的系统，电池本身也有高频噪声。模拟电路中的模拟信号更经受不住来自电源的干扰。

5、注意印刷线板与元器件的高频特性

在高频情况下，印刷线路板上的引线，过孔，电阻、电容、接插件的分布电感与电容等不可忽略。电容的分布电感不可忽略，电感的分布电容不可忽略。电阻产生对高频信号的反射，引线的分布电容会起作用，当长度大于噪声频率相应波长的 1/20 时，就产生天线效应，噪声通过引线向外发射。

印刷线路板的过孔大约引起 0.6pf的电容。

一个集成电路本身的封装材料引入 2~6pf电容。

一个线路板上的接插件，有 520nH的分布电感。一个双列直扦的 24 引脚集成电路扦座，引入 4~18nH的分布电感。

这些小的分布参数对于这行较低频率下的微控制器系统中是可以忽略不计的；而对于高速系

统必须予以特别注意。

6、元件布置要合理分区

元件在印刷线路板上排列的位置要充分考虑抗电磁干扰问题，原则之一是各部件之间的引线要尽量短。在布局上，要把模拟信号部分，高速数字电路部分，噪声源部分（如继电器，大电流开关等）这三部分合理地分开，使相互间的信号耦合为最小。

7、处理好接地线

印刷电路板上，电源线和地线最重要。克服电磁干扰，最主要的手段就是接地。

对于双面板，地线布置特别讲究，通过采用单点接地法，电源和地是从电源的两端接到印刷线路板上来的，电源一个接点，地一个接点。印刷线路板上，要有多个返回地线，这些都会聚到回电源的那个接点上，就是所谓单点接地。所谓模拟地、数字地、大功率器件地开分，是指布线分开，而最后都汇集到这个接地点上来。与印刷线路板以外的信号相连时，通常采用屏蔽电缆。对于高频和数字信号，屏蔽电缆两端都接地。低频模拟信号用的屏蔽电缆，一端接地为好。

对噪声和干扰非常敏感的电路或高频噪声特别严重的电路应该用金属罩屏蔽起来。

8、用好去耦电容。

好的高频去耦电容可以去除高到 1GHZ的高频成份。陶瓷片电容或多层陶瓷电容的高频特性较好。设计印刷线路板时，每个集成电路的电源，地之间都要加一个去耦电容。去耦电容有两个作用：一方面是本集成电路的蓄能电容，提供和吸收该集成电路开门关门瞬间的充放电能；另一方面旁路掉该器件的高频噪声。数字电路中典型的去耦电容为 0.1uf的去耦电容有5nH分布电感，它的并行共振频率大约在 7MHz左右，也就是说对于 10MHz 以下的噪声有较好的去耦作用，对 40MHz以上的噪声几乎不起作用。

1uf，10uf电容，并行共振频率在 20MHz以上，去除高频率噪声的效果要好一些。在电源进入印刷板的地方和一个 1uf或 10uf的去高频电容往往是有利的，即使是用电池供电的系统也需要这种电容。

每 10 片左右的集成电路要加一片充放电电容，或称为蓄放电容，电容大小可选 10uf。最好不用电解电容，电解电容是两层溥膜卷起来的，这种卷起来的结构在高频时表现为电感，最好使用胆电容或聚碳酸酐电容。

去耦电容值的选取并不严格，可按C=1/f计算；即 10MHz取 0.1uf，对微控制器构成的系统，取 0.1~0.01uf之间都可以。

三、降低噪声与电磁干扰的一些经验。

1. 能用低速芯片就不用高速的，高速芯片用在关键地方。
2. 可用串一个电阻的办法，降低控制电路上下沿跳变速率。
3. 尽量为继电器等提供某种形式的阻尼。
4. 使用满足系统要求的最低频率时钟。
5. 时钟产生器尽量靠近到用该时钟的器件。石英晶体振荡器外壳要接地。
6. 用地线将时钟区圈起来，时钟线尽量短。
7. I/O驱动电路尽量靠近印刷板边，让其尽快离开印刷板。对进入印制板的信号要加滤波。
8. 高噪声区来的信号也要加滤波，同时用串终端电阻的办法，减小信号反射。
9. MCD无用端要接高，或接地，或定义成输出端，集成电路上该接电源地的端都要接，不要悬空。
10. 闲置不用的门电路输入端不要悬空，闲置不用的运放正输入端接地，负输入端接输出端。
11. 印制板尽量使用 45 折线而不用 90 折线布线以减小高频信号对外的发射与耦合。
12. 印制板按频率和电流开关特性分区，噪声元件与非噪声元件要距离再远一些。
13. 单面板和双面板用单点接电源和单点接地、电源线、地线尽量粗，经济是能承受的话用

多层板以减小电源，地的容生电感。

14. 时钟、总线、片选信号要远离I/O线和接插件。
15. 模拟电压输入线、参考电压端要尽量远离数字电路信号线，特别是时钟。
16. 对A/D类器件，数字部分与模拟部分宁可统一下也不要交叉。
17. 时钟线垂直于I/O线比平行I/O线干扰小，时钟元件引脚远离I/O电缆。
18. 元件引脚尽量短，去耦电容引脚尽量短。
19. 关键的线要尽量粗，并在两边加上保护地。高速线要短要直。
20. 对噪声敏感的线不要与大电流，高速开关线平行。
21. 石英晶体下面以及对噪声敏感的器件下面不要走线。
22. 弱信号电路，低频电路周围不要形成电流环路。
23. 任何信号都不要形成环路，如不可避免，让环路区尽量小。
24. 每个集成电路一个去耦电容。每个电解电容边上都要加一个小的高频旁路电容
25. 用大容量的钽电容或聚酷电容而不用电解电容作电路充放电储能电容。使用管状电容时，外壳要接地

第六讲 keil c 的基础知识

很多人对 keilc 不是很了解 我收集了一些资料，提供如下：

Keil C51 vs 标准 C

深入理解并应用 C51 对标准 ANSIC 的扩展是学习 C51 的关键之一。因为大多数扩展功能都是直接针对 8051 系列 CPU 硬件的。大致有以下 8 类：

1 8051 存储类型及存储区域

2 存储模式

3 存储器类型声明

4 变量类型声明

5 位变量与位寻址

6 特殊功能寄存器(SFR)

7 C51 指针

8 函数属性

具体说明如下(8031 为缺省 CPU)。

1. 第一节 Keil C51 扩展关键字

C51 V4.0 版本有以下扩展关键字(共 19 个)：

_at_ idata sfr16 alien interrupt small

bdata large _task_ Code bit pdata

using reentrant xdata compact sbit data sfr

2. 第二节 内存区域(Memory Areas)：

1. 1. Pragram Area：

由 Code 说明可有多达 64kBytes 的程序存储器

2. 2. Internal Data Memory:

内部数据存储器可用以下关键字说明：

data：直接寻址区，为内部 RAM 的低 128 字节 00H～7FH

idata：间接寻址区，包括整个内部 RAM 区 00H～FFH

bdata：可位寻址区， 20H～2FH

3. 3. External Data Memory

外部 RAM 视使用情况可由以下关键字标识：

xdata：可指定多达 64KB 的外部直接寻址区，地址范围 0000H～0FFFFH

pdata：能访问 1 页(25bBytes)的外部 RAM，主要用于紧凑模式(Compact Model)。

4. 4. Speciac Function Register Memory

8051 提供 128Bytes 的 SFR 寻址区，这区域可位寻址、字节寻址或字寻址，用以控制定时器、计数器、串口、I/O 及其它部件，可由以下几种关键字说明：

sfr：字节寻址 比如 sfr P0=0x80;为 P0 口地址为 80H，“＝”后 H～FFH 之间的常数。

sfr16：字寻址，如 sfr16 T2=0xcc;指定 Timer2 口地址 T2L=0xcc T2H=0xCD

sbit：位寻址，如 sbit EA=″0xAF″;指定第 0xAF 位为 EA，即中断允许

还可以有如下定义方法：

sbit OV=PSW^2; (定义 OV 为 PSW 的第 2 位)

sbit OV＝0XD0^2; (同上)

或 bit OV-＝0xD2(同上)。

存储类型与存储区关系

data > 可寻址片内 ram

bdata > 可位寻址的片内 ram

idata > 可寻址片内 ram，允许访问全部内部 ram

pdata > 分页寻址片外 ram (MOVX @R0) (256 BYTE/页)

xdata > 可寻址片外 ram (64k 地址范围)

code > 程序存储区 (64k 地址范围),对应 MOVC @DPTR

二、指针类型和存储区的关系对变量进行声明时可以指定变量的存储类型如:

uchar data x 和 data uchar x 相等价都是在内 ram 区分配一个字节的变量。同样对于指针变量的声明,因涉及到指针变量本身的存储位置和指针所指向的存储区位置不同而进行相应的存储区类型关键字的使用如：uchar xdata * data pstr 是指在内 ram 区分配一个

指针变量("*"号后的 data 关键字的作用)，而且这个指针本身指向 xdata 区("*"前 xdata 关键字的作用)，可能初学 C51 时有点不好懂也不好记。没关系，我们马上就可以看到对应“*”前后不同的关键字的使用在编译时出现什么情况。.....

uchar xdata tmp[10]; //在外 ram 区开辟 10 个字节的内存空间，地址是外 ram 的 0x0000－0x0009

.....第 1 种情况:uchar data * data pstr;

pstr="tmp";首先要提醒大家这样的代码是有 bug 的, 他不能通过这种方式正确的访问到 tmp 空间。 为什么？我们把编译后看到下面的汇编代码：MOV 0x08,#tmp(0x00) ;0x08 是指针 pstr 的存储地址看到了吗！本来访问外 ram 需要 2 byte 来寻址 64k 空间，但因为使用 data 关键字(在"*"号前的那个)，所以按 KeilC 编译环境来说就把他编译成指向内 ram 的指针变量了，这也是初学 C51 的朋友们不理解各个存储类型的关键字定义而造成的 bug。特别是当工程中的默认的存储区类为 large 时，又把 tmp[10] 声明为 uchar tmp[10] 时，这样的 bug 是很隐秘的不容易被发现。第 2 种情况:uchar xdata * data pstr;

pstr = tmp;这种情况是没问题的，这样的使用方法是指在内 ram 分配一个指针变量("*"号后的 data 关键字的作用),而且这个指针本身指向 xdata 区("*"前 xdata 关键字的作用)。编译后的汇编代码如下。MOV 0x08,#tmp(0x00) ;0x08 和 0x09 是在内 ram 区分配的 pstr 指针变量地址空间

MOV 0x09,#tmp(0x00)这种情况应该是在这里所有介绍各种情况中效率最高的访问外 ram 的方法了，请大家记住他。第 3 种情况:uchar xdata * xdata pstr;

pstr="tmp";这中情况也是对的，但效率不如第 2 种情况。编译后的汇编代码如下。MOV DPTR, #0x000A ;0x000A,0x000B 是在外 ram 区分配的 pstr 指针变量地址空间

```
MOV A, #tmp(0x00)

MOV @DPTR, A

INC DPTR

MOV A, #tmp(0x00)
```

MOVX @DPTR, A 这种方式一般用在内 ram 资源相对紧张而且对效率要求不高的项目中。第 4 种情况:uchar data * xdata pstr;

pstr="tmp";如果详细看了第 1 种情况的读者发现这种写法和第 1 种很相似，是的，同第 1 种情况一样这样也是有 bug 的，但是这次是把 pstr 分配到了外 ram 区了。编译后的汇编代码如下。MOV DPTR, #0x000A ;0x000A 是在外 ram 区分配的 pstr 指针变量的地址空间

MOV A, #tmp(0x00)

MOVX @DPTR, A 第 5 种情况:uchar * data pstr;

pstr="tmp";大家注意到"*"前的关键字声明没有了，是的这样会发生什么事呢？下面这么写呢！对了用齐豫的一首老歌名来说就是 “请跟我来”，请跟我来看看编译后的汇编代码，有人问这不是在讲 C51 吗？ 为什么还要给我们看汇编代码。C51 要想用好就要尽可能提升 C51

编译后的效率，看看编译后的汇编会帮助大家尽快成为生产高效 C51 代码的高手的。还是看代码吧！MOV 0x08, #0X01 ;0x08－0x0A 是在内 ram 区分配的 pstr 指针变量的地址空间

MOV 0x09, #tmp(0x00)

MOV 0x0A, #tmp(0x00)注意：这是新介绍给大家的，大家会疑问为什么在前面的几种情况的 pstr 指针变量都用 2 byte 空间而到这里就用 3 byte 空间了呢？这是 KeilC 的一个系统内部处理，在 KeilC 中一个指针变量最多占用 3 byte 空间，对于没有声明指针指向存储空间类型的指针，系统编译代码时都强制加载一个字节的指针类型分辩值。具体的对应关系可以参考 KeilC 的 help 中 C51 User's Guide。第 6 种情况:uchar * pstr;

pstr="tmp";这是最直接最简单的指针变量声明，但他的效率也最低。还是那句话，大家一起说好吗！编译后的汇编代码如下。MOV DPTR, #0x000A ;0x000A－0x000C 是在外 ram 区分配的 pstr 指针变量地址空间

MOV A, #0x01

MOV @DPTR, A

INC DPTR

MOV DPTR, #0x000A

MOV A, #tmp(0x00)

```
MOV @DPTR, A

INC DPTR

MOV A, #tmp(0x00)
```

MOVX @DPTR, A 这种情况很类似第 5 种和第 3 种情况的组合，既把 pstr 分配在外 ram 空间了又增加了指针类型的分辨值。 小结一下：大家看到了以上的 6 种情况，其中效率最高的是第 2 种情况，既可以正确访问 ram 区又节约了代码，效率最差的是第 6 种，但不是说大家只使用第 2 种方式就可以了，还要因情况而定，一般说来应用 51 系列的系统架构的内部 ram 资源都很紧张，最好大家在定义函数内部或程序段内部的局部变量使用内 ram，而尽量不要把全局变量声明为内 ram 区中。所以对于全局指针变量我建议使用第 3 种情况，而对于局部的指针变量使用第 2 种

本征库函数(intrinsic routines)和非本征证库函数

C51 强大功能及其高效率的重要体现之一在于其丰富的可直接调用的库函数，多使用库函数使程序代码简单，结构清晰，易于调试和维护，下面介绍 C51 的库函数系统。
C51 提供的本征函数是指编译时直接将固定的代码插入当前行，而不是用 ACALL 和 LCALL 语句来实现,这样就大大提供了函数访问的效率,而非本征函数则必须由 ACALL 及 LCALL 调用。
C51 的本征库函数只有 9 个，数目虽少，但都非常有用，列如下：
crol,_cror_：将 char 型变量循环向左(右)移动指定位数后返回
iror,_irol_：将 int 型变量循环向左(右)移动指定位数后返回
lrol,_lror_：将 long 型变量循环向左(右)移动指定位数后返回
nop： 相当于插入 NOP
testbit： 相当于 JBC bitvar 测试该位变量并跳转同时清除。
chkfloat： 测试并返回源点数状态。
使用时，必须包含#incluclе <intrins.h>一行。
如不说明，下面谈到的库函数均指非本征库函数。

第二节 几类重要库函数

1. 专用寄存器 include 文件
例如 8031、8051 均为 REG51.h 其中包括了所有 8051 的 SFR 及其位定义，一般系统都必须包括本文件。
2. 绝对地址 include 文件 absacc.h
该文件中实际只定义了几个宏，以确定各存储空间的绝对地址。
3. 动态内存分配函数，位于 stdlib.h 中

4. 缓冲区处理函数位于“string.h”中
其中包括拷贝比较移动等函数如：
memccpy memchr memcmp memcpy memmove memset
这样很方便地对缓冲区进行处理。

5. 输入输出流函数，位于“stdio.h”中
流函数通 8051 的串口或用户定义的 I/O 口读写数据，缺省为 8051 串口，如要修改，比如改为 LCD 显示，可修改 lib 目录中的 getkey.c 及 putchar.c 源文件，然后在库中替换它们即可。

Keil C51 库函数原型列表

1. CTYPE.H

```
bit isalnum(char c);
bit isalpha(char c);
bit iscntrl(char c);
bit isdigit(char c);
bit isgraph(char c);
bit islower(char c);
bit isprint(char c);
bit ispunct(char c);
bit isspace(char c);
bit isupper(char c);
bit isxdigit(char c);
bit toascii(char c);
bit toint(char c);
char tolower(char c);
char __tolower(char c);
char toupper(char c);
char __toupper(char c);
```

2. INTRINS.H

```
unsigned char _crol_(unsigned char c,unsigned char b);
unsigned char _cror_(unsigned char c,unsigned char b);
unsigned char _chkfloat_(float ual);
unsigned int _irol_(unsigned int i,unsigned char b);
unsigned int _iror_(unsigned int i,unsigned char b);
unsigned long _irol_(unsigned long l,unsigned char b);
unsigned long _iror_(unsigned long L,unsigned char b);
void _nop_(void);
bit _testbit_(bit b);
```

3. STDIO.H

```
char getchar(void);
char _getkey(void);
char *gets(char * string,int len);
int printf(const char * fmtstr[,argument]…);
char putchar(char c);
int puts (const char * string);
int scanf(const char * fmtstr.[,argument]…);
int sprintf(char * buffer,const char *fmtstr[;argument]);
int sscanf(char *buffer,const char * fmtstr[,argument]);
```

```
char ungetchar(char c);
void vprintf (const char *fmtstr,char * argptr);
void vsprintf(char *buffer,const char * fmtstr,char * argptr);
```

4. STDLIB.H

```
float atof(void * string);
int atoi(void * string);
long atol(void * string);
void * calloc(unsigned int num,unsigned int len);
void free(void xdata *p);
void init_mempool(void *data *p,unsigned int size);
void *malloc (unsigned int size);
int rand(void);
void *realloc (void xdata *p,unsigned int size);
void srand (int seed);
```

5. STRING.H

```
void *memccpy (void *dest,void *src,char c,int len);
void *memchr (void *buf,char c,int len);
char memcmp(void *buf1,void *buf2,int len);
void *memcopy (void *dest,void *SRC,int len);
void *memmove (void *dest,void *src,int len);
void *memset (void *buf,char c,int len);
char *strcat (char *dest,char *src);
char *strchr (const char *string,char c);
char strcmp (char *string1,char *string2);
char *strcpy (char *dest,char *src);
int strcspn(char *src,char * set);
int strlen (char *src);
char *strncat (char 8dest,char *src,int len);
char strncmp(char *string1,char *string2,int len);
char strncpy (char *dest,char *src,int len);
char *strpbrk (char *string,char *set);
int strpos (const char *string,char c);
char *strrchr (const char *string,char c);
char *strrpbrk (char *string,char *set);
int strrpos (const char *string,char c);
int strspn(char *string,char *set);
```