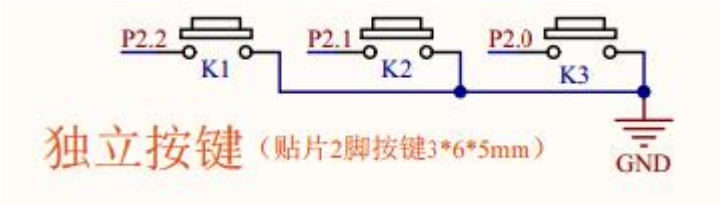
## 第九十二节： 独立按键的四大要素（自锁，消抖，非阻塞，清零式滤波）。

### 【92.1　独立按键的硬件电路简介。】




上图 92.1.1　独立按键电路

按键有两种驱动方式，一种是独立按键，一种是矩阵按键。1 个独立按键要占用 1 个 IO 口，IO 口不能共用。而矩阵按键的 IO 口是分时片选复用的，用少量的 IO 口就可以驱动翻倍级别的按键数量。比如，用 8 个 IO 口只能驱动 8 个独立按键，但是却可以驱动 16 个矩阵按键（4x4）。因此，按键少的时候就用独立按键，按键多的时候就用矩阵按键。这两种按键的驱动本质是一样的，都是靠识别输入信号的下降沿（或上升沿）来识别按键的触发。

独立按键的硬件原理基础，如上图，P2.2 这个 IO 口，在按键 K1 没有被按下的时候，P2.2 口因为单片机内部自带上拉电阻把电平拉高，此时 P2.2 口是高电平的输入状态。当按键 K1 被按下的时候，按键 K1 左右像一根导线连接到电源的负极（GND），直接把原来 P2.2 口的电平拉低，此时 P2.2 口变成了低电平的输入状态。编写按键驱动程序，就是要识别这个电平从高到低的过程，这个过程也叫下降沿。多说一句，51 单片机的 P1,P2,P3 口是内部自带上拉电阻的，而 P0 口是内部没有上拉电阻的，需要外接上拉电阻。除此之外，很多单片机内部其实都没有上拉电阻的，因此，建议大家在做独立按键电路的时候，养成一个习惯，凡是按键输入状态都外接上拉电阻。

识别按键的下降沿触发有四大要素：自锁，消抖，非阻塞，清零式滤波。

“自锁”，按键一旦进入到低电平，就要“自锁”起来，避免不断触发按键，只有当按键被松开变成高电平的时候，才及时“解锁”为下一次触发做准备。

“消抖”，按键是一个机械触点器件，在接触的瞬间必然存在微观上的机械抖动，反馈到电平的瞬间就是“高，低，高，低...”这种不稳定的电平状态是一种干扰，但是，按键一旦按下去稳定了之后，这种状态就消失，电平就一直保持稳定的低电平。消抖的本质就是滤波，要把这种接触的瞬间抖动过滤掉，避免按键的“一按多触发”。

“非阻塞”，在处理消抖的时候，必须用到延时，如果此时用阻塞的 delay 延时就会影响其它任务的运行效率，因此，用非阻塞的定时延时更加有优越性。

“清零式滤波”，在消抖的时候，有两种境界，第一种境界是判断两次电平的状态，中间插入“固定的时间”延时，这种方法前后一共判断了两次，第一次是识别到低电平就进入延时的状态，第二次是延时后再确认一次是否继续是低电平的状态，这种方法的不足是，“固定的时间”全凭经验值，但是不同的按键它们的抖动时间长度是不同的，除此之外，前后才判断了两次，在软件的抗干扰能力上也弱了很多，“密码等级”不够高。第二种境界就是“清零式滤波”，“清零式滤波”非常巧妙，抗扰能力超强，它能自动过滤不同按键的“抖动时间”，然后再进入一个“稳定时间”的“N 次识别判断”，更加巧妙的是，在“抖动时间”和“稳定时间”两者时间内，只要发现一次是高电平的干扰，就马上自动清零计时器，重新开始计时。“稳定时间”一般取 20ms 到 30ms 之间，而“抖动时间”是隐藏的，在代码上并没有直接描写出来，但是却无形地融入了代码之中，只有慢慢体会才能发现它的存在。

具体的代码如下，实现的功能是按一次 K1 或者 K2 按键，就触发一次蜂鸣器鸣叫。

```
#include "REG52.H"

#define KEY_VOICE_TIME   50 //按键触发后发出的声音长度
#define KEY_FILTER_TIME  25  //按键滤波的“稳定时间”25ms

void T0_time();
void SystemInitial(void) ;
void Delay(unsigned long u32DelayTime) ;
void PeripheralInitial(void) ;

void BeepOpen(void);
void BeepClose(void);
void VoiceScan(void);
void KeyScan(void);    //按键识别的驱动函数，放在定时中断里
void KeyTask(void);    //按键任务函数，放在主函数内

sbit P3_4=P3^4;
sbit KEY_INPUT1=P2^2;  //K1 按键识别的输入口。
sbit KEY_INPUT2=P2^1;  //K2 按键识别的输入口。

volatile unsigned char vGu8BeepTimerFlag=0;
volatile unsigned int vGu16BeepTimerCnt=0;

volatile unsigned char vGu8KeySec=0;  //按键的触发序号，全局变量意味着是其它函数的接口。

void main()
{
    SystemInitial();
    Delay(10000);
    PeripheralInitial();
    while(1)
    {
       KeyTask();    //按键任务函数
    }
}

void T0_time() interrupt 1
{
    VoiceScan();
    KeyScan();    //按键识别的驱动函数

    TH0=0xfc;
    TL0=0x66;
```

```
}


void SystemInitial(void)
{
    TMOD=0x01;
    TH0=0xfc;
    TL0=0x66;
    EA=1;
    ET0=1;
    TR0=1;
}

void Delay(unsigned long u32DelayTime)
{
    for(;u32DelayTime>0;u32DelayTime--);
}

void PeripheralInitial(void)
{

}

void BeepOpen(void)
{
    P3_4=0;
}

void BeepClose(void)
{
    P3_4=1;
}

void VoiceScan(void)
{

      static unsigned char Su8Lock=0;

      if(1==vGu8BeepTimerFlag&&vGu16BeepTimerCnt>0)
      {
          if(0==Su8Lock)
          {
               Su8Lock=1;
               BeepOpen();
```

```
          }
          else
          {

               vGu16BeepTimerCnt--;

               if(0==vGu16BeepTimerCnt)
               {
                   Su8Lock=0;
                    BeepClose();
               }

          }
      }
}

/* 注释一：
* 独立按键扫描的详细过程，以按键 K1 为例，如下：
* 第一步：平时没有按键被触发时，按键的自锁标志,去抖动延时计数器一直被清零。
* 第二步：一旦有按键被按下，去抖动延时计数器开始在定时中断函数里累加，在还没累加到
*         阀值 KEY_FILTER_TIME 时，如果在这期间由于受外界干扰或者按键抖动，而使
*         IO 口突然瞬间触发成高电平，这个时候马上把延时计数器 Su16KeyCnt1 清零了,这个过程
*         非常巧妙，非常有效地去除瞬间的杂波干扰。以后凡是用到开关感应器的时候，
*         都可以用类似这样的方法去干扰。
* 第三步：如果按键按下的时间达到阀值 KEY_FILTER_TIME 时，则触发按键，把编号 vGu8KeySec 赋值。
*         同时，马上把自锁标志 Su8KeyLock1 置 1，防止按住按键不松手后一直触发。
* 第四步：等按键松开后，自锁标志 Su8KeyLock1 及时清零（解锁），为下一次自锁做准备。
* 第五步：以上整个过程，就是识别按键 IO 口下降沿触发的过程。
*/
void KeyScan(void)  //此函数放在定时中断里每 1ms 扫描一次
{
   static unsigned char Su8KeyLock1; //1 号按键的自锁
   static unsigned int  Su16KeyCnt1; //1 号按键的计时器
   static unsigned char Su8KeyLock2; //2 号按键的自锁
   static unsigned int  Su16KeyCnt2; //2 号按键的计时器

   //1 号按键
   if(0!=KEY_INPUT1)//IO 是高电平，说明按键没有被按下，这时要及时清零一些标志位
   {
      Su8KeyLock1=0; //按键解锁
      Su16KeyCnt1=0; //按键去抖动延时计数器清零，此行非常巧妙，是全场的亮点。
   }
   else if(0==Su8KeyLock1)//有按键按下，且是第一次被按下。这行很多初学者有疑问，请看专题分析。
   {
```

```
      Su16KeyCnt1++; //累加定时中断次数
      if(Su16KeyCnt1>=KEY_FILTER_TIME) //滤波的“稳定时间”KEY_FILTER_TIME，长度是 25ms。
      {
         Su8KeyLock1=1;  //按键的自锁,避免一直触发
         vGu8KeySec=1;    //触发 1 号键
      }
   }

   //2 号按键
   if(0!=KEY_INPUT2)
   {
      Su8KeyLock2=0;
      Su16KeyCnt2=0;
   }
   else if(0==Su8KeyLock2)
   {
      Su16KeyCnt2++;
      if(Su16KeyCnt2>=KEY_FILTER_TIME)
      {
         Su8KeyLock2=1;
         vGu8KeySec=2;    //触发 2 号键
      }
   }



}

void KeyTask(void)    //按键任务函数，放在主函数内
{
    if(0==vGu8KeySec)
    {
        return;//按键的触发序号是 0 意味着无按键触发，直接退出当前函数，不执行此函数下面的代码
    }

    switch(vGu8KeySec) //根据不同的按键触发序号执行对应的代码
    {
       case 1:     //1 号按键

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME;  //触发按键后，发出固定长度的声音
            vGu8BeepTimerFlag=1;
            vGu8KeySec=0;  //响应按键服务处理程序后，按键编号必须清零，避免一致触发
            break;
```

```
      case 2:     //2 号按键

            vGu8BeepTimerFlag=0;
            vGu16BeepTimerCnt=KEY_VOICE_TIME;  //触发按键后，发出固定长度的声音
            vGu8BeepTimerFlag=1;
            vGu8KeySec=0;  //响应按键服务处理程序后，按键编号必须清零，避免一致触发
            break;

    }
}
```

## 【92.2　专题分析：else if(0==Su8KeyLock1)。】

疑问：

```
  if(0!=KEY_INPUT1)
  {
     Su8KeyLock1=0;
     Su16KeyCnt1=0;
  }
  else if(0==Su8KeyLock1)//有按键按下，且是第一次被按下。为什么？为什么？为什么？
  {
     Su16KeyCnt1++;
     if(Su16KeyCnt1>KEY_FILTER_TIME)
     {
        Su8KeyLock1=1;
        vGu8KeySec=1;
     }
  }
```

解答：

```
首先，我们要明白 C 语言的语法中，
if(条件 1)
{

}
else if(条件 2)
{

}
```

以上语句是一对组合语句，不能分开来看。当（条件 1）成立的时候，它是绝对不会判断（条件 2）的。当（条件 1）不成立的时候，才会判断（条件 2）。

回到刚才的问题，当程序执行到（条件 2） else if(0==Su8KeyLock1)的时候，就已经默认了（条件 1）if(0!=KEY_INPUT1)不成立，这个条件不成立，就意味着 0==KEY_INPUT1，也就是有按键被按下，因此，这里的 else if(0==Su8KeyLock1)等效于 else if(0==Su8KeyLock1&&0==KEY_INPUT1)，而 Su8KeyLock1 是一个自

锁标志位，一旦按键被触发后，这个标志位会变 1，防止按键按住不松手的时候不断触发按键。这样，按键只能按一次触发一次，松开手后再按一次，又触发一次。

## 【92.3　专题分析：if(0!=KEY_INPUT1)。】

疑问：为什么不用 if(1==KEY_INPUT1)而用 if(0!=KEY_INPUT1)？

解答：其实两者在功能上是完全等效的，在这里都可以用。之所以本教程优先选用后者 if(0!=KEY_INPUT1)，是因为考虑到了代码在不同单片机平台上的可移植性和兼容性。很多 32 位的单片机提供的是库函数，库函数返回的按键状态是一个字节变量来表示，当被按下的时候是 0，但是，当没有按下的时候并不一定等于 1，而是一个“非 0”的数值。

## 【92.4　专题分析：把 KeyScan 函数放在定时器中断里。】

疑问：为什么把 KeyScan 函数放在定时器中断里？

解答：中断函数里放的函数或者代码越少越好，但是 KeyScan 函数是特殊的函数，是涉及到 IO 口输入信号的滤波，滤波就涉及到时间的及时性与均匀性，放在定时中断函数里更加能保证时间的一致性。比如，蜂鸣器驱动，动态数码管驱动，按键扫描驱动，我个人都习惯放在定时中断函数里。

## 【92.5　专题分析：if(0==vGu8KeySec)return。】

疑问：if(0==vGu8KeySec)return 是不是多此一举？

解答：在 KeyTask 函数这里，if(0==vGu8KeySec)return 这行代码删掉，对程序功能是没有影响的，这里之所以多插入这行判断语句，是因为，当按键多达几十个的时候，避免主函数每次进入 KeyTask 函数，都挨个扫描判断 switch 的状态进行多次判断，如果增加了这行 if(0==vGu8KeySec)return 代码，就可以直接退出省事，在理论上感觉更加运行高效。其实，不同单片机不同的 C 编译器可能对 switch 语句的翻译不一样，因此，这里的是不是更加高效我不敢保证。但是可以保证的是，加了这行代码也没有其它副作用。