C51 单片机及 C 语言知识点必备秘籍

——单片机学习知识点

单片机学习知识点全攻略（一）

导语：单片机对于初学者来说确实很难理解，不少学过单片机的同学或电子爱好者，甚至在毕业时仍旧是一无所获。基于此，电子发烧友网将整合《单片机关键知识点全攻略》，共分为四个系列，以飨读者，敬请期待！此系列对于业内电子工程师也有收藏和参考价值。

单片机关键知识点一览：

系列一

- 1：单片机简叙

- 2：单片机引脚介绍
- 3：单片机存储器结构
- 4：第一个单片机小程序
- 5：单片机延时程序分析
- 6：单片机并行口结构
- 7：单片机的特殊功能寄存器

系列二

- 8：单片机寻址方式与指令系统
- 9：单片机数据传递类指令
- 10：单片机数据传送类指令
- 11：单片机算术运算指令
- 12：单片机逻辑运算类指令
- 13：单片机逻辑与或异或指令祥解
- 14：单片机条件转移指令

系列三

- 15：单片机位操作指令

- 16：单片机定时器与计数器
- 17：单片机定时器/计数器的方式
- 18：单片机的中断系统
- 19：单片机定时器、中断试验
- 20：单片机定时/计数器实验
- 21：单片机串行口介绍

系列四

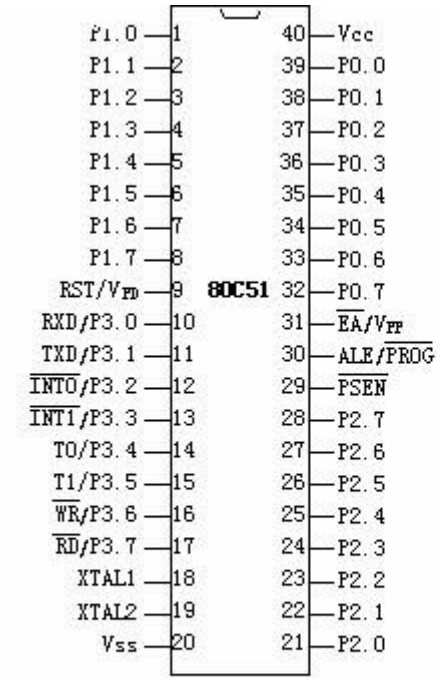
- 22：单片机串行口通信程序设计
- 23：LED 数码管静态显示接口与编
- 24：动态扫描显示接口电路及程序
- 25：单片机键盘接口程序设计
- 26：单片机矩阵式键盘接口技术及
- 27：关于单片机的一些基本概念
- 28：实际案例实践——单片机音乐程序设计

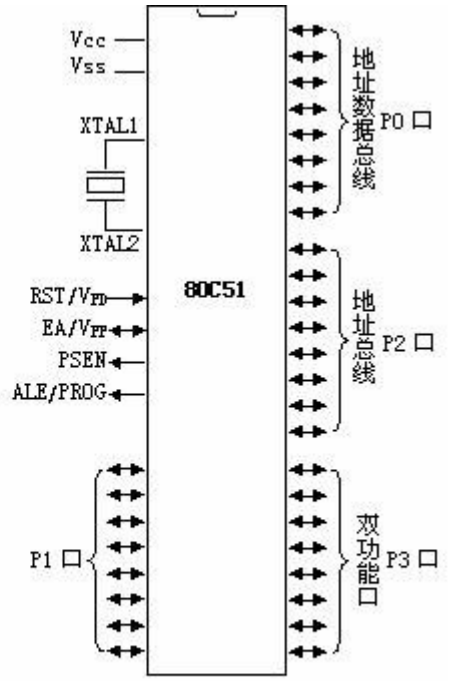
1：单片机简叙

什么是单片机 一台能够工作的计算机要有这样几个部份构成:CPU(进行运算、控制)、RAM(数据存储)、ROM(程序存储)、输入/输出设备(例如:串行口、并行输 出口等)。在个人计算机上这些部份被分成若干块芯片,安装一个称之为主板的印刷线路板上。而在单片机中,这些部份,全部被做到一块集成电路芯片中了,所以 就称为单片(单芯片)机,而且有一些单片机中除了上述部份外,还集成了其它部份如 A/D,D/A 等。

单片机是一种控制芯片,一个微型的计算机,而加上晶振,存储器,地址锁存器,逻辑门,七段译码器(显示器),按钮(类似键盘),扩展芯片,接口等那是单片机系统。

2:单片机引脚介绍





单片机的 40 个引脚大致可分为 4 类：电源、时钟、控制和 I/O 引脚。

1. 电源：

(1) VCC - 芯片电源，接+5V；

(2) VSS - 接地端；

2. 时钟：XTAL1、XTAL2 - 晶体振荡电路反相输入端和输出端。

3. 控制线：控制线共有 4 根，

(1) ALE/PROG：地址锁存允许/片内 EPROM 编程脉冲

① ALE 功能：用来锁存 P0 口送出的低 8 位地址

② PROG 功能：片内有 EPROM 的芯片，在 EPROM 编程期间，此引脚输入编程脉冲。

(2) PSEN：外 ROM 读选通信号。

(3) RST/VPD：复位/备用电源。

① RST（Reset）功能：复位信号输入端。

② VPD 功能：在 Vcc 掉电情况下，接备用电源。

(4) EA/Vpp：内外 ROM 选择/片内 EPROM 编程电源。

① EA 功能：内外 ROM 选择端。

② Vpp 功能：片内有 EPROM 的芯片，在 EPROM 编程期间，施加编程电源 Vpp。

4. I/O 线

80C51 共有 4 个 8 位并行 I/O 端口：P0、P1、P2、P3 口，共 32 个引脚。P3 口还具有第二功能，用于特殊信号输入输出和控制信号（属控制总线）。

拿到一块芯片，想要使用它，首先必须要知道怎样连线，我们用的一块称之为 89C51 的芯片，下面我们就看一下如何给它连线。

1、 电源：这当然是必不可少的了。单片机使用的是 5V 电源，其中正极接 40 管脚，负极（地）接 20 管脚。

2、 振荡电路：单片机是一种时序电路，必须供给脉冲信号才能正常工作，在单片机内部已集成了振荡器，使用晶体振荡器，接 18、19 脚。只要买来晶体震荡器，电容，连上就能了，按图 1 接上即可。

3、 复位管脚：按图 1 中画法连好，至于复位是何含义及为何需要复要复位，在单片机功能中介绍。

4、 EA 管脚：EA 管脚接到正电源端。 至此，一个单片机就接好，通上电，单片机就开始工作了。

我们的第一个任务是要用单片机点亮一只发光二极管 LED，显然，这个 LED 必须要和单片机的某个管脚相连，不然单片机就没法控制它了，那么和哪个管脚相连呢？单片机上除了刚才用掉的 5 个管脚，还有 35 个，我们将这个 LED 和 1 脚相连。（见图 1，其中 R1 是限流电阻）

按照这个图的接法，当 1 脚是高电平时，LED 不亮，只有 1 脚是低电平时，LED 才发亮。因此要 1 脚我们要能够控制，也就是说，我们要能够让 1 管脚按要 求变为高或低电平。即然我们要控制 1 脚，就得给它起个名字，总不能就叫它一脚吧？叫它什么名字呢？设计 51 芯片的 INTEL 公司已经起好了，就叫它 P1.0，这是规定，不能由我们来更改。

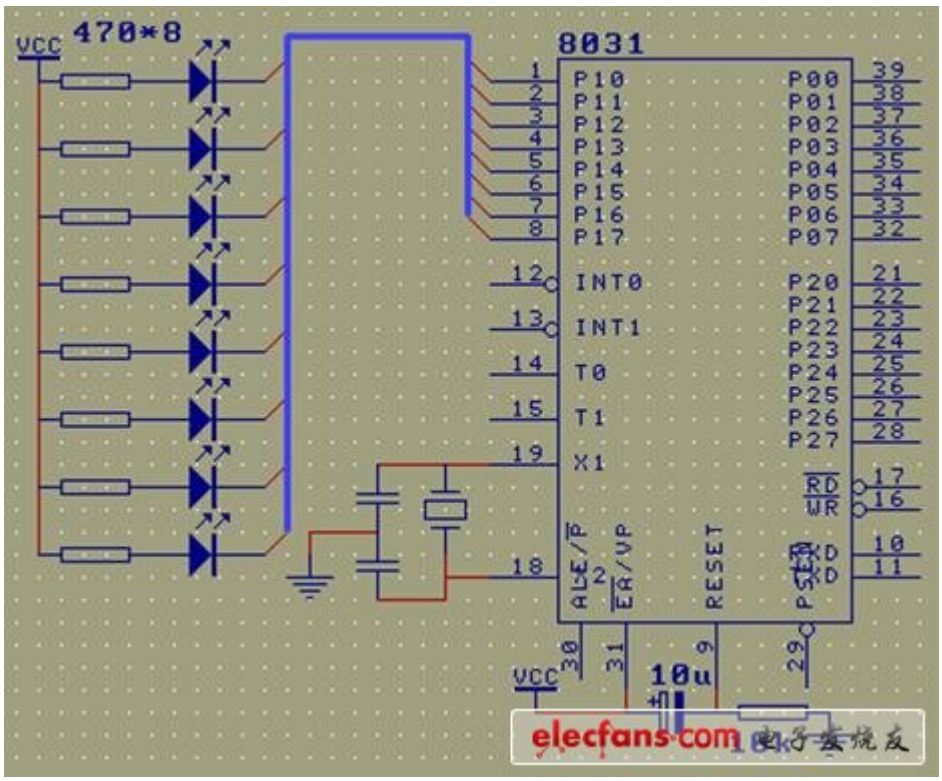


图 1

名字有了，我们又怎样让它变‘高’或变‘低’呢？叫人做事，说一声就能，这叫发布命令，要计算机做事，也得要向计算机发命令，计算机能听得懂的命令称 之为计算机的指令。让一个管脚输出高电平的指令是 SETB，让一个管脚输出低电平的指令是 CLR。因此，我们要 P1.0 输出高电平，只要写 SETB P1.0，要 P1.0 输出低电平，只要写 CLR P1.0 就能了。

现在我们已经有办法让计算机去将 P10 输出高或低电平了，但是我们怎样才能计算机执行这条指令呢？总不能也对计算机也说一声了事吧。要解决这个问题，还得有几步要走。第一，计算机看不懂 SETB CLR 之类的指令，我们得把指令翻译成计算机能懂的方式，再让

计算机去读。计算机能懂什么呢？它只懂一样东西——数字。因此我们得把 SETB P1.0 变为（D2H，90H ），把 CLR P1.0 变为 （C2H，90H ），至于为什么是这两个数字，这也是由 51 芯片的设计者--INTEL 规定的，我们不去研究。第二步，在得到这两个数字后，怎样让这两个数字进入单片机的 内部呢？这要借助于一个硬件工具“编程器”。如果你还不知道是什么是编程器，我来介绍一下，就是把你在电脑上写出来来的代码用汇编器等编译器生成的一个目 标烧写到单片机的 eprom 里面去的工具，80c51 这种类型的单片机编程是一件很麻烦的事情，必要要先装到编程器上编程后才能在设备上使用，而目前最新 的 89s51 单片机居然在线编程（isp）功能，不用拔出来利用简单的电路就可以实现把代码写入单片机内部，本站有详细的 at89s51 编程器制作教程

我们将编程器与电脑连好，运行编程器的软件，然后在编缉区内写入（D2H，90H）见图 2，

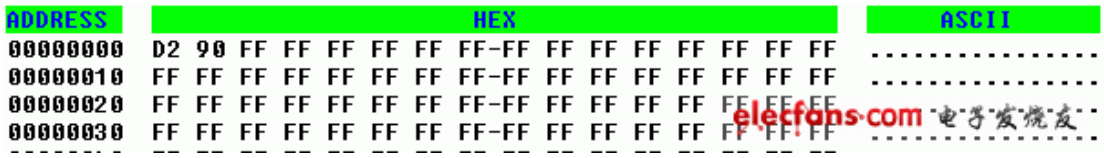


图 2

写入……好，拿下片子，把片子插入做好的电路板，接通电源……什么？灯不亮？这就对了，因为我们写进去的指令就是让 P10 输出高电平，灯当然不亮，要 是亮就错了。现在我们再拨下这块芯片，重新放回到编程器上，将编缉区的内容改为（C2H，90H），也就是 CLR P1.0，写片，拿下片子，把片子插进电路板，接电，好，灯亮了。因为我们写入的（）就是让 P10 输出低电平的指令。这样我们看到，硬件电路的连线没有做 任何改变，只要改变写入单片机中的内容，就能改变电路的输出效果。

3：单片机存储器结构

单片机内部存储结构分析

我们来思考一个问题，当我们在编程器中把一条指令写进单片要内部，然后取下单片机，单片机就可以执行这条指令，那么这条指令一定保存在单片机的某个地 方，并且这个地方在单片机掉电后依然可以保持这条指令不会丢失，这是个什么地方呢？这个地方就是单片机内部的只读存储器即 ROM（READ ONLY MEMORY）。为什么称它为只读存储器呢？刚才我们不是明明把两个数字写进去了吗？原来在 89C51 中的 ROM 是一种电可擦除的 ROM，称为 FLASH ROM，刚才我们是用的编程器，在特殊的条件下由外部设备对 ROM 进行写的操作，在单片机正常工作条件下，只能从那面读，不能把数据写进去，所以我们还是 把它称为 ROM。

数的本质和物理现象：我们知道，计算机能进行数学运算，这可令我们非常的难以理解，计算机吗，我们虽不了解它的组成，但它总只是一些电子元器件，怎么 能进行数学运算呢？我们做数学题如 37+45 是这样做的，先在纸上写 37，然后在下面写 45，然后大脑运算，最后写出结果，运算的原材料：37、45 和结 果：82 都是写在纸上的，计算机中又是放在什么地方呢？为了解决这个问题，先让我们做一个实验：这里有一盏灯，我们知道灯要么亮，要么不亮，就有两种状 态，我们能用’0’和’1’来代替这两种状态，规定亮为’1’，不亮为’0’。现在放上两盏灯，一共有几种状态呢？我们列表来看一下：

状态	○○	○●	●○	●●
表达	0 0	0 1	1 0	elecfans.com 电子发烧友

请大家自己写上 3 盏灯的情况 000 001 010 011 100 101 110 111

我们来看，这个 000，001，101 不就是我们学过的的二进制数吗？本来，灯的亮和灭只是一种物理现象，可当我们把它们按一按的次序排更好后，灯的亮和灭就代表了数字了。让我们再抽象一步， 灯为什么会亮呢？看电路 1，是因为输出电路输出高电平，给灯通了电。因此，灯亮和灭就能用电路的输出是高电平还是低电平来替代了。这样，数字就和电平的 高、低联系上了。（请想一下，我们还看到过什么样的类似的例程呢？（海军之）灯语、旗语，电报，甚至红、绿灯）

什么是位：

通过上面的实验我们已经知道：一盏灯亮或者说一根线的电平的高低，能代表两种状态：0 和 1。实际上这就是一个二进制位，因此我们就把一根线称之为一“位”，用 BIT 表示。

什么是字节：

一根线能表于 0 和 1，两根线能表达 00，01，10，11 四种状态，也就是能表于 0 到 3，而三根能表达 0-7，计算机中常常用 8 根线放在一起，同时计 数，就能表过到 0-255 一共 256 种状态。这 8 根线或者 8 位就称之为一个字节（BYTE）。不要问我为什么是 8 根而不是其它数，因为我也不知道。（计算 机世界是一本人造的世界，不是自然界，很多事情你无法问为什么，只能说：它是一种规定，大家在以后的学习过程中也要注意这个问题）

存储器的工作原理：

1、存储器构造

存储器就是用来存放数据的地方。它是利用电平的高低来存放数据的，也就是说，它存放的实际上是电平的高、低，而不是我们所习惯认为的 1234 这样的数字，这样，我们的一个谜团就解开了，计算机也没什么神秘的吗。

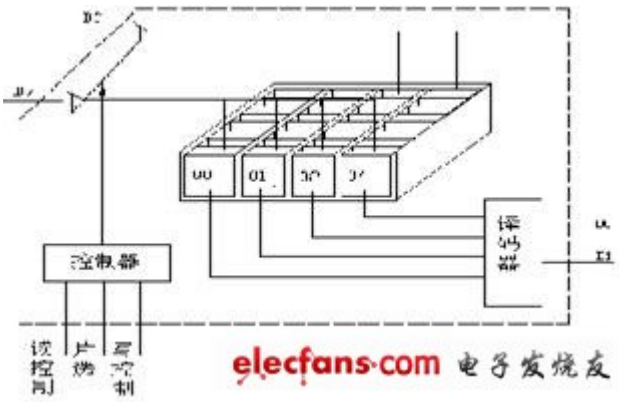


图 2

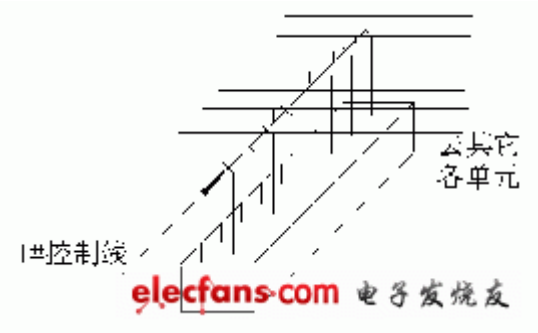


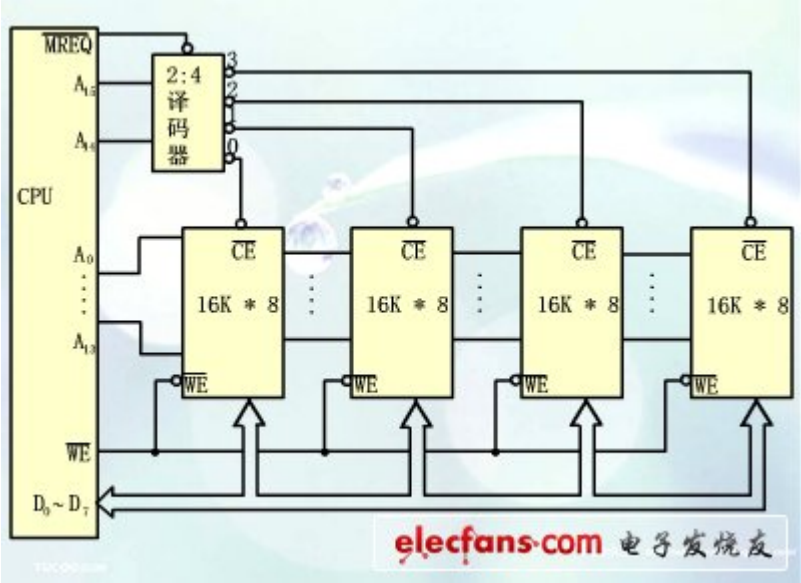
图 3

让我们看图 2。单片机里面都有这样的存储器，这是一个存储器的示意图：一个存储器就象一个个的小抽屉，一个小抽屉里有八个小格子，每个小格子就是用来 存放“电荷”的，电荷通过与它相连的电线传进来或释放掉，至于电荷在小格子里是怎样存的，就不用我们操心了，你能把电线想象成水管，小格子里的电荷就象是 水，那就好理解了。存储器中的每个小抽屉就是一个放数据的地方，我们称之为一个“单元”。

有了这么一个构造，我们就能开始存放数据了，想要放进一个数据 12，也就是 00001100，我们只要把第二号和第三号小格子里存满电荷，而其它小格 子里的电荷给放掉就行了（看图 3）。可是问题出来了，看图 2，一个存储器有好多单元，线是并联的，在放入电荷的时候，会将电荷放入所有的单元中，而释放电 荷的时候，会把每个单元中的电荷都放掉，这样的话，不管存储器有多少个单元，都只能放同一个数，这当然不是我们所希望的，因此，要在结构上稍作变化，看图 2，在每个单元上有个控制线，我想要把数据放进哪个单元，就给一个信号这个单元的控制线，这个控制线就把开关打开，这样电荷就能自由流动了，而其它单元控 制线上没有信号，所以开关不打开，不会受到影响，这样，只要控制不一样单元的控制线，就能向各单元写入不一样的数据了，同样，如果要某个单元中取数据，也 只要打开对应的控制开关就行了。

2、存储器译码

那么，我们怎样来控制各个单元的控制线呢？这个还不简单，把每个单元元的控制线都引到集成电路的外面不就行了吗？事情可没那么简单，一片 27512 存 储器中有 65536 个单元，把每根线都引出来，这个集成电路就得有 6 万多个脚？不行，怎么办？要想法减少线的数量。我们有一种办法称这为译码，简单介绍一 下：一根线能代表 2 种状态，2 根线能代表 4 种状态，3 根线能代表几种，256 种状态又需要几根线代表？8 种，8 根线，所以 65536 种状态我们只需要 16 根线就能代表了。



3、存储器的选片及总线的概念

至此，译码的问题解决了，让我们再来关注另外一个问题。送入每个单元的八根线是用从什么地方来的呢？它就是从计算机上接过来的，一般地，这八根线除了 接一个存储器之外，还要接其它的器件，如图 4 所示。这样问题就出来了，这八根线既然不是存储器和计算机之间专用的，如果总是将某个单元接在这八根线上，就 不好了，比如这个存储器单元中的数值是 0FFH 另一个存储器的单元是 00H，那么这根线到底是处于高电平，还是低电平？岂非要打架看谁历害了？所以我们要 让它们分离。办法当然很简单，当外面的线接到集成电路的管脚进来后，不直接接到各单元去，中间再加一组开关（参考图 4 ）就行了。

平时我们让开关关闭着，如果确实是要向这个存储器中写入数据，或要从存储器中读出数据，再让开关接通就行了。这组开关由三根引线选择：读控制 端、写控制端和片选端。要将数据写入片中，先选中该片，然后发出写信号，开关就合上了，并将传过来的数据（电荷）写入片中。如果要读，先选中该片，然后发 出读信号，开关合上，数据就被送出去了。注意图 4，读和写信号同时还接入到另一个存储器，但是由于片选端不一样，所以虽有读或写信号，但没有片选信号，所 以另一个存储器不会“误会”而开门，造成冲突。那么会不一样时选中两片芯片呢？只要是设计好的系统就不会，因为它是由计算控制的，而不是我们人来控制的， 如果真的出现同时出现选中两片的情况，那就是电路出了故障了，这不在我们的讨论之列。

从上面的介绍中我们已经看到，用来传递数据的八根线并不是专用的，而是很多器件大家共用的，所以我们称之为数据总线，总线英文名为 BUS，总即公交车道，谁者能走。而十六根地址线也是连在一起的，称之为地址总线。

半导体存储器的分类

按功能能分为只读和随机存取存储器两大类。所谓只读，从字面上理解就是只能从里面读，不能写进去，它类似于我们的书本，发到我们手回之后，我们只能读里面的内容，不能随意更改书本上的内容。只读存储器的英文缩写为 ROM（READ ONLY MEMORY）

所谓随机存取存储器，即随时能改写，也能读出里面的数据，它类似于我们的黑板，我能随时写东西上去，也能用黑板擦擦掉重写。随机存储器的英文缩写为 RAM（READ RANDOM MEMORY）这两种存储器的英文缩写一定要记牢。

注意：所谓的只读和随机存取都是指在正常工作情况下而言，也就是在使用这块存储器的时候，而不是指制造这块芯片的时候。不然，只读存储器中的数据是怎 么来的呢？其实这个道理也很好理解，书本拿到我们手里是不能改了，能当它还是原材料——白纸的时候，当然能由印刷厂印上去了。

顺便解释一下其它几个常见的概念。

PROM，称之为可编程存储器。这就象我们的练习本，买来的时候是空白的，能写东西上去，可一旦写上去，就擦不掉了，所以它只能用写一次，要是写错了，就报销了。（现在已经被淘汰）

EPROM，称之为紫外线擦除的可编程只读存储器。它里面的内容写上去之后，如果觉得不满意，能用一种特殊的办法去掉后重写，这就是用紫外线照射，紫 外线就象“消字灵”，能把字去掉，然后再重写。当然消的次数多了，也就不灵光了，所以这种芯片能擦除的次数也是有限的——几百次吧。（现在已经被淘汰）

EEPROM，也叫 E2PROM 称之为电可擦可编程只读存储器，它和 EEPROM 类似，写上去的东西也能擦掉重写，但它要方便一些，不需要光照了，只要用电就能擦除或者重新改写数据，所以就方便许多，而且寿命也很长（几万到几十万次不等）。

FLASH，称之为闪速存储器，属于 EEPROM 的改进产品，它的最大特点是必须按块(Block)擦除（每个区块的大小不定，不同厂家的产品有不同的规格）， 而 EEPROM 则可以一次只擦除一个字节（Byte）。FLASH 现在常用于大容量存储，比如 u 盘

再次强调，这里的所有的写都不是指在正常工作条件下。不管是 PROM 还是 EPROM，它们的写都要有特殊的条件，一般我们用一种称之为“编程器”的设备来做这项工作，一旦把它装到它的工作位置，就不能随便改写了。

4：第一个单片机小程序

上一次我们的程序实在是没什么用，要灯亮还要重写一下片子，下面我们要让灯持续地闪烁，这就有一定的实用价值了，比如能把它当成汽车上的一个信号灯用 了。怎样才能让灯持续地闪烁呢？实际上就是要灯亮一段时间，再灭一段时间，也就是说要 P10 持续地输出高和低电平。怎样实现这个要求呢？请考虑用下面的指 令是否可行：

```
SETB P10

CLR P10 ……
```

这是不行的，有两个问题，第一，计算机执行指令的时间很快，执行完 SETB P10 后，灯是灭了，但在极短时间（微秒级）后，计算机又执行了 CLR P10 指令，灯又亮了，所以根本分辨不出灯曾灭过。第二，在执行完 CLR P10 后，不会再去执行 SETB P10 指令，所以以后再也没有机会让灭了。

为了解决这两个问题，我们能做如下设想，第一，在执行完 SETB P10 后，延时一段时间（几秒或零点几秒）再执行第二条指令，就能分辨出灯曾灭过了。第二在执行完第二条指令后，让计算机再去执行第一条指令，持续地在原 地兜圈，我们称之为“循环”，这样就能完成任务了。

以下先给出程序（后面括号中的数字是为了便于讲解而写的，实际不用输入）：

```
；主程序：
LOOP：  SETB P10          ；（1）
LCALL DELAY      ；（2）
CLR P10          ；（3）
LCALL DELAY      ；（4）
AJMP LOOP        ；（5）
；以下子程序
DELAY：  MOV R7，#250     ；（6）
D1：  MOV R6，#250        ；（7）
D2：  DJNZ R6，D2         ；（8）
```

```
DJNZ R7，D1          ；（9）

RET                  ；（10）

END                  ；（11）
```

按上面的设想分析一下前面的五条指令。

第一条是让灯灭，第二条应当是延时，第三条是让灯亮，第四条和第二条一模一样，也是延时，第五条应当是转去执行第一条指令。第二和第四条实现的原理稍 后谈，先看第五条，LJMP 是一条指令，意思是转移，往什么地方转移呢？后面跟的是 LOOP，看一下，什么地方还有 LOOP，对了，在第一条指令的前面有 一个 LOOP，所以很直观地，我们能认识到，它要转到第一条指令处。这个第一条指令前面的 LOOP 被称之为标号，它的用途就是给这一行起一个名字，便于使 用。是否一定要给它起名叫 LOOP 呢？当然不是，起什么名字，完全由编程序的人决定，能称它为 A，X 等等，当然，这个时候，第五条指令 LJMP 后面的名字 也得跟着改了。

第二条和第四条指令的用途是延时，它是怎样实现的呢？指令的形式是 LCALL，这条指令称为调用子程序指令，看一下指令后面跟的是什么，DELAY， 找一下 DELAY，在第六条指令的前面，显然，这也是一个标号。这条指令的作用是这样的：当执行 LCALL 指令时，程序就转到 LCALL 后面的标号所标定 的程序处执行，如果在执行指令的过程中遇到 RET 指令，则程序就返回到 LCALL 指令的下面的一条指令继续执行，从第六行开始的指令中，能看到确实有 RET 指令。在执行第二条指令后，将转去执行第 6 条指令，而在执行完 6，7，8，9 条指令后将遇到第 10 条令：RET，执行该条指令后，程序将回来执行第 三条指令，即将 P10 清零，使灯亮，然后又是第四条指令，执行第四条指令就是转去执行第 6，7，8，9，10 条指令，然后回来执行第 5 条指令，第 5 条指令 就是让程序回到第 1 条开始执行，如此周而复始，灯就在持续地亮、灭了。

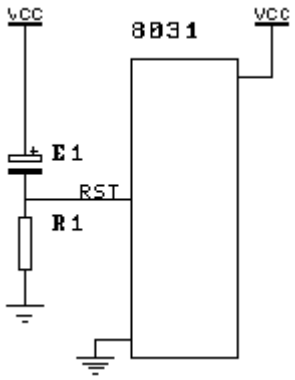
在标号 DELAY 标志的这一行到 RET 这一行中的所有程序，这是一段延时程序，大概延时零点几秒，至于具体的时间，以后我们再学习如何计算。 程序的最后一行是 END，这不是一条指令，它只是告诉我们程序到此结束，它被称为“伪指令”。

单片机内部结构分析：为了知道延时程序是如何工作的，我们必需首先了解延时程序中出现的一些符号，就从 R1 开始，R1 被称之为工作寄存器。什么是工作 寄存器呢？让我们从现实生活中来找找答案。如果出一道数学题：123+567，让你回答结果是多少，你会马上答出是 690，再看下面一道 题：123+567+562，要让你要上回答，就不这么不难了吧？我们会怎样做呢？如果有张纸，就不难了，我们先算出 123+567=690，把 690 写 在纸上，然后再算 690+562 得到结果是 1552。这其中 1552 是我们想要的结果，而 690 并非我们所要的结果，但是为了得到最终结果，我们又不得不 先算出 690，并记下来，这其实是一个中间结果，计算机中做运算和这个类似，为了要得到最终结果，一般要做很多步的中间结果，这些中间结果要有个地方放才 行，把它们放哪呢？放在前面提到过的 ROM 中能吗？显然不行，因为计算机要将结果写进去，而 ROM 是不能写的，所以在单片机中另有一个区域称为 RAM 区（RAM 是随机存取存储器的英文缩写),它能将数据写进去。 特别地，在 MCS-51 单片机中，将 RAM 中分出一块区域，称为工作寄存器区。

5：单片机延时程序分析

上一次课中，我们已经知道，程序中的符号 R7、R6 是代表了一个个的 RAM 单元，是用来放一些数据的，下面我们再来看一下其它符号的含义。

```
DELAY:  MOV R7, #250      ; （6）
D1:  MOV R6, #250         ; （7）
D2:  DJNZ R6, D2          ; （8）
DJNZ R7, D1               ; （9）
RET                       ; （10）
```



〈单片机延时程序〉

MOV：这是一条指令，意思是传递数据。说到传递，我们都很清楚，传东西要从一本人的手上传到另一本人的手上，也就是说要有一个接受者，一个传递者和 一样东西。从指令 MOV R7，#250 中来分析，R7 是一个接受者，250 是被传递的数，传递者在这条指令中被省略了(注意:并不是每一条传递指令都会省的,事实上大部份数据传 递指令都会有传递者)。它的意义也很明显：将数据 250 送到 R7 中去，因此执行完这条指令后，R7 单元中的值就应当是 250。在 250 前面有个#号，这又 是什么意思呢？这个#就是用来说明 250 就是一个被传递的东西本身，而不是传递者。那么 MOV R6，#250 是什么意思，应当不用分析了吧。

DJNZ：这是另一条指令，我们来看一下这条指令后面跟着的两个东西，一个是 R6，一个是 D2，R6 我们当然已知是什么了，查一下 D2 是什么。D2 在 本行的前面，我们已学过，这称之为标号。标号的用途是什么呢？就是给本行起一个名字。DJNZ 指令的执行过程是这样的，它将其后面的第一个参数中的值减 1，然后看一下，这个值是否等于 0，如果等于 0，就往下执行，如果不等于 0，就转移，转到什么地方去呢？可能大家已猜到了，转到第二个参数所指定的地方去 （请大家用自己的话讲一下这条语句是怎样执行的）。本条指令的最终执行结果就是，在原地转圈 250 次。

执行完了 DJNZ R6，D2 之后（也就是 R6 的值等于 0 之后），就会去执行下面一行，也就是 DJNZ R7，D1，请大家自行分析一下这句话执行的结果。（转去执行 MOV R6，#250，同时 R7 中的值减 1），最终 DJNZ R6，D2 这句话将被执行 250*250=62500 次，执行这么多次同一条指令干吗？就是为了延时。

一个问题：如果在 R6 中放入 0，会有什么样的结果。

二、时序分析：

前面我们介绍了延时程序，但这还不完善，因为，我们只知道 DJNZ R6，D2 这句话会被执行 62500 次，但是执行这么多次需要多长时间呢？是否满足我们的要求呢？我们还不知道，所以下面要来解决这个问题。

先提一个问题：我们学校里什么是最重要的。（铃声）校长能出差，老师能休息，但学校一日无铃声必定大乱。整个学校就是在铃声的统一指挥下，步调一致， 统一协调地工作着。这个铃是按一定的时间安排来响的，我们能称之为“时序&#0;&#0;时间的次序”。一个由人组成的单位尚且要有一定 的时序，计算机当然更要有严格的时序。事实上，计算机更象一个大钟，什么时候分针动，什么时候秒针动，什么时候时针动，都有严格的规定，一点也不能乱。计 算机要完成的事更复杂，所以它的时序也更复杂。

我们已知，计算机工作时，是一条一条地从 ROM 中取指令，然后一步一步地执行，我们规定：计算机访问一次存储器的时间，称之为一个机器周期。这是一个 时间基准，好象我们人用“秒”作为我们的时间基准一样，为什么不干脆用“秒”，多好，很习惯，学下去我们就会知道用“秒”反而不习惯。

一个机器周期包括 12 个时钟周期。下面让我们算一下一个机器周期是多长时间吧。设一个单片机工作于 12M 晶体震荡器，它的时钟周期是 1/12（微 秒）。它的一个机器周期是 12*（1/12）也就是 1 微秒。（请计算一个工作于 6M 晶体震荡器的单片机，它的机器周期是多少）。

MCS-51 单片机的所有指令中，有一些完成得比较快，只要一个机器周期就行了，有一些完成得比较慢，得要 2 个机器周期，还有两条指令要 4 个机器周期 才行。这也不难再解，

不是吗？我让你扫地的执行要完成总得比要你完成擦黑板的指令时间要长。为了恒量指令执行时间的长短，又引入一个新的概念：指令周期。 所谓指令周期就是指执行一条指令的时间。INTEL 对每一条指令都给出了它的指令周期数，这些数据，大部份不需要我们去记忆，但是有一些指令是需要记住 的，如 DJNZ 指令是双周期指令。

下面让我们来计算刚才的延时。首先必须要知道晶体震荡器的频率，我们设所用晶体震荡器为 12M，则一个机器周期就是 1 微秒。而 DJNZ 指令是双周期指令，所以执行一次要 2 个微秒。一共执行 62500 次，正好 125000 微秒，也就是 125 毫秒。

练习：设计一个延时 100 毫秒的延时程序。

要点分析：1、一个单元中的数是否能超过 255。2、如何分配两个数。

三、复位电路

一、复位方式

1. 复位条件

RST 引脚保持 2 个机器周期以上的高电平。

2. 复位电路

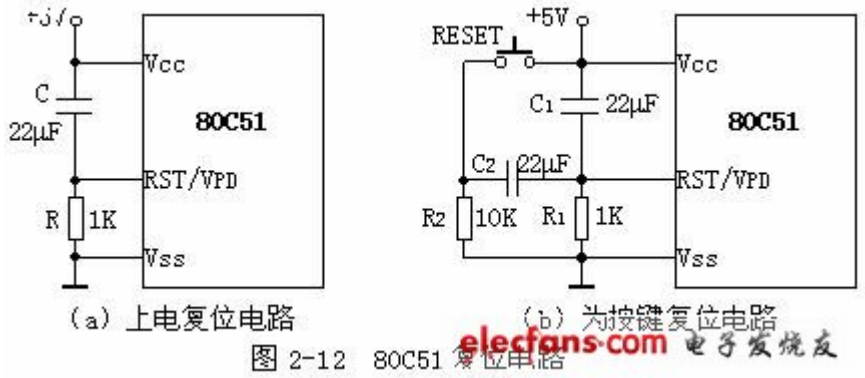


图 2-12 80C51 复位电路

〈单片机复位电路〉

3. 复位后 CPU 状态

PC： 0000H TMOD： 00H

Acc： 00H TCON： 00H

B： 00H TH0： 00H

PSW： 00H TL0： 00H

SP： 07H TH1： 00H

DPTR：0000H TL1： 00H

P0～P3：FFH SCON： 00H

IP：×××00000B SBUF： 不定

IE：0××00000B PCON： 0×××0000B

任何单片机在工作之前都要有个复位的过程，复位是什么意思呢？它就象是我们上课之前打的预备铃。预备铃一响，大家就自动地从操场、其它地方进入教室 了，在这一段时间里，是没有老师干预的，对单片机来说，是程序还没有开始执行，是在做准备工作。显然，准备工作不需要太长的时间，复位只需要 5ms 的时间 就能了。如何进行复位呢？只要在单片机的 RST 管脚上加上高电平，就能了，按上面所说，时间不少于 5ms。为了达到这个要求，能用很多种办法，这里供给一 种供参考，见图 1。实际上，我们在上一次实验的图中已见到过了。

这种复位电路的工作原理是：通电时，电容两端相当于是短路，于是 RST 管脚上为高电平，然后电源通过电阻对电容充电，RST 端电压慢慢下降，降到一定程序，即为低电平，单片机开始正常工作。

6：单片机并行口结构

上两次我们做过两个实验，都是让 P1.0 这个管脚使灯亮，我们能设想：既然 P1.0 能让灯亮，那么其它的管脚可不能呢？看一下图 1，它是 8031 单片 机管脚的说明，在 P1.0 旁边有 P1.1，P1.2….P1.7，它们是否都能让灯亮呢？除了以 P1 开头的外，还有以 P0，P2，P3 开头的，数一下，一 共是 32 个管脚，前面我们以学过 7 个管脚，加上这 32 个这 39 个了。它们都以 P 字开头，只是后面的数字不一样，它们是否有什么联系呢？它们能不能都让灯亮 呢？在我们的实验板上，除了 P10 之外，还有 P11 -》 P17 都与 LED 相连，下面让我们来做一个实验，程序如下：

```
MAIN:  MOV P1，#0FFH

LCALL DELAY

MOV P1，#00H

LCALL DELAY

LJMP MAIN
```

```
DELAY：MOV R7，#250

D1：  MOV R6，#250

D2：  DJNZ R6，D2

DJNZ R7，D1

RET

END
```

将这段程序转为机器码，用编程器写入单片机中，结果如何？通电以后我们能看到 8 只 LED 全部在闪动。因此，P10-》P17 是全部能点亮灯的。事实 上，凡以 P 开头的这 32 个管脚都是能点亮灯的，也就是说：这 32 个管脚都能作为输出使用，如果不用来点亮 LED，能用来控制继电器，能用来控制其它的执行 机构。

程序分析：这段程序和前面做过的程序比较，只有两处不一样：第一句：原来是 SETB P1.0，现在改为 MOV P1，#0FFH，第三句：原来是 CLR P1.0，现在改为 MOV P1.0，#00H。从中能看出，P1 是 P1.0-》P1.7 的全体的代表，一个 P1 就表示了所有的这八个管脚了。当然用的指令也不一样了，是用 MOV 指 令。为什么用这条指令？看图 2，我们把 P1 作为一个整体，就把它当作是一个存储器的单元，对一个单元送进一个数能用 MOV 指令。

二、第四个实验

除了能作为输出外，这 32 个管脚还能做什么呢？下面再来做一个单片机实验，源程序如下：

```
MAIN：  MOV P3，#0FFH

LOOP：  MOV A，P3
```

MOV P1，A

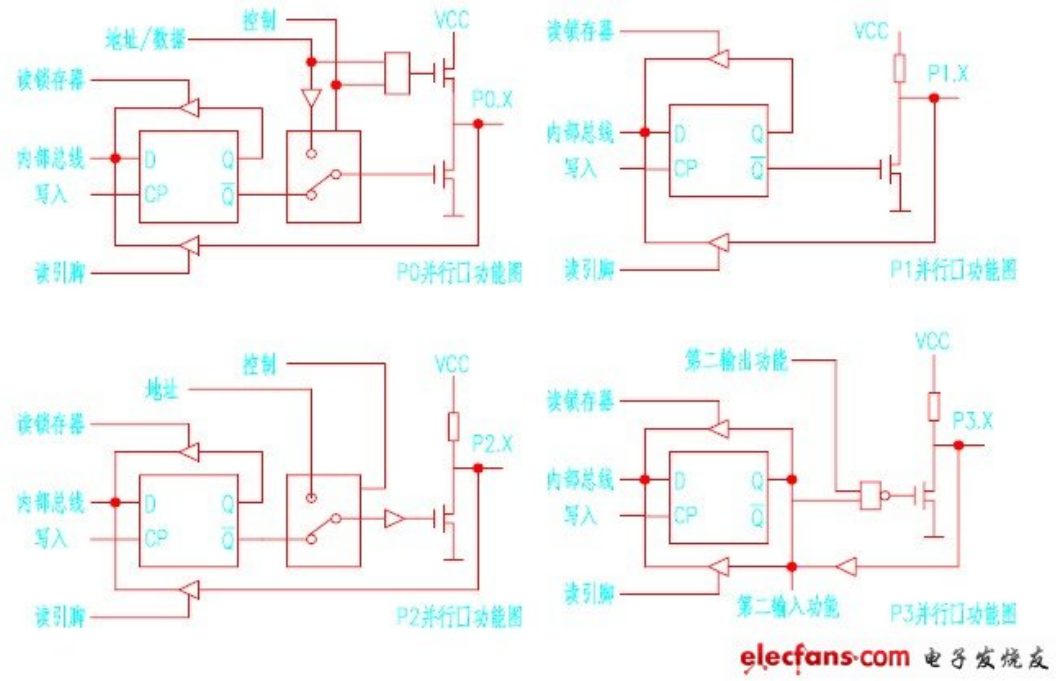
LJMP LOOP

先看一下这个实验的结果：所有灯全部不亮，然后我按下一个按钮，第（1）个灯亮了，再按下另一个按钮，第（2）个灯亮了，松开按钮灯就灭了。从这个实验现象结合电路来分析一下程序。

从硬件电路的连线能看出，有四个按钮被接入到 P3 口的 P32，P33，P34，P35。第一条指令的用途我们能猜到：使 P3 口全部为高电平。第二条指 令是 MOV A，P3，其中 MOV 已经知道，是送数的意思，这条指令的意思就是将 P3 口的数送到 A 中去，我们能把 A 当成是一个中间单元（看图 3），第三句话是将 A 中的数又送到 P1 口 去，第四句话是循环，就是持续地重复这个过程，这我们已见过。当我们按下第一个按钮时，第（3）只灯亮了，所以 P12 口应当输出是低电平，为什么 P12 口 会输出低电平呢？我们看一下有什么被送到了 P1 口，只有从 P3 口进来的数送到 A，又被送到了 P1 口，所以，肯定是 P3 口进来的数使得 P12 位输出电平的。 P3 口的 P32 位的按钮被按下，使得 P32 位的电平为低，通过程序，又使 P12 口输出低电平，所以 P3 口起来了一个输入的作用。验证：按第二、三、四个按 钮，同时按下 2 个、3 个、4 个按钮都能得到同样的结论，所以 P3 口确实起到了输入作用，这样，我们能看到，以 P 字开头的管脚，不仅能用作输出，还能用作输 入，其它的管脚是否能呢？是的，都能。这 32 个管脚就称之为并行口，下面我们就对并行口的结构作一个分析，看一下它是怎样实现输入和输出的。

并行口结构分析：

1、输出结构



《并行口结构图》

先看 P1 口的一位的结构示意图（只画出了输出部份）：从图中能看出，开关的打开和合上代表了管脚输出的高和低，如果开关合上了，则管脚输出就是低，如 果开关打开了，则输出高电平，这个开关是由一根线来控制的，这根数据总线是出自于 CPU，让我们回想一下，数据总线是一根大家公用的线，很多的器件和它连 在一起，在不一样的时候，不一样的器件当然需要不一样的信号，如某一时刻我们让这个管脚输出高电平，并要求保持若干时间，在这段时间里，计算机当然在忙个 不停，在与其它器件进行联络，这根控制线上的电平未必能保持原来的值不变，输出就会发生变化了。怎么解决这个问题呢？我们在存储器一节中学过，存储器中是 能存放电荷的，我们不妨也加一个小的存储器的单元，并在它的前面加一个开关，要让这一位输出时，就把开关打开，信号就进入存储器的单元，然后马上关闭开 关，这样这一位的状态就被保存下来，直到下一次命令让它把开关再打开为

止。这样就能使这一位的状态与别的器件无关了，这么一个小单元，我们给它一个很形象 的名字，称之为“锁存器”。

2、输入结构

这是并行口的一位的输出结构示意图，再看，除了输出之外，还有两根线，一根从外部管脚接入，另一根从锁存器的输出接出，分别标明读管脚和读锁存器。这 两根线是用于从外部接收信号的，为什么要两根呢？原来，在 51 单片机中输入有两种方式，分别称为‘读管脚’和‘读锁存器’，第一种方式是将管脚作为输入， 那是真正地从外部管脚读进输入的值，第二种方式是该管脚处于输出状态时，有时需要改变这一位的状态，则并不需要真正地读管脚状态，而只是读入锁存器的状 态，然后作某种变换后再输出。

请注意输入结构图，如果将这一根引线作为输入口使用，我们并不能保证在任何时刻都能得到正确的结果（为什么？）参考图 2 输入示意图。接在外部的开关如 果打开，则应当是输入 1，而如果闭合开关，则输入 0，但是，如果单片机内部的开关是闭合的，那么不管外部的开关是开还是闭，单片机接受到的数据都是 0。可 见，要让这一端口作为输入使用，要先做一个‘准备工作’，就是先让内部的开关断开，也就是让端口输出‘1’才行。正因为要先做这么一个准备工作，所以我们 称之为“准双向 I/O 口”。

以上是 P1 口的一位的结构，P1 口其它各位的结构与之相同，而其它三个口：P0、P2、P3 则除入作为输入输出口之外还有其它用途，所以结构要稍复杂 一些，但其用于输入、输出的结构是相同的。看图（）。对我们来说，这些附加的功能不必由我们来控制，所以我们就不去关心它了。

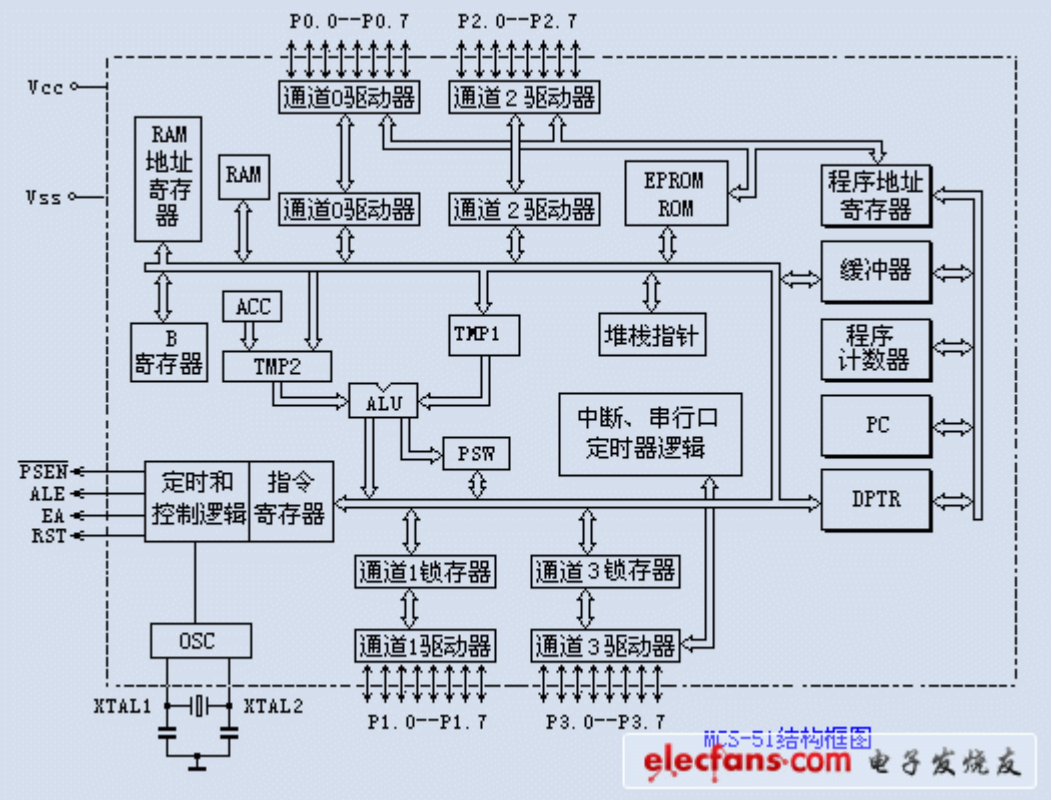
7：单片机的特殊功能寄存器

通过前面的学习，我们已知单片机的内部有 ROM、有 RAM、有并行 I/O 口，那么，除了这些东西之外，单片机内部究竟还有些什么，这些个零碎的东西怎么连在一起的，让我们来对单片机内部的寄存器作一个完整的功能分析吧！

下图中我们能看出，在 51 单片机内部有一个 CPU 用来运算、控制，有四个并行 I/O 口，分别是 P0、P1、P2、P3，有 ROM，用来存放程序，有 RAM，用来存放中间结果，此外还有

定时/计数器，串行 I/O 口，中断系统，以及一个内部的时钟电路。在一个 51 单片机的内部包含了这么多的东西。

《单片机内部结构图》



对上面的图进行进一步的分析，我们已知，对并行 I/O 口的读写只要将数据送入到对应 I/O 口的锁存器就能了，那么对于定时/计数器，串行 I/O 口等怎 么用呢？在单片机中有一些独立的存储单元是用来控制这些器件的，被称之为特殊功能寄存器(SFR)。事实上，我们已接触过 P1 这个特殊功能寄存器了，还有 哪些呢？

看下表 1

符号	地址	功能介绍
B	F0H	B寄存器
ACC	E0H	累加器
PSW	D0H	程序状态字
IP	B8H	中断优先级控制寄存器
P3	B0H	P3口锁存器
IE	A8H	中断允许控制寄存器
P2	A0H	P2口锁存器
SBUF	99H	串行口锁存器
SCON	98H	串行口控制寄存器
P1	90H	P1口锁存器
TH1	8DH	定时器/计数器1（高8位）
TH0	8CH	定时器/计数器1（低8位）

表 1

《特殊功能寄存器地址映象表（一）》

SFR 名称	符号	位地址/位定义名/位编号								字节地址
		D7	D6	D5	D4	D3	D2	D1	D0	
B 寄存器	B	F7H	F6H	F5H	F4H	F3H	F2H	F1H	F0H	(F0H)
累加器 A	Acc	E7H	E6H	E5H	E4H	E3H	E2H	E1H	E0H	(E0H)
		Acc.7	Acc.6	Acc.5	Acc.4	Acc.3	Acc.2	Acc.1	Acc.0	
程序状态字寄存器	PSW	D7H	D6H	D5H	D4H	D3H	D2H	D1H	D0H	(D0H)
		Cy	AC	F0	RS1	RS0	OV	F1	P	
		PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0	
中断优先级控制寄存器	IP	BFH	BEH	BDH	BCH	BBH	BAH	B9H	B8H	(B8H)
					PS	PT1	PX1	PT0	PX0	
I/O 端口 3	P3	B7H	B6H	B5H	B4H	B3H	B2H	B1H	B0H	(B0H)
		P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0	
中断允许控制寄存器	IE	AFH	AEH	ADH	ACH	ABH	AAH	A9H	A8H	(A8H)
		EA			ES	ET1	EX1	ET0	EX0	
I/O 端口 2	P2	A7H	A6H	A5H	A4H	A3H	A2H	A1H	A0H	(A0H)
		P2.7	P2.6	P2.5	P2.4	P2.3	P2.2	P2.1	P2.0	

《特殊功能寄存器地址映象表（二）》

串行数据缓冲器	SBUF								99H
串行控制寄存器	SCON	9FH	9EH	9DH	9CH	9BH	9AH	99H	98H	(98H)
		SM0	SM1	SM2	REN	TB8	RB8	TI	RI	
I/O 端口 1	P1	97H	96H	95H	94H	93H	92H	91H	90H	(90H)
		P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0	
定时/计数器 1（高字节）	TH1								8DH
定时/计数器 0（高字节）	TH0								8CH
定时/计数器 1（低字节）	TL1								8BH
定时/计数器 0（低字节）	TL0								8AH
定时/计数器方式选择	TMOD	GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0	89H
定时/计数器控制寄存器	TCON	8FH	8EH	8DH	8CH	8BH	8AH	89H	88H	(88H)
		TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	
电源控制及波特率选择	PCON	SMOD				G71				87H

《特殊功能寄存器地址映象表（三）》

数据指针（高字节）	DPH								83H
数据指针（低字节）	DPL								82H
堆栈指针	SP								81H
I/O 端口 0	P0	87H	86H	85H	84H	83H	82H	81H	80H	(80H)
		P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0	

下面，我们介绍一下几个常用的 SFR，看图 2。

ACC：累加器，常常用 A 表示。这是个什么东西，可不能从名字上理解，它是一个寄存器，而不是一个做加法的东西，为什么给它这么一个名字呢？或许是因 为在运算器做运算

时其中一个数一定是在 ACC 中的缘故吧。它的名字特殊，身份也特殊，稍后我们将学到指令，能发现，所有的运算类指令都离不开它。

2、B：一个寄存器。在做乘、除法时放乘数或除数，不做乘除法时，随你怎么用。

3、PSW：程序状态字。这是一个很重要的东西，里面放了 CPU 工作时的很多状态，借此，我们能了解 CPU 的当前状态，并作出对应的处理。它的各位功能请看表 2

D7	D6	D5	D4	D3	D2	D1	D0
CY	AC	F0	RS1	RS0	OV		P

表 2

PSW 也称为标志寄存器，了解这个对于了解单片机原理非常的重要，存放各有关标志。其结构和定义如下：

位编号	PSW.7	PSW.6	PSW.5	PSW.4	PSW.3	PSW.2	PSW.1	PSW.0
位地址	D7H	D6H	D5H	D4H	D3H	D2H	D1H	D0H
位定义名	Cy	AC	F0	RS1	RS0			

下面我们逐一介绍 sfr 各位的用途

（1）CY：进位标志。用于表示 Acc.7 有否向更高位进位。8051 中的运算器是一种 8 位的运算器，我们知道，8 位运算器只能表示到 0-255，如 果做加法的话，两数相加可能会超过 255，这样最高位就会丢失，造成运算的错误，怎么办？最高位就进到这里来。这样就没事了。

例：78H+97H（01111000+10010111）

（2）AC：辅助进位标志也叫半进位标志。 用于表示 Acc.3 有否向 Acc.4 进位

例：57H+3AH（01010111+00111010）

（3）F0：用户标志位，由我们（编程人员）决定什么时候用，什么时候不用。

（4）RS1、RS0：工作寄存器组选择位。这个我们已知了。

RS1、RS0 = 00 —— 0 区（00H～07H）

RS1、RS0 = 01 —— 1 区（08H～0FH）

RS1、RS0 = 10 —— 2 区（10H～17H）

RS1、RS0 = 11 —— 3 区（18H～1FH）

（5）OV：溢出标志位。 表示 Acc 在有符号数算术运算中的溢出，什么是溢出我们稍后再谈吧。

（6）P：奇偶校验位：它用来表示 ALU 运算结果中二进制数位“1”的个数的奇偶性。若为奇数，则 P=1，不然为 0。

例：某运算结果是 78H（01111000），显然 1 的个数为偶数，所以 P=0。

4、DPTR（DPH、DPL）：数据指针，能用它来访问外部数据存储器中的任一单元，如果不用，也能作为通用寄存器来用，由我们自己决定如何使用。 16 位，由两个 8 位寄存器 DPH、DPL 组成。主要用于存放一个 16 位地址，作为访问外部存储器（外 RAM 和 ROM）的地址指针。

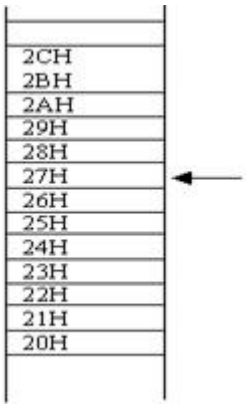
5、P0、P1、P2、P3：这个我们已经知道，是四个并行输入/输出口的寄存器。它里面的内容对应着管脚的输出。

6、SP：堆栈指针。（专用于指出堆栈顶部数据的地址。）

堆栈介绍：日常生活中，我们都注意到过这样的现象，家里洗的碗，一只一只摞起来，最晚放上去的放在最上面，而最早放上去的则放在最下面，在取的时候正 好相反，先从最上面取，这种现象我们用一句话来概括：“先进后出，后进先出”。请大家想想，还有什么地方有这种现象？其实比比皆是，建筑工地上堆放的砖 头、材料，仓库里放的货物，都是“先进后出，后进先出”，这实际是一种存取物品的规则，我们称之为“堆栈”。

在单片机中，我们也能在 RAM 中构造这样一个区域，用来存放数据，这个区域存放数据的规则就是“先进后出，后进先出”，我们称之为“堆栈”。为什么需 要这样来存放数据呢？存储器本身不是能按地址来存放数据吗？对，知道了地址的确就能知道里面的内容，但如果我们需要存放的是一批数据，每一个数据都需要知 道地址那不是麻烦吗？如果我们让数据一个接一个地放置，那么我们只要知道第一个数据所在地址单元就能了（看图 2）如果第一个数据在 27H，那么第二、三个 就在 28H、29H 了。所以利用堆栈这种办法来放数据能简化操作

那么 51 中堆栈什么地方呢？单片机中能存放数据的区域有限，我们不能够专门分配一块地方做堆栈，所以就在内存（RAM）中开辟一块地方，用于堆栈，但 是用内存的哪一块呢？还是不好定，因为 51 是一种通用的单片机，各人的实际需求各不相同，有人需要多一些堆栈，而有人则不需要那么多，所以怎么分配都不合 适，怎样来解决这个问题？分不好干脆就不分了，把分的权利给用户（编程者），根据自己的需要去定吧，所以 51 单片机中堆栈的位置是能变化的。而这种变化就 体现在 SP 中值的变化，看图 2，SP 中的值等于 27H 不就相当于是一个指针指向 27H 单元吗？当然在真正的 51 机中，开始指针所指的位置并非就是数据存放 的位置，而是数据存放的前一个位置，比如一开始指针是指向 27H 单元的，那么第一个数据的位置是 28H 单元，而不是 27H 单元，为什么会这样，我们在学堆 栈命令时再说明。其它的 SFR，我们在用到时再介绍。



单片机学习知识点全攻略（二）

本单片机系列关键知识点一览：

系列二

- 8：单片机寻址方式与指令系统

- 9：单片机数据传递类指令
- 10：单片机数据传送类指令
- 11：单片机算术运算指令
- 12：单片机逻辑运算类指令
- 13：单片机逻辑与或异或指令祥解
- 14：单片机条件转移指令

8、单片机寻址方式与指令系统

通过前面的学习，我们已经了解了单片机内部的结构，并且也已经知道，要控制单片机，让它为我们干学，要用指令，我们已学了几条指令，但很零散，从现在开始，我们将要系统地学习 8051 单片机的指令部份。

一、概述

1、指令的格式

我们已知，要让计算机做事，就得给计算机以指令，并且我们已知，计算机很“笨”，只能懂得数字，如前面我们写进机器的 75H，90H，00H 等等，所 以指令的第一种格式就是机器码格式，也说是数字的形式。但这种形式实在是为难我们人了，太难记了，于是有另一种格式，助记符格式，如 MOV P1，#0FFH，这样就好记了。 这两种格式之间的关系呢，我们不难理解，本质上它们完全等价，只是形式不一样而已。

2、汇编

我们写指令使用汇编格式，而计算机和单片机只懂机器码格式，所以要将我们写的汇编格式的指令转换为机器码格式，这种转换有两种办法：手工汇编和机器汇 编。手工汇编

实际上就是查表，因为这两种格式纯粹是格式不一样，所以是一一对应的，查一张表格就行了。不过手工查表总是嫌麻烦，所以就有了计算机软件，用 计算机软件来替代手工查表，这就是机器汇编。

二、单片机的寻址

让我们先来复习一下我们学过的一些指令：MOV P1，#0FFH，MOV R7，#0FFH 这些指令都是将一些数据送到对应的位置中去，为什么要送数据呢？第一个因为送入的数能让灯全灭掉，第二个是为了要实现延时，从这里我们能 看出来，在用单片机的编程语言编程时，经常要用到数据的传递，事实上数据传递是单片机编程时的一项重要工作，一共有 28 条指令（单片机共 111 条指令）。 下面我们就从数据传递类指令开始吧。

分析一下 MOV P1，#0FFH 这条指令，我们不难得出结论，第一个词 MOV 是命令动词，也就是决定做什么事情的，MOV 是 MOVE 少写了一个 E，所以就是“传递”，这 就是指令，规定做什么事情，后面还有一些参数，分析一下，数据传递必须要有一个“源”也就是你要送什么数，必须要有一个“目的”，也就是你这个数要送到什 么地方去，显然在上面那条单片机指令中，要送的数（源）就是 0FFH，而要送达的地方（目的地）就是 P1 这个寄存器。在数据传递类指令中，均将目的地写在 指令的后面，而将源写在最后。

这条指令中，送给 P1 是这个数本身，换言之，做完这条指令后，我们能明确地知道，P1 中的值是 0FFH，但是并不是任何时候都能直接给出数本身的。例如，在我们前面给出的单片机延时程序例是这样写的：

```
MAIN:  SETB P1.0          ;（1）
LCALL DELAY ;（2）
CLR P1.0            ;（3）
LCALL DELAY      ;（4）
AJMP MAIN        ;（5）
```

```
；以下子程序
DELAY：  MOV R7，#250      ；（6）
D1：  MOV R6，#250      ；（7）
D2：  DJNZ R6，D2        ；（8）
DJNZ R7，D1       ；（9）
RET                ；（10）
END                ；（11）
```

表 1

```
-------------------------------------------------
MAIN：  SETB P1.0          ；（1）
MOV 30H，#255
LCALL DELAY ；
CLR P1.0           ；（3）
MOV 30H，#200
LCALL DELAY      ；（4）
AJMP MAIN        ；（5）
```

```
；以下子程序
DELAY：  MOV R7，30H      ；（6）
D1：  MOV R6，#250      ；（7）
D2：  DJNZ R6，D2        ；（8）
DJNZ R7，D1       ；（9）
RET                ；（10）
END                ；（11）
```

这样一来，我每次调用延时程序延时的时间都是相同的（大致都是 0.13S），如果我提出这样的要求：灯亮后延时时间为 0.13S 灯灭，灯灭后延时 0.1 秒灯亮，如此循环，这样的程序还能满足要求吗？不能，怎么办？我们能把延时程序改成这样（见表 2）：调用则见表 2 中的主程，也就是先把一个数送入 30H，在子程序中 R7 中的值并不固定，而是根据 30H 单元中传过来的数确定。这样就能满足要求。

从这里我们能得出结论，在数据传递中要找到被传递的数，很多时候，这个数并不能直接给出，需要变化，这就引出了一个概念：如何寻找操作数，我们把寻找 操作数所在单元的地址称之为寻址。在这里我们直接使用数所在单元的地址找到了操作数，所以称这种办法为直接寻址。除了这种办法之外，还有一种，如果我们把 数放在工作寄存器中，从工作寄存器中寻找数据，则称之为寄存器寻址。例：MOV A，R0 就是将 R0 工作寄存器中的数据送到累加器 A 中去。提一个问题：我们知道，工作寄存器就是内存单元的一部份，如果我们选择工作寄存器组 0，则 R0 就 是 RAM 的 00H 单元，那么这样一来，MOV A，00H，和 MOV A，R0 不就没什么区别了吗？为什么要加以区别呢？的确，这两条指令执行的结果是完全相同的，都是将 00H 单元中的内容送到 A 中去，但是执行的过程不一 样，执行第一条指令需要 2 个周期，而第二条则只需要 1 个周期，第一条指令变成最终的目标码要两个字节（E5H 00H），而第二条则只要一个字节（E8h）就能了。

这么斤斤计较！不就差了一个周期吗，如果是 12M 的晶体震荡器的话，也就 1 个微秒时间了，一个字节又能有多少？

不对，如果这条指令只执行一次，也许无所谓，但一条指令如果执行上 1000 次，就是 1 毫秒，如果要执行 1000000 万次，就是 1S 的误差，这就很可观了，单片机做的是实时控制的事，所以必须如此“斤斤计较”。字节数同样如此。

再来提一个问题，现在我们已知，寻找操作数能通过直接给的方式（立即寻址）和直接给出数所在单元地址的方式（直接寻址），这就够了吗？

看这个问题，要求从 30H 单元开始，取 20 个数，分别送入 A 累加器。

就我们目前掌握的办法而言，要从 30H 单元取数，就用 MOV A，30H，那么下一个数呢？是 31H 单元的，怎么取呢？还是只能用 MOV A，31H，那么 20 个数，不是得 20 条指令才能写完吗？这里只有 20 个数，如果要送 200 个或 2000 个数，那岂不要写上 200 条或 2000 条命令？这 未免太笨了吧。为什么会出现这样的状况？是因为我们只会把地址写在指令中，所以就没办法了，如果我们不是把地址直接写在指令中，而是把地址放在另外一个寄 存器单元中，根据这个寄存器单元中的数值决定该到哪个单元中取数据，比如，当前这个寄存器中的值是 30H，那么就到 30H 单元中去取，如果是 31H 就到 31H 单元中去取，就能解决这个问题了。怎么个解决法呢？既然是看的寄存器中的值，那么我们就能通过一定的办法让这里面的值发生变化，比如取完一个数后， 将这个寄存器单元中的值加 1，还是执行同一条指令，可是取数的对象却不一样了，不是吗。通过例程来说明吧。

```
MOV R7，#20

MOV R0，#30H

LOOP：MOV A，@R0

INC R0

DJNZ R7，LOOP
```

这个例程中大部份指令我们是能看懂的，第一句，是将立即数 20 送到 R7 中，执行完后 R7 中的值应当是 20。第二句是将立即数 30H 送入 R0 工作寄存器 中，所以执行完后，R0 单元中的值是 30H，第三句，这是看一下 R0 单元中是什么值，把这个值作为地址，取这个地址单元的内容送入 A 中，此时，执行这条指 令的结果就相当于 MOV A，30H。第四句，没学过，就是把 R0 中的值加 1，因此执行完后，R0 中的值就是 31H，第五句，学过，将 R7 中的值减 1，看是否等于 0，不等于 0，则 转到标号 LOOP 处继续执行，因此，执行完这句后，将转去执行 MOV A，@R0 这句话，此时相当于执行了 MOV A，31H（因为此时的 R0 中的值已是 31H 了），如此，直到 R7 中的值逐次相减等于 0，也就是循环 20 次为止，就实现了我们的要求：从 30H 单元开始将 20 个数据送入 A 中。

这也是一种寻找数据的办法，由于数据是间接地被找到的，所以就称之为间址寻址。注意，在间址寻址中，只能用 R0 或 R1 存放等寻找的数据。

9、单片机数据传递类指令

单片机数据传递类指令

（3）以直接地址为目的操作数的指令

MOV direct，A 例： MOV 20H，A

MOV direct，Rn MOV 20H，R1

MOV direct1，direct2 MOV 20H，30H

MOV direct，@Ri MOV 20H，@R1

MOV direct，#data MOV 20H，#34H

（4）以间接地址为目的操作数的指令

MOV @Ri，A 例：MOV @R0，A

MOV @Ri，direct MOV @R1，20H

MOV @Ri，#data MOV @R0，#34H

（5）十六位数的传递指令

MOV DPTR，#data16

8051 是一种 8 位机，这是唯一的一条 16 位立即数传递指令，其功能是将一个 16 位的立即数送入 DPTR 中去。其中高 8 位送入 DPH，低 8 位送入 DPL。例：MOV DPTR，#1234H，则执行完了之后 DPH 中的值为 12H，DPL 中的值为 34H。反之，如果我们分别向 DPH，DPL 送数，则结果也一样。如有下面 两条指令：MOV DPH，#35H，MOV DPL，#12H。则就相当于执行了 MOV DPTR，#3512H。

数据传递类指令综合练习：

给出每条指令执行后的结果

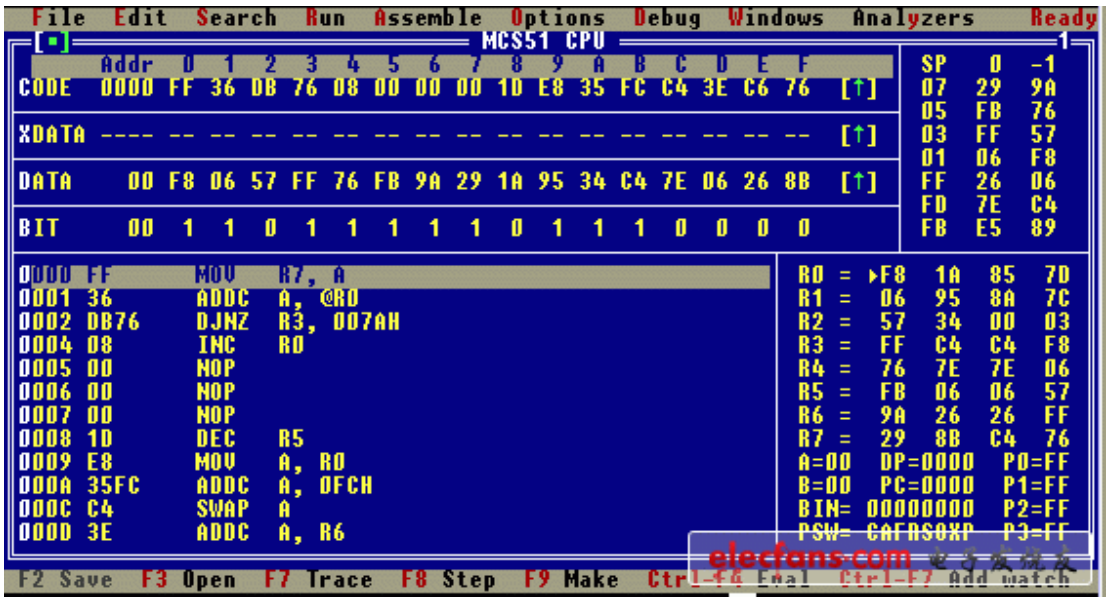
上机练习：

MOV 23H,#30H	(23h)=30h		
MOV 12H,#34H	(12h)=34h	MOV 45H,34H	(45H)=34H
MOV R0,#23H	(R0)=23H	MOV DPTR,#6712H	(DPR)=6712
MOV R7,#22H	(R7)=22H	MOV 12H,DPH	(12H)=67
MOV R1,12H	(R1)=12H	MOV R0,DPL	(R0)=12H
MOV A,@R0	(A)=30H	MOV A,@R0	(A)=67H
MOV 34H,@R1	(34H)=34H		

说明：用括号括起来代表内容，如（23H）则代表内部 RAM23H 单元中的值，（A）则代表累加器 A 单元中的值。

进入 DOS 状态，进入 WAVE 所在的目录，例 D：\WAVE

键入 MCS51，出现如下画面



《单片机数据传递指令》图 1

按 File-》Open，出现对话框后，在 Name 处输入一个文件名（见图 2），如果是下面列表中已存在的，则打开这个文件，如果不存在这个文件，则新建一个文件（见图 3）

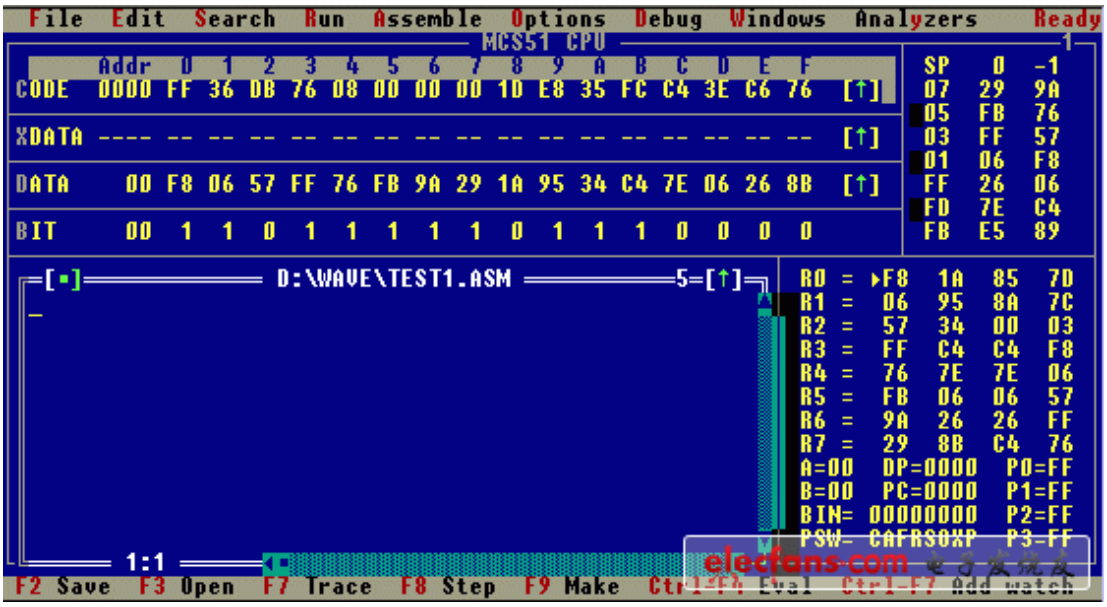
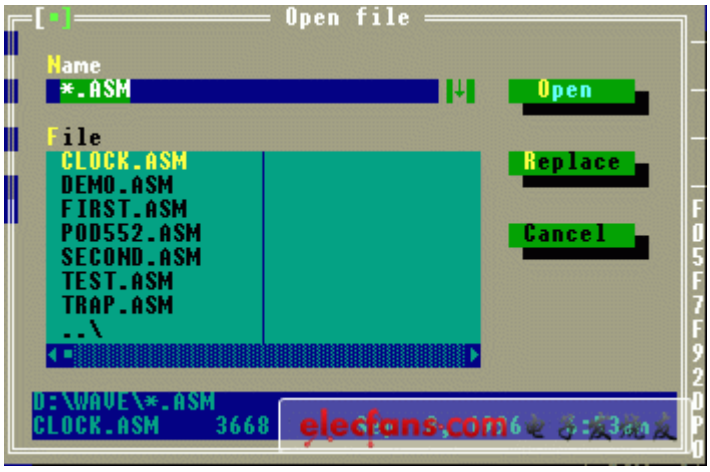


图 2



在空白处将上面的程序输入。见图 4。用 ALT+A 汇编通过。用 F8 即可单步执行，在执行过程中注意观察屏幕左边的工作寄存器及 A 累加器中的值的变化。

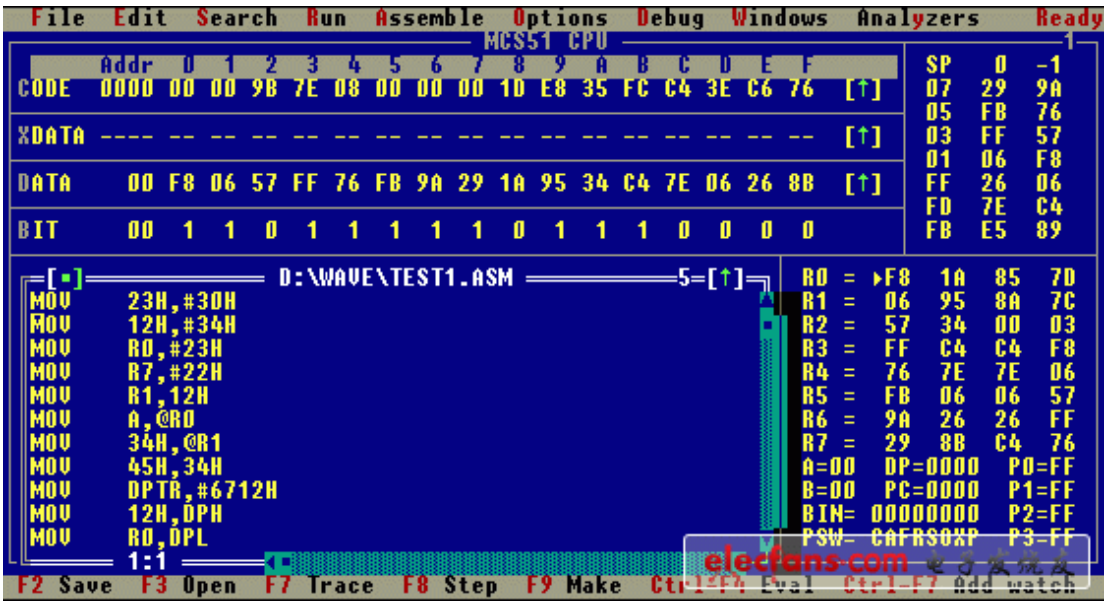


图 4

内存中值的变化在此是看不到的，可以用如下方法观察（看图 5）：将鼠标移到 DATA，双击，则光标进入此行，此时可以键盘上的上下光标键上下翻动来观 察内存值的变化。本行的最前面 DATA 后面的数据代表的是“一段”的开始地址，如现在为 20H，再看屏幕的最上方，数字从 0 到 F，显示两者相加就等于真正 的地址值，如现在图上所示的内存 20H、21H、22H、23H 中的值分别是 FBH 、0EH、E8H、30H。

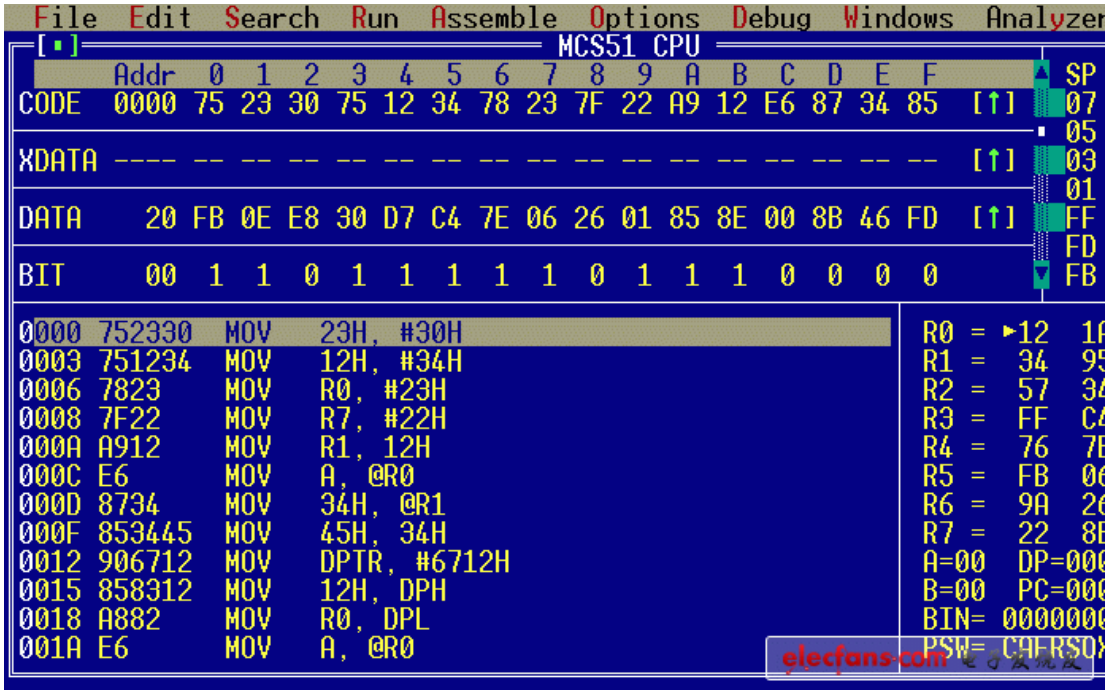


图 5

6、当运行完程序后，即进入它的反汇编区，不是我们想要的东西。为了再从头开始，可以用 CTRL+F2 功能键复位 PC 值。注意此时不会看到原来的窗 口，为看到原来的窗口，请用 ALT+4 或 ALT+5 等来切换。当然以上操作也可以菜单进行。CTRL+F2 是程序复位，用 RUN 菜单。窗口用 WINDOWS 菜单。

此次大家就用用熟这个软件吧，说实话，我并不很喜欢它，操作起来不方便，但给我的机器只能上这个,没办法,下次再给网友单独介绍一个好一点的吧。现在最好的是keil 。

10、单片机数据传送类指令

单片机的累加器 A 与片外 RAM 之间的数据传递类指令

MOVX A，@Ri

MOVX @Ri，A

MOVX A，@DPTR

MOVX @DPTR，A

说明：

1）在 51 系列单片机中，与外部存储器 RAM 打交道的只能是 A 累加器。所有需要传送入外部 RAM 的数据必需要通过 A 送去，而所有要读入的外部 RAM 中 的数据也必需通过 A 读入。在此我们能看出内外部 RAM 的区别了，内部 RAM 间能直接进行数据的传递，而外部则不行，比如，要将外部 RAM 中某一单元（设为 0100H 单元的数据）送入另一个单元（设为 0200H 单元），也必须先将 0100H 单元中的内容读入 A，然后再传送到 0200H 单元中去。

要读或写外部的 RAM，当然也必须要知道 RAM 的地址，在后两条单片机指令中，地址是被直接放在 DPTR 中的。而前两条指令，由于 Ri（即 R0 或 R1）只是一个 8 位的寄存器，所以只供给低 8 位地址。因为有时扩展的外部 RAM 的数量比较少，少于或等于 256 个，就只需要供给 8 位地址就够了。

使用时应当首先将要读或写的地址送入 DPTR 或 Ri 中，然后再用读写命令。

例：将单片机外部 RAM 中 100H 单元中的内容送入外部 RAM 中 200H 单元中。

MOV DPTR，#0100H

MOVX A，@DPTR

MOV DPTR，#0200H

MOVX @DPTR，A

程序存储器向累加器 A 传送指令

MOVC A，@A+DPTR 本指令是将 ROM 中的数送入 A 中。本指令也被称为单片机查表指令，常用此指令来查一个已做好在 ROM 中的表格 说明：

此条指令引出一个新的寻址办法：变址寻址。本指令是要在 ROM 的一个地址单元中找出数据，显然必须知道这个单元的地址，这个单元的地址是这样确定的： 在执行本指令立脚点 DPTR 中有一个数，A 中有一个数，执行指令时，将 A 和 DPTR 中的数加起为，就成为要查找的单元的地址。

查找到的结果被放在 A 中，因此，本条指令执行前后，A 中的值不一定相同。

例：有一个数在 R0 中，要求用查表的办法确定它的平方值（此数的取值范围是 0-5）

```
MOV DPTR，#TABLE

MOV A，R0

MOVC A，@A+DPTR

TABLE： DB 0，1，4，9，16，25
```

设 R0 中的值为 2，送入 A 中，而 DPTR 中的值则为 TABLE，则最终确定的 ROM 单元的地址就是 TABLE+2，也就是到这个单元中去取数，取到的是 4，显然它正是 2 的平方。其它数据也能类推。

标号的真实含义：从这个地方也能看到另一个问题，我们使用了标号来替代具体的单元地址。事实上，标号的真实含义就是地址数值。在这里它代表 了，0，1，4，9，16，25 这几个数据在 ROM 中存放的起点位置。而在以前我们学过的如 LCALL DELAY 单片机指令中，DELAY 则代表了以 DELAY 为标号的那段程序在 ROM 中存放的起始地址。事实上，CPU 正是通过这个地址才找到这段程序的。

能通过以下的例程再来看一看标号的含义：

```
MOV DPTR，#100H

MOV A，R0

MOVC A，@A+DPTR

ORG 0100H.

DB 0，1，4，9，16，25
```

如果 R0 中的值为 2，则最终地址为 100H+2 为 102H，到 102H 单元中找到的是 4。这个能看懂了吧？

那为什么不这样写程序，要用标号呢？不是增加疑惑吗？

如果这样写程序的话，在写程序时，我们就必须确定这张表格在 ROM 中的具体的位置，如果写完程序后，又想在这段程序前插入一段程序，那么这张表格的位 置就又要变了，要改 ORG 100H 这句话了，我们是经常需要修改程序的，那多麻烦，所以就用标号来替代，只要一编译程序，位置就自动发生变化，我们把这个麻烦事交给计算 机&#0;&#0;指我们用的电脑去做了。

堆栈操作

```
PUSH direct

POP direct
```

第一条指令称之为推入，就是将 direct 中的内容送入堆栈中，第二条指令称之为弹出，就是将堆栈中的内容送回到 direct 中。推入指令的执行过程 是，首先将 SP 中的值加 1，然后把 SP 中的值当作地址，将 direct 中的值送进以 SP 中的值为地址的 RAM 单元中。例：

```
MOV SP，#5FH

MOV A，#100

MOV B，#20

PUSH ACC

PUSH B
```

则执行第一条 PUSH ACC 指令是这样的：将 SP 中的值加 1，即变为 60H，然后将 A 中的值送到 60H 单元中，因此执行完本条指令后， 内存 60H 单元的值就是 100，同样，执行 PUSH B 时，是将 SP+1，即变为 61H，然后将 B 中的值送入到 61H 单元中，即执行完本条指令后，61H 单元中的值变为 20。

POP 指令的在单片机中执行是这样的，首先将 SP 中的值作为地址，并将此地址中的数送到 POP 指令后面的那个 direct 中，然后 SP 减 1。

接上例：

```
POP B

POP ACC
```

则执行过程是：将 SP 中的值（现在是 61H）作为地址，取 61H 单元中的数值（现在是 20），送到 B 中，所以执行完本条指令后 B 中的值是 20，然后将 SP 减 1，因此本条指令执行完后，SP 的值变为 60H，然后执行 POP ACC，将 SP 中的值（60H）作为地址，从该地址中取数（现在是 100），并送到 ACC 中，所以执行完本条指令后，ACC 中的值是 100。

这有什么意义呢？ACC 中的值本来就是 100，B 中的值本来就是 20，是的，在本例中，的确没有意义，但在实际工作中，则在 PUSH B 后一般要执行其他指令，而且这些指令会把

A 中的值，B 中的值改掉，所以在程序的结束，如果我们要把 A 和 B 中的值恢复原值，那么这些指令就有意义了。

还有一个问题，如果我不用堆栈，比如说在 PUSH ACC 指令处用 MOV 60H，A，在 PUSH B 处用指令 MOV 61H，B，然后用 MOV A，60H，MOV B，61H 来替代两条 POP 指令，不是也一样吗？是的，从结果上看是一样的，但是从过程看是不一样的，PUSH 和 POP 指令都是单字节，单周期指令，而 MOV 指令则是双字节，双周期指令。更何况，堆栈的作用不止于此，所以一般的计算机上都设有堆栈，单片机也是一样，而我们在编写子程序，需要保存数据时， 常常也不采用后面的办法，而是用堆栈的办法来实现。

例：写出以下单片机程序的运行结果

```
MOV 30H，#12

MOV 31H，#23

PUSH 30H

PUSH 31H

POP 30H

POP 31H
```

结果是 30H 中的值变为 23，而 31H 中的值则变为 12。也就两者进行了数据交换。从这个例程能看出：使用堆栈时，入栈的书写次序和出栈的书写次序必须相反，才能保证数据被送回原位，不然就要出错了。

作业：在 MCS51 下执行上面的例程，注意观察内存窗口和堆栈窗口的变化。

11、单片机算术运算指令

不带进位位的单片机加法指令

ADD A，#DATA ;例：ADD A，#10H

ADD A，direct ;例：ADD A，10H

ADD A，Rn ;例：ADD A，R7

ADD A，@Ri ;例：ADD A，@R0

用途：将 A 中的值与其后面的值相加，最终结果否是回到 A 中。

例：MOV A，#30H

ADD A，#10H

则执行完本条指令后，A 中的值为 40H。

下面的题目自行练习

MOV 34H，#10H

MOV R0，#13H

MOV A，34H

ADD A，R0

MOV R1，#34H

ADD A，@R1

带进位位的加法指令

ADDC A，Rn

ADDC A，direct

ADDC A，@Ri

ADDC A，#data

用途：将 A 中的值和其后面的值相加，并且加上进位位 C 中的值。

说明：由于 51 单片机是一种 8 位机，所以只能做 8 位的数学运算，但 8 位运算的范围只有 0-255，这在实际工作中是不够的，因此就要进行扩展，一般是 将 2 个 8 位的数学运算合起来，成为一个 16 位的运算，这样，能表达的数的范围就能达到 0-65535。如何合并呢？其实很简单，让我们看一个 10 进制数的 例程：

66+78。

这两个数相加，我们根本不在意这的过程，但事实上我们是这样做的：先做 6+8（低位），然后再做 6+7，这是高位。做了两次加法，只是我们做的时候并 没有刻意分成两次加法来做罢了，或者说我们并没有意识到我们做了两次加法。之所以要分成两次来做，是因为这两个数超过了一位数所能表达的范置（0-9）。

在做低位时产生了进位，我们做的时候是在适当的位置点一下，然后在做高位加法是将这一点加进去。那么计算机中做 16 位加法时同样如此，先做低 8 位的， 如果两数相加产生了进位，也要“点一下”做个标记，这个标记就是进位位 C，在 PSW 中。在进行高位加法是将这个 C 加进去。例：1067H+10A0H，先 做 67H+A0H=107H，而 107H 显然超过了 0FFH，因此最终保存在 A 中的是 7，而 1 则到了 PSW 中的 CY 位了，换言之，CY 就相当于是 100H。然后再做 10H+10H+CY，结果是 21H，所以最终的结果是 2107H。

带借位的单片机减法指令

SUBB A，Rn

SUBB A，direct

SUBB A，@Ri

SUBB A，#data

设（每个 H，（R2）=55H，CY=1，执行指令 SUBB A，R2 之后，A 中的值为 73H。

说明：没有不带借位的单片机减法指令，如果需要做不带位的减法指令（在做第一次相减时），只要将 CY 清零即可。

乘法指令

MUL AB

此单片机指令的功能是将 A 和 B 中的两个 8 位无符号数相乘，两数相乘结果一般比较大，因此最终结果用 1 个 16 位数来表达，其中高 8 位放在 B 中，低 8 位放在 A 中。在乘积大于 FFFFFH（65535）时，OV 置 1（溢出），不然 OV 为 0，而 CY 总是 0。

例：（A）=4EH，（B）=5DH，执行指令

MUL AB 后，乘积是 1C56H，所以在 B 中放的是 1CH，而 A 中放的则是 56H。

除法指令

DIV AB

此单片机指令的功能是将 A 中的 8 位无符号数除了 B 中的 8 位无符号数（A/B）。除法一般会出现小数，但计算机中可没法直接表达小数，它用的是我们小学 生还没接触到小数

时用的商和余数的概念，如 13/5，其商是 2，余数是 3。除了以后，商放在 A 中，余数放在 B 中。CY 和 OV 都是 0。如果在做除法前 B 中的 值是 00H，也就是除数为 0，那么 OV=1。

加 1 指令

INC A

INC Rn

INC direct

INC @Ri

INC DPTR

用途很简单，就是将后面目标中的值加 1。例：（A）=12H，（R0）=33H，（21H）=32H，（34H）=22H，DPTR=1234H。执行下面的指令：

INC A （A）=13H

INC R2 （R0）=34H

INC 21H （21H）=33H

INC @R0 （34H）=23H

INC DPTR （ DPTR）=1235H

后结果如上所示。

说明：从结果上看 INC A 和 ADD A，#1 差不多，但 INC A 是单字节，单周期指令，而 ADD #1 则是双字节，双周期指令，而且 INC A 不会影响 PSW 位，如（A）=0FFH，INC A 后

（A）=00H，而 CY 依然保持不变。如果是 ADD A ，#1，则（A）=00H，而 CY 一定是 1。因此加 1 指令并不适合做加法，事实上它主要是用来做计数、地址增加等用途。另外，加法类指令都是以 A 为核心 的&#0;&#0;其中一个数必须放在 A 中，而运算结果也必须放在 A 中，而加 1 类指令的对象则广泛得多，能是寄存器、内存地址、间址寻址 的地址等等。

减 1 指令

减 1 指令

DEC A

DEC RN

DEC direct

DEC @Ri

与加 1 指令类似，就不多说了。

综合练习：

```
MOV A，#12H
MOV R0，#24H
MOV 21H，#56H
ADD A，#12H
MOV DPTR，#4316H
ADD A，DPH
```

```
ADD A，R0

CLR C

SUBB A，DPL

SUBB A，#25H

INC A

SETB C

ADDC A，21H

INC R0

SUBB A，R0

MOV 24H，#16H

CLR C

ADD A，@R0
```

先写出每步运行结果，然后将以上题目建入，并在软件仿真中运行，观察寄存器及有关单元的内容的变化，是否与自己的预想结果相同。

12、单片机逻辑运算类指令

对单片机的累加器 A 的逻辑操作：

CLR A ；将 A 中的值清 0，单周期单字节指令，与 MOV A，#00H 效果相同。

CPL A ；将 A 中的值按位取反

RL A ；将 A 中的值逻辑左移

RLC A ；将 A 中的值加上进位位进行逻辑左移

RR A ；将 A 中的值进行逻辑右移

RRC A ；将 A 中的值加上进位位进行逻辑右移

SWAP A ；将 A 中的值高、低 4 位交换。

例：（A）=73H，则执行 CPL A，这样进行：

73H 化为二进制为 01110011，

逐位取反即为 10001100，也就是 8CH。

RL A 是将（A）中的值的第 7 位送到第 0 位，第 0 位送 1 位，依次类推。

例:A 中的值为 68H,执行 RL A。68H 化为二进制为 01101000,按上图进行移动。01101000 化为 11010000，即 D0H。

RLC A，是将（A）中的值带上进位位（C）进行移位。

例：A 中的值为 68H，C 中的值为 1，则执行 RLC A

1 01101000 后，结果是 0 11010001，也就是 C 进位位的值变成了 0，而（A）则变成了 D1H。

RR A 和 RRC A 就不多谈了，请大家参考上面两个例程自行练习吧。

SWAP A，是将 A 中的值的高、低 4 位进行交换。

例：（A）=39H，则执行 SWAP A 之后，A 中的值就是 93H。怎么正好是这么前后交换呢？因为这是一个 16 进制数，每 1 个 16 进位数字代表 4 个二进位。注意，如果是这样的：（A）=39，后面没 H，执行 SWAP A 之后，可不是（A）=93。要将它化成二进制再算：39 化为二进制是 10111，也就是 0001，0111 高 4 位是 0001，低 4 位是 0111，交换后 是 01110001，也就是 71H，即 113。

练习，已知（A）=39H，执行下列单片机指令后写出每步的结果

```
CPL A

RL A

CLR C

RRC A

SETB C

RLC A

SWAP A
```

通过前面的学习，我们已经掌握了相当一部份的单片机指令，大家对这些枯燥的单片机指令可能也有些厌烦了，下面让我们轻松一下，做个实验。

实验五：

```
ORG 0000H

LJMP START
```

```
ORG 30H
START:
MOV SP, #5FH
MOV A, #80H
LOOP:
MOV P1, A
RL A
LCALL DELAY
LJMP LOOP
delay:
mov r7, #255
d1:  mov r6, #255
d2:  nop
nop
nop
nop
```

```
djnz r6, d2

djnz r7, d1

ret

END
```

先让我们将程序写入片中，装进实验板，看一看现象。

看到的是一个暗点流动的现象，让我们来分析一下吧。

前而的 ORG 0000H、LJMP START、ORG 30H 等我们稍后分析。从 START 开始，MOV SP，#5FH，这是初始化堆栈，在本程序中有无此句无关紧要，不过我们慢慢开始接触正规的编程，我也就慢慢给大家培养习惯吧。

MOV A，#80H，将 80H 这个数送到 A 中去。干什么呢？不知道，往下看。

MOV P1，A。将 A 中的值送到 P1 端口去。此时 A 中的值是 80H，所以送出去的也就是 80H，因此 P1 口的值是 80H，也就是 10000000B，通过前面的 分析，我们应当知道，此时 P1。7 接的 LED 是不亮的，而其它的 LED 都是亮的，所以就形成了一个“暗点”。继续看，RL A，RL A 是将 A 中的值进行左移，算一下，移之后的结果是什么？对了，是 01H，也就是 00000001B，这样，应当是接在 P1。0 上的 LED 不亮，而其它的都 亮了，从现象上看“暗点”流到了后面。然后是调用延时程序，这个我们很熟悉了，让这个“暗点”“暗”一会儿。然后又调转到 LOOP 处（LJMP LOOP）。请大家计算一下，下面该哪个灯不亮了。。。。。对了，应当是接在 P1。1 上灯不亮了。这样依次循环，就形成了“暗点流动”这一现象。

问题：

如何实现亮点流动？

如何改变流动的方向？

答案：

1、将 A 中的初始值改为 7FH 即可。

2、将 RL A 改为 RR A 即可。

13、单片机逻辑与或异或指令详解

ANL A，Rn ;A 与 Rn 中的值按位‘与’，结果送入 A 中

ANL A，direct ;A 与 direct 中的值按位‘与’，结果送入 A 中

ANL A，@Ri ;A 与间址寻址单元@Ri 中的值按位‘与’，结果送入 A 中

ANL A，#data ;A 与立即数 data 按位‘与’，结果送入 A 中

ANL direct，A ;direct 中值与 A 中的值按位‘与’，结果送入 direct 中

ANL direct，#data ;direct 中的值与立即数 data 按位‘与’，结果送入 direct 中。

这几条指令的关键是知道什么是逻辑与。这里的逻辑与是指按位与

例：71H 和 56H 相与则将两数写成二进制形式：

（71H） 01110001

（56H） 00100110

结果 00100000 即 20H，从上面的式子能看出，两个参与运算的值只要其中有一个位上是 0，则这位的结果就是 0，两个同是 1，结果才是 1。

理解了逻辑与的运算规则，结果自然就出来了。看每条指令后面的注释

下面再举一些例程来看。

MOV A，#45H ；（A）=45H

MOV R1，#25H ；（R1）=25H

MOV 25H，#79H ；（25H）=79H

ANL A，@R1 ;45H 与 79H 按位与，结果送入 A 中为 41H （A）=41H

ANL 25H，#15H ;25H 中的值（79H）与 15H 相与结果为（25H）=11H）

ANL 25H，A ;25H 中的值（11H）与 A 中的值（41H）相与，结果为（25H）=11H

在知道了逻辑与指令的功能后，逻辑或和逻辑异或的功能就很简单了。逻辑或是按位“或”，即有“1”为 1，全“0”为 0。例：

10011000

或 01100001

结果 11111001

而异或则是按位“异或”，相同为“0”，相异为“1”。例：

10011000

异或 01100001

结果 11111001

而所有的或指令，就是将与指仿中的 ANL 换成 ORL，而异或指令则是将 ANL 换成 XRL。即

或指令：

ORL A，Rn ;A 和 Rn 中的值按位‘或’，结果送入 A 中

ORL A，direct ;A 和与间址寻址单元@Ri 中的值按位‘或’，结果送入 A 中

ORL A，#data ;A 和立 direct 中的值按位‘或’，结果送入 A 中

ORL A，@Ri ;A 和即数 data 按位‘或’，结果送入 A 中

ORL direct，A ;direct 中值和 A 中的值按位‘或’，结果送入 direct 中

ORL direct，#data ;direct 中的值和立即数 data 按位‘或’，结果送入 direct 中。

异或指令：

XRL A，Rn ;A 和 Rn 中的值按位‘异或’，结果送入 A 中

XRL A，direct ;A 和 direct 中的值按位‘异或’，结果送入 A 中

XRL A，@Ri ;A 和间址寻址单元@Ri 中的值按位‘异或’，结果送入 A 中

XRL A，#data ;A 和立即数 data 按位‘异或’，结果送入 A 中

XRL direct，A ;direct 中值和 A 中的值按位‘异或’，结果送入 direct 中

XRL direct，#data ;direct 中的值和立即数 data 按位‘异或’，结果送入 direct 中。

练习：

MOV A，#24H

MOV R0，#37H

ORL A，R0

XRL A，#29H

MOV 35H，#10H

ORL 35H，#29H

MOV R0，#35H

ANL A，@R0

14、控制转移类指令

无条件转移类指令

短转移类指令

AJMP addr11

长转移类指令

LJMP addr16

相对转移指令

SJMP rel

上面的三条指令，如果要仔细分析的话，区别较大，但开始学习时，可不理会这么多，统统理解成：JMP 标号，也就是跳转到一个标号处。事实上，LJMP 标号，在前面的例程中我们已接触过，并且也知道如何来使用了。而 AJMP 和 SJMP 也是一样。那么他们的区别何在呢？在于跳转的范围不一样。好比跳 远，LJMP 一下就能跳 64K 这么远（当然近了更没关系了）。而 AJMP 最多只能跳 2K 距离，而 SJMP 则最多只能跳 256 这么远。原则上，所有用 SJMP 或 AJMP 的地方都能用 LJMP 来替代。因此在开始学习时，需要跳转时 能全用 LJMP，除了一个场合。什么场合呢？先了解一下 AJMP，AJMP 是一条双字节指令，也就说这条指令本身占用存储器(ROM)的两个单元。而 LJMP 则是三字节指令，即这条指令占用存储器(ROM)的三个单元。下面是第四条跳转指令。

间接转移指令

JMP @A+DPTR

这条指令的用途也是跳转，转到什么地方去呢？这可不能由标号简单地决定了。让我们从一个实际的例程入手吧。

```
MOV DPTR，#TAB ;将 TAB 所代表的地址送入 DPTR
MOV A，R0 ;从 R0 中取数（详见下面说明）
MOV B，#2
MUL A，B ;A 中的值乘 2（详见下面的说明）
JMP A，@A+DPTR ;跳转
TAB： AJMP S1 ;跳转表格
AJMP S2
```

AJMP S3

应用背景介绍：在单片机开发中，经常要用到键盘，见上面的 9 个按钮的键盘。我们的要求是：当按下功能键 A……。.G 时去完成不一样的功能。这用程序设计的语言来表达的话，就是：按下不一样的键去执行不一样的程序段，以完成不一样的功能。怎么样来实现呢？

前面的程序读入的是按钮的值，如按下‘A’键后获得的键值是 0，按下‘B’键后获得的值是‘1’等等，然后根据不一样的值进行跳转，如键值为 0 就转到 S1 执行，为 1 就转到 S2 执行。。。。如何来实现这一功能呢？

先从程序的下面看起，是若干个 AJMP 语句，这若干个 AJMP 语句最后在存储器中是这样存放的（见图 3），也就是每个 AJMP 语句都占用了两个存储器 的空间，并且是连续存放的。而 AJMP S1 存放的地址是 TAB，到底 TAB 等于多少，我们不需要知道，把它留给汇编程序来算好了。

下面我们来看这段程序的执行过程：第一句 MOV DPTR，#TAB 执行完了之后，DPTR 中的值就是 TAB，第二句是 MOV A，R0，我们假设 R0 是由按钮处理程序获得的键值，比如按下 A 键，R0 中的值是 0，按下 B 键，R0 中的值是 1，以此类推，现在我们假设按下的是 B 键，则 执行完第二条指令后，A 中的值就是 1。并且按我们的分析，按下 B 后应当执行 S2 这段程序，让我们来看一看是否是这样呢？第三条、第四条指令是将 A 中的值乘 2，即执行完第 4 条指令后 A 中的值是 2。下面就执行 JMP @A+DPTR 了，现在 DPTR 中的值是 TAB，而 A+DPTR 后就是 TAB+2，因此，执行此句程序后，将会跳到 TAB+2 这个地址继续执行。看一看在 TAB+2 这个地址里面放的是什么？就是 AJMP S2 这条指令。因此，马上又执行 AJMP S2 指令，程序将跳到 S2 处往下执行，这与我们的要求相符合。

请大家自行分析按下键“A”、“C”、“D”……之后的情况。

这样我们用 JMP @A+DPTR 就实现了按下一键跳到对应的程序段去执行的这样一个要求。再问大家一个问题，为什么取得键值后要乘 2？如果例程下面的所有指令换成 LJMP，即：

LJMP S1，LJMP S2……这段程序还能正确地执行吗？如果不能，应该怎么改？

14、单片机条件转移指令

条件转移指令是指在满足一定条件时进行相对转移。

判 A 内容是否为 0 转移指令

```
JZ rel

JNZ rel
```

第一指令的功能是：如果(A)=0，则转移，不然次序执行(执行本指令的下一条指令)。转移到什么地方去呢？如果按照传统的办法，就要算偏移量，很麻烦，好在现在我们能借助于机器汇编了。因此这第指令我们能这样理解：JZ 标号。即转移到标号处。下面举一例说明：

```
MOV A，R0

JZ L1

MOV R1，#00H

AJMP L2

L1： MOV R1，#0FFH

L2： SJMP L2

END
```

在执行上面这段程序前如果 R0 中的值是 0 的话，就转移到 L1 执行，因此最终的执行结果是 R1 中的值为 0FFH。而如果 R0 中的值不等于 0，则次序执行，也就是执行 MOV R1，#00H 指令。最终的执行结果是 R1 中的值等于 0。

第一条指令的功能清楚了，第二条当然就好理解了，如果 A 中的值不等于 0，就转移。把上面的那个例程中的 JZ 改成 JNZ 试试吧，看看程序执行的结果是什么？

比较转移指令

```
CJNE A，#data，rel

CJNE A，direct，rel

CJNE Rn，#data，rel

CJNE @Ri，#data，rel
```

第一条指令的功能是将 A 中的值和立即数 data 比较，如果两者相等，就次序执行（执行本指令的下一条指令），如果不相等，就转移，同样地，我们能将 rel 理解成标号，即：CJNE A，#data，标号。这样利用这条指令，我们就能判断两数是否相等，这在很多场合是非常有用的。但有时还想得知两数比较之后哪个大，哪个小，本条指令也 具有这样的功能，如果两数不相等，则 CPU 还会反映出哪个数大，哪个数小，这是用 CY（进位位）来实现的。如果前面的数（A 中的）大，则 CY=0，不然 CY=1，因此在程序转移后再次利用 CY 就可判断出 A 中的数比 data 大还是小了。

例：

```
MOV A，R0

CJNE A，#10H，L1

MOV R1，#0FFH

AJMP L3

L1： JC L2
```

```
MOV R1，#0AAH

AJMP L3

L2： MOV R1，#0FFH

L3： SJMP L3
```

上面的程序中有一条单片机指令我们还没学过，即 JC，这条指令的原型是 JC rel，作用和上面的 JZ 类似，但是它是判 CY 是 0，还是 1 进行转移，如果 CY=1，则转移到 JC 后面的标号处执行，如果 CY=0 则次序执行（执行它的下面一条指令）。

分析一下上面的程序，如果（A）=10H，则次序执行，即 R1=0。如果（A）不等于 10H，则转到 L1 处继续执行，在 L1 处，再次进行判断，如果 （A）》10H，则 CY=1，将次序执行，即执行 MOV R1，#0AAH 指令，而如果（A）《10H，则将转移到 L2 处指行，即执行 MOV R1，#0FFH 指令。因此最终结果是：本程序执行前，如果（R0）=10H，则（R1）=00H，如果（R0）》10H，则（R1）=0AAH，如果 （R0）《10H，则（R1）=0FFH。

弄懂了这条指令，其它的几条就类似了，第二条是把 A 当中的值和直接地址中的值比较，第三条则是将直接地址中的值和立即数比较，第四条是将间址寻址得到的数和立即数比较，这里就不详谈了，下面给出几个对应的例程。

```
CJNE A，10H ;把 A 中的值和 10H 中的值比较（注意和上题的区别）

CJNE 10H，#35H ;把 10H 中的值和 35H 中的值比较

CJNE @R0，#35H ;把 R0 中的值作为地址，从此地址中取数并和 35H 比较

循环转移指令

DJNZ Rn，rel
```

DJNZ direct，rel

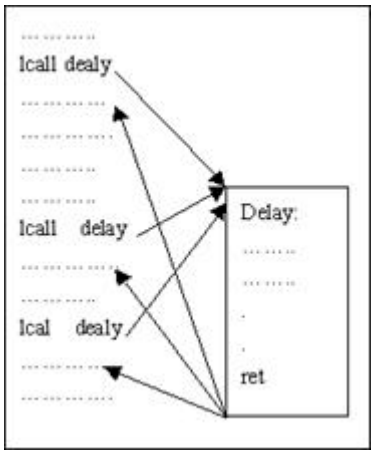
第一条指令在前面的例程中有详细的分析，这里就不多谈了。第二条指令，只是将 Rn 改成直接地址，其它一样，也不多说了，给一个例程。

DJNZ 10H，LOOP

3．调用与返回指令

（1）主程序与子程序 在前面的灯的实验中，我们已用到过了子程序，只是我们并没有明确地介绍。子程序是干什么用的，为什么要用子程序技术呢？举个例程，我们数据老师布置了 10 道算术题，经过观察，每一道题中都包含一个（3*5+2）*3 的运算，我们能有两种选择，第一种，每做一道题，都把这个算式算一遍，第二种选择，我们能先 把这个结果算出来，也就是 51，放在一边，然后要用到这个算式时就将 51 代进去。这两种办法哪种更好呢？不必多言。设计程序时也是这样，有时一个功能会在 程序的不一样地方反复使用，我们就能把这个功能做成一段程序，每次需要用到这个功能时就“调用”一下。

（2）调用及回过程：主程序调用了子程序，子程序执行完之后必须再回到主程序继续执行，不能“一去不回头”，那么回到什么地方呢？是回到调用子程序的下面一条指令继续执行(当然啦,要是还回到这条指令,不又要再调用子程序了吗？那可就没完没了了……)。参考图 1



调用指令

LCALL addr16 ;长调用指令

ACALL addr11 ;短调用指令

上面两条指令都是在主程序中调用子程序，两者有一定的区别，但在开始学习单片机的这些指令时，能不加以区别，而且能用 LCALL 标号，ACALL 标号，来理解，即调用子程序。

（5）返回指令则说了，子程序执行完后必须回到主程序，如何返回呢？只要执行一条返回指令就能了，即执行 ret 指令

4．空操作指令

nop 就是 空操作，就是什么事也不干，停一个周期，一般用作短时间的延时。

单片机学习知识点全攻略（三）

参阅相关系列

单片机学习知识点全攻略（一）

单片机学习知识点全攻略（二）

系列三主要知识点：

- 15：单片机位操作指令
- 16：单片机定时器与计数器
- 17：单片机定时器/计数器的方式
- 18：单片机的中断系统
- 19：单片机定时器、中断试验
- 20：单片机定时/计数器实验
- 21：单片机串行口介绍

15、单片机位操作指令

前面那些流水灯的例程，我们已经习惯了“位”一位就是一盏灯的亮和灭，而我们学的指令却全都是用“字节”来介绍的：字节的移动、加法、减法、逻辑运 算、移位等等。用字节来处理一些数学问题，比如说：控制冰箱的温度、电视的音量等等很直观，能直接用数值来表在。可是如果用它来控制一些开关的打开和合 上，灯的亮和灭，就有些不直接了，记得我们上次课上的流水灯的例程吗？我们知道送往 P1 口的数值后并不能马上知道哪个灯亮和来灭，而是要化成二进制才知 道。工业中有很多场合需要处理这类开关输出，继电器吸合，用字节来处理就显示有些麻烦，所以在 8031 单片机中特意引入一个位处理机制。

位寻址区

在 8031 中，有一部份 RAM 和一部份 SFR 是具有位寻址功能的，也就是说这些 RAM 的每一个位都有自己的地址，能直接用这个地址来对此进行操作。

内部 RAM 的 20H-2FH 这 16 个字节，就是 8031 的位寻址区。看图 1。可见这里面的每一个 RAM 中的每个位我们都可能直接用位地址来找到它们，而不必用字节地址，然后再用逻辑指令的方式。

能位寻址的特殊功能寄存器

8031 中有一些 SFR 是能进行位寻址的，这些 SFR 的特点是其字节地址均可被 8 整除，如 A 累加器，B 寄存器、PSW、IP（中断优先级控制寄存 器）、IE（中断允许控制寄存器）、SCON（串行口控制寄存器）、TCON（定时器/计数器控制寄存器）、P0-P3（I/O 端口锁存器）。以上的一些 SFR 我们还不熟，等我们讲解相关内容时再作详细解释。

位操作指令

MCS-51 单片机的硬件结构中，有一个位处理器（又称布尔处理器），它有一套位变量处理的指令集。在进行位处理时，CY（就是我们前面讲的进位位） 称“位累加器”。有自己的位 RAM，也就是我们刚讲的内部 RAM 的 20H-2FH 这 16 个字节单元即 128 个位单元，还有自己的位 I/O 空间（即 P0.0…。.P0.7，P1.0…….P1.7，P2.0……。.P2.7，P3.0……。.P3.7）。当然在物理实体上它们与原来的以字节寻址用的 RAM，及端口是完全相同的，或者说这些 RAM 及端口都能有两种使用办法。

位传送指令

MOV C，BIT

MOV BIT，C

这组指令的功能是实现位累加器（CY）和其它位地址之间的数据传递。

例：MOV P1.0，CY ;将 CY 中的状态送到 P1.0 管脚上去（如果是做算术运算，我们就能通过观察知道现在 CY 是多少啦）。

MOV P1.0，CY ;将 P1.0 的状态送给 CY。

位修正指令

位清 0 指令

CLR C ;使 CY=0

CLR bit ;使指令的位地址等于 0。例：CLR P1.0 ;即使 P1.0 变为 0

位置 1 指令

SETB C ;使 CY=1

SETB bit ;使指定的位地址等于 1。例：SETB P1.0 ;使 P.0 变为 1

位取反指令

CPL C ;使 CY 等于原来的相反的值，由 1 变为 0，由 0 变为 1。

CPL bit ;使指定的位的值等于原来相反的值，由 0 变为 1，由 1 变为 0。

例：CPL P1.0

以我们做过的实验为例，如果原来灯是亮的，则执行本指令后灯灭，反之原来灯是灭的，执行本指令后灯亮。

位逻辑运算指令

位与指令

ANL C，bit ;CY 与指定的位地址的值相与，结果送回 CY

ANL C，/bit ;先将指定的位地址中的值取出后取反，再和 CY 相与，结果送回 CY，但注意，指定的位地址中的值本身并不发生变化。

例：ANL C，/P1.0

设执行本指令前，CY=1，P1.0 等于 1（灯灭），则执行完本指令后 CY=0，而 P1.0 也是等于 1。

可用下列程序验证：

```
ORG 0000H

AJMP START

ORG 30H

START： MOV SP，#5FH

MOV P1，#0FFH

SETB C

ANL C，/P1.0

MOV P1.1，C ;将做完的结果送 P1.1，结果应当是 P1.1 上的灯亮，而 P1.0 上的灯还是不亮
```

位或指令

```
ORL C，bit

ORL C，/bit
```

这个的功能大家自行分析吧，然后对照上面的例程，编一个验证程序，看看你相得对吗？

位条件转移指令

判 CY 转移指令

JC rel

JNC rel

第一条指令的功能是如果 CY 等于 1 就转移，如果不等于 1 就次序执行。那么转移到什么地方去呢？我们能这样理解：JC 标号，如果等于 1 就转到标号处执行。这条指令我们在上节课中已讲到，不再重复。

第二条指令则和第一条指令相反，即如果 CY=0 就转移，不等于 0 就次序执行，当然，我们也同样理解： JNC 标号

判位变量转移指令

JB bit，rel

JNB bit，rel

第一条指令是如果指定的 bit 位中的值是 1，则转移，不然次序执行。同样，我们能这样理解这条指令：JB bit，标号

第二条指令请大家先自行分析

下面我们举个例程说明：

ORG 0000H

```
LJMP START

ORG 30H

START：MOV SP，#5FH

MOV P1，#0FFH

MOV P3，#0FFH

L1： JNB P3.2，L2 ;P3.2 上接有一只按钮，它按下时，P3.2=0

JNB P3.3，L3 ;P3.3 上接有一只按钮，它按下时，P3.3=0

LJM P L1

L2： MOV P1，#00H

LJMP L1

L3： MOV P1，#0FFH

LJMP L1

END
```

把上面的例程写入片子，看看有什么现象……

按下接在 P3.2 上的按钮，P1 口的灯全亮了，松开或再按，灯并不熄灭，然后按下接在 P3.3 上的按钮，灯就全灭了。这像什么？这不就是工业现场经常用到的“启动”、“停止”的功能吗？

怎么做到的呢？一开始，将 0FFH 送入 P3 口，这样，P3 的所有引线都处于高电平，然后执行 L1，如果 P3.2 是高电平（键没有按下），则次序执行 JNB P3.3，L3 语句，同样，如果 P3.3 是高电平（键没有按下），则次序执行 LJMP L1 语句。这样就不停地检测 P3.2、P3.3，如果有一次 P3.2 上的按钮按下去了，则转移到 L2，执行 MOV P1，#00H，使灯全亮，然后又转去 L1，再次循环，直到检测到 P3.3 为 0，则转 L3，执行 MOV P1，#0FFH，例灯全灭，再转去 L1，如此循环不已。大家能否稍加改动，将本程序用 JB 指令改写？

16、单片机定时器与计数器

一、计数概念的引入

从选票的统计谈起：画“正”。这就是计数，生活中计数的例程处处可见。例：录音机上的计数器、家里面用的电度表、汽车上的里程表等等，再举一个工业生 产中的例程，线缆行业在电线生产出来之后要计米，也就是测量长度，怎么测法呢？用尺量？不现实，太长不说，要一边做一边量呢，怎么办呢？行业中有很巧妙的 办法，用一个周长是 1 米的轮子，将电缆绕在上面一周，由线带轮转，这样轮转一周不就是线长 1 米嘛，所以只要记下轮转了多少圈，就能知道走过的线有多长了。

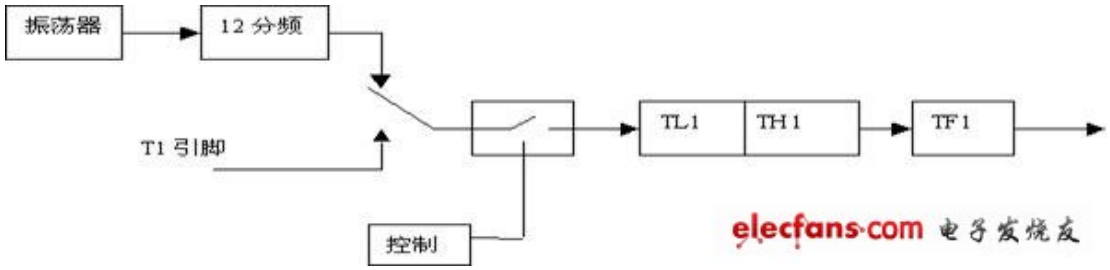
二、计数器的容量

从一个生活中的例程看起：一个水盆在水龙头下，水龙没关紧，水一滴滴地滴入盆中。水滴持续落下，盆的容量是有限的，过一段时间之后，水就会逐渐变满。 录音机上的计数器最多只计到 999…。那么单片机中的计数器有多大的容量呢？8031 单片机中有两个计数器，分别称之为 T0 和 T1，这两个计数器分别是由 两个 8 位的 RAM 单元组成的，即每个计数器都是 16 位的计数器，最大的计数量是 65536。

三、定时

8031 中的计数器除了能作为计数之用外，还能用作时钟，时钟的用途当然很大，如打铃器，电视机定时关机，空调定时开关等等，那么计数器是如何作为定时器来用的呢？

一个闹钟，我将它定时在 1 个小时后闹响，换言之，也能说是秒针走了（3600）次，所以时间就转化为秒针走的次数的，也就是计数的次数了，可见，计数的次数和时间之间的确十分相关。那么它们的关系是什么呢？那就是秒针每一次走动的时间正好是 1 秒。



《单片机定时器记数器结构》

结论：只要计数脉冲的间隔相等，则计数值就代表了时间的流逝。由此，单片机中的定时器和计数器是一个东西，只不过计数器是记录的外界发生的事情，而定 时器则是由单片机供给一个非常稳定的计数源。那么供给组定时器的是计数源是什么呢？看图 1，原来就是由单片机的晶体震荡器经过 12 分频后获得的一个脉冲 源。晶体震荡器的频率当然很准，所以这个计数脉冲的时间间隔也很准。问题：一个 12M 的晶体震荡器，它供给给计数器的脉冲时间间隔是多少呢？当然这很不 难，就是 12M/12 等于 1M，也就是 1 个微秒。结论：计数脉冲的间隔与晶体震荡器有关，12M 的晶体震荡器，计数脉冲的间隔是 1 微秒。

四、溢出

让我们再来看水滴的例程，当水持续落下，盆中的水持续变满，最终有一滴水使得盆中的水满了。这个时候如果再有一滴水落下，就会发生什么现象？水会漫出来，用个术语来讲就是“溢出”。

水溢出是流到地上，而计数器溢出后将使得 TF0 变为“1”。至于 TF0 是什么我们稍后再谈。一旦 TF0 由 0 变成 1，就是产生了变化，产生了变化就会引 发事件，就象定时的时

间一到，闹钟就会响一样。至于会引发什么事件，我们下次课再介绍，现在我们来研究另一个问题：要有多少个计数脉冲才会使 TF0 由 0 变 为 1。

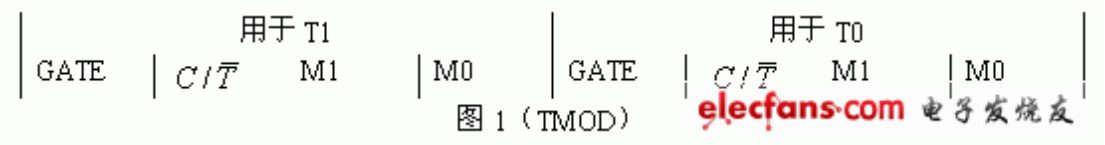
五、任意定时及计数的办法 刚才已研究过，计数器的容量是 16 位，也就是最大的计数值到 65536，因此计数计到 65536 就会产生溢出。这个没有问题，问题是我们现实生活中，经常 会有少于 65536 个计数值的要求，如包装线上，一打为 12 瓶，一瓶药片为 100 粒，怎么样来满足这个要求呢？

提示：如果是一个空的盆要 1 万滴水滴进去才会满，我在开始滴水之前就先放入一勺水，还需要 10000 滴嘛？对了，我们采用预置数的办法，我要计 100，那我就先放进 65436，再来 100 个脉冲，不就到了 65536 了吗。定时也是如此，每个脉冲是 1 微秒，则计满 65536 个脉冲需时 65.536 毫秒，但现在我只要 10 毫秒就能了，怎么办？10 个毫秒为 10000 个微秒，所以，只要在计数器里面放进 55536 就能了。

17、单片机定时器/计数器的方式

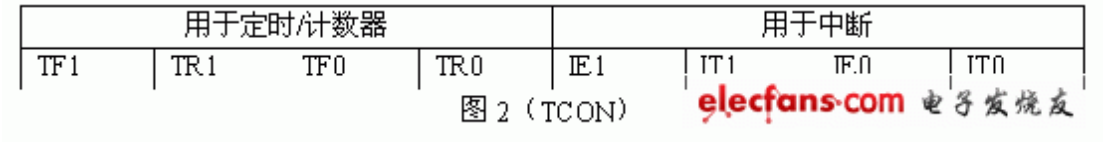
从上一节我们已经得知，单片机中的定时/计数器都能有多种用途，那么我怎样才能让它们工作于我所需要的用途呢？这就要通过定时/计数器的方式控制字来设置。

在单片机中有两个特殊功能寄存器与定时/计数有关，这就是 TMOD 和 TCON。顺便说一下，TMOD 和 TCON 是名称，我们在写程序时就能直接用这个 名称来指定它们，当然也能直接用它们的地址 89H 和 88H 来指定它们（其实用名称也就是直接用地址，汇编软件帮你翻译一下而已）。



《TMOD 结构》

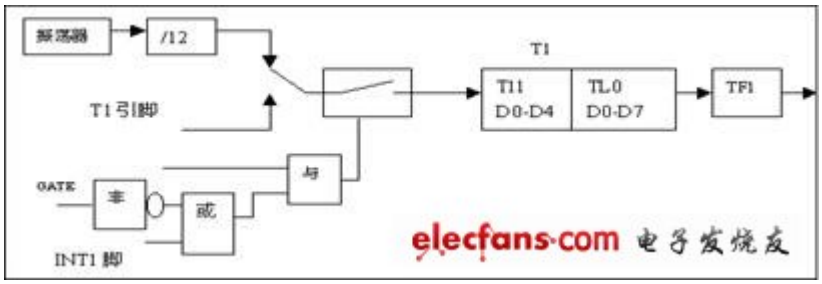
从图 1 中我们能看出，TMOD 被分成两部份，每部份 4 位。分别用于控制 T1 和 T0，至于这里面是什么意思，我们下面介绍。



《TCON 结构》

从图 2 中我们能看出，TCON 也被分成两部份，高 4 位用于定时/计数器，低 4 位则用于中断（我们暂不管）。而 TF1（0）我们上节课已提到了，当计数溢出后 TF1（0）就由 0 变为 1。原来 TF1（0）在这儿！那么 TR0、TR1 又是什么呢？看上节课的图。

计数脉冲要进入计数器还真不不难，有层层关要通过，最起码，就是 TR0（1）要为 1，开关才能合上，脉冲才能过来。因此，TR0（1）称之为运行控制 位，可用指令 SETB 来置位以启动计数器/定时器运行，用指令 CLR 来关闭定时/计数器的工作，一切尽在自己的掌握中。



《单片机定时器/计数器结构》

定时/计数器的四种工作方式

工作方式 0

定时器/计数器的工作方式 0 称之为 13 位定时/计数方式。它由 TL（1/0）的低 5 位和 TH（0/1）的 8 位组成 13 位的计数器，此时 TL（1/0）的高 3 位未用。

我们用这个图来讨论几个问题：

M1M0：定时/计数器一共有四种工作方式，就是用 M1M0 来控制的，2 位正好是四种组合。

C/T：前面我们说过，定时/计数器即可作定时用也可用计数用，到底作什么用，由我们根据需要自行决定，也说是决定权在我 们&#0;&#0;编程者。如果 C/T 为 0 就是用作定时器（开关往上打），如果 C/T 为 1 就是用作计数器（开关往下打）。顺便提一下：一 个定时/计数器同一时刻要么作定时用，要么作计数用，不能同时用的，这是个极普通的常识，几乎没有教材会提这一点，但很多开始学习者却会有此困惑。

GATE：看图，当我们选择了定时或计数工作方式后，定时/计数脉冲却不一定能到达计数器端，中间还有一个开关，显然这个开关不合上，计数脉冲就没法过去，那么开关什么时候过去呢？有两种情况

GATE=0，分析一下逻辑，GATE 非后是 1，进入或门，或门总是输出 1，和或门的另一个输入端 INT1 无关，在这种情况下，开关的打开、合上只取 决于 TR1，只要 TR1 是 1，开关就合上，计数脉冲得以畅通无阻，而如果 TR1 等于 0 则开关打开，计数脉冲无法通过，因此定时/计数是否工作，只取决于 TR1。

GATE=1，在此种情况下，计数脉冲通路上的开关不仅要由 TR1 来控制，而且还要受到 INT1 管脚的控制，只有 TR1 为 1，且 INT1 管脚也是高电平，开关才合上，计数脉冲才得以通过。这个特性能用来测量一个信号的高电平的宽度，想想看，怎么测？

为什 么在这种模式下只用 13 位呢？干吗不用 16 位，这是为了和 51 机的前辈 48 系列兼容而设的一种工作式，如果你觉得用得不顺手，那就干脆用第二种工作方式。

工作方式 1

工作方式 1 是 16 位的定时/计数方式，将 M1M0 设为 01 即可，其它特性与工作方式 0 相同。

工作方式 2

在介绍这种式方式之前先让我们思考一个问题：上一次课我们提到过任意计数及任意定时的问题，比如我要计 1000 个数，可是 16 位的计数器要计到 65536 才满，怎么办呢？我们讨论后得出的办法是用预置数，先在计数器里放上 64536，再来 1000 个脉冲，不就行了吗？是的，但是计满了之后我们又 该怎么办呢？要知道，计数总是持续重复的，流水线上计满后马上又要开始下一次计数，下一次的计数还是 1000 吗？当计满并溢出后，计数器里面的值变成了 0（为什么，能参考前面课程的说明），因此下一次将要计满 65536 后才会溢出，这可不符合要求，怎么办？当然办法很简单，就是每次一溢出时执行一段程序（这常常是需要的，要不然要溢出干吗？）能在这段程序中做把预置数 64536 送入计数器中的事情。所以采用工作方式 0 或 1 都要在溢出后做一个重置预置数的 工作，做工作当然就得要时间，一般来说这点时间不算什么，可是有一些场合我们还是要计较的，所以就有了第三种工作方式&#0;&#0;自 动再装入预置数的工作方式。

既然要自动得新装入预置数，那么预置数就得放在一个地方，要不然装什么呢？那么预置数放在什么地方呢？它放在 T（0/1）的高 8 位，那么这样高 8 位不 就不能参与计数了吗？是的，在工作方式 2，只有低 8 位参与计数，而高 8 位不参与计数，用作预置数的存放，这样计数范围就小多了，当然做任可事总有代价的， 关键是看值不值，如果我根本不需要计那么多数，那么就能用这种方式。看图 4，每当计数溢出，就会打开 T（0/1）的高、低 8 位之间的开关，计预置数进入低 8 位。这是由硬件自动完成的，不需要由人工干预。

常常这种式作方式用于波特率发生器（我们将在串行接口中讲解），用于这种用途时，定时器就是为了供给一个时间基准。计数溢出后不需要做事情，要做的仅仅只有一件，就是重新装入预置数，再开始计数，而且中间不要任何延迟，可见这个任务用工作方式 2 来完成是最妙不过了。

工作方式 3

这种式作方式之下，定时/计数器 0 被拆成 2 个独立的定时/计数器来用。其中，TL0 能组成 8 位的定时器或计数器的工作方式，而 TH0 则只能作为定时器 来用。我们知道作定时、计数器来用，需要控制，计满后溢出需要有溢出标记，T0 被分成两个来用，那就要两套控制及、溢出标记了，从何而来呢？TL0 还是用 原来的 T0 的标记，而 TH0 则借用 T1 的标记。如此 T1 不是无标记、控制可用了吗？是的。

一般情况处，只有在 T1 以工作方式 2 运行（当波特率发生器用）时，才让 T0 工作于方式 3 的。

定时器/计数器的定时/计数范围

工作方式 0：13 位定时/计数方式，因此，最多能计到 2 的 13 次方，也就是 8192 次。

工作方式 1：16 位定时/计数方式，因此，最多能计到 2 的 16 次方，也就是 65536 次。

工作方式 2 和工作方式 3，都是 8 位的定时/计数方式，因此，最多能计到 2 的 8 次方，也说是 256 次。

预置值计算：用最大计数量减去需要的计数次数即可。

例：流水线上一个包装是 12 盒，要求每到 12 盒就产生一个动作，用单片机的工作方式 0 来控制，应当预置多大的值呢？对了，就是 8192-12=8180。

以上是计数，明白了这个道理，定时也是一样。这在前面的课程已提到，我们不再重复，请参考前面的例程。

18、单片机的中断系统

有关单片机中断系统的概念：什么是中断，我们从一个生活中的例程引入。你正在家中看书，突然电话铃响了，你放下书本，去接电话，和来电话的人交谈，然 后放下电话，回来继续看你的书。这就是生活中的“中断”的现象，就是正常的工作过程被外部的事件打断了。仔细研究一下生活中的中断，对于我们学习单片机的 中断也很有好处。

第一、什么可经引起中断，生活中很多事件能引起中断：有人按了门铃了，电话铃响了，你的闹钟闹响了，你烧的水开了…。等等诸如此类的事件，我们把能引 起中断的称之为中断源，单片机中也有一些能引起中断的事件，8031 中一共有 5 个：两个外部中断，两个计数/定时器中断，一个串行口中断。

第二、中断的嵌套与优先级处理：设想一下，我们正在看书，电话铃响了，同时又有人按了门铃，你该先做那样呢？如果你正是在等一个很重要的电话，你一般 不会去理会门铃的，而反之，你正在等一个重要的客人，则可能就不会去理会电话了。如果不是这两者（即不等电话，也不是等人上门），你可能会按你常常的习惯 去处理。总之这里存在一个优先级的问题，单片机中也是如此，也有优先级的问题。优先级的问题不仅仅发生在两个中断同时产生的情况，也发生在一个中断已产 生，又有一个中断产生的情况，比如你正接电话，有人按门铃的情况，或你正开门与人交谈，又有电话响了情况。考虑一下我们会怎么办吧。

第三、中断的响应过程：当有事件产生，进入中断之前我们必须先记住现在看书的第几页了，或拿一个书签放在当前页的位置，然后去处理不一样的事情（因为 处理完了，我们还要回来继续看书）：电话铃响我们要到放电话的地方去，门铃响我们要到门那边去，也说是不一样的中断，我们要在不一样的地点处理，而这个地 点常常还是固定的。计算机中也是采用的这种办法，五个中断源，每个中断产生后都到一个固定的地方去找处理这个中断的程序，当然在去之前首先要保存下面将执 行的指令的地址，以便处理完中断后回到原来的地方继续往下执行程序。具体地说，中断响应能分为以下几个步骤：1、保护断点，即保存下一将要执行的指令的地 址，就是把这个地址送入堆栈。2、寻找中断入口，根据 5 个不一样的中断源所产生的中断，查找 5 个不一样的入口地址。以上工作是由计算机自动完成的，与编程 者无关。在这 5 个入口地址处存放有中断处理程序（这是程序编写时放在那儿的，如果没把中断程序放在那儿，就错了，中断程序就不能被执行到）。3、执行中断 处理程序。4、中断返回：执行完中断指令后，就从中断处返回到主程序，继续执行。究竟单片机是怎么样找到中断程序所在位置，又怎么返回的呢？我们稍后再 谈。

MCS-51 单片机中断系统的结构：

5 个中断源的符号、名称及产生的条件如下。

INT0：外部中断 0，由 P3．2 端口线引入，低电平或下跳沿引起。

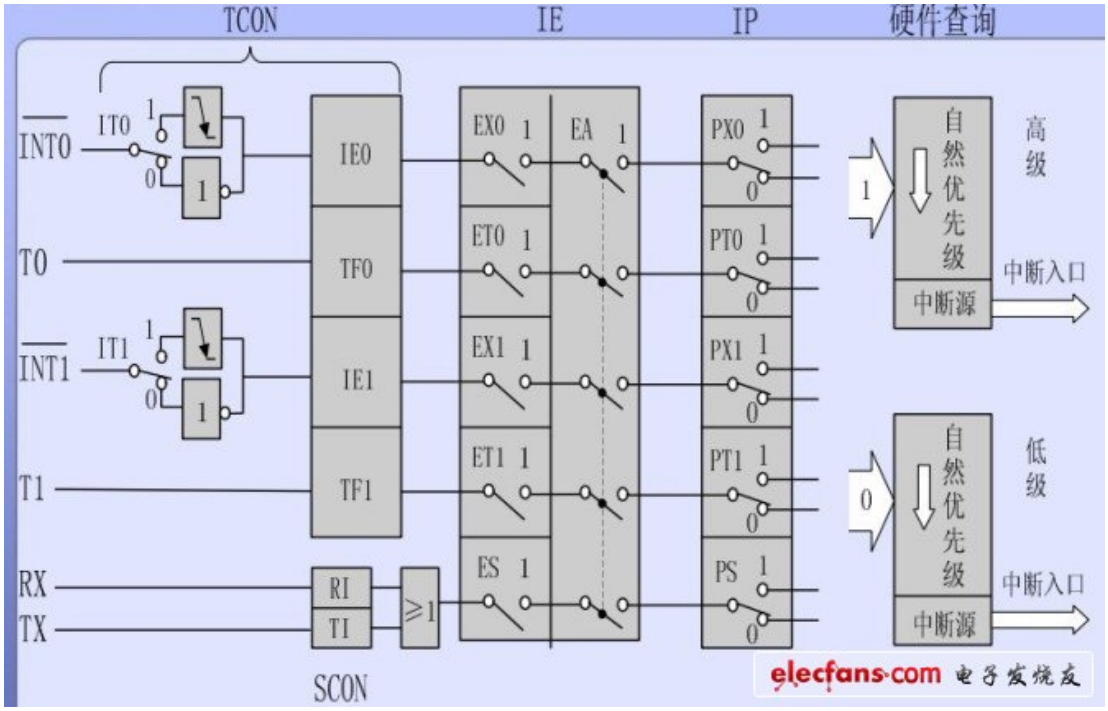
INT1：外部中断 1，由 P3．3 端口线引入，低电平或下跳沿引起。

T0：定时器 / 计数器 0 中断，由 T0 计满回零引起。

T1：定时器 / 计数器 1 中断，由 T1 计满回零引起。

TI / RI：串行 I / O 中断，串行端口完成一帧字符发送 / 接收后引起。

整个中断系统的结构框图见下图一所示。



《51 单片机中断系统结构》

如图所示，由与中断有关的特殊功能寄存器、中断入口、次序查询逻辑电路等组成，包括 5 个中断请求源，4 个用于中断控制的寄存器 IE、IP、ECON 和 SCON 来控制中断类弄、中断的开、关和各种中断源的优先级确定。

中断请求源：

（1）外部中断请求源：即外中断 0 和 1，经由外部管脚引入的，在单片机上有两个管脚，名称为 INT0、INT1，也就是 P3.2、P3.3 这两个管 脚。在内部的 TCON 中有四位是与外中断有关的。IT0：INT0 触发方式控制位，可由软件进和置位和复位，IT0=0，INT0 为低电平触发方 式，IT0=1，INT0 为负跳变触发方式。这两种方式的差异将在以后再谈。IE0：INT0 中断请求标志位。当有外部的中断请求时，这位就会置 1（这由 硬件来完成），在 CPU 响应中断后，由硬件将 IE0 清 0。IT1、IE1 的用途和 IT0、IE0 相同。（2）内部中断请求源 TF0：定时器 T0 的溢出中断 标记，当 T0 计数产生溢出时，由硬件置位 TF0。当 CPU 响应中断后，再由硬件将 TF0 清 0。TF1：与 TF0 类似。TI、RI：串行口发送、接收中断，在串行口中再讲解。2、中断允许寄存器 IE 在 MCS－51 中断系统中，中断的允许或禁止是由片内可进行位寻址的 8 位中断允许寄存器 IE 来控制的。见下表 EAX

其中 EA 是总开关，如果它等于 0，则所有中断都不允许。ES－串行口中断允许 ET1－定时器 1 中断允许 EX1－外中断 1 中断允许。ET0－定时器 0 中断允许 EX0－外中断 0 中断允许。如果我们要设置允许外中断 1，定时器 1 中断允许，其它不允许，则 IE 能是 EAX

即 8CH，当然，我们也能用位操作指令 SETB EA

SETB ET1SETB EX1

来实现它。3、五个中断源的自然优先级与中断服务入口地址外中断 0：0003H 定时器 0：000BH 外中断 1：0013H 定时器 1：001BH 串行口：0023H 它们的自然优先级由高到低排列。写到这里，大家应当明白，为什么前面有一些程序一始我们这样写：

```
ORG 0000HLJMP START

ORG 0030H
```

START：。

这样写的目的，就是为了让出中断源所占用的向量地址。当然，在程序中没用中断时，直接从 0000H 开始写程序，在原理上并没有错，但在实际工作中最好 不这样做。优先级：单片机采用了自然优先级和人工设置高、低优先级的策略，即能由程序员设定那些中断是高优先级、哪些中断是低优先级，由于只有两级，必有 一些中断处于同一级别，处于同一级别的，就由自然优先级确定。

开机时，每个中断都处于低优先级，我们能用指令对优先级进行设置。看表 2 中断优先级中由中断优先级寄存器 IP 来高置的，IP 中某位设为 1，对应的中断就是高优先级，不然就是低优先级。

XX

X

PS

PT1

PX1

PT0

PX0

例：设有如下要求，将 T0、外中断 1 设为高优先级，其它为低优先级，求 IP 的值。IP 的首 3 位没用，可任意取值，设为 000，后面根据要求写就能了 XX

因此，最终，IP 的值就是 06H。例：在上例中，如果 5 个中断请求同时发生，求中断响应的次序。响应次序为：定时器 0－＞外中断 1－＞外中断 0－＞实时器 1－＞串行中断。

MCS－51 的中断响应过程：

1、中断响应的条件：讲到这儿，我们依然对于计算机响应中断感到神奇，我们人能响应外界的事件，是因为我们有多种“传感器“－－眼、耳能接受不一样的 信息，计算机是如何做到这点的呢？其实说穿了，一点都不希奇，MCS51 工作时，在每个机器周期中都会去查询一下各个中断标记，看他们是否是“1“，如果 是 1，就说明有中断请求了，所以所谓中断，其实也是查询，不过是每个周期都查一下而已。这要换成人来说，就相当于你在看书的时候，每一秒钟都会抬起头来看 一看，查问一下，是不是有人按门铃，是否有电话。。。。很蠢，不是吗？可计算机本来就是这样，它根本没人聪明。了解了上述中断的过程，就不难解中断响应的 条件了。在下列三种情况之一时，CPU 将封锁对中断的响应：

CPU 正在处理一个同级或更高级别的中断请求。

现行的机器周期不是当前正执行指令的最后一个周期。我们知道，单片机有单周期、双周期、三周期指令，当前执行指令是单字节没有关系，如果是双字节或四字节的，就要等整条指令都执行完了，才能响应中断（因为中断查询是在每个机器周期都可能查到的）。

当前正执行的指令是返回批令（RETI）或访问 IP、IE 寄存器的指令，则 CPU 至少再执行一条指令才应中断。这些都是与中断有关的，如果正访问 IP、IE 则可能会开、关中断或改变中断的优先级，而中断返回指令则说明本次中断还没有处理完，所以都要等本指令处理结束，再执行一条指令才能响应中断。

2、中断响应过程 CPU 响应中断时，首先把当前指令的下一条指令（就是中断返回后将要执行的指令）的地址送入堆栈，然后根据中断标记，将对应的中断入 口地址送入 PC，PC 是程序指针，CPU 取指令就根据 PC 中的值，PC 中是什么值，就会到什么地方去取指令，所以程序就会转到中断入口处继续执行。这些工 作都是由硬件来完成的，不必我们去考虑。这里还有个问题，大家是否注意到，每个中断向量地址只间隔了 8 个单元，如 0003－000B，在如此少的空间中如 何完成中断程序呢？很简单，你在中断处安排一个 LJMP 指令，不就能把中断程序跳转到任何地方了吗？一个完整的主程序看起来应该是这样的：

```
ORG 0000HLJMP START
```

ORG 0003H

LJMP INT0 ；转外中断 0ORG 000BH

RETI ；没有用定时器 0 中断，在此放一条 RETI，万一 “不小心“产生了中断，也不会有太大的后果。。

中断程序完成后，一定要执行一条 RETI 指令，执行这条指令后，CPU 将会把堆栈中保存着的地址取出，送回 PC，那么程序就会从主程序的中断处继续往 下执行了。注意：CPU 所做的保护工作是很有限的，只保护了一个地址，而其它的所有东西都不保护，所以如果你在主程序中用到了如 A、PSW 等，在中断程序 中又要用它们，还要保证回到主程序后这里面的数据还是没执行中断以前的数据，就得自己保护起来。

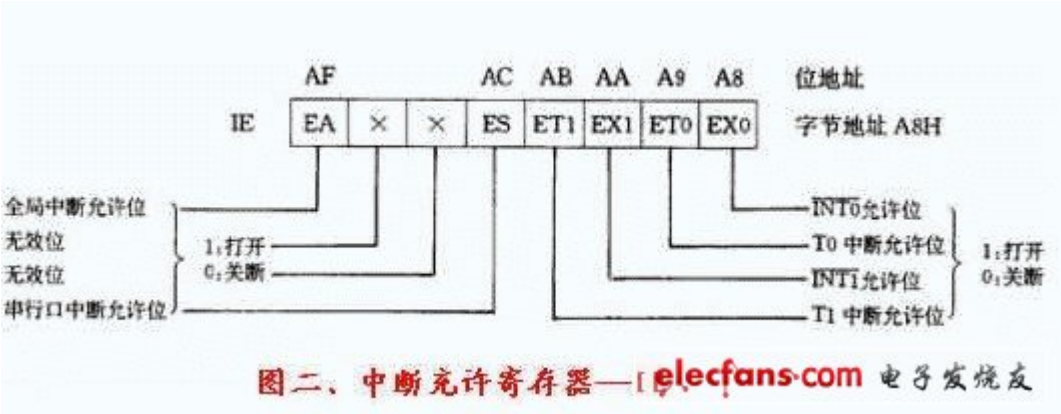
中断系统的控制寄存器：

中断系统有两个控制寄存器 IE 和 IP，它们分别用来设定各个中断源的打开 / 关闭和中断优先级。此外，在 TCON 中另有 4 位用于选择引起外部中断的条件并作为标志位。

1．中断允许寄存器--IE

IE 在特殊功能寄存器中，字节地址为 A8H，位地址（由低位到高位）分别是 A8H-AFH。

IE 用来打开或关断各中断源的中断请求，基本格式如下图二所示：



图二、中断允许寄存器—IE elecfans.com 电子发烧友

EA：全局中断允许位。EA＝0，关闭全部中断；EA＝1，打开全局中断控制，在此条件下，由各个中断控制位确定相应中断的打开或关闭。

×：无效位。

ES：串行 I／O 中断允许位。ES＝1，打开串行 I／O 中断；ES＝0，关闭串行 I／O 中断。

ET1；定时器／计数器 1 中断允许位。ET1＝1，打开 T1 中断；ET1＝0，关闭 T1 中断。

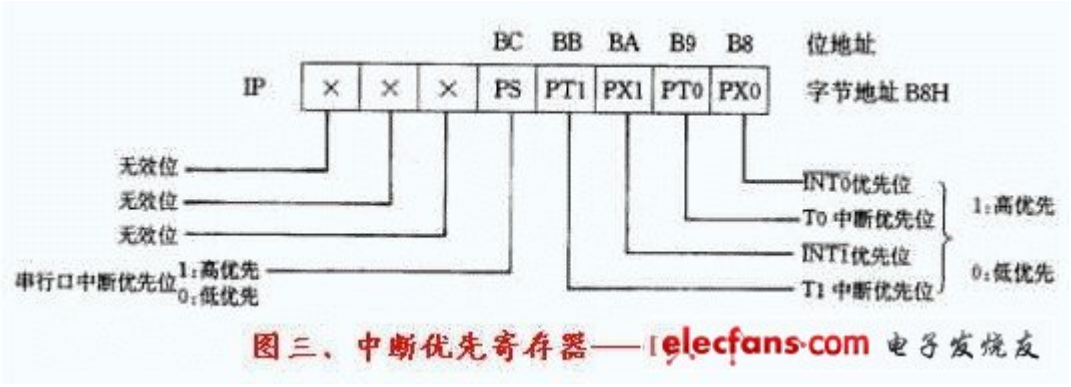
EX1：外部中断 1 中断允许位。EX1＝1，打开 INT1；EX1＝0，关闭 INT1。

ET0：定时器／计数器 0 中断允许位。ET0＝1，打开 T0 中断；ET0＝0，关闭 T0 中断。

EX0：外部中断 0 中断允许位。Ex0＝1，打开 INT0;EX0=0，关闭 INT0.

中断优先寄存器--IP：

IP 在特殊功能寄存器中，字节地址为 B8H，位地址（由低位到高位）分别是 B8H 一 BFH，IP 用来设定各个中断源属于两级中断中的哪一级，IP 的基本格式如下图三所示：



×：无效位。

PS：串行 I／O 中断优先级控制位。PS＝1，高优先级；PS＝0，低优先级。

PT1：定时器／计数器 1 中断优先级控制位。PT1＝1，高优先级；PT1＝0，低优先级。

Px1：外部中断 1 中断优先级控制位。Px1＝1，高优先级；PX1＝0，低优先级。

PT0：定时器／计数器 o 中断优先级控制位。PT0＝1，高优先级；PT0＝0，低优先级。

Px0：外部中断 0 中断优先级控制位。Px0＝1，高优先级；Px0＝0，伤优先级。

在 MCS-51 单片机系列中，高级中断能够打断低级中断以形成中断嵌套；同级中断之间，或低级对高级中断则不能形成中断嵌套。若几个同级中断同时向 CPU 请求中断响应，则 CPU 按如下顺序确定响应的先后顺序：

INT0 一 T0---INT1 一 T1 一 RI／T1.

中断的响应过程

若某个中断源通过编程设置，处于被打开的状态，并满足中断响应的条件，而且①当前正在执行的那条指令已被执行完

1、当前未响应同级或高级中断

2、不是在操作 IE，IP 中断控制寄存器或执行 REH 指令则单片机响应此中断。

在正常的情况下，从中断请求信号有效开始，到中断得到响应，通常需要 3 个机器周期到 8 个机器周期。中断得到响应后，自动清除中断请求标志（对串行 I／0 端口的中断标志，要用软件清除），将断点即程序计数器之值（PC）压入堆栈（以备恢复用）；然后把相应的中断入口地址装入 PC，使程序转入到相应的 中断服务程序中去执行。

各个中断源在程序存储器中的中断入口地址如下：

中断源 入口地址

INT0（外部中断 0） 0003H

TF0（T0 中断） 000BH

INT1（外部中断 1） 0013H

TF1（T1 中断） 001BH

RI／TI（串行口中断） 0023H

由于各个中断入口地址相隔甚近，不便于存放各个较长的中断服务程序，故通常在中断入口地址开始的二三个单元中，安排一条转移类指令，以转入到安排在那儿的中断服务程序。以 T1 中断为例，其过程下如图四所示。

由于 5 个中断源各有其中断请求标志 0，TF0，IE1，TF1 以及 RI／TI，在中断源满足中断请求的条件下，各标志自动置 1，以向 CPU 请求中断。 如果某一中断源提出中断请求

后，CPU 不能立即响应，只要该中断请求标志不被软件人为清除，中断请求的状态就将一直保持，直到 CPU 响应了中断为止，对串 行口中断而言，这一过程与其它 4 个中断的不同之处在于；即使 CPU 响应了中断，其中断标志 RI / TI 也不会自动清零，必须在中断服务程序中设置清除 RI / TI 的指令后，才会再一次地提出中断请求。

CPU 的现场保护和恢复必须由被响应的相应中断服务程序去完成，当执行 RETI 中断返回指令后，断点值自动从栈顶 2 字节弹出，并装入 PC 寄存器，使 CPU 继续执行被打断了的程序。

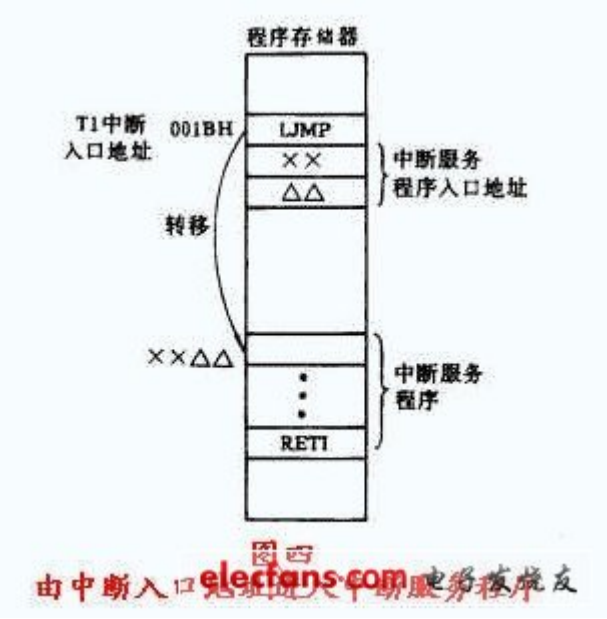
下面给出一个应用定时器中断的实例。

现要求编制一段程序，使 P1．0 端口线上输出周期为 2ms 的方波脉冲。设单片机晶振频率

Fosc＝6MHZ.

1、方法：利用定时器 T0 作 1ms 定时，达到定时值后引起中断，在中断服务程序中，使 P1．0 的状态取一次反，并再次定时 1ms。

2、定时初值:机器周期MC＝12/fosc＝2us。所以定时 1ms 所需的机器周期个数为 500D，亦即 01F4H。设 T0 为工作方式 1（16 位方式），则定时初值是（01F4H）求补＝FE0CH



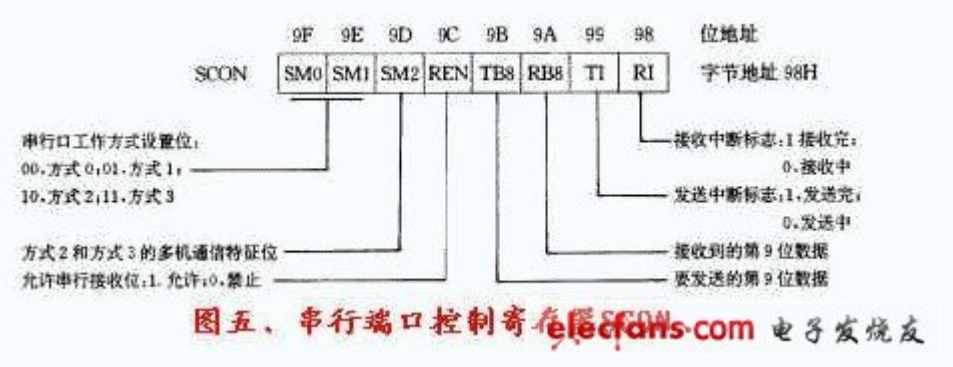
串行端口的控制寄存器：

```
START:  MOV TMOD, #01H   ; T0为定时器状态，工作方式1
        MOV TL0, #0CH    ; T0的低位定时初值
        MOV TH0, #0FEH   ; T0的高位定时初值
        MOV TCON, #10H   ; 打开T0
        SETB ET0         ; 1ET0，即允许T0中断
        SETB EA          ; 1EA，即充许全局中断
        AJMP $           ; 动态暂存
000BH:  AJMP IST0        ; 转入T0中断服务程序入口地址IST0
IST0:   MOV TL0, #0CH    ; 重置定时器初值
        MOV TH0, #0FEH   ; 重置定时器初值
        CPL P1.0         ; P1.0取反
        RET1             ; 中断返回
```


串行端口共有 2 个控制寄存器 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、波特率的大小，以及作为运行的中断标志等。

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 的格式如图五所示。



图五、串行端口控制寄存器 SCON

SMo，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 的多机通讯控制位

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：

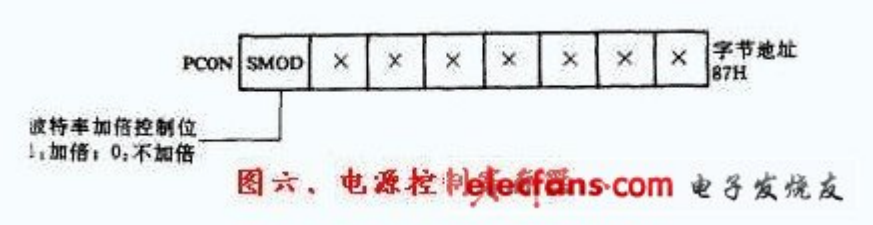
发送中断标志位。发送前必须用软件清零，发送过程中 TI 保持零电平，发送完一帧数据后，由硬件自动置 1。如要再发送，必须用软件再清零。

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。

电源控制寄存器 PCON

PCON 的字节地址为 87H，无位地址，PCON 的格式如图六所示。需指出的是，对 80C31 单片机而言，PCON 还有几位有效控制位。



图六、电源控制寄存器 elecfans.com 电子发烧友

SMOD：波特率加倍位。在计算串行方式 1，2，3 的波特率时；0---不加倍；1---加倍。

串行中断的应用特点：

8031 单片机的串行 I／O 端口是一个中断源，有两个中断标志 RI 和 TI，RI 用于接收，TI 用于发送。

串行端口无论在何种工作方式下，发送／接收前都必须对 TI／RI 清零。当一帧数据发送／接收完后，TI/RI 自动置 1，如要再发送／接收，必须先用软件将其清除。

在串行中断被打开的条件下，对方式 0 和方式 1 来说，一帧数据发送／接收完后，除置位 TI／RI 外，还会引起串行中断请求，并执行串行中侧目务程序。但对方式 2 和方式 3 的接收机而言，还要视 SM2 和 RB8 的状态，才可确定 RI 是否被置位以及串行中断的开放：

SM2 RB8 接收机中断标志与中断状态

0 1 激活 RI，引起中断

1 0 不激活 RI，不引起中断

1 1 激活 RI，引起中断

单片机正是利用方式 2，3 的这一特点，实现多机间的通信。串行端口的常用应用方法见相关章节。

波特率的确定：

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

对方式 2 而言，波特率的计算式为 2SMOD • fosc / 64。当 SMOD＝0 时，波特率为 fm / 64；当 SMOD＝1 时，波特率为 fosc / 32。在此方式下，程控设置 SMOD 位的状态后，波特率就确定了，不需要再作其它设置。

对方式 1 和方式 3 来说，波特率的计算式为 2SMOD / 32×T1 溢出率，根据 SMOD 状态位的不同，波特率有 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，因而波特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。由于在这种方式下，T1 的溢出率（次 / 秒）计算式可表达成：

$$T1\ 溢出率=\frac{f_{osc}}{12(256-x)}$$

式中 x 为设置的定时初值，于是波特率(位/秒)表达式为：

$$BR=\frac{2^{SMOD}}{32}\times\frac{f_{osc}}{12(256-x)}$$

由上式可见，选取不同的 x 初值，就可得到不同的波特率。

把上式变换一下形式，就可根据所要求的波特率 BR，算出 T1 定时器初值的大小来，其计算式为：

$$x=256-\frac{2^{SMOD}\cdot f_{osc}}{384\cdot BR}$$

例如，以产生 1200 位/秒为例，求定时器 T1 定时初值 x 的大小。

设 $f_{osc}=6MHz$，SMOD=0，则

$$1200/秒=\frac{1}{32}\times\frac{6\times10^{6}Hz}{12(256-x)}$$

解得

$$x=243D=F3H$$

下面一段主程序和中断服务程序，是利用串行方式 1 从数据 00H 开始连续不断增大地串行发送一片数据的程序例。设单片机晶振的频率为 6MHZ，波特率为 1200 位 / 秒。

```
ORG 2000H          ;1200位/秒的定时器初值
MOV TL1,#0F3H
MOV TH1,#0F3H      ;使SMOD=0
MOV PCON,#00H      ;T1方式2
MOV TMOD,#20H
SETB EA
CLR ET1            ;关闭T1中断
SETB ES            ;开串行中断
SETB TR1           ;开T1定时
MOV SCON,#40H      ;串行方式1
CLR A
MOV SBUF,A         ;串行发送
JNB T1,$           ;等待发送完
CLR T1,            ;清标志
SJMP $
ORG 0023H          ;串行中断入口地址
MOV SBUF,A         ;连续发送
JNB T1,$
INC A
CLR T1
RET1               ;中断返回
```


19、单片机定时器、中断试验

我们在学单片机时我们第一个例程就是灯的闪烁，那是用延时程序做的，现在回想起来，这样做不很恰当，为什么呢？我们的主程序做了灯的闪烁，就不能再干其它的事了，难道单片机只能这样工作吗？当然不是，我们能用定时器来实现灯的闪烁的功能。

例 1：查询方式

```
ORG 0000H

AJMP START

ORG 30H

START:

MOV P1，#0FFH ;关所 灯

MOV TMOD，#00000001B ;定时/计数器 0 工作于方式 1

MOV TH0，#15H

MOV TL0，#0A0H ;即数 5536

SETB TR0 ;定时/计数器 0 开始运行

LOOP:JBC TF0，NEXT ;如果 TF0 等于 1，则清 TF0 并转 NEXT 处

AJMP LOOP ;不然跳转到 LOOP 处运行

NEXT:CPL P1.0

MOV TH0，#15H
```

```
MOV TL0，#9FH;重置定时/计数器的初值

AJMP LOOP

END AJMP LOOP

END
```

键入程序，看到了什么？灯在闪烁了，这可是用定时器做的，不再是主程序的循环了。简单地分析一下程序，为什么用 JBC 呢？TF0 是定时/计数器 0 的溢 出标记位，当定时器产生溢出后，该位由 0 变 1，所以查询该位就可知字时时间是否已到。该位为 1 后，要用软件将标记位清 0，以便下一次定时是间到时该位由 0 变 1，所以用了 JBC 指令，该指位在判 1 转移的同时，还将该位清 0。

以上程序是能实现灯的闪烁了，可是主程序除了让灯闪烁外，还是不能做其他的事啊！不，不对，我们能在 LOOP：……和 AJMP LOOP 指令之间插入一些指令来做其他的事情，只要保证执行这些指令的时间少于定时时间就行了。那我们在用软件延时程序的时候不是也能用一些指令来替代 DJNZ 吗？是的，但是那就要求你精确计算所用指令的时间，然后再减去对应的 DJNZ 循环次数，很不方便，而现在只要求所用指令的时间少于定时时间就行，显然要求低了。当然，这样的办法还是不好，所以我们常用以下的办法来实现。

程序 2：用中断实现

```
ORG 0000H ， http://www.51hei.com

AJMP START

ORG 000BH ;定时器 0 的中断向量地址

AJMP TIME0 ;跳转到真正的定时器程序处

ORG 30H
```

```
START:
MOV P1, #0FFH ;关所 灯
MOV TMOD, #00000001B ;定时/计数器 0 工作于方式 1
MOV TH0, #15H
MOV TL0, #0A0H ;即数 5536
SETB EA ;开总中断允许
SETB ET0 ;开定时/计数器 0 允许
SETB TR0 ;定时/计数器 0 开始运行
LOOP: AJMP LOOP ;真正工作时，这里可写任意程序
TIME0: ;定时器 0 的中断处理程序
PUSH ACC
PUSH PSW ;将 PSW 和 ACC 推入堆栈保护
CPL P1.0
MOV TH0, #15H
MOV TL0, #0A0H ;重置定时常数
POP PSW
```

```
POP ACC

RETI

END
```

上面的例程中，定时时间一到，TF0 由 0 变 1，就会引发中断，CPU 将自动转至 000B 处寻找程序并执行，由于留给定时器中断的空间只有 8 个字节，显 然不足以写下所有有中断处理程序，所以在 000B 处安排一条跳转指令，转到实际处理中断的程序处，这样，中断程序能写在任意地方，也能写任意长度了。进入 定时中断后，首先要保存当前的一些状态，程序中只演示了保存存 ACC 和 PSW，实际工作中应该根据需要将可能会改变的单元的值都推入堆栈进行保护（本程序 中实际不需保存护任何值，这里只作个演示）。

上面的两个单片机程序运行后，我们发现灯的闪烁非常快，根本分辨不出来，只是视觉上感到灯有些晃动而已，为什么呢？我们能计算一下，定时器中预置的数 是 5536，所以每计 60000 个脉冲就是定时时间到，这 60000 个脉冲的时间是多少呢？我们的晶体震荡器是 12M，所以就是 60000 微秒，即 60 毫 秒，因此速度是非常快的。如果我想实现一个 1S 的定时，该怎么办呢？在该晶体震荡器濒率下，最长的定时也就是 65。536 个毫秒啊！上面给出一个例程。

```
ORG 0000H

AJMP START

ORG 000BH ;定时器 0 的中断向量地址

AJMP TIME0 ;跳转到真正的定时器程序处

ORG 30H

START:
```

```
MOV P1，#0FFH ;关所 灯
MOV 30H，#00H ;软件计数器预清 0
MOV TMOD，#00000001B ;定时/计数器 0 工作于方式 1
MOV TH0，#3CH
MOV TL0，#0B0H ;即数 15536
SETB EA ;开总中断允许
SETB ET0 ;开定时/计数器 0 允许
SETB TR0 ;定时/计数器 0 开始运行
LOOP： AJMP LOOP ;真正工作时，这里可写任意程序
TIME0： ;定时器 0 的中断处理程序
PUSH ACC
PUSH PSW ;将 PSW 和 ACC 推入堆栈保护
INC 30H
MOV A，30H
CJNE A，#20，T_RET ;30H 单元中的值到了 20 了吗？
T_L1： CPL P1.0 ;到了，取反 P10
```

```
MOV 30H，#0 ;清软件计数器
T_RET：
MOV TH0，#15H
MOV TL0，#9FH ;重置定时常数
POP PSW
POP ACC
RETI
END
```

先自己分析一下，看看是怎么实现的？这里采用了软件计数器的概念，思路是这样的，先用定时/计数器 0 做一个 50 毫秒的定时器，定时是间到了以后并不是 立即取反 P10，而是将软件计数器中的值加 1，如果软件计数器计到了 20，就取反 P10，并清掉软件计数器中的值，不然直接返回，这样，就变成了 20 次定 时中断才取反一次 P10，因此定时时间就延长了成了 20*50 即 1000 毫秒了。

这个思路在工程中是非常有用的，有的时候我们需要若干个定时器，可 51 中总共才有 2 个，怎么办呢？其实，只要这几个定时的时间有一定的公约数，我们就 能用软件定时器加以实现，如我要实现 P10 口所接灯按 1S 每次，而 P11 口所接灯按 2S 每次闪烁，怎么实现呢？对了我们用两个计数器，一个在它计到 20 时，取反 P10，并清零，就如上面所示，另一个计到 40 取反 P11，然后清 0，不就行了吗？这部份的程序如下

```
ORG 0000H
AJMP START
```

```
ORG 000BH ;定时器 0 的中断向量地址
AJMP TIME0 ;跳转到真正的定时器程序处
ORG 30H
START:
MOV P1, #0FFH ;关所 灯
MOV 30H, #00H ;软件计数器预清 0
MOV TMOD, #00000001B ;定时/计数器 0 工作于方式 1
MOV TH0, #3CH
MOV TL0, #0B0H ;即数 15536
SETB EA ;开总中断允许
SETB ET0 ;开定时/计数器 0 允许
SETB TR0 ;定时/计数器 0 开始运行
LOOP: AJMP LOOP ;真正工作时，这里可写任意程序
TIME0: ;定时器 0 的中断处理程序
PUSH ACC
PUSH PSW ;将 PSW 和 ACC 推入堆栈保护
```

```
INC 30H
INC 31H ;两个计数器都加 1
MOV A，30H
CJNE A，#20，T_NEXT ;30H 单元中的值到了 20 了吗?
T_L1： CPL P1.0 ;到了，取反 P10
MOV 30H，#0 ;清软件计数器
T_NEXT:
MOV A，31H
CJNE A，#40，T_RET ;31h 单元中的值到 40 了吗?
T_L2:
CPL P1.1
MOV 31H，#0 ;到了，取反 P11，清计数器，返回
T_RET:
MOV TH0，#15H
MOV TL0，#9FH ;重置定时常数
POP PSW
```

```
POP ACC

RETI

END
```

您能用定时器的办法实现前面讲的流水灯吗？试试看。

20、单片机定时/计数器实验

前面我们做了定时器的实验，现在来看一看计数实验，在工作中计数常常会有两种要求：第一、将计数的值显示出来，第二、计数值到一定程度即中断报警。第 一种如各种计数器、里程表，第二种如前面例中讲到的生产线上的计数。先看第一种吧。我们的硬件中是这样连线的：324 组成的振荡器连到定时/计数器 1 的外 部管脚 T1 上面，我们就利用这个来做一个计数实验，要将计数的值显示出来，当然最好用数码管了，可我们还没讲到这一部份，为了避免把问题复杂化，我们用 P1 口的 8 个 LED 来显示计到的数据。

程序如下：

```
ORG 0000H ， http://www.51hei.com

AJMP START

ORG 30H

START：

MOV SP，#5FH

MOV TMOD，#01000000B ;定时/计数器 1 作计数用，0 不用全置 0

SETB TR1 ;启动计数器 1 开始运行。
```

```
LOOP:  MOV A, TL0

MOV P1, A

AJMP LOOP

END
```

在硬件上用线将 324 的输出与 T1 连通（印板上有焊盘）运行这种程序，注意将板按正确的位置放置（LM324 放在左手边，LED 排列是按从高位到低们排列）看到什么？随着 324 后接的 LED 的闪烁，单片机的 8 只 LED 也在持续变化，注意观察，是不是按二进制：

```
00000000

00000001

00000010

00000011
```

这样的次序在变呢？这就对了，这就是 TL0 中的数据。

程序二：

```
ORG 0000H

AJMP START

ORG 001BH

AJMP TIMER1 ;定时器 1 的中断处理
```

```
ORG 30H
START:  MOV SP, #5FH
MOV TMOD, #01010000B ;定时/计数器 1 作计数用，模式 1，0 不用全置 0
MOV TH1, #0FFH
MOV TL1, #0FAH ;预置值，要求每计到 6 个脉冲即为一个事件
SETB EA
SETB ET1 ;开总中断和定时器 1 中断允许
SETB TR1 ;启动计数器 1 开始运行。
AJMP $
TIMER1:
PUSH ACC
PUSH PSW
CPL P1.0 ;计数值到，即取反 P1.0
MOV TH1, #0FFH
MOV TL1, #0FAH ;重置计数初值
POP PSW
```

```
POP ACC

RETI

END
```

上面这个单片机程序完成的工作很简单，就是在每 6 个脉冲到来后取反一次 P1。0，因此实验的结果应当是：LM324 后接的 LED 亮、灭 6 次，则 P1。0 口所接 LED 亮或灭一次。这实际就是我们上面讲的计数器的第二种应用。

程序三：外部中断实验

```
ORG 0000H

AJMP START

ORG 0003H ;外部中断地直入口

AJMP INT0

ORG 30H

START： MOV SP，#5FH

MOV P1，#0FFH ;灯全灭

MOV P3，#0FFH ;P3 口置高电平

SETB EA

SETB EX0
```

```
AJMP $

INT0:

PUSH ACC

PUSH PSW

CPL P1.0

POP PSW

POP ACC

RETI

END
```

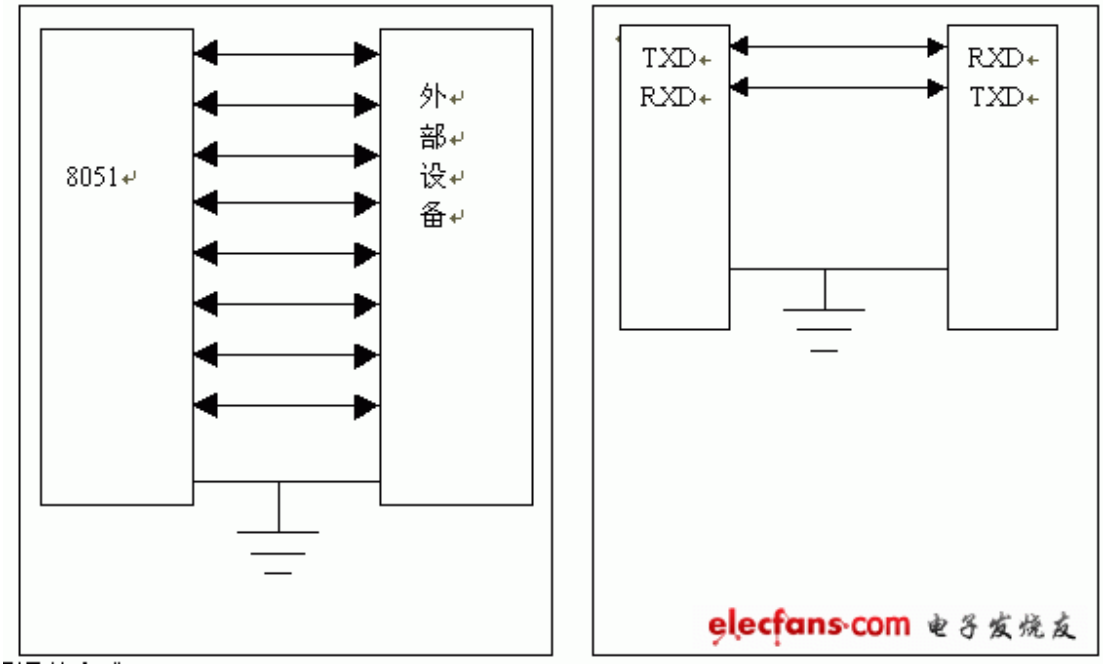
本程序的功能很简单，按一次按钮 1（接在 12 管脚上的）就引发一次中断 0，取反一次 P1。0，因此理论上按一下灯亮，按一下灯灭，但在实际做实验时， 可能会发觉有时不“灵”，按了它没反应，但在大部份时候是对的，这是怎么回事呢？我们在讲解键盘时再作解释，这个程序本身是没有问题的。

21、单片机串行口介绍

介绍：串行口是单片机与外界进行信息交换的工具。

8051 单片机的通信方式有两种：

并行通信：数据的各位同时发送或接收。 串行通信：数据一位一位次序发送或接收。参看下图：



串行通信的方式：

异步通信：它用一个起始位表示字符的开始，用停止位表示字符的结束。其每帧的格式如下：

在一帧格式中，先是一个起始位 0，然后是 8 个数据位，规定低位在前，高位在后，接下来是奇偶校验位（能省略），最后是停止位 1。用这种格式表示字符，则字符能一个接一个地传送。

在异步通信中，CPU 与外设之间必须有两项规定，即字符格式和波特率。字符格式的规定是双方能够在对同一种 0 和 1 的串理解成同一种意义。原则上字符格式能由通信的双方自由制定，但从通用、方便的角度出发，一般还是使用一些标准为好，如采用 ASCII 标准。

波特率即数据传送的速率，其定义是每秒钟传送的二进制数的位数。例如，数据传送的速率是 120 字符/s，而每个字符如上述规定包含 10 数位，则传送波特率为 1200 波特。

同步通信：在同步通信中，每个字符要用起始位和停止位作为字符开始和结束的标志，占用了时间；所以在数据块传递时，为了提高速度，常去掉这些标志，采用同步传送。由于数据块传递开始要用同步字符来指示，同时要求由时钟来实现发送端与接收端之间的同步，故硬件较复杂。

通信方向：在串行通信中，把通信接口只能发送或接收的单向传送办法叫单工传送；而把数据在甲乙两机之间的双向传递，称之为双工传送。在双工传送方式中又分为半双工传送和全双工传送。半双工传送是两机之间不能同时进行发送和接收，任一时该，只能发或者只能收信息。

2．8051 单片机的串行接口结构

8051 单片机串行接口是一个可编程的全双工串行通信接口。它可用作异步通信方式（UART），与串行传送信息的外部设备相连接，或用于通过标准异步通信协议进行全双工的 8051 多机系统也能通过同步方式，使用 TTL 或 CMOS 移位寄存器来扩充 I/O 口。

8051 单片机通过管脚 RXD（P3.0，串行数据接收端）和管脚 TXD（P3.1，串行数据发送端）与外界通信。SBUF 是串行口缓冲寄存器，包括 发送寄存器和接收寄存器。它们有相同名字和地址空间，但不会出现冲突，因为它们两个一个只能被 CPU 读出数据，一个只能被 CPU 写入数据。

串行口的控制与状态寄存器

串行口控制寄存器 SCON

它用于定义串行口的工作方式及实施接收和发送控制。字节地址为 98H，其各位定义如下表：

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

SM0、SM1：串行口工作方式选择位，其定义如下：

SM0、SM1	工作方式	功能描述	波特率
0 0	方式0	8位移位寄存器	Fosc/12
0 1	方式1	10位UART	可变
1 0	方式2	11位UART	Fosc/64或fosc/32
1 1	方式3	11位UART	可变

其中fosc为晶体震荡器频率

SM2：多机通信控制位。在方式 0 时，SM2 一定要等于 0。在方式 1 中，当（SM2）=1 则只有接收到有效停止位时，RI 才置 1。在方式 2 或方式 3 当（SM2）=1 且接收到的第九位数据 RB8=0 时，RI 才置 1。

REN：接收允许控制位。由软件置位以允许接收，又由软件清 0 来禁止接收。

TB8： 是要发送数据的第 9 位。在方式 2 或方式 3 中，要发送的第 9 位数据，根据需要由软件置 1 或清 0。例如，可约定作为奇偶校验位，或在多机通信中作为区别地址帧或数据帧的标志位。

RB8：接收到的数据的第 9 位。在方式 0 中不使用 RB8。在方式 1 中，若（SM2）=0，RB8 为接收到的停止位。在方式 2 或方式 3 中，RB8 为接收到的第 9 位数据。

TI：发送中断标志。在方式 0 中，第 8 位发送结束时，由硬件置位。在其它方式的发送停止位前，由硬件置位。TI 置位既表示一帧信息发送结束，同时也是 申请中断，可根据需要，用软件查询的办法获得数据已发送完毕的信息，或用中断的方式来发送下一个数据。TI 必须用软件清 0。

RI：接收中断标志位。在方式 0，当接收完第 8 位数据后，由硬件置位。在其它方式中，在接收到停止位的中间时刻由硬件置位（例外情况见于 SM2 的说明）。RI 置位表示一帧数据接收完毕，可用查询的办法获知或者用中断的办法获知。RI 也必须用软件清 0。

特殊功能寄存器 PCON

PCON 是为了在 CHMOS 的 80C51 单片机上实现电源控制而附加的。其中最高位是 SMOD。

串行口的工作方式

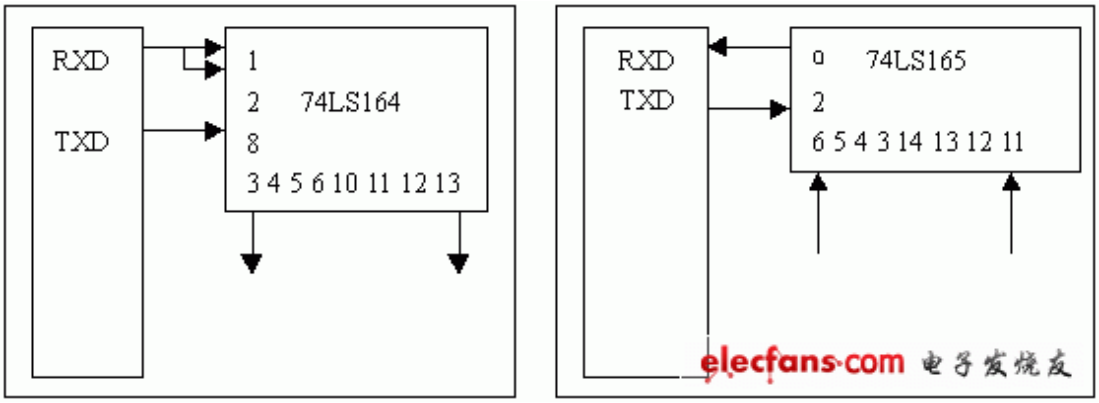
8051 单片机的全双工串行口可编程为 4 种工作方式，现分述如下：

方式 0 为移位寄存器输入/输出方式。可外接移位寄存器以扩展 I/O 口，也能外接同步输入/输出设备。8 位串行数据者是从 RXD 输入或输出，TXD 用来输出同步脉冲。

输出 串行数据从 RXD 管脚输出，TXD 管脚输出移位脉冲。CPU 将数据写入发送寄存器时，立即启动发送，将 8 位数据以 fos/12 的固定波特率从 RXD 输出，低位在前，高位在后。发送完一帧数据后，发送中断标志 TI 由硬件置位。

输入 当串行口以方式 0 接收时，先置位允许接收控制位 REN。此时，RXD 为串行数据输入端，TXD 仍为同步脉冲移位输出端。当（RI）=0 和（REN）=1 同时满足时，开始接收。当接收到第 8 位数据时，将数据移入接收寄存器，并由硬件置位 RI。

下面两图分别是方式 0 扩展输出和输入的接线图。



《单片机串行口接线图》

方式 1 为波特率可变的 10 位异步通信接口方式。发送或接收一帧信息，包括 1 个起始位 0，8 个数据位和 1 个停止位 1。

输出 当 CPU 执行一条指令将数据写入发送缓冲 SBUF 时，就启动发送。串行数据从 TXD 管脚输出，发送完一帧数据后，就由硬件置位 TI。

输入 在（REN）=1 时，串行口采样 RXD 管脚，当采样到 1 至 0 的跳变时，确认是开始位 0，就开始接收一帧数据。只有当（RI）=0 且停止位为 1 或者 （SM2）=0 时，停止位才进入 RB8，8 位数据才能进入接收寄存器，并由硬件置位中断标志 RI；不然信息丢失。所以在方式 1 接收时，应先用软件清零 RI 和 SM2 标志。

方式 2

方式月为固定波特率的 11 位 UART 方式。它比方式 1 增加了一位可程控为 1 或 0 的第 9 位数据。

输出： 发送的串行数据由 TXD 端输出一帧信息为 11 位，附加的第 9 位来自 SCON 寄存器的 TB8 位，用软件置位或复位。它可作为多机通信中地址/数据信息的标志 位，也能作

为数据的奇偶校验位。当 CPU 执行一条数据写入 SUBF 的指令时，就启动发送器发送。发送一帧信息后，置位中断标志 TI。

输入： 在（REN）=1 时，串行口采样 RXD 管脚，当采样到 1 至 0 的跳变时，确认是开始位 0，就开始接收一帧数据。在接收到附加的第 9 位数据后，当（RI）=0 或者（SM2）=0 时，第 9 位数据才进入 RB8，8 位数据才能进入接收寄存器，并由硬件置位中断标志 RI；不然信息丢失。且不置位 RI。再过一位时间后， 不管上述条件时否满足，接收电路即行复位，并重新检测 RXD 上从 1 到 0 的跳变。

工作方式 3

方式 3 为波特率可变的 11 位 UART 方式。除波特率外，其余与方式 2 相同。

波特率选择

如前所述，在串行通信中，收发双方的数据传送率（波特率）要有一定的约定。在 8051 串行口的四种工作方式中，方式 0 和 2 的波特率是固定的，而方式 1 和 3 的波特率是可变的，由定时器 T1 的溢出率控制。

方式 0

方式 0 的波特率固定为主振频率的 1/12。

方式 2

方式 2 的波特率由 PCON 中的选择位 SMOD 来决定，可由下式表示：

波特率=2 的 SMOD 次方除以 64 再乘一个 fosc，也就是当 SMOD=1 时，波特率为 1/32fosc，当 SMOD=0 时， 波特率为 1/64fosc

3．方式 1 和方式 3

定时器 T1 作为波特率发生器，其公式如下：

波特率= $\frac{2^{smod}}{32}\times$ 定时器 T1 溢出率

T1 溢出率= T1 计数率/产生溢出所需的周期数

式中 T1 计数率取决于它工作在定时器状态还是计数器状态。当工作于定时器状态时，T1 计数率为 fosc/12;当工作于计数器状态时，T1 计数率为外部输入频率，此频率应小于 fosc/24。产生溢出所需周期与定时器 T1 的工作方式、T1 的预置值有关。

定时器 T1 工作于方式 0：溢出所需周期数=8192-x

定时器 T1 工作于方式 1：溢出所需周期数=65536-x

定时器 T1 工作于方式 2：溢出所需周期数=256-x

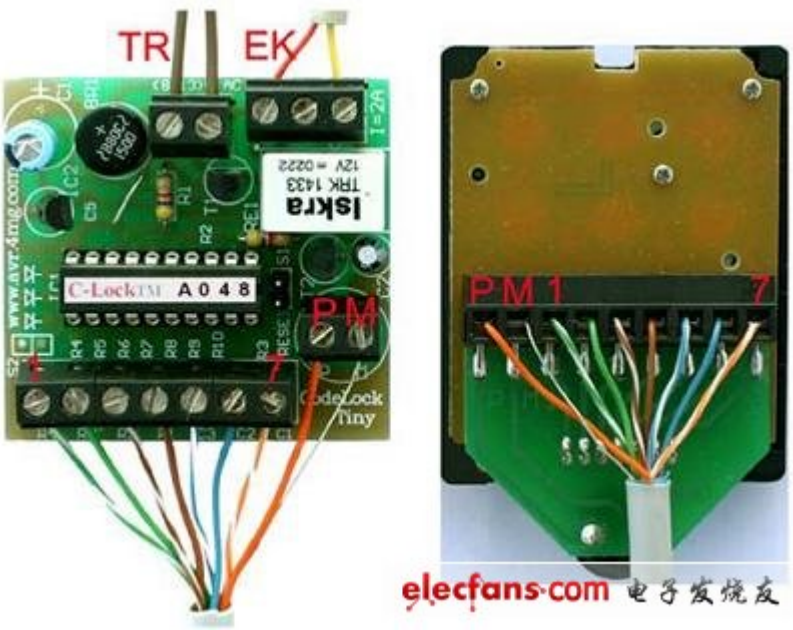
因为方式 2 为自动重装入初值的 8 位定时器/计数器模式，所以用它来做波特率发生器最恰当。

当时钟频率选用 11.0592MHZ 时，取易获得标准的波特率，所以很多单片机系统选用这个看起来“怪”的晶体震荡器就是这个道理。

下表列出了定时器 T1 工作于方式 2 常用波特率及初值。

常用波特率	Fosc(MHZ)	SMOD	TH1初值
19200	11.0592	1	FDH
9600	11.0592	0	FDH
4800	11.0592	0	FAH
2400	11.0592	0	F4h
1200	11.0592	0	

导语：本期是本次单片机学习知识点的最终回，我们会列出前三回一起方便读者回顾学习。本次主要知识点为单片机串口通信、接口和实际案例实践——单片机 音乐程序设计的学习。单片机对于初学者来说确实很难理解，不少学过单片机的同学或电子爱好者，甚至在毕业时仍旧是一无所获。基于此，电子发烧友网将整合《单片机学习知识点全攻略》，共分为四个系列，以飨读者，敬请期待！此系列对于业内电子工程师也有收藏和参考价值。



参阅相关系列

单片机学习知识点全攻略（一）

单片机学习知识点全攻略（二）

单片机学习知识点全攻略（三）

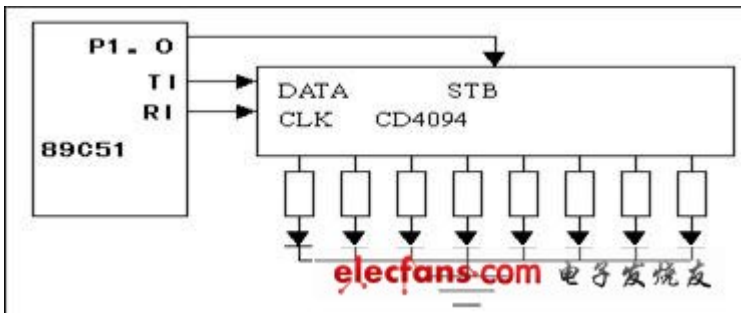
系列四

- 22：单片机串行口通信程序设计
- 23：LED 数码管静态显示接口与编

- 24：动态扫描显示接口电路及程序
- 25：单片机键盘接口程序设计
- 26：单片机矩阵式键盘接口技术及
- 27：关于单片机的一些基本概念
- 28：实际案例实践——单片机音乐程序设计

22、单片机串行口通信程序设计

1．串行口方式 0 应用编程 8051 单片机串行口方式 0 为移位寄存器方式，外接一个串入并出的移位寄存器，就能扩展一个并行口。



《单片机串行口通信程序设计硬件连接图》

例：用 8051 单片机串行口外接 CD4094 扩展 8 位并行输出口，如图所示，8 位并行口的各位都接一个发光二极管，要求发光管呈流水灯状态。 串行口方式 0 的数据传送可采用中断方式，也可采用查询方式，无论哪种方式，都要借助于 TI 或 RI 标志。串行发送时，能靠 TI 置位（发完一帧数据后）引起中 断申请，在中断服务程序中发送下一帧数据，或者通过查询 TI 的状态，只要 TI 为 0 就继续查询，TI 为 1 就结束查询，发送下一帧数据。在串行接收时，则由 RI 引起中断或对 RI 查询来确定何时接收下一帧数据。无论采用什么

方式，在开始通信之前，都要先对控制寄存器 SCON 进行初始化。在方式 0 中将，将 00H 送 SCON 就能了。

```
----------------单片机串行口通信程序设计列子-------------------------

ORG 2000H

START:  MOV SCON, #00H ;置串行口工作方式 0

MOV A, #80H ;最高位灯先亮

CLR P1.0 ;关闭并行输出（避象传输过程中，各 LED 的“暗红”现象）

OUT0:  MOV SBUF, A ;开始串行输出

OUT1:  JNB TI, OUT1 ;输出完否

CLR TI ;完了，清 TI 标志，以备下次发送

SETB P1.0 ;打开并行口输出

ACALL DELAY ;延时一段时间

RR A ;循环右移

CLR P1.0 ;关闭并行输出

JMP OUT0 ;循环
```

说明：DELAY 延时子程序能用前面我们讲 P1 口流水灯时用的延时子程序，这里就不给出了。

二、串行口异步通信

```
org 0000H
AJMP START
ORG 30H
START:
mov SP, #5fh ;
mov TMOD, #20h ;T1:  工作模式 2
mov PCON, #80h ;SMOD=1
mov TH1, #0FDH ;初始化波特率（参见表）
mov SCON, #50h ;Standard UART settings
MOV R0, #0AAH ;准备送出的数
SETB REN ;允许接收
SETB TR1 ;T1 开始工作
WAIT:
MOV A, R0
CPL A
```

```
MOV R0, A
MOV SBUF, A
LCALL DELAY
JBC TI, WAIT1 ;如果 TI 等于 1，则清 TI 并转 WAIT1
AJMP WAIT
WAIT1:  JBC RI, READ ;如果 RI 等于 1，则清 RI 并转 READ
AJMP WAIT1
READ:
MOV A, SBUF ;将取得的数送 P1 口
MOV P1, A
LJMP WAIT
DELAY:  ;延时子程序
MOV R7, #0ffH
DJNZ R7, $
RET
END
```

将程序编译通过，写入芯片，插入实验板，用通读电缆将实验板与主机的串行口相连就能实验了。上面的程序功能很简单，就是每隔一段时间向主机轮流送数 55H 和 AAH，并把主机送去的数送到 P1 口。能在 PC 端用串行口精灵来做实验。串行口精灵在我主页上有下载。运行串行口精灵后，按主界面上的“设置参 数”按钮进入“设置参数”对话框，按下面的参数进行设置。注意，我的机器上用的是串行口 2，如果你不是串行口 2，请自行更改串行口的设置。

设置完后，按确定返回主界面，注意右边有一个下拉列表，应当选中“按 16 进制”。然后按“开始发送”、“开始接收”就能了。按此设置，实验板上应当有 两只灯亮，6 只灯灭。大家能自行更改设置参数中的发送字符如 55，00，FF 等等，观察灯的亮灭，并分析原因，也能在主界面上更改下拉列表中的“按 16 进 制”为“按 10 进制”或“按 ASCII 字符”来观察现象，并仔细分析。这对于大家理解 16 进制、10 进制、ASCII 字符也是很有好处的。程序本身很简 单，又有注释，这里就不详加说明了。

三、上述程序的中断版本

```
org 0000H

AJMP START
```

```
org 0023h
AJMP SERIAL ;
ORG 30H
START:
mov SP, #5fh ;
mov TMOD, #20h ;T1:  工作模式 2
mov PCON, #80h ;SMOD=1
mov TH1, #0FDH ;初始化波特率（参见表）
mov SCON, #50h ;Standard UART settings
MOV R0, #0AAH ;准备送出的数
SETB REN ;允许接收
SETB TR1 ;T1 开始工作
SETB EA ;开总中断
SETB ES ;开串行口中断
SJMP $
SERIAL:
```

```
MOV A，SBUF

MOV P1，A

CLR RI

RETI

END
```

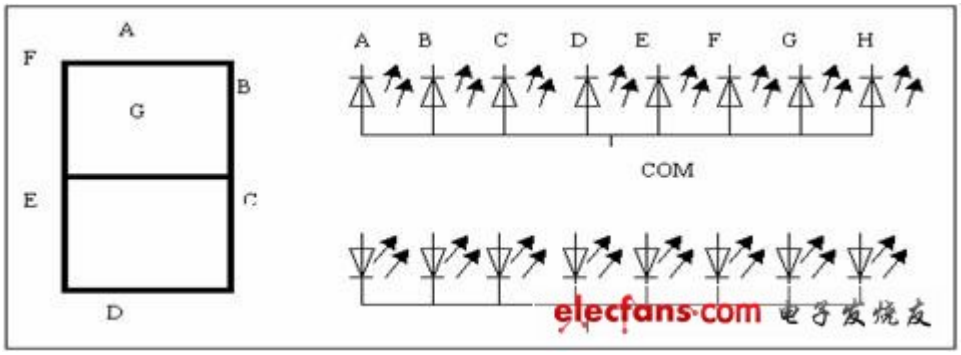
本程序没有写入发送程序，大家能自行添加。

23、LED 数码管静态显示接口与编程

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

引言：还记得我们小时候玩的“火柴棒游戏”吗，几根火柴棒组合起来，能拼成各种各样的图形，LED 数码管显示器实际上也是这么一个东西。

八段 LED 数码管显示器



《单片机静态显示接口》

八段 LED 数码管显示器由 8 个发光二极管组成。基中 7 个长条形的发光管排列成“日”字形，另一个贺点形的发光管在数码管显示器的右下角作为显示小数点 用，它能显示各种数字及部份英文字母。LED 数码管显示器有两种不一样的形式：一种是 8 个发光二极管的阳极都连在一起的，称之为共阳极 LED 数码管显示 器；另一种是 8 个发光二极管的阴极都连在一起的，称之为共阴极 LED 数码管显示器。如下图所示。`

共阴和共阳结构的 LED 数码管显示器各笔划段名和安排位置是相同的。当二极管导通时，对应的笔划段发亮，由发亮的笔划段组合而显示的各种字符。8 个笔 划段 hgfedcba 对应于一个字节（8 位）的 D7 D6 D5 D4 D3 D2 D1 D0，于是用 8 位二进制码就能表示欲显示字符的字形代码。例如，对于共阴 LED 数码管显示器，当公共阴极接地（为零电平），而阳极 hgfedcba 各段为 0111011 时，数码管显示器显示“P”字符，即对于共阴极 LED 数码管显示器，“P”字符的字形码是 73H。如果是共阳 LED 数码管显示器，公共阳极 接高电平，显示“P”字符的字形代码应为 10001100（8CH）。这里必须注意的是：很多产品为方便接线，常不按规则的办法去对应字段与位的关系，这 个时候字形码就必须根据接线来自行设计了，后面我们会给出一个例程。

在单片机应用系统中，数码管显示器显示常用两种办法：静态显示和动态扫描显示。所谓静态显示，就是每一个数码管显示器都要占用单独的具有锁存功能的 I/O 接口用于笔划段字形代码。这样单片机只要把要显示的字形代码发送到接口电路，就不用管它了，直到要显示新的数据时，再发送新的字形码，因此，使用这 种办法单片机中 CPU 的开销小。能供给单独锁存的 I/O 接口电路很多，这里以常用的串并转换电路 74LS164 为例，介绍一种常用静态显示电路，以使大家 对静态显示有一定的了解。

MCS-51 单片机串行口方式押为移们寄存器方式，外接 6 片 74LS164 作为 6 位 LED 数码管显示器的静态显示接口，把8031 的RXD 作为数据输 出线，TXD 作为移位时钟脉冲。74LS164 为 TTL 单向 8 位移位寄存器，可实现串行输入，并行输出。其中 A、B（第 1、2 脚）为串行数据输入端，2 个 管脚按逻辑与运算规律输入信号，公一个输入信号时可并接。T（第 8 脚）为时钟输入端，可连接到串行口的 TXD 端。每一个时钟信号的上升沿加到 T 端时，移位 寄存器移一位，8 个时钟脉冲过后，8 位二进制数全部移入 74LS164 中。R（第 9 脚）为

复位端，当 R=0 时，移位寄存器各位复 0，只有当 R=1 时，时钟 脉冲才起作用。Q1…Q8（第 3-6 和 10-13 管脚）并行输出端分别接 LED 数码管显示器的 hg---a 各段对应的管脚上。关于 74LS164 还能作如 下的介绍：所谓时钟脉冲端，其实就是需要高、低、高、低的脉冲，不管这个脉冲是怎么来的，比如，我们用根电线，一端接 T，一端用手拿着，分别接高电平、低 电平，那也是给出时钟脉冲，在 74LS164 获得时钟脉冲的瞬间（再讲清楚点，是在脉冲的沿），如果数据输入端（第 1，2 管脚）是高电平，则就会有一个 1 进入到 74LS164 的内部，如果数据输入端是低电平，则就会有一个 0 进入其内部。在给出了 8 个脉冲后，最先进入 74LS164 的第一个数据到达了最高 位，然后再来一个脉冲会有什么发生呢？再来一个脉冲，第一个脉冲就会从最高位移出，就象车站排队买票，栏杆就那么长，要从后面进去一本人，前面必须要从前 面走出去一本人才行。

搞清了这一点，下面让我们来看电路，6 片 7LS164 首尾相串，而时钟端则接在一起，这样，当输入 8 个脉冲时，从单片机 RXD 端输出的数据就进入到了 第一片 74LS164 中了，而当第二个 8 个脉冲到来后，这个数据就进入了第二片 74LS164，而新的数据则进入了第一片 74LS164，这样，当第六个 8 个脉冲完成后，首次送出的数据被送到了最左面的 164 中，其他数据依次出现在第一、二、三、四、五片 74LS164 中。有个问题，在第一个脉冲到来时， 除了第一片 74LS164 中接收数据外，其他各片在干吗呢？它们也在接收数据，因为它们的时钟端都是被接在一起的，可是数据还没有送到其他各片呢，它们在 接收什么数据呢？。。。。。。其实所谓数据不过是一种说法而已，实际就是电平的高低，当第一个脉冲到来时，第一片 164 固然是从单片机接收数据了，而其它 各片也接到前一片的 Q8 上，而 Q8 是一根电线，在数字电路中它只可能有两种状态：低电平或高电平，也就是“0”和“1”。所以它的下一片 74LS164 也 相当于是在接收数据啊。只是接收的全部是 0 或 1 而已。这个问题放在这儿说明，可能有朋友不屑一顾，而有的朋友可能还是不清楚，这实际上涉及到数的本质的问 题，如果不懂的，请仔细思考，并找一些数字电路的数，理解 164 的工作原理，再来看这个问题，或者去看看我的另一篇文章《27 课：关于单片机的一些基本概 念》的文章。务必搞懂，搞懂了这一点，你的级别就高过开始学习者，可谓入门者了。

入口：把要显示的数分别放在显示缓冲区 60H-65H 共 6 个单元中，并且分别对应各个数码管 LED0-LED5。

出口：将预置在显示缓冲区中的 6 个数成对应的显示字形码，然后输出到数码管显示器中显示。

单片机 led 显示程序如下：

```
DISP:  MOV SCON, #00H ;初始化串行口方式 0
MOV R1, #06H ;显示 6 位数
MOV R0, #65H ;60H-65H 为显示缓冲区
MOV DPTR, #SEGTAB ;字形表的入口地址
LOOP:
MOV A, @R0 ;取最高位的待显示数据
MOVC A, @A+DPTR ;查表获取字形码
MOV SBUF, A ;送串行口显示
DELAY:  JNB TI, DELAY ;等待发送完毕
CLR TI ;清发送标志
DEC R0 ;指针下移一位，准备取下一个待显示数
DJNZ R1, LOOP ;直到 6 个数据全显示完。
RET
SETTAB:  ;字形表，前面有介绍，以后我们再介绍字形表的制作。
```

```
DB 03H 9FH 27H 0DH 99H 49H 41H 1FH 01H 09H 0FFH

; 0 1 2 3 4 5 6 7 8 9 消隐码

单片机显示测试用主程序

ORG 0000H

AJMP START

ORG 30H

START:  MOV SP, #6FH

MOV 65H, #0

MOV 64H, #1

MOV 63H, #2

MOV 62H, #3

MOV 61H, #4

MOV 60H, #5

LCALL DISP

SJMP $
```

如果按图示数码管排列，则以上主程序将显示的是 543210，想想看，如果要显示 012345 该怎样送数？

下面我们来分析一下字形表的制作问题。先就上述“标准”的图形来看吧。写出数据位和字形的对应关系并列一个表如下（设为共阳型，也就是对应的输出位为 0 时笔段亮）

如何，字形表会做了吧，就是这样列个表格，根据要求（0 亮或 1 亮）写出对应位的 0 和 1，就成了。做个练习，写出 A-F 的字形码吧。

如果为了接线方便而打乱了接线的次序，那么字形表又该如何接呢？也很简单，一样地列表啊。以新实验板为例，共阳型。接线如下：

P0.7 P0.6 P0.5 P0.4 P0.3 P0.2 P0.1 P0.0

C E H D G F A B

则字形码如下所示：

;0 00101000 28H

;1 01111110 7EH

;2 10100100 0A4H

;3 01100100 64H

;4 01110010 72H

;5 01100001 61H

;6 00100001 21H

;7 01111100 7CH

;8 00100000 20H

;9 01100000 60H

作为练习，大家写出 A-F 的字形代码。

24、动态扫描显示接口电路及程序

在单片机系统中动态扫描显示接口是单片机中应用最为广泛的一种显示方式之一。其接口电路是把所有显示器的 8 个笔划段 a-h 同名端连在一起，而每一个显 示器的公共极 COM 是各自独立地受 I/O 线控制。CPU 向字段输出口送出字形码时，所有显示器接收到相同的字形码，但究竟是那个显示器亮，则取决于 COM 端，而这一端是由 I/O 控制的，所以我们就能自行决定何时显示哪一位了。而所谓动态扫描就是指我们采用分时的办法，轮流控制各个显示器的 COM 端，使各个 显示器轮流点亮。在 http://www.51hei.com 还有很多关于单片机显示接口的文章，大家可以参考一下

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

下图所示就是我们的单片机实验板上的动态扫描接口。由 89c51 的 P0 口能灌入较大的电流，所以我们采用共阳的数码管，并且不用限流电阻，而只是用两 只 1N4004 进行降压后给数码管供电，这里仅用了两只，实际上还能扩充。它们的公共端则由 PNP 型三极管 8550 控制，显然，如果 8550 导通，则对应 的数码管就能亮，而如果 8550 截止，则对应的数码管就不可能亮，8550 是由 P2.7，P2.6 控制的。这样我们就能通过控制 P27、P26 达到控制某 个数码管亮或灭的目的。

下面的这个单片机程序，就是用实验板上的数码管显示 0 和 1。

```
FIRST EQU P2.7 ;第一位数码管的位控制

SECOND EQU P2.6 ;第二位数码管的位控制

DISPBUFF EQU 5AH ;显示缓冲区为 5AH 和 5BH
```

```
ORG 0000H
AJMP START
ORG 30H
START:
MOV SP, #5FH ;设置堆栈
MOV P1, #0FFH
MOV P0, #0FFH
MOV P2, #0FFH ;初始化，所显示器，LED 灭
MOV DISPBUFF, #0 ;第一位显示 0
MOV DISPBUFF+1, #1 ;第二握显示 1
LOOP:
LCALL DISP ;调用显示程序
AJMP LOOP
;主程序到此结束
DISP:
PUSH ACC ;ACC 入栈
```

```
PUSH PSW ;PSW 入栈
MOV A，DISPBUFF ;取第一个待显示数
MOV DPTR，#DISPTAB ;字形表首地址
MOVC A，@A+DPTR ;取字形码
MOV P0，A ;将字形码送 P0 位（段口）
CLR FIRST ;开第一位显示器位口
LCALL DELAY ;延时 1 毫秒
SETB FIRST ;关闭第一位显示器（开始准备第二位的数据）
MOV A，DISPBUFF+1 ;取显示缓冲区的第二位
MOV DPTR，#DISPTAB
MOVC A，@A+DPTR
MOV P0，A ;将第二个字形码送 P0 口
CLR SECOND ;开第二位显示器
LCALL DELAY ;延时
SETB SECOND ;关第二位显示
POP PSW
```

```
POP ACC
RET
DELAY:  ;延时 1 毫秒
PUSH PSW
SETB RS0
MOV R7, #50
D1:  MOV R6, #10
D2:  DJNZ R6, $
DJNZ R7, D1
POP PSW
RET
DISPTAB:DB 28H, 7EH, 0a4H, 64H, 72H, 61H, 21H, 7CH, 20H, 60H
END
```

从上面的单片机例程中能看出，动态扫描显示必须由 CPU 持续地调用显示程序，才能保证持续持续的显示。

上面的这个程序能实现数字的显示，但不太实用，为什么呢？这里仅是显示两个数字，并没有做其他的工作，因此，两个数码管轮流显示 1 毫秒，没有问题，实 际的工作中，当然不可能只显示两个数字，还是要做其他的事情的，这样在二次调用显示程序之间的时间

间隔就不一不定了，如果时间间隔比较长，就会使显示不连 续。而实际工作中是很难保证所有工作都能在很短时间内完成的。况且这个显示程序也有点“浪费”，每个数码管显示都要占用 1 个毫秒的时间，这在很多合是不允 许的，怎么办呢？我们能借助于定时器，定时时间一到，产生中断，点亮一个数码管，然后马上返回，这个数码管就会一直亮到下一次定时时间到，而不用调用延时 程序了，这段时间能留给主程序干其他的事。到下一次定时时间到则显示下一个数码管，这样就很少浪费了。

```
Counter EQU 59H ;计数器，显示程序通过它得知现正显示哪个数码管

FIRST EQU P2.7 ;第一位数码管的位控制

SECOND EQU P2.6 ;第二位数码管的位控制

DISPBUFF EQU 5AH ;显示缓冲区为 5AH 和 5BH

ORG 0000H

AJMP START

ORG 000BH ;定时器 T0 的入口

AJMP DISP ;显示程序

ORG 30H

START:

MOV SP, #5FH ;设置堆栈

MOV P1, #0FFH

MOV P0, #0FFH
```

```
MOV P2，#0FFH ;初始化，所显示器，LED 灭
MOV TMOD，#00000001B ;定时器 T0 工作于模式 1（16 位定时/计数模式）
MOV TH0，#HIGH（65536-2000）
MOV TL0，#LOW（65536-2000）
SETB TR0
SETB EA
SETB ET0
MOV Counter，#0 ;计数器初始化
MOV DISPBUFF，#0 ;第一位始终显示 0
MOV A，#0
LOOP：
MOV DISPBUFF+1，A ;第二位轮流显示 0-9
INC A
LCALL DELAY
CJNE A，#10，LOOP
MOV A，#0
```

```
AJMP LOOP ;在此中间能按排任意程序，这里仅作示范。
;主程序到此结束
DISP:  ;定时器 T0 的中断响应程序
PUSH ACC ;ACC 入栈
PUSH PSW ;PSW 入栈
MOV TH0，#HIGH（65536-2000） ;定时时间为 2000 个周期，约 2170 微秒（11.0592M）
MOV TL0，#LOW（65536-2000）
SETB FIRST
SETB SECOND ;关显示
MOV A，#DISPBUFF ;显示缓冲区首地址
ADD A，Counter
MOV R0，A
MOV A，@R0 ;根据计数器的值取对应的显示缓冲区的值
MOV DPTR，#DISPTAB ;字形表首地址
MOVC A，@A+DPTR ;取字形码
MOV P0，A ;将字形码送 P0 位（段口）
```

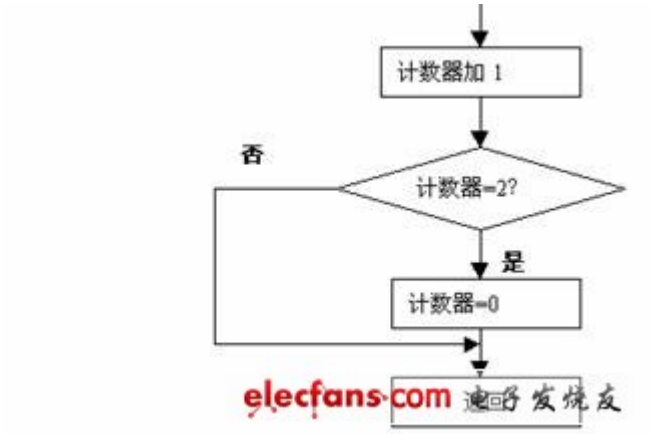
```
MOV A，Counter ;取计数器的值
JZ DISPFIRST ;如果是 0 则显示第一位
CLR SECOND ;不然显示第二位
AJMP DISPNEXT
DISPFIRST：
CLR FIRST ;显示第一位
DISPNEXT：
INC Counter ;计数器加 1
MOV A，Counter
DEC A ;如果计数器计到 2，则让它回 0
DEC A
JZ RSTCOUNT
AJMP DISPEXIT
RSTCOUNT：
MOV Counter，#0 ;计数器的值只能是 0 或 1
DISPEXIT：
```

```
POP PSW
POP ACC
RETI
DELAY:  ;延时 130 毫秒
PUSH PSW
SETB RS0
MOV R7, #255
D1:  MOV R6, #255
D2:  NOP
NOP
NOP
NOP
DJNZ R6, D2
DJNZ R7, D1
POP PSW
RET
```

```
DISPTAB:DB 28H, 7EH, 0a4H, 64H, 72H, 61H, 21H, 7CH, 20H, 60H

END
```

从上面的单片机程序能看出，动态显示和静态显示相比，程序稍有点复杂，不过，这是值得的。这个程序有一定的通用性，只要改变端口的值及计数器的值就能显示更多位数了。下面给出显示程序的流程图。

《动态扫描程序框图》



25、单片机键盘接口程序设计

键盘是由若干按钮组成的开关矩阵，它是单片机系统中最常用的输入设备，用户能通过键盘向计算机输入指令、地址和数据。一般单片机系统中采和非编码键盘，非编码键盘是由软件来识别键盘上的闭合键，它具有结构简单，使用灵活等特点，因此被广泛应用于单片机系统。

按钮开关的抖动问题

组成键盘的按钮有触点式和非触点式两种，单片机中应用的一般是由机械触点组成的。在下图中，当开

《键盘结构图》

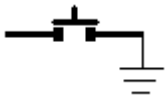


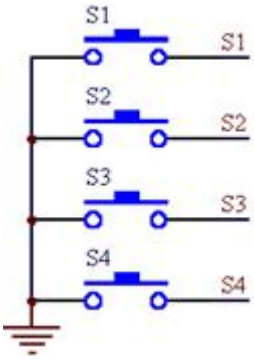
图 1

图 2

关 S 未被按下时，P1。0 输入为高电平，S 闭合后，P1。0 输入为低电平。由于按钮是机械触点，当机械触点断开、闭合时，会有抖动动，P1。0 输入端 的波形如图 2 所示。这种抖动对于人来说是感觉不到的，但对计算机来说，则是完全能感应到的，因为计算机处理的速度是在微秒级，而机械抖动的时间至少是毫秒 级，对计算机而言，这已是一个“漫长”的时间了。前面我们讲到中断时曾有个问题，就是说按钮有时灵，有时不灵，其实就是这个原因，你只按了一次按钮，可是 计算机却已执行了多次中断的过程，如果执行的次数正好是奇数次，那么结果正如你所料，如果执行的次数是偶数次，那就不对了。

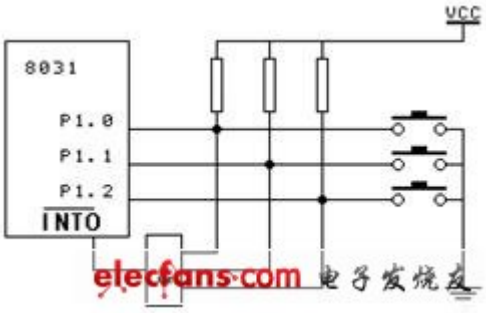
为使 CPU 能正确地读出 P1 口的状态，对每一次按钮只作一次响应，就必须考虑如何去除抖动，常用的去抖动的办法有两种：硬件办法和软件办法。单片机中 常用软件法，因此，对于硬件办法我们不介绍。软件法其实很简单，就是在单片机获得 P1。0 口为低的信息后，不是立即认定 S1 已被按下，而是延时 10 毫秒或 更长一些时间后再次检测 P1。0 口，如果仍为低，说明 S1 的确按下了，这实际上是避开了按钮按下时的抖动时间。而在检测到按钮释放后（P1。0 为高）再延 时 5-10 个毫秒，消除后沿的抖动，然后再对键值处理。不过一般情况下，我们常常不对按钮释放的后沿进行处理，实践证明，也能满足一定的要求。当然，实际 应用中，对按钮的要求也是千差万别，要根据不一样的需要来编制处理程序，但以上是消除键抖动的原则。

键盘与单片机的连接



《键盘连接》

图 3



《单片机与键盘接口图》

图 4

1、通过 1/0 口连接。将每个按钮的一端接到单片机的 I/0 口，另一端接地，这是最简单的办法，如图 3 所示是实验板上按钮的接法，四个按钮分别接到 P3.2 、P3.3、P3.4 和 P3.5。对于这种键各程序能采用持续查询的办法，功能就是：检测是否有键闭合，如有键

闭合，则去除键抖动，判断键号并转入对应的 键处理。下面给出一个例程。其功能很简单，四个键定义如下：

P3.2：开始，按此键则灯开始流动（由上而下）

P3.3：停止，按此键则停止流动，所有灯为暗

P3.4：上，按此键则灯由上向下流动

P3.5：下，按此键则灯由下向上流动

```
UpDown EQU 00H ;上下行标志
StartEnd EQU 01H ;起动及停止标志
LAMPCODE EQU 21H ;存放流动的数据代码
ORG 0000H
AJMP MAIN
ORG 30H
MAIN:
MOV SP, #5FH
MOV P1, #0FFH
CLR UpDown ;启动时处于向上的状态
CLR StartEnd ;启动时处于停止状态
```

```
MOV LAMPCODE，#0FEH ;单灯流动的代码
LOOP：
ACALL KEY ;调用键盘程序
JNB F0，LNEXT ;如果无键按下，则继续
ACALL KEYPROC ;不然调用键盘处理程序
LNEXT：
ACALL LAMP ;调用灯显示程序
AJMP LOOP ;反复循环，主程序到此结束
DELAY：
MOV R7，#100
D1： MOV R6，#100
DJNZ R6，$
DJNZ R7，D1
RET
;----------------------------------------延时程序，键盘处理中调用
KEYPROC：
```

```
    MOV A，B ;从 B 寄存器中获取键值

    JB ACC.2，KeyStart ;分析键的代码，某位被按下，则该位为 1（因为在键盘程序中已取反）

    JB ACC.3，KeyOver

    JB ACC.4，KeyUp

    JB ACC.5，KeyDown

    AJMP KEY_RET

    KeyStart：

    SETB StartEnd ;第一个键按下后的处理

    AJMP KEY_RET

    KeyOver：

    CLR StartEnd ;第二个键按下后的处理

    AJMP KEY_RET

    KeyUp： SETB UpDown ;第三个键按下后的处理

    AJMP KEY_RET

    KeyDown：

    CLR UpDown ;第四个键按下后的处理
```

```
KEY_RET:RET

KEY：

CLR F0 ;清 F0，表示无键按下。

ORL P3，#00111100B ;将 P3 口的接有键的四位置 1

MOV A，P3 ;取 P3 的值

ORL A，#11000011B ;将其余 4 位置 1

CPL A ;取反

JZ K_RET ;如果为 0 则一定无键按下

ACALL DELAY ;不然延时去键抖

ORL P3，#00111100B

MOV A，P3

ORL A，#11000011B

CPL A

JZ K_RET

MOV B，A ;确实有键按下，将键值存入 B 中

SETB F0 ;设置有键按下的标志
```

```
K_RET:
ORL P3, #00111100B ;此处循环等待键的释放
MOV A, P3
ORL A, #11000011B
CPL A
JZ K_RET1 ;直到读取的数据取反后为 0 说明键释放了，才从键盘处理程序中返回
AJMP K_RET
K_RET1:
RET
D500MS:  ;流水灯的延迟时间
PUSH PSW
SETB RS0
MOV R7, #200
D51:  MOV R6, #250
D52:  NOP
NOP
```

```
NOP
NOP
DJNZ R6, D52
DJNZ R7, D51
POP PSW
RET
LAMP:
JB StartEnd, LampStart ;如果 StartEnd=1, 则启动
MOV P1, #0FFH
AJMP LAMPRET ;不然关闭所有显示, 返回
LampStart:
JB UpDown, LAMPUP ;如果 UpDown=1, 则向上流动
MOV A, LAMPCODE
RL A ;实际就是左移位而已
MOV LAMPCODE, A
MOV P1, A
```

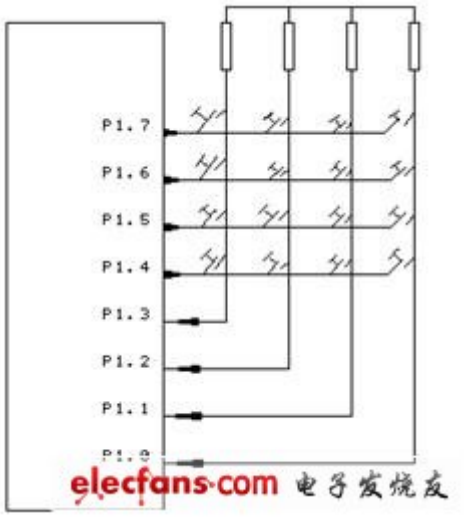
```
LCALL D500MS

AJMP LAMPRET

LAMPUP：

MOV A，LAMPCODE

RR A ;向下流动实际就是右移

MOV LAMPCODE，A

MOV P1，A

LCALL D500MS

LAMPRET：

RET

END
```

以上程序功能很简单，但它演示了一个单片机键盘处理程序的基本思路，程序本身很简单，也不很实用，实际工作中还会有好多要考虑的因素，比如主循环每次 都调用灯的循环程序，会造成按钮反应“迟钝”，而如果一直按着键不放，则灯不会再流动，一直要到松开手为止，等等，大家能仔细考虑一下这些问题，再想想有 什么好的解决办法。

2、采用中断方式：如图 4 所示。各个按钮都接到一个与非上，当有任何一个按钮按下时，都会使与门输出为低电平，从而引起单片机的中断，它的好处是不用在主程序中持续地循环查询，如果有键按下，单片机再去做对应的处理

26、矩阵式键盘接口技术及程序设计

在单片机系统中键盘中按钮数量较多时，为了减少 I/O 口的占用，常常将按钮排列成矩阵形式，如图 1 所示。在矩阵式键盘中，每条水平线和垂直线在交叉处 不直接连通，而是通过一个按钮加以连接。这样，一个端口（如 P1 口）就能组成 4*4=16 个按钮，比之直接将端口线用于键盘多出了一倍，而且线数越多，区 别越明显，比如再多加一条线就能组成 20 键的键盘，而直接用端口线则只能多出一键（9 键）。由此可见，在需要的键数比较多时，采用矩阵法来做键盘是合理 的。



《单片机矩阵式键盘接口技术及编程接口图》

矩阵式结构的键盘显然比直接法要复杂一些，识别也要复杂一些，上图中，列线通过电阻接正电源，并将行线所接的单片机的 I/O 口作为输出端，而列线所接 的 I/O 口则作为输入。这样，当按钮没有按下时，所有的输出端都是高电平，代表无键按下。行线输出是低电平，一旦有键按下，则输入线就会被拉低，这样，通 过读入输入线的状态就可得知是否有键按下了。具体的识别及编程办法如下所述。

矩阵式键盘的按钮识别办法

确定矩阵式键盘上何键被按下介绍一种“行扫描法”。

行扫描法 行扫描法又称为逐行（或列）扫描查询法，是一种最常用的按钮识别办法，如上图所示键盘，介绍过程如下。

判断键盘中有无键按下 将全部行线 Y0-Y3 置低电平，然后检测列线的状态。只要有一列的电平为低，则表示键盘中有键被按下，而且闭合的键位于低电平线与 4 根行线相交叉的 4 个按钮之中。若所有列线均为高电平，则键盘中无键按下。

判断闭合键所在的位置 在确认有键按下后，即可进入确定具体闭合键的过程。其办法是：依次将行线置为低电平，即在置某根行线为低电平时，其它线为高电平。在确定某根行线位置为低 电平后，再逐行检测各列线的电平状态。若某列为低，则该列线与置为低电平的行线交叉处的按钮就是闭合的按钮。

下面给出一个具体的例程：

图仍如上所示。8031 单片机的 P1 口用作键盘 I/O 口，键盘的列线接到 P1 口的低 4 位，键盘的行线接到 P1 口的高 4 位。列线 P1.0-P1.3 分别 接有 4 个上拉电阻到正电源+5V，并把列线 P1.0-P1.3 设置为输入线，行线 P1.4-P.17 设置为输出线。4 根行线和 4 根列线形成 16 个相交点。

检测当前是否有键被按下。检测的办法是 P1.4-P1.7 输出全“0”，读取 P1.0-P1.3 的状态，若 P1.0-P1.3 为全“1”，则无键闭合，不然有键闭合。

去除键抖动。当检测到有键按下后，延时一段时间再做下一步的检测判断。

若有键被按下，应识别出是哪一个键闭合。办法是对键盘的行线进行扫描。P1.4-P1.7 按下述 4 种组合依次输出：

P1.7 1 1 1 0

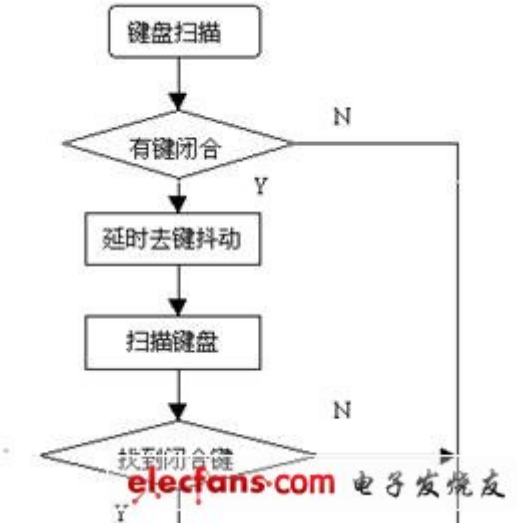
P1.6 1 1 0 1

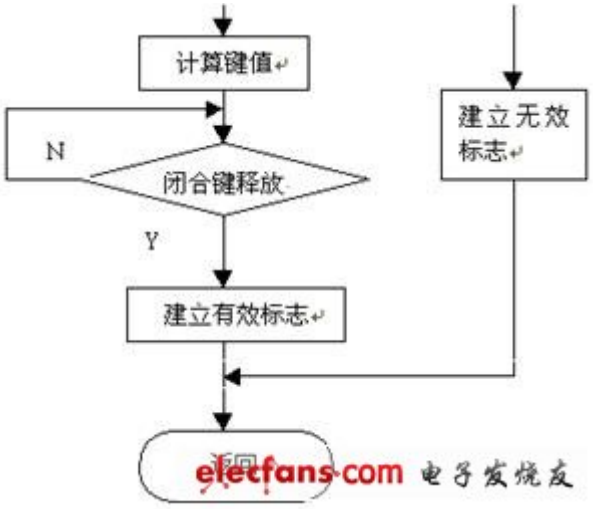
P1.5 1 0 1 1

P1.4 0 1 1 1

在每组行输出时读取 P1.0-P1.3，若全为“1”，则表示为“0”这一行没有键闭合，不然有键闭合。由此得到闭合键的行值和列值，然后可采用计算法或查表法将闭合键的行值和列值转换成所定义的键值

为了保证键每闭合一次 CPU 仅作一次处理，必须却除键释放时的抖动。





《单片机矩阵式键盘接口技术及编程》

键盘扫描程序：

从以上分析得到单片机键盘扫描程序的流程图如图 2 所示。程序如下

```
SCAN:  MOV P1, #0FH
MOV A, P1
ANL A, #0FH
CJNE A, #0FH, NEXT1
SJMP NEXT3
NEXT1:  ACALL D20MS
```

```
MOV A, #0EFH
NEXT2:  MOV R1, A
MOV P1, A
MOV A, P1
ANL A, #0FH
CJNE A, #0FH, KCODE;
MOV A, R1
SETB C
RLC A
JC NEXT2
NEXT3:  MOV R0, #00H
RET
KCODE:  MOV B, #0FBH
NEXT4:  RRC A
INC B
JC NEXT4
```

```
MOV A，R1
SWAP A
NEXT5： RRC A
INC B
INC B
INC B
INC B
JC NEXT5
NEXT6： MOV A，P1
ANL A，#0FH
CJNE A，#0FH，NEXT6
MOV R0，#0FFH
RET
```

键盘处理程序就作这么一个简单的介绍，实际上，键盘、显示处理是很复杂的，它一般占到一个应用程序的大部份代码，可见其重要性，但说到，这种复杂并不 来自于单片机的本身，而是来自于操作者的习惯等等问题，因此，在编写键盘处理程序之前，最好先把它从逻辑上理清，然后用适当的算法表示出来，最后再去写代 码，这样，才能快速有效地写好代码。

27、关于单片机的一些基本概念

随着电子技术的迅速发展，计算机已深入地渗透到我们的生活中，许多电子爱好者开始学习单片机知识，但单片机的内容比较抽象，相对电子爱好者已熟悉的模 拟电路、数字电路，单片机中有一些新的概念，这些概念非常基本以至于一般作者不屑去谈，教材自然也不会很深入地讲解这些概念，但这些内容又是学习中必须要 理解的，下面就结合本人的学习、教学经验，对这些最基本概念作一说明，希望对自学者有所帮助。

一、总线：我们知道，一个电路总是由元器件通过电线连接而成的，在模拟电路中，连线并不成为一个问题，因为各器件间一般是串行关系，各器件之间的连线 并不很多，但计算机电路却不一样，它是以微处理器为核心，各器件都要与微处理器相连，各器件之间的工作必须相互协调，所以就需要的连线就很多了，如果仍如 同模拟电路一样，在各微处理器和各器件间单独连线，则线的数量将多得惊人，所以在微处理机中引入了总线的概念，各个器件共同享用连线，所有器件的 8 根数据 线全部接到 8 根公用的线上，即相当于各个器件并联起来，但仅这样还不行，如果有两个器件同时送出数据，一个为 0，一个为 1，那么，接收方接收到的究竟是什 么呢？这种情况是不允许的，所以要通过控制线进行控制，使器件分时工作，任何时候只能有一个器件发送数据（能有多个器件同时接收）。器件的数据线也就被称 为数据总线，器件所有的控制线被称为控制总线。在单片机内部或者外部存储器及其它器件中有存储单元，这些存储单元要被分配地址，才能使用，分配地址当然也 是以电信号的形式给出的，由于存储单元比较多，所以，用于地址分配的线也较多，这些线被称为地址总线。

二、数据、地址、指令：之所以将这三者放在一起，是因为这三者的本质都是一样的——数字，或者说都是一串‘0’和‘1’组成的序列。换言之，地址、指 令也都是数据。指令：由单片机芯片的设计者规定的一种数字，它与我们常用的指令助记符有着严格的一一对应关系，不能由单片机的开发者更改。地址：是寻找单 片机内部、外部的存储单元、输入输出口的依据，内部单元的地址值已由芯片设计者规定好，不可更改，外部的单元能由单片机开发者自行决定，但有一些地址单元 是一定要有的（详见程序的执行过程）。数据：这是由微处理机处理的对象，在各种不一样的应用电路中各不相同，一般而言，被处理的数据可能有这么几种情况：

1·地址（如 MOV DPTR，#1000H），即地址 1000H 送入 DPTR。

2·方式字或控制字（如 MOV TMOD，#3），3 即是控制字。

3·常数（如 MOV TH0，#10H）10H 即定时常数。

4·实际输出值（如 P1 口接彩灯，要灯全亮，则执行指令：MOV P1，#0FFH，要灯全暗，则执行指令：MOV P1，#00H）这里 0FFH 和 00H 都是实际输出值。又如用于 LED 的字形码，也是实际输出的值。

理解了地址、指令的本质，就不难理解程序运行过程中为什么会跑飞，会把数据当成指令来执行了。

三、P0 口、P2 口和 P3 的第二功能使用办法 开始学习时一般对 P0 口、P2 口和 P3 口的第二功能使用办法迷惑不解，认为第二功能和原功能之间要有一个切换的过程，或者说要有一条指令，事实上，各端口 的第二功能完全是自动的，不需要用指令来转换。如 P3.6、P3.7 分别是 WR、RD 信号，当微片理机外接 RAM 或有外部 I/O 口时，它们被用作第二功 能，不能作为通用 I/O 口使用，只要一微处理机一执行到 MOVX 指令，就会有对应的信号从 P3.6 或 P3.7 送出，不需要事先用指令说明。事实上‘不能作 为通用 I/O 口使用’也并不是‘不能’而是（使用者）‘不会’将其作为通用 I/O 口使用。你完全能在指令中按排一条 SETB P3.7 的指令，并且当单片机执行到这条指令时，也会使 P3.7 变为高电平，但使用者不会这么去做，因为这常常这会导致系统的崩溃（即死机）。

四、程序的执行过程 单片机在通电复位后 8051 内的程序计数器（PC）中的值为‘0000’，所以程序总是从‘0000’单元开始执行，也就是说：在系统的 ROM 中一定要存在‘0000’这个单元，并且在‘0000’单元中存放的一定是一条指令。

五、堆栈 堆栈是一个区域，是用来存放数据的，这个区域本身没有任何特殊之处，就是内部 RAM 的一部份，特殊的是它存放和取用数据的方式，即所谓的‘先进后出，后进 先出’，并且堆栈有特殊的数据传输指令，即‘PUSH’和‘POP’，有一个特殊的专为其服务的单元，即堆栈指针 SP，每当执一次 PUSH 指令时，SP 就 （在原来值的基础上）自动加 1，每当执行一次 POP 指令，SP 就（在原来值的基础上）自动减 1。由于 SP 中的值能用指令加

以改变，所以只要在程序开始阶段 更改了 SP 的值，就能把堆栈设置在规定的内存单元中，如在程序开始时，用一条 MOV SP，#5FH 指令，就时把堆栈设置在从内存单元 60H 开始的单元中。一般程序的开头总有这么一条设置堆栈指针的指令，因为开机时，SP 的初始值为 07H，这样就使堆栈从 08H 单元开始往后，而 08H 到 1FH 这个区域正是 8031 的第二、三、四工作寄存器区，经常要被使用，这会造成数据的浑乱。不一 样作者编写程序时，初始化堆栈指令也不完全相同，这是作者的习惯问题。当设置好堆栈区后，并不意味着该区域成为一种专用内存，它还是能象普通内存区域一样 使用，只是一般情况下编程者不会把它当成普通内存用了。

六、单片机的开发过程 这里所说的开发过程并不是一般书中所说的从任务分析开始，我们假设已设计并制作好硬件，下面就是编写软件的工作。在编写软件之前，首先要确定一些常数、地 址，事实上这些常数、地址在设计阶段已被直接或间接地确定下来了。如当某器件的连线设计好后，其地址也就被确定了，当器件的功能被确定下来后，其控制字也 就被确定了。然后用文本编缉器（如 EDIT、CCED 等）编写软件，编写好后，用编译器对源程序文件编译，查错，直到没有语法错误，除了极简单的程序外， 一般应用仿真机对软件进行调试，直到程序运行正确为止。运行正确后，就能写片（将程序固化在 EPROM 中）。在源程序被编译后，生成了扩展名为 HEX 的目 标文件，一般编程器能够识别这种格式的文件，只要将此文件调入即可写片。在此，为使大家对整个过程有个认识，举一例说明：

表 1

```
ORG 0000H

LJMP START

ORG 040H

START：

MOV SP，#5FH ；设堆栈

LOOP：
```

```
NOP

LJMP  LOOP  ；循环

END  ；结束
```

表 2

```
：03000000020040BB

：0700400075815F000200431F
```

表 3

```
02 00 40 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF 75 81 5F 00 02 00 43
```

表 1 为源程序，表 2 是汇编后得到的 HEX 文件，表 3 是由 HEX 文件转换成的目标文件，也就是最终写入 EPROM 的文件，它由编程器转换得到，也能由 HEXBIN 一类的程序转换得到。学过手工汇编者应当不难找出表 3 与表 1 的一一对应关系，值得注意的是从 02 00 40 后开始的一长串‘FF’，直到 75 81，这是由于伪指令：ORG 040H 造成的结果。

七、仿真、仿真机 仿真是单片机开发过程中非常重要的一个环节，除了一些极简单的任务，一般产品开发过程中都要进行仿真，仿真的主要目的是进行软件调试，当然借助仿真机，也 能进行一些硬件排错。一块单片机应用电路板包括单片机部份及为达到使用目的而设计的应用电路，仿真就是利用仿真机来代替应用电路板（称目标机）的单片机部 份，对应用电路部份进行测试、调试。仿真有 CPU 仿真和 ROM 仿真两种，所谓 CPU 仿真是指用仿真机代替目标机的 CPU，由仿真机向目标机的应用电路部份 供给各种信号、数据，进行调试的办法。这种仿真能通过单步运行、连续运行等多种办法来运行程序，并能观察到单片机内部的变化，便于改正程序中的错误。所谓 ROM 仿真，就是用仿真机代替目标机的 ROM，目标机的 CPU 工作时，从仿真机中读取程序，并执行。这种仿真其实就是将仿真机当成一

片 EPROM，只是省 去了擦片、写片的麻烦，并没有多少调试手段可言。常常这是二种不一样类型的仿真机，也就是说，一台仿真机不能既做 CPU 仿真，又做 ROM 仿真。可能的情况 下，当然以 CPU 仿真好。以上是本人对单片机的理解，如有不对之处，请诸位大侠多多指点。发表您的高论。

28、单片机音乐程序设计

利用单片机（或单板机）奏乐大概是无线电爱好者感兴趣的问题之一。本文从单片机的基本发间实验出发，谈谈音乐程序的设计原理，并给出具体实例，以供参考。

单片机的基本发音实验

我们知道，声音的频谱范围约在几十到几千赫兹，若能利用程序来控制单处机某个口线的“高”电平或低电平，则在该口线上就能产生一定频率的矩形波，接上喇叭就能发出一定频率的声音，若再利用延时程序控制“高”“低”电平的持续时间，就能改变输出频率，从而改变音调。

例如，要产生 200HZ 的音频信号，按图 1 接入喇叭（若属临时实验，也可将喇叭直接接在 P1 口线上），实验程序为：

其中子程序 DEL 为延时子程序，当 R3 为 1 时，延时时间约为 20us，R3 中存放延时常数，对 200HZ 音频，其周期为 1/200 秒，即 5ms。这 样，当 P1.4 的高电平或低电平的持续时间为 2.5ms，即 R3 的时间常数取 2500/20=125（7DH）时，就能发出 200HZ 的音调。将上述程序 键入学习机，并持续修改 R3 的常数能感到音调的变化。乐曲中，每一音符对应着确定的频率，表 1 给出 C 调时各音符频率及其对应的时间常数。读者能根据表 1 所 供给的常数，将其 16 进制代码送入 R3，反复练习体会。根据表 1 能奏出音符。仅这还不够，要准确奏出一首曲子，必须准确地控制乐曲节奏，即一音符的持续时 间。

音符的节拍我们能用定时器 T0 来控制，送入不一样的初值，就能产生不一样的定时时间。便如某歌曲的节奏为每分钟 94 拍，即一拍为 0.64 秒。其它节拍与时间的对应关系见表 2。

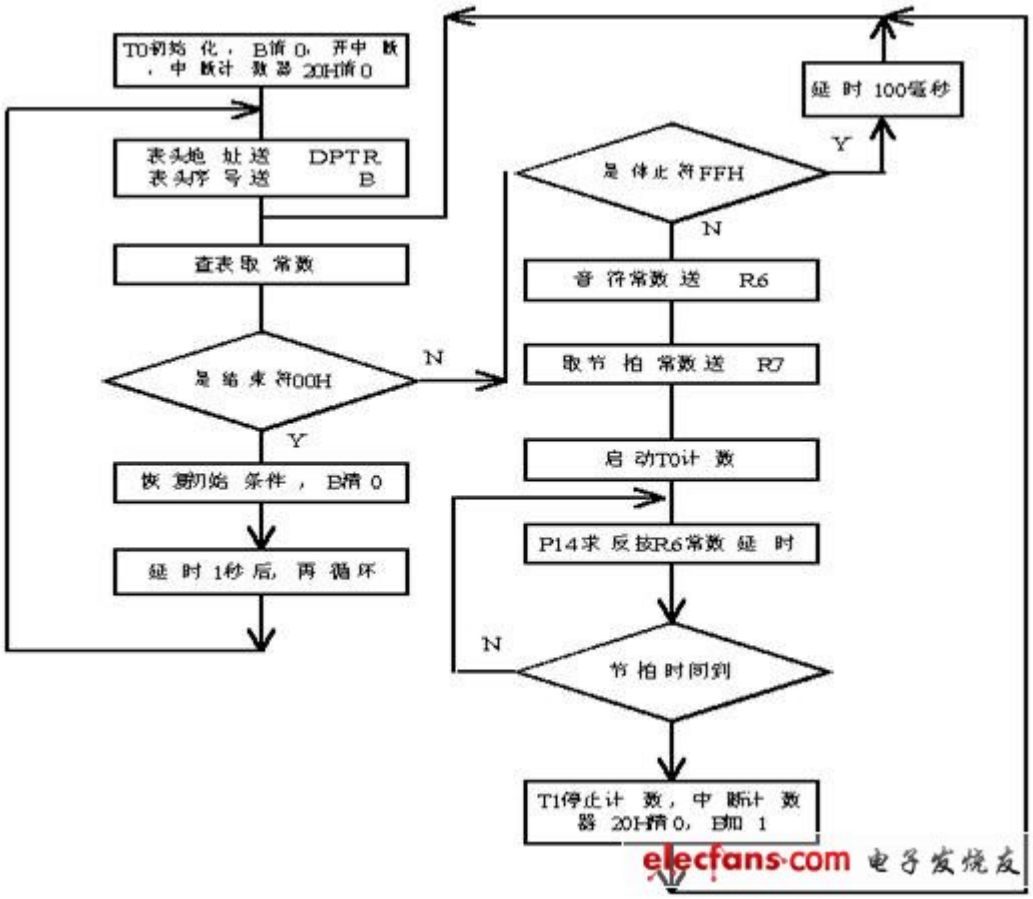
但时，由于 T0 的最大定时时间只能为 131 毫秒，因此不可能直接用改变 T0 的时间初值来实现不一样节拍。我们能用 T0 来产生 10 毫秒的时间基准，然后 设置一个中断计数器，通过判别中断计数器的值来控制节拍时间的长短。表 2 中也给出了各种节拍所对应的时间常数。例如对 1/4 拍音符，定时时间为 0.16 秒，对应的时间常数为 16（即 10H）；对 3 拍音符，定时时间为 1.92 秒，对应时间长数为 192（即 C0H）。

我们将每一音符的时间常数和其对应的节拍常数作为一组，按次序将乐曲中的所有常数排列成一个表，然后由查表程序依次取出，产生音符并控制节奏，就能实 现演奏效果。此外，结束符和体止符能分别用代码 00H 和 FFH 来表示，若查表结果为 00H，则表示曲子终了；若查表结果为 FFH，则产生对应的停顿效果。 为了产生手弹的节奏感，在某些音符（例如两个相同音符）音插入一个时间单位的频率略有不一样的音符。

下面给出程序序清单，可直接在 TD-III 型学习机上演奏，对其它不一样型号的学习机，只需对应地改变一下地址即可。本程序演奏的是民歌“八月桂花遍 地开”，C 调，节奏为 94 拍/分。读者也能自行找出一首歌，按表 1 和表 2 给定的常数，将乐曲翻译成码表输入机器，而程序不变。本实验办法简便，即使不懂音 乐的人，将一首陌生的曲子翻译成代码也是易事，和着机器的演奏学唱一首歌曲，其趣味无穷。

程序清单（略，请参看源程序的说明）。

程序框图如图 2 所示。



《单片机音乐程序的设计图》

本课由单片机教程网提供，有问题指出。

硬件连接说明：

随便找一个仿真机或者什么单片机实验板，只要能工作的就行，将程序输入，运行，然后找个音箱（你计算机旁边应当就有一对吧）拨出插头，插头的前端接在 P1。0 上，后面部分找根线接单片机的地，就应当有声了，然后怎么改进硬件连接就是你的事了。。。。

音乐程序汇编代码代码 1 ----------Voice.asm----------

```
ORG 0000H

LJMP START

ORG 000BH

INC 20H ;中断服务，中断计数器加 1

MOV TH0，#0D8H

MOV TL0，#0EFH ;12M 晶振，形成 10 毫秒中断

RETI

START:

MOV SP，#50H

MOV TH0，#0D8H

MOV TL0，#0EFH

MOV TMOD，#01H

MOV IE，#82H
```

```
MUSIC0：
NOP
MOV DPTR，#DAT ;表头地址送 DPTR
MOV 20H，#00H ;中断计数器清 0
MOV B，#00H ;表序号清 0
MUSIC1：
NOP
CLR A
MOVC A，@A+DPTR ;查表取代码
JZ END0 ;是 00H，则结束
CJNE A，#0FFH，MUSIC5
LJMP MUSIC3
MUSIC5：
NOP
MOV R6，A
INC DPTR
```

```
MOV A，B
MOVC A，@A+DPTR ;取节拍代码送 R7
MOV R7，A
SETB TR0 ;启动计数
MUSIC2：
NOP
CPL P1.0
MOV A，R6
MOV R3，A
LCALL DEL
MOV A，R7
CJNE A，20H，MUSIC2 ;中断计数器（20H）=R7 否？
;不等，则继续循环
MOV 20H，#00H ;等于，则取下一代码
INC DPTR
; INC B
```

```
LJMP MUSIC1

MUSIC3：

NOP

CLR TR0 ;休止 100 毫秒

MOV R2，#0DH

MUSIC4：

NOP

MOV R3，#0FFH

LCALL DEL

DJNZ R2，MUSIC4

INC DPTR

LJMP MUSIC1

END0：

NOP

MOV R2，#64H ;歌曲结束，延时 1 秒后继续

MUSIC6：
```

```
MOV R3, #00H
LCALL DEL
DJNZ R2, MUSIC6
LJMP MUSIC0
DEL:
NOP
DEL3:
MOV R4, #02H
DEL4:
NOP
DJNZ R4, DEL4
NOP
DJNZ R3, DEL3
RET
NOP
DAT:
```

db 26h，20h，20h，20h，20h，20h，26h，10h，20h，10h，20h，80h，26h，20h，30h，20h

db 30h，20h，39h，10h，30h，10h，30h，80h，26h，20h，20h，20h，20h，20h，1ch，20h

db 20h，80h，2bh，20h，26h，20h，20h，20h，2bh，10h，26h，10h，2bh，80h，26h，20h

db 30h，20h，30h，20h，39h，10h，26h，10h，26h，60h，40h，10h，39h，10h，26h，20h

db 30h，20h，30h，20h，39h，10h，26h，10h，26h，80h，26h，20h，2bh，10h，2bh，10h

db 2bh，20h，30h，10h，39h，10h，26h，10h，2bh，10h，2bh，20h，2bh，40h，40h，20h

db 20h，10h，20h，10h，2bh，10h，26h，30h，30h，80h，18h，20h，18h，20h，26h，20h

db 20h，20h，20h，40h，26h，20h，2bh，20h，30h，20h，30h，20h，1ch，20h，20h，20h

db 20h，80h，1ch，20h，1ch，20h，1ch，20h，30h，20h，30h，60h，39h，10h，30h，10h

db 20h，20h，2bh，10h，26h，10h，2bh，10h，26h，10h，26h，10h，2bh，10h，2bh，80h

db 18h，20h，18h，20h，26h，20h，20h，20h，20h，60h，26h，10h，2bh，20h，30h，20h

db 30h，20h，1ch，20h，20h，20h，20h，80h，26h，20h，30h，10h，30h，10h，30h，20h

db 39h，20h，26h，10h，2bh，10h，2bh，20h，2bh，40h，40h，10h，40h，10h，20h，10h

db 20h，10h，2bh，10h，26h，30h，30h，80h，00H

END

音乐程序汇编代码代码 2 ----------Voice1.asm----------

;标题 ‘八月桂花香’发声程序

;摘要 详见‘无线电’92 年 3 期

;作者 周振安

ORG 0000H

LJMP START

ORG 000BH

INC 20H ;中断服务，中断计数器加 1

MOV TH0，#0D8H

MOV TL0，#0EFH ;12M 晶振，形成 10 毫秒中断

RETI

```
START:
MOV SP, #50H
MOV TH0, #0D8H
MOV TL0, #0EFH
MOV TMOD, #01H
MOV IE, #82H
MUSIC0:
NOP
MOV DPTR, #DAT ;表头地址送 DPTR
MOV 20H, #00H ;中断计数器清 0
MOV B, #00H ;表序号清 0
MUSIC1:
NOP
CLR A
MOVC A, @A+DPTR ;查表取代码
JZ END0 ;是 00H, 则结束
```

```
CJNE A, #0FFH, MUSIC5
LJMP MUSIC3
MUSIC5:
NOP
MOV R6, A
INC DPTR
MOV A, B
MOVC A, @A+DPTR ;取节拍代码送 R7
MOV R7, A
SETB TR0 ;启动计数
MUSIC2:
NOP
CPL P1.0
MOV A, R6
MOV R3, A
LCALL DEL
```

```
MOV A，R7
CJNE A，20H，MUSIC2 ;中断计数器（20H）=R7 否?
;不等，则继续循环
MOV 20H，#00H ;等于，则取下一代码
INC DPTR
; INC B
LJMP MUSIC1
MUSIC3：
NOP
CLR TR0 ;休止 100 毫秒
MOV R2，#0DH
MUSIC4：
NOP
MOV R3，#0FFH
LCALL DEL
DJNZ R2，MUSIC4
```

```
INC DPTR
LJMP MUSIC1
END0：
NOP
MOV R2，#64H ;歌曲结束，延时 1 秒后继续
MUSIC6：
MOV R3，#00H
LCALL DEL
DJNZ R2，MUSIC6
LJMP MUSIC0
DEL：
NOP
DEL3：
MOV R4，#02H
DEL4：
NOP
```

```
DJNZ R4，DEL4
NOP
DJNZ R3，DEL3
RET
NOP
DAT：
DB 18H，  30H，  1CH，  10H
DB 20H，  40H，  1CH，  10H
DB 18H，  10H，  20H，  10H
DB 1CH，  10H，  18H，  40H
DB 1CH，  20H，  20H，  20H
DB 1CH，  20H，  18H，  20H
DB 20H，  80H，  0FFH，  20H
DB 30H，  1CH，  10H ，  18H
DB 20H，  15H，  20H ，  1CH
DB 20H，  20H，  20H ，  26H
```

```
DB 40H,  20H ,  20H ,  2BH
DB 20H,  26H,  20H ,  20H
DB 20H,  30H ,  80H ,  0FFH
DB 20H,  20H,  1CH ,  10H
DB 18H,  10H,  20H ,  20H
DB 26H,  20H ,  2BH ,  20H
DB 30H,  20H ,  2BH ,  40H
DB 20H,  20H ,  1CH ,  10H
DB 18H,  10H,  20H,  20H
DB 26H,  20H ,  2BH,  20H
DB 30H,  20H,  2BH ,  40H
DB 20H,  30H,  1CH ,  10H
DB 18H,  20H ,  15H ,  20H
DB 1CH,  20H ,  20H ,  20H
DB 26H,  40H,  20H ,  20H
DB 2BH,  20H,  26H ,  20H
```

```
DB 20H,  20H,  30H ,  80H
DB 20H,  30H,  1CH ,  10H
DB 20H,  10H,  1CH ,  10H
DB 20H,  20H,  26H ,  20H
DB 2BH,  20H,  30H ,  20H
DB 2BH,  40H,  20H ,  15H
DB 1FH,  05H,  20H ,  10H
DB 1CH,  10H,  20H ,  20H
DB 26H,  20H,  2BH ,  20H
DB 30H,  20H,  2BH ,  40H
DB 20H,  30H,  1CH ,  10H
DB 18H,  20H ,  15H ,  20H
DB 1CH,  20H ,  20H ,  20H
DB 26H,  40H,  20H ,  20H
DB 2BH,  20H,  26H ,  20H
DB 20H,  20H,  30H,  30H
```

```
DB 20H,  30H,  1CH,  10H
DB 18H,  40H,  1CH,  20H
DB 20H,  20H,  26H,  40H
DB 13H,  60H,  18H ,  20H
DB 15H,  40H,  13H ,  40H
DB 18H,  80H,  00H
end
```

——单片机 C 语言知识点

单片机 C 语言知识点全攻略（一）

单片机C语言知识点全攻略（一）

欢迎每日工程师交

电子发烧友网讯：继《单片机学习知识点全攻略》得到广大读者好评，根据有网友提出美中不足的是所用单片机编程语言为汇编，基于此，电子发烧友网再接再厉再次为读者诚挚奉上非常详尽的《单片机 C 语言知识点全攻略》系列单片机 C 语言学习教程，本教程共分为四部分，主要知识点如下所示。

第一部分知识点：

- 第一课 建立你的第一个 KeilC51 项目
- 第二课 C51HEX 文件的生成和单片机
- 第三课 C51 数据类型
- 第四课 C51 常量

第二部分知识点：

- 第五课 C51 变量
- 第六课 C51 运算符和表达式
- 第七课 运算符和表达式（关系运算符）
- 第八课 运算符和表达式（位运算符）
- 第九课 C51 运算符和表达式（指针和地址运算符）

第三部分知识点：

- 第十课 C51 表达式语句及仿真器
- 第十一课 C51 复合语句和条件语句

- 第十二课 C51 开关分支语句
- 第十三课 C51 循环语句
- 第十四课 C51 函数

第四部分知识点：

- 第十五课 C51 数组的使用
- 第十六课 C51 指针的使用
- 第十七课 C51 结构、联合和枚举的使用
- 附录（运算符优先级和结合性等）

c 语言是很好用的结构化语言，80 年代后，c 也能用在单片机上了。本站为了方便大家学习制作了一个单片机 c 语言教程，如果你是新手先看下下面的 c51 介绍吧，过去长期困扰人们的所谓“高级语言产生代码太长，运行速度太慢，运行效率不高，所以不适合单片机使用”keil 公司出品的单片机 c 语言集成开发环 境成功的解决了这个难题，使得单片机 c 语言的效率大大的提高，而且在关键部位还能嵌入汇编语言代码，从而挖掘程序的最高潜力。

目前，8051 上的 C 语言的代码长度，已经做到了汇编水平的 1.2~1.5 倍。4K 字节以上的程度，C 语言的优势更能得到发挥。至于运行速度的问题， 只要有好的仿真器，找出关键的代码，再进一步做一下人工优化，就可很容易达到美满。单片机 c 语言是高效的单片机开发语言，本站提供的单片机 c 语言教程共 17 课时，由浅入深，看完了这些教程你就基本了解了 c51，能进行一般的单片机 c 语言程序设计了，现在让我们开始学习吧。

第一课、建立你的第一个 KeilC51 项目

随着单片机技术的不断发展，以单片机 C 语言为主流的高级语言也不断被更多的单片机爱好者和工程师所喜爱。使用 C51 肯定要使用到编译器，以便把写好的 C 程序编译为机器码，这样单片机才能执行编写好的程序。KEIL uVISION2 是众多单片机应用开发软件中优秀的软件之一，它支持众多不一样公司的 MCS51 架构的芯片，它集编辑，编译，仿真等于一体，同时还支持，PLM，汇编和 C 语言的程序设计，它的界面和常用的微软 VC++的界面相似，界面友好，易学易用，在调试程序，软件仿真方面也有很强大的功能。本站提供的单片机 c 语言教程都是基于 keilc51 的。

下面结合 8051 介绍单片机 C 语言的优越性：

• 无须懂得单片机的具体硬件，也能够编出符合硬件实际的专业水平的程序；

• 不懂得单片机的指令集，也能够编写完美的单片机程序；

• 不同函数的数据实行覆盖，有效利用片上有限的 RAM 空间；

• 提供 auto、static、const 等存储类型和专门针对 8051 单片机的 data、idata、pdata、xdata、code 等存储类型，自动为变量合理地分配地址；

• C 语言提供复杂的数据类型（数组、结构、联合、枚举、指针等），极大地增强了程序处理能力和灵活性；

• 提供 small、compact、large 等编译模式，以适应片上存储器的大小；

• 中断服务程序的现场保护和恢复，中断向量表的填写，是直接与单片机相关的，都由 C 编译器代办；

• 程序具有坚固性：数据被破坏是导致程序运行异常的重要因素。C 语言对数据进行了许多专业性的处理，避免了运行中间非异步的破坏

• 提供常用的标准函数库，以供用户直接使用；

• 有严格的句法检查，错误很少，可容易地在高级语言的水平上迅速地被排掉；

• 可方便地接受多种实用程序的服务：如片上资源的初始化有专门的实用程序自动生成；再如，有实时多任务操作系统可调度多道任务，简化用户编程，提高运行的安全性等等。

• 头文件中定义宏、说明复杂数据类型和函数原型，有利于程序的移植和支持单片机的系列化产品的开发；

以上简单介绍了 KEILC51 软件，要使用 KEILC51 软件，必需先要安装它，这也是学习单片机编程语言所要求的第一步——建立学习环境。

安装好后，您是不是想建立自己的第一个单片机 C 语言程序项目呢？下面就让我们一起来建立一个小程序吧，请根据教程一步步的来，你绝对可以在短时间内熟悉 c51 的。

首先当然是运行 KEIL 软件，接着按下面的步骤建立您的第一个项目：

（1）点击 Project 菜单，选择弹出的下拉式菜单中的 New Project，如图 1—2。接着弹 出一个标准 Windows 文件对话窗口，如图 1—3。在“文件名”中输入您的第一个 C 程序项 目名称，这里我们用“test”。“保存”后的文件扩展名为 uv2，这是 KEIL uVision2 项目文件扩展名，以后能直接点击此文件以打开先前做的项目。

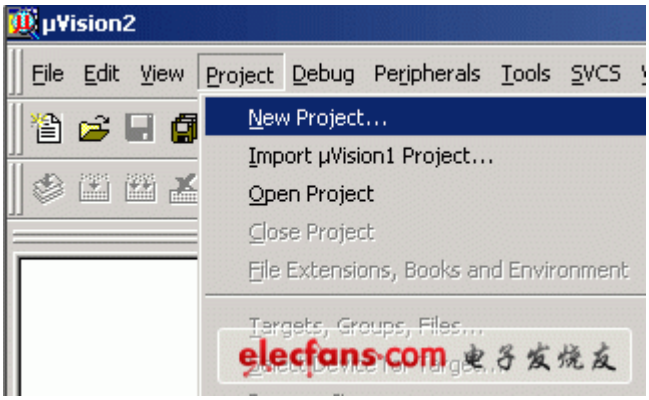


图 1－2 New Project 菜单

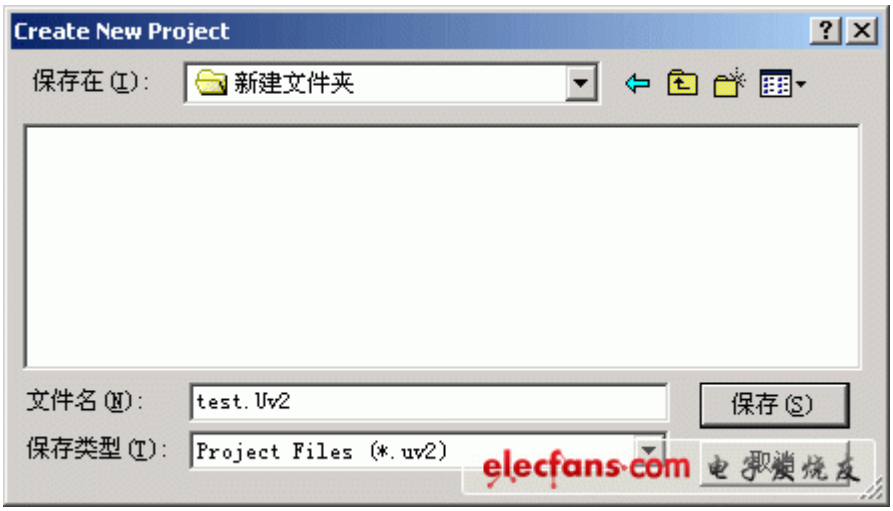


图 1－3 文件窗口

（2）选择所要的单片机，这里选择常用的 Ateml 公司的 AT89c51。而且本单片机 c 语言教程里的大部分程序都是基于此芯片的，此时屏幕如图 1－4 所示。AT89c51 有什么功能、特点呢？看图中右边有简单的介绍。完成上面步骤后，就可 以进行程序的编写了。

（3）首先在项目中创建新的程序文件或加入旧程序文件。如果您没有现成的程序，那么就要新建一个程序文件。在 KEIL 中有一些程序的 Demo，在这里我们还是以一个 C 程序 为例介绍如何新建一个 C 程序和如何加到您的第一个项目中吧。点击图 1－5 中 1 的新建文 件的快捷按钮，在 2 中出现一个新的文字编辑窗口，这个操作也能通过菜单 File －New 或 快捷键 Ctrl+N 来实现。好了，现在能编写程序了。

下面是经典的一段程序，呵，如果您看过别的程序书也许也有类似的程序：

```
#include 《AT89X51.H》
```

```
#include 《stdio.h》

void main (void)

{

SCON = 0x50; //串口方式 1，允许接收

TMOD = 0x20; //定时器 1 定时方式 2

TCON = 0x40; //设定时器 1 开始计数

TH1 = 0xE8; //11.0592MHz 1200 波特率

TL1 = 0xE8;

TI = 1;

TR1 = 1; //启动定时器

while (1)

printf (Hello World! \n) ;; //显示 Hello World

}
```

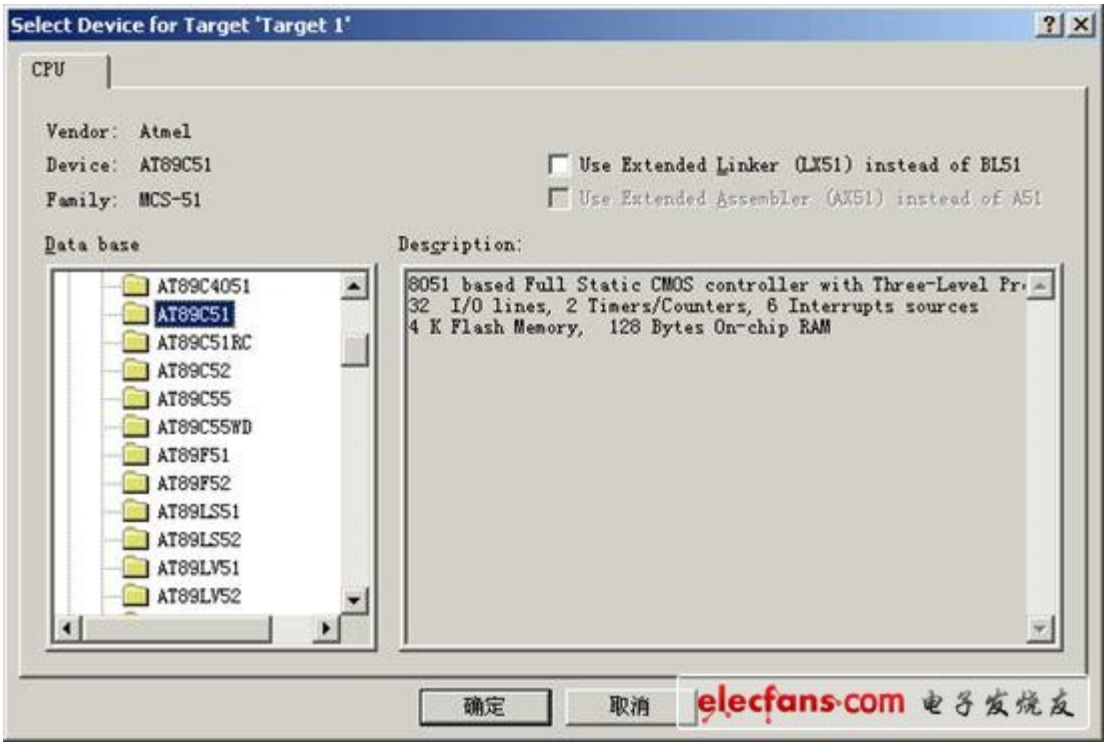


图 1—4 选取芯片

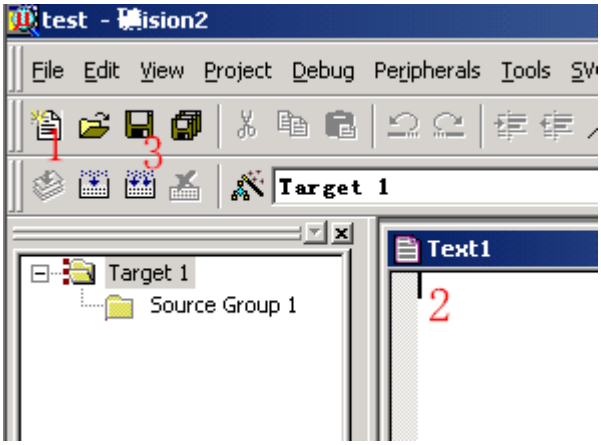


图 1－5 新建程序文件

这段程序的功能是不断从串行口输出“Hello World!”字符，先不管程序的语法和意思吧，先 看看如何把它加入到项目中和如何编译试运行。

(4)点击图 1－5 中的 3 保存新建的程序,也能用菜单 File－Save 或快捷键 Ctrl+S

进行保存。因是新文件所以保存时会弹出类似图 1－3 的文件操作窗口，把第一个程序命名

为 test1.c，保存在项目所在的目录中，这个时候您会发现程序单词有了不一样的颜色,说明 KEIL 的 C 语言语法检查生效了。如图 1－6 鼠标在屏幕左边的 Source Group1 文件夹图标上右击弹出 菜单，在这里能做在项目中增加减少文件等操作。选“Add File to Group ‘Source Group 1’” 弹出文件窗口，选择刚刚保存的文件，按 ADD 按钮，关闭文件窗，程序文件已加到项目中了。这个时候在 Source Group1 文件夹图标左边出现了一个小+号说明，文件组中有了文件，点击它能展开查看。

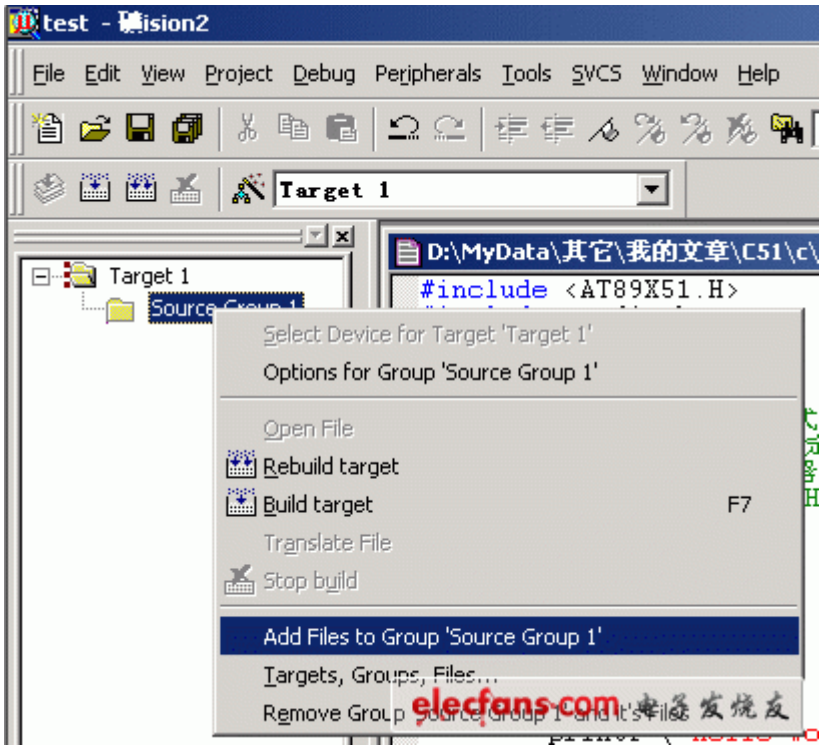


图 1－6 把文件加入到项目文件组中

（5）C 程序文件已被加到了项目中了，下面就剩下编译运行了。这个项目只是用做学习新建程序项目和编译运行仿真的基本方法，所以使用软件默认的编译设置，它不会生成用 于芯片烧写的 HEX 文件。先来看图 1－7 吧，图中 1、2、3 都是编译按钮，不一样是 1 是用 于编译单个文件。2 是编译链接当前项目，如果先前编译过一次之后文件没有做动编辑改动， 这个时候再点击是不会再次重新编译的。3 是重新编译，每点击一次均会再次编译链接一次，不 管程序是否有改动。在 3 右边的是停止编译按钮，只有点击了前三个中的任一个，停止按钮 才会生效。5 是菜单中的它们。在 4 中能看到编译的错误信息和使用的系统资源情况等， 以后我们要查错就靠它了。6 是有一个小放大镜的按钮，这就是开启\关闭调试模式的按钮， 它也存在于菜单 Debug－Start\Stop Debug Session，快捷键为 Ctrl+F5。

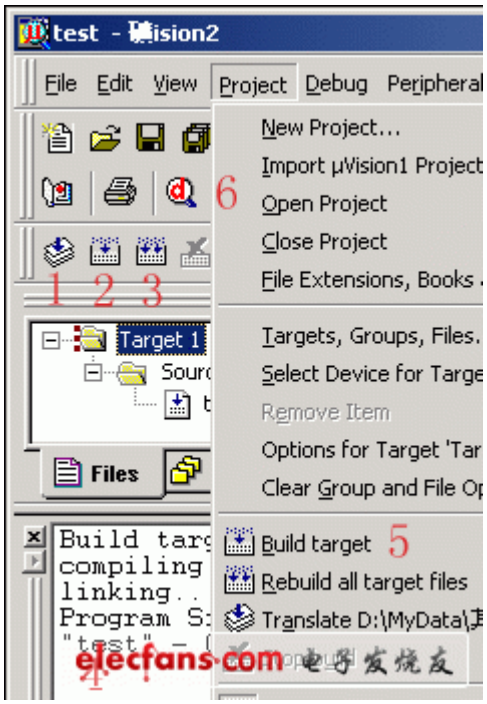


图 1－7 编译程序

（6）进入调试模式，软件窗口样式大致如图 1－8 所示。图中 1 为运行，当程序处于停止 状态时才有效，2 为停止，程序处于运行状态时才有效。3 是复位，模拟芯片的复位,程序 回到最开头处执行。按 4 能打开 5 中的串行调试窗口,这个窗口能看到从 51 芯片的串 行口输入输出的字符，这里的第一个项目也正是在这里看运行结果。这些在菜单中也有。首 先按 4 打开串行调试窗口，再按运行键，这个时候就能看到串行调试窗口中不断的打印“Hello World! ”。最后要停止程序运行回到文件编辑模式中，就要先按停止按钮再按开启\关闭调试 模式按钮。然后就能进行关闭 KEIL 等相关操作了。

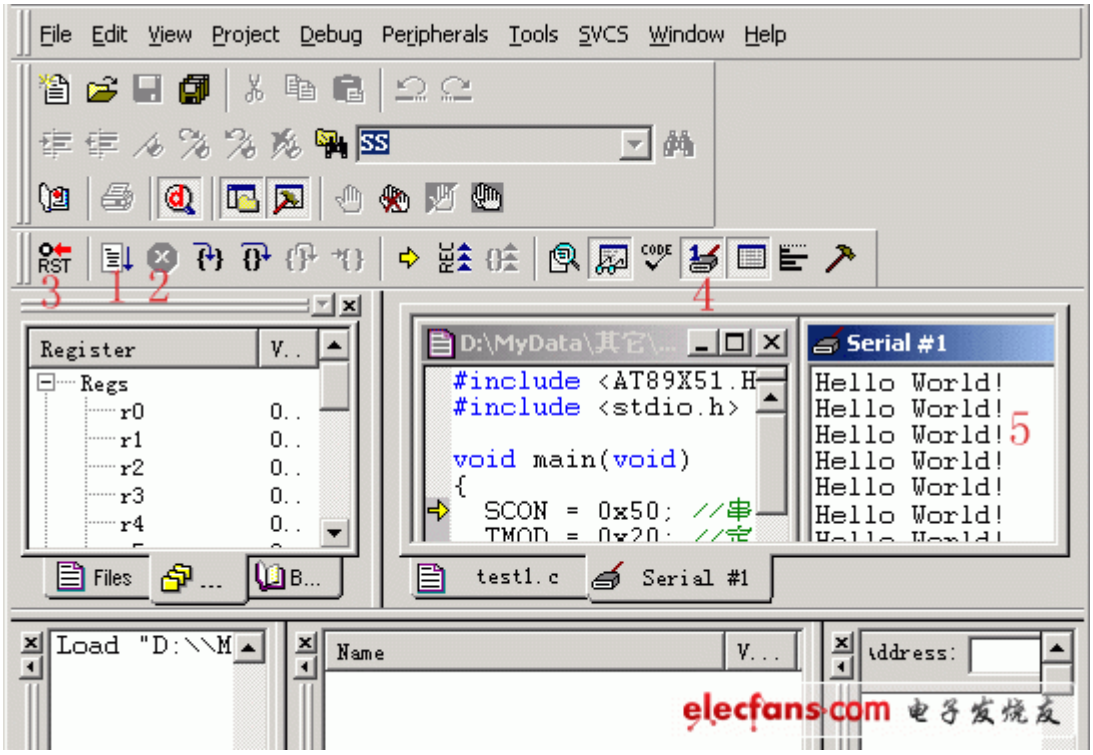


图 1—8 调试运行程序

第二课、C51HEX 文件的生成和单片机最小系统

上一篇建立了第一个单片机 C 语言项目，但为了让编译好的程序能通过编程器写入 51 芯 片中，要先用编译器生成 HEX 文件，下面来看看如何用 KEIL uVISION2 来编译生成用于烧写 芯片的 HEX 文件。HEX 文件格式是 Intel 公司提出的按地址排列的数据信息，数据宽度为字 节，所有数据使用 16 进制数字表示， 常用来保存单片机或其他处理器的目标程序代码。它保 存物理程序存储区中的目标代码映象。一般的编程器都支持这种格式。我们先来打开第一个 项目，打开它的所在目录，找到 test.Uv2 的文件就能打开先前的项目了。然后右击图 2—1 中的 1 项目文件夹，弹出项目功能菜单，选 Options for Target’ Target1’，弹出项目选项设置窗口， 同样先选中项目文件夹图标，这个时候在

Project 菜单中也有一样的菜单可选。打开项目选项窗口， 转到 Output 选项页图 2—2 所示，图中 1 是选择编译输出的路径，2 是设置编译输出生成的文件 名，3 则是决定是否要创建 HEX 文件，选中它就能输出 HEX 文件到指定的路径中。选好了？ 好，我们再将它重新编译一次，很快在编译信息窗口中就显示 HEX 文件创建到指定的路径 中了，如图 2—3。这样我们就可用自己的编程器所附带的软件去读取并烧到芯片了，再用实 验板看结果，至于编程器或仿真器品种繁多具体方法就看它的说明书了，这里也不做讨论。

（技巧：一、在图 2—1 中的 1 里的项目文件树形目录中，先选中对象，再单击它就可对它进 行重命名操作，双击文件图标便可打开文件。二、在 Project 下拉菜单的最下方有最近编辑过

的项目路径保存，这里能快速打开最近在编辑的项目。）

图 2—1 项目功能菜单

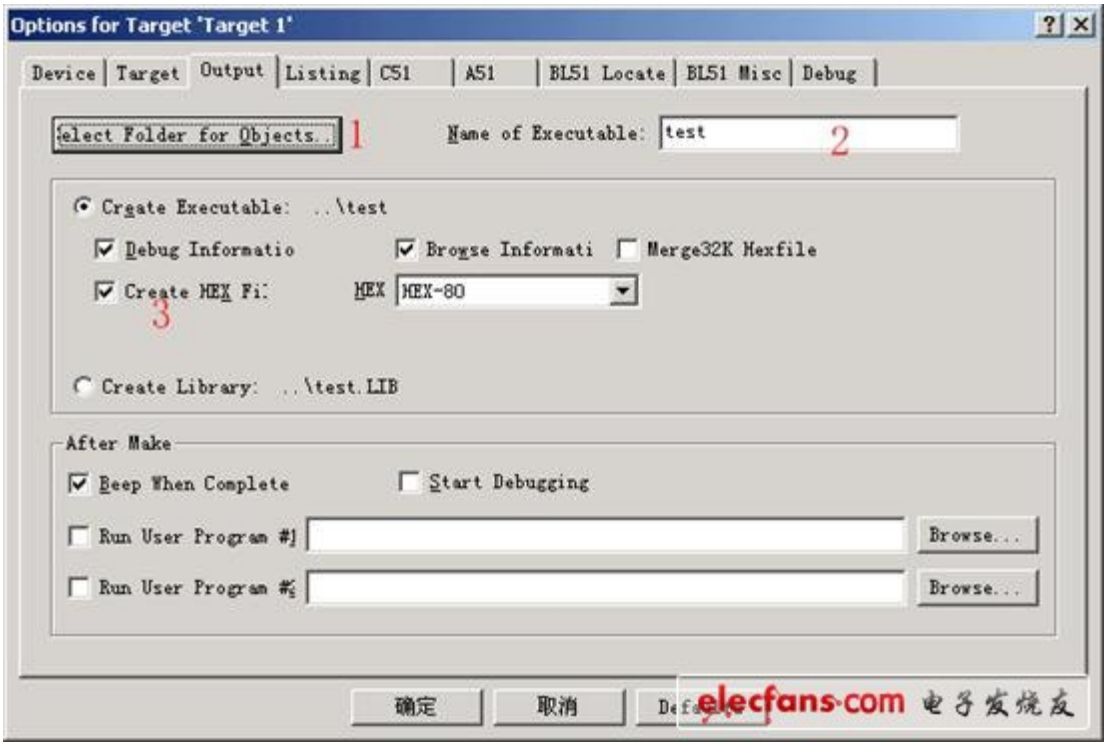


图 2—2 项目选项窗口

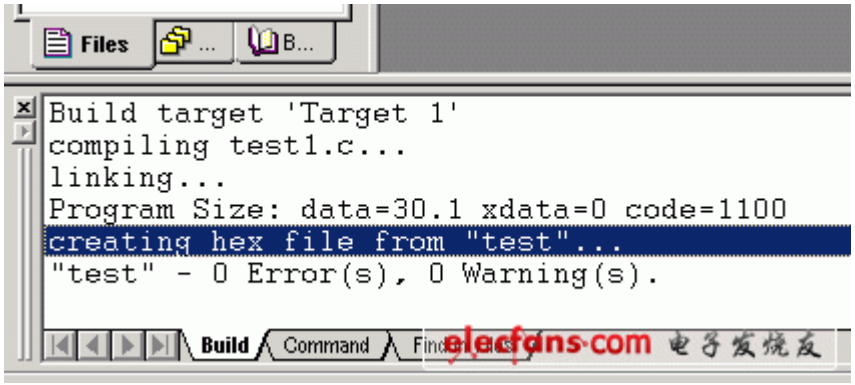


图 2—3 编译信息窗口

或许您已把编译好的文件烧到了芯片上，如果您购买或自制了带串行口输出元件的学习实 验板，那您就能把串行口和 PC 机串行口相联用串行口调试软件或 Windows 的超级终端，将其波特 率设为 1200，就能看到不停输出的“Hello World! ”字样。如果您还没有实验板，那这 里先说说 AT89c51 的最小化系统，再以一实例程序验证最小化系统是否在运行，这个最小化 系统也易于自制用于实验。图 2—4 便是 AT89c51 的最小化系统，不过为了让我们能看出它 是在运行的，加了一个电阻和一个 LED，用以显示它的状态，晶体震荡器能根据自己的情况使用， 一般实验板上是用 11.0592MHz 或 12MHz，使用前者的好外是能产生标准的串行口波特率，后 者则一个机器周期为 1 微秒，便于做精确定时。在自己做实验里，注意的是 VCC 是+5V 的， 不能高于此值，不然将损坏单片机，太低则不能正常工作。在 31 脚要接高电平，这样我们 才能执行片内的程序，如接低电平则使用片外的程序存储器。下面建一个新的项目名为 OneLED 来验证最小化系统是否能工作（所有的例程都可在笔者的主页下面下载到，网址： http://www.51hei.com 。程序如下：

```
#include 《 AT89X51.h》 //预处理命令

void main（void） //主函数名

{

//这是第一种注释方式

unsigned int a; //定义变量 a 为 int 类型

/* 这是第二种注释方式

*/

do{ //do while 组成循环
```

```
    for （a=0; a《50000; a++）; //这是一个循环 P1_0 = 0; //设 P1.0 口为低电平，点亮 LED for （a=0; a《50000; a++）; //这是一个循环 P1_0 = 1; //设 P1.0 口为高电平，熄灭 LED

    }

    while（1）;

    }
```

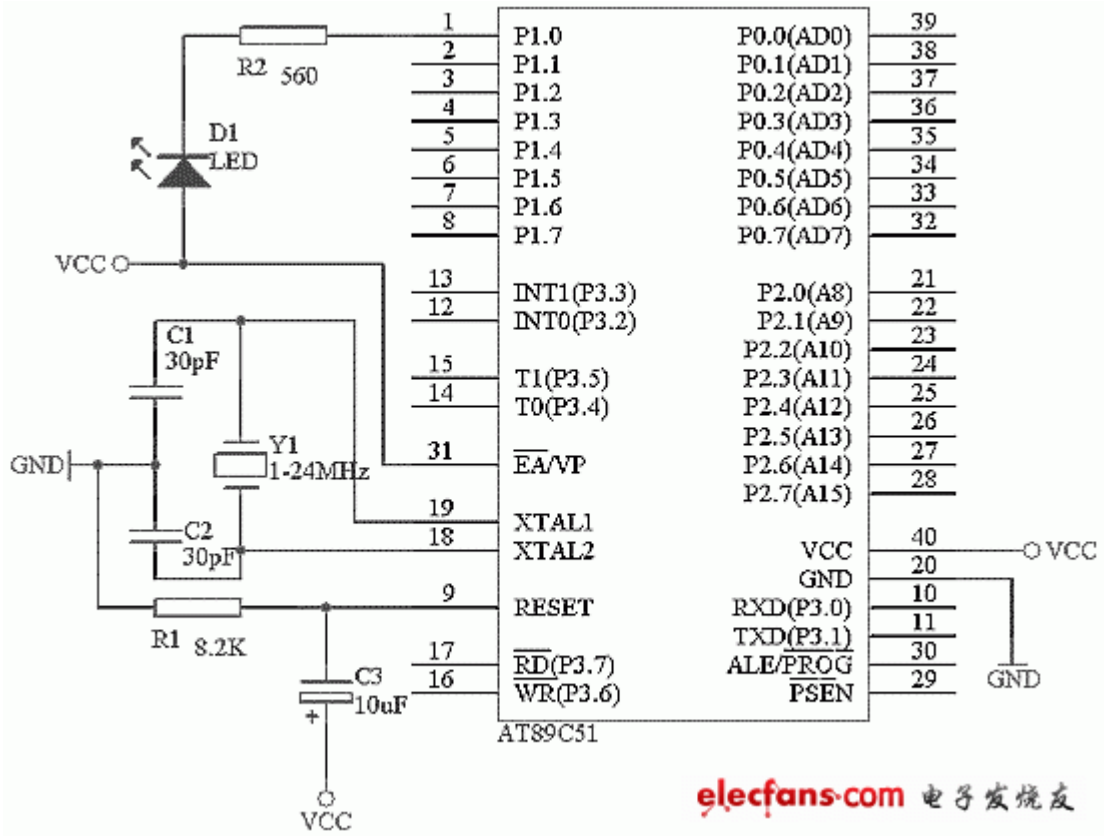


图 2－4 AT89c51 最小化系统

这里先讲讲 KEIL C 编译器所支持的注释语句。一种是以“//”符号开始的语句，符号之后 的语句都被视为注释，直到有回车换行。另一种是在“/*”和“*/”符号之内的为注释。注 释不会被 C 编译器所编译。一个 C 应用程序中应有一个 main 主函数，main 函数能调用别

的功能函数，但其它功能函数不允许调用 main 函数。不论 main 函数放在程序中的那个位置， 总是先被执行。用上面学到的知识编译写好的 OneLED 程序，并把它烧到刚做好的最小化系 统中。上电，刚开始时 LED 是不亮的（因为上电复位后所有的 IO 口都置 1 引脚为高电平）， 然后延时一段时间（for （a=0；a《50000；a++）这句在运行），LED 亮，再延时，LED 熄灭， 然后交替亮、灭。第一个真正的小实验就做完，如果没有这样的效果那么您就要认真检查一下电路或编译烧写的步骤了。

第三课、C51 数据类型

每写一个程序，总离不开数据的应用，在学习 c51 语言的过程中掌握理解数据类型也是 很关键的。先看表 3－1，表中列出了 KEIL uVision2 单片机 c 语言编译器所支持的数据类型。在标准 C 语言中基本的数据类型为 char，int，short，long，float 和 double，而在 c51 编译器中 int 和 short 相同，float 和 double 相同，这里就不列出说明了。下面来看看它们的具体定 义：

数据类型	长　　度	值　　域
unsigned　char	单字节	0～255
signed　char	单字节	-128～+127
unsigned　int	双字节	0～65535
signed　int	双字节	-32768～+32767
unsigned　long	四字节	0～4294967295
signed　long	四字节	-2147483648～+2147483647
float	四字节	±1.175494E-38～±3.402823E+38
*	1～3 字节	对象的地址
bit	位	0 或 1
sfr	单字节	0～255
sfr16	双字节	0～65535
sbit	位	0 或 1

表 3－1 KEIL uVision2 单片机 c 语言编译器所支持的数据类型

1.　char 字符类型

char 类型的长度是一个字节，通常用于定义处理字符数据的变量或常量。分无符号字符类型 unsigned char 和有符号字符类型 signed char，默认值为 signed char 类型。unsigned char 类型用字节中所有的位来表示数值，所能表达的数值范围是 0～255。signed char 类型用字节中最高位字节表示数据的符号，“0”表示正数，“1”表示负数，负数用补码表示。所能表示的数值范围是-128～+127。unsigned char 常用于处理 ASCII 字符或用于处理小于或等于 255 的整型数。

＊正数的补码与原码相同，负二进制数的补码等于它的绝对值按位取反后加 1。

2.　int 整型

int 整型长度为两个字节，用于存放一个双字节数据。分有符号 int 整型数 signed int 和无符号整型数 unsigned int，默认值为 signed int 类型。signed int 表示的数值范 围是-32768～+32767，字节中最高位表示数据的符号，“0”表示正数，“1”表示负数。 unsigned int 表示的数值范围是 0～65535。

先停一下来写个小程序看看 unsigned char 和 unsigned int 用于延时的不一样效果，说 明它们的长度是不一样的，学习它们的使用方法。依旧用上一篇的最小化系统做实验，不过要加多 一个电阻和 LED，如图 3－1。实验中用 D1 的点亮表明正在用 unsigned int 数值延时，用

D2 点亮表明正在用 unsigned char 数值延时。

图 3－1 第 3 课实验用电路 把这个项目称为 TwoLED，实验程序如下：

```
#include 《AT89X51.h》 //预处理命令

void main（void） //主函数名

{

unsigned int a; //定义变量 a 为 unsigned int 类型

unsigned char b; //定义变量 b 为 unsigned char 类型

do

{ //do while 组成循环

for （a=0; a《65535; a++）

P1_0 = 0; //65535 次设 P1.0 口为低电平，点亮 LED P1_0 = 1; //设 P1.0 口为高电平，熄灭 LED
```

```
    for （a=0; a《30000; a++）; //空循环

    for （b=0; b《255; b++）

    P1_1 = 0; //255 次设 P1.1 口为低电平，点亮 LED P1_1 = 1; //设 P1.1 口为高电平，熄灭 LED

    for （a=0; a《30000; a++）; //空循环

    }

    while（1）;

    }
```

同样编译烧写，上电运行您就能看到结果了。很明显 D1 点亮的时间长于 D2 点亮的时间。

这里必须要讲的是，当定义一个变量为特定的数据类型时，在程序使用该变量不应使它的值 超过数据类型的值域。如本例中的变量 b 不能赋超出 0～255 的值，如 for（b=0; b《255; b++） 改为 for （b=0; b《256; b++），编译是能通过的，但运行时就会有问题出现，就是说 b 的 值永远都是小于 256 的，所以无法跳出循环执行下一句 P1_1 = 1，从而造成死循环。同理 a 的值不应超出 0～65535。

3. long 长整型

long 长整型长度为四个字节，用于存放一个四字节数据。分有符号 long 长整型 signed long 和无符号长整型 unsigned long，默认值为 signed long 类型。signed int 表示 的数值范围是-2147483648～+2147483647，字节中最高位表示数据的符号，“0”表示正 数，“1”表示负数。unsigned long 表示的数值范围是 0～4294967295。

4. float 浮点型

float 浮点型在十进制中具有 7 位有效数字，是符合 IEEE－754 标准的单精度浮点型数 据，占用四个字节。因浮点数的结构较复杂在以后的章节中再做详细的讨论。

5.* 指针型 指针型本身就是一个变量，在这个变量中存放的指向另一个数据的地址。这个指针变量 要占据一定的内存单元，对不一样的处理器长度也不尽相同，在 c51 中它的长度一般为 1～

3 个字节。指针变量也具有类型，在以后的课程中有专门一课做探讨，这里就不多说了。

6. bit 位标量

bit 位标量是 c51 编译器的一种扩充数据类型，利用它可定义一个位标量，但不能定义 位指针，也不能定义位数组。它的值是一个二进制位，不是 0 就是 1，类似一些高级语 言中的 Boolean 类型中的 True 和 False。

7. sfr 特殊功能寄存器

sfr 也是一种扩充数据类型，占用一个内存单元，值域为 0～255。利用它能访问 51 单片机内部的所有特殊功能寄存器。如用 sfr P1 = 0x90 这一句定 P1 为 P1 端口在片内 的寄存器，在后面的语句中用以用 P1 = 255（对 P1 端口的所有引脚置高电平）之类的 语句来操作特殊功能寄存器。

8. sfr16 16 位特殊功能寄存器

sfr16 占用两个内存单元，值域为 0～65535。sfr16 和 sfr 一样用于操作特殊功能寄存 器，所不一样的是它用于操作占两个字节的寄存器，如定时器 T0 和 T1。

9. sbit 可录址位

sbit 同样是 单片机 c 语言 中的一种扩充数据类型，利用它能访问芯片内部的 RAM 中的可寻址

位或特殊功能寄存器中的可寻址位。如先前定义了

```
sfr P1 = 0x90; //因 P1 端口的寄存器是可位寻址的，所以能定义

sbit P1_1 = P1^1; //P1_1 为 P1 中的 P1.1 引脚
```

//同样我们能用 P1.1 的地址去写，如 sbit P1_1 = 0x91; 这样在以后的程序语句中就能用 P1_1 来对 P1.1 引脚进行读写操作了。通常这些能 直接使用系统供给的预处理文件，里面已定义好各特殊功能寄存器的简单名字，直接引 用能省去一点时间，我自己是一直用的。当然您也能自己写自己的定义文件，用您 认为好记的名字。

第四课、C51 常量

上一篇学习了 KEIL c 单片机 c 语言 编译器所支持的数据类型。而这些 c51 数据类型又是怎么用在常量和变量的定义中的呢？又有什么要注意的吗？常量就是在程序运行过程中不能改变值的量，而变 量是能在程序运行过程中不断变化的量。变量的定义能使用所有 c51 编译器支持的数据类型，而常量的数据类型只有整型、浮点型、字符型、字符串型和位标量。 这一篇学习常量定义和使用方法，而下一篇则学习单片机 c 语言的变量。

常量的数据类型说明是这样的

1. 整型常量能表示为十进制如 123，0，－89 等。十六进制则以 0x 开头如 0x34，-0x3B 等。长整型就在数字后面加字母 L，如 104L，034L，0xF340 等。

2. 浮点型常量可分为 十进 制和指数表示形式 。十 进制由数字和小数点组成，如 0.888，3345.345，0.0 等，整数或小数部分为 0，能省略但必须有小数点。指数表 示形式为［±］数字［。数字］e［±］数字，［］中的内容为可选项，其中内容根据具体情 况可有可无，但其余部分必须有，如 125e3，7e9，-3.0e-3。

3. 字符型常量是单引号内的字符，如‘a’，‘d’等，不能显示的控制字符，能 在该字符前面加一个反斜杠“\”组成专用转义字符。常用转义字符表请看表 4－1。

4. 字符串型常量由双引号内的字符组成，如“test”，“OK”等。当引号内的没有字 符时，为空字符串。在使用特殊字符时同样要使用转义字符如双引号。在 C 中字符 串常量是做为字符类型数组来处理的，在存储字符串时系统会在字符串尾部加上\o 转义字符以作为该字符串的结束符。字符串常量“A”和字符常量‘A’是不一样的， 前者在存储时多占用一个字节的字间。

5. 位标量，它的值是一个二进制。

表 4－1 常用转义字符表

转义字符	含义	ASCII 码（16/10 进制）
\o	空字符(NULL)	00H/0
\n	换行符(LF)	0AH/10
\r	回车符(CR)	0DH/13
\t	水平制表符(HT)	09H/9
\b	退格符(BS)	08H/8
\f	换页符(FF)	0CH/12
\’	单引号	27H/39
\”	双引号	22H/34
\\	反斜杠	5CH/92

常量可用在不必改变值的场合，如固定的数据表，字库等。常量的定义方式有几种，下 面来加以说明。

```
#difine False 0x0; //用预定义语句能定义常量

#difine True 0x1; //这里定义 False 为 0，True 为 1

//在程序中用到 False 编译时自动用 0 替换，同理 True 替换为 1

unsigned int code a=100; //这一句用 code 把 a 定义在程序存储器中并赋值
```

const unsigned int c=100; //用 const 定义 c 为无符号 int 常量并赋值 以上两句它们的值都保存在程序存储器中，而程序存储器在运行中是不允许被修改的，

所以如果在这两句后面用了类似 a=110，a++这样的赋值语句，编译时将会出错。

下面写个跑马灯程序来实验一下典型的常量使用方法。先来看看电路图吧。它是在上一篇的

实验电路的基础上增加几个 LED 组成的，也就是用 P1 口的全部引脚分别驱动一个 LED，电 路如图 4－1 所示。

新建一个 RunLED 的项目，主程序如下：

```
#include 《AT89X51.H》 //预处理文件里面定义了特殊寄存器的名称如 P1 口定义为 P1

void main (void)

{

//定义花样数据

const unsigned char design [32] ={0xFF, 0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F,

0x7F, 0xBF, 0xDF, 0xEF, 0xF7, 0xFB, 0xFD, 0xFE, 0xFF,

0xFF, 0xFE, 0xFC, 0xF8, 0xF0, 0xE0, 0xC0, 0x80, 0x0,

0xE7, 0xDB, 0xBD, 0x7E, 0xFF};

unsigned int a; //定义循环用的变量
```

```
unsigned char b; //在 c51 编程中因内存有限尽可能注意变量类型的使用
//尽可能使用少字节的类型，在大型的程序中很受用
do{
for （b=0; b《32; b++）
{
}
}while（1）;
}
for（a=0; a《30000; a++）; //延时一段时间
P1 = design [b]; //读已定义的花样数据并写花样数据到 P1 口
```

程序中的花样数据能自以去定义，因这里我们的 LED 要 AT89c51 的 P1 引脚为低电平才 会点亮,所以我们要向 P1 口的各引脚写数据 0 对应连接的 LED 才会被点亮,P1 口的八个引 脚刚好对应 P1 口特殊寄存器的八个二进位，如向 P1 口定数据 0xFE，转成二进制就是

11111110，最低位 D0 为 0 这里 P1.0 引脚输出低电平，LED1 被点亮。如此类推，大家不难算 出自己想要做的效果了。大家编译烧写看看，效果就出来，显示的速度您能根据需要调整 延时 a 的值，不要超过变量类型的值域就很行了。哦，您还没有实验板？那如何能知道程 序运行的结果呢？呵，不用急，这就来说说用 KEIL uVision2 的软件仿真来调试 IO 口输出输入程序。

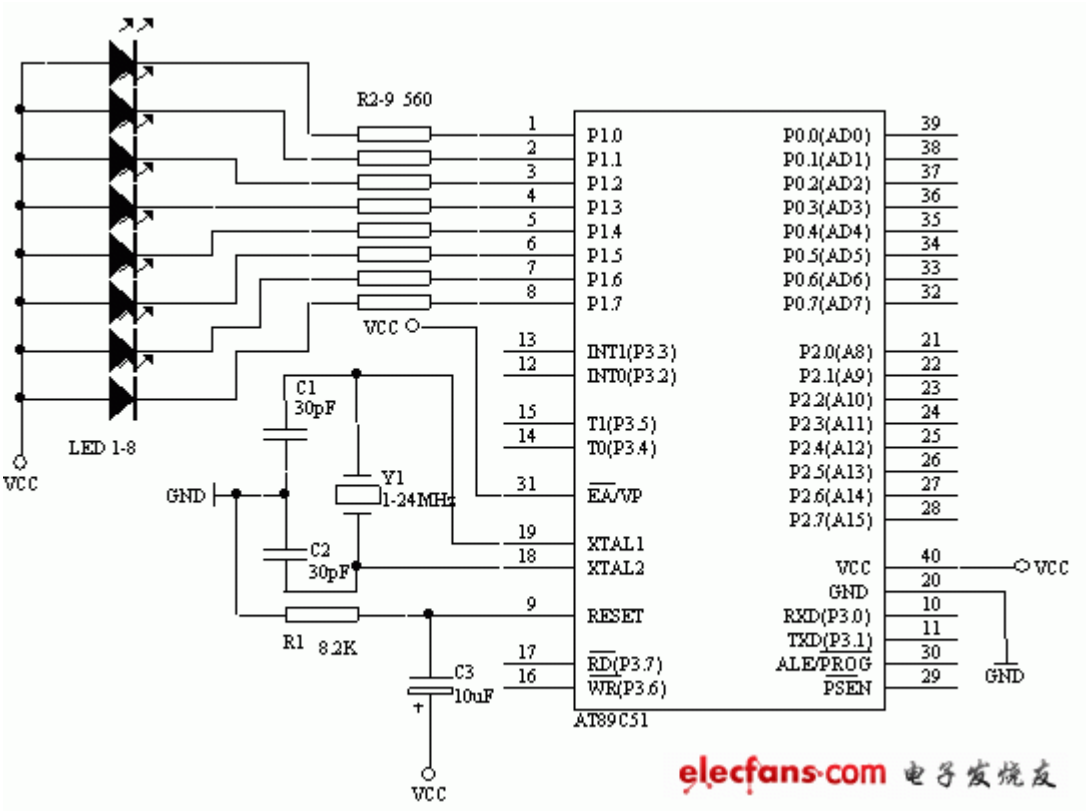


图 4－1 八路跑马灯电路 编译运行上面的程序，然后按外部设备菜单 Peripherals－I/O Ports－Port1 就打开

Port1 的调试窗口了，如图 4－3 中的 2。这个时候程序运行了，但我们并不能在 Port1 调试窗口 上看到有会什么效果，这个时候能用鼠标左击图 4－3 中 1 旁边绿色的方条，点一下就有一个 小红方格再点一下又没有了，哪一句语句前有小方格程序运行到那一句时就停止了，就是设 置调试断点，同样图 4－2 中的 1 也是同样功能，分别是增加/移除断点、移除所有断点、允 许/禁止断点、禁止所有断点，菜单也有一样的功能，另外菜单中还有 Breakpoints 可打开 断点设置窗口它的功能更强大，不过这里先不用它。在“P1 = design [b] ;”这一句设置一 个断点这个时候程序运行到这里就停住了，再留意

一下 Port1 调试窗口，再按图 5-2 中的 2 的运 行键，程序又运行到设置断点的地方停住了，这个时候 Port1 调试窗口的状态又不一样了。也就是说 Port1 调试窗口模拟了 P1 口的电平状态，打勾为高电平，不打勾则为低电平，窗口中 P1 为 P1 寄存器的状态，Pins 为引脚的状态，注意的是如果是读引脚值之前必须把引脚对应的 寄存器置 1 才能正确读取。图 4－2 中 2 旁边的{}样的按钮分别为单步入，步越，步出和 执行到当前行。图中 3 为显示下一句将要执行的语句。图 4－3 中的 3 是 Watches 窗口可查 看各变量的当前值，数组和字串是显示其头一个地址，如本例中的 design 数组是保存在 code 存储区的首地址为 D:0x08，能在图中 4 Memory 存储器查看窗口中的 Address 地址中打入 D:0x08 就能查看到 design 各数据和存放地址了。如果你的 uVision2 没有显示这些窗口，能在 View 菜单中打开在图 4－2 中 3 后面一栏的查看窗口快捷栏中打开。

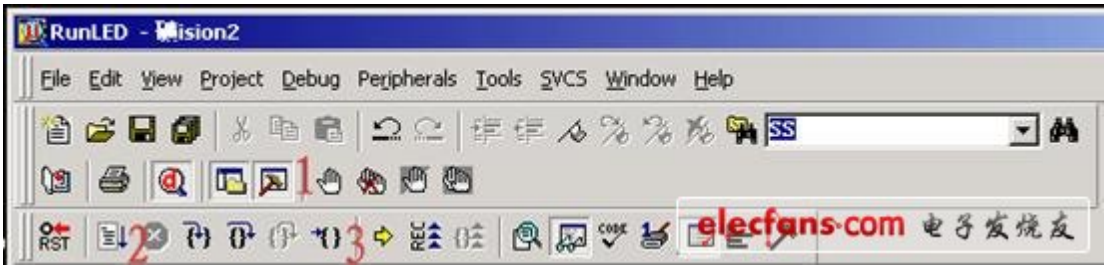


图 4－2 调试用快捷菜单栏

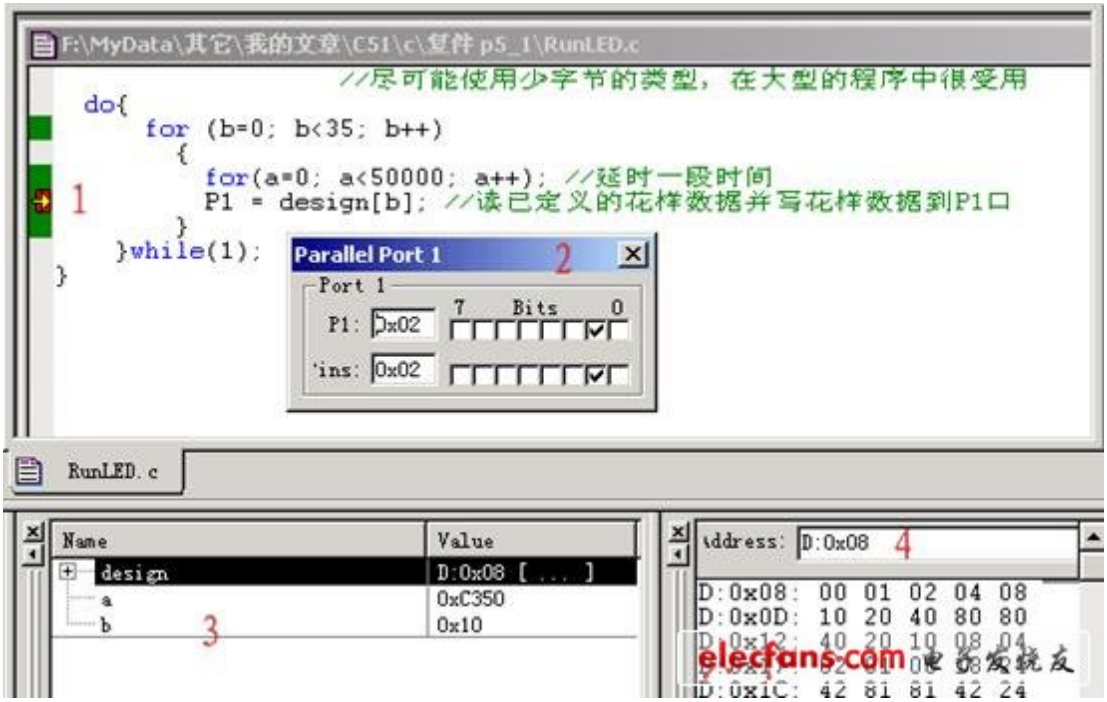


图 4－3 各调试窗口

单片机 C 语言知识点全攻略（二）

电子发烧友网讯：继《单片机学习知识点全攻略》得到广大读者好评，根据有网友提出美中不足的是所用单片机编程语言为汇编，基于此，电子发烧友网再接再厉再次为读者诚挚奉上非常详尽的《单片机 C 语言知识点全攻略》系列单片机 C 语言学习教程，本教程共分为四部分，主要知识点如下所示。

参阅相关系列文章，

单片机 C 语言知识点全攻略（一）

第二部分知识点：

- 第五课 C51 变量
- 第六课 C51 运算符和表达式

- 第七课 运算符和表达式（关系运算符）
- 第八课 运算符和表达式（位运算符）
- 第九课 C51 运算符和表达式（指针和地址运算符）

第五课、C51 变量

上课所提到变量就是一种在程序执行过程中其值能不断变化的量。要在程序中使用变量必须先用标识符作为变量名，并指出所用的数据类型和存储模式，这样编译系统才能为变量分配相应的存储空间。定义一个变量的格式如下：

［存储种类］ 数据类型 ［存储器类型］ 变量名表

在定义格式中除了数据类型和变量名表是必要的，其它都是可选项。存储种类有四种：自动（auto），外部（extern），静态（static）和寄 存器（register），缺省类型为自动（auto）。这些存储种类的具体含义和使用方法，将在第七课《变量的存储》中进一步进行学习。

而这里的数据类型则是和我们在第四课中学习到的名种数据类型的定义是一样的。说明了一个变量的数据类型后，还可选择说明该变量的存储器类型。存储器类 型的说明就是指定该变量在单片机 c 语言硬件系统中所使用的存储区域，并在编译时准确的定位。表 6－1 中是 KEIL uVision2 所能认别的存储器类型。注意的是在 AT89c51 芯片中 RAM 只有低 128 位，位于 80H 到 FFH 的高 128 位则在 52 芯片中才有用，并 和特殊寄存器地址重叠。特殊寄存器（SFR）的地址表请看附录二 AT89c51 特殊功能寄存器列表

表6－1　存储器类型

存储器类型	说　明
data	直接访问内部数据存储器（128字节),访问速度最快
bdata	可位寻址内部数据存储器（16字节），允许位与字节混合访问
idata	间接访问内部数据存储器（256字节），允许访问全部内部地址
pdata	分页访问外部数据存储器（256字节），用MOVX @Ri指令访问
xdata	外部数据存储器(64KB)，用MOVX @DPTR指令访问
code	程序存储器（64KB）,用MOVC @A+DPTR指令访问

如果省略存储器类型，系统则会按编译模式 SMALL，COMPACT 或 LARGE 所规定的默认存储器类型去指定变量的存储区域。无论什么存储模式都能 声明变量在任何的 8051 存储区范围，然而把最常用的命令如循环计数器和队列索引放在内部数据区能显著的提高系统性能。还有要指出的就是变量的存储种类与 存储器类型是完全无关的。

数据存储模式

存储模式决定了没有明确指定存储类型的变量，函数参数等的缺省存储区域，共三种：

1. 1. Small 模式

所有缺省变量参数均装入内部 RAM，优点是访问速度快，缺点是空间有限，只适用于小程序。

2. 2. Compact 模式

所有缺省变量均位于外部 RAM 区的一页（256Bytes），具体哪一页可由 P2 口指定，在 STARTUP.A51 文件中说明，也可用 pdata 指定，优点是空间较 Small 为宽裕速度较 Small 慢，较 large 要快，是一种中间状态。

3. 3. large 模式

所有缺省变量可放在多达 64KB 的外部 RAM 区，优点是空间大，可存变量多，缺点是速度较慢。

提示：存储模式在单片机 c 语言编译器选项中选择。

之前提到简单提到 sfr，sfr16，sbit 定义变量的方法，下面我们再来仔细看看。

sfr 和 sfr16 能直接对 51 单片机的特殊寄存器进行定义，定义方法如下：

sfr 特殊功能寄存器名= 特殊功能寄存器地址常数;

sfr16 特殊功能寄存器名= 特殊功能寄存器地址常数;

我们能这样定义 AT89c51 的 P1 口

sfr P1 = 0x90; //定义 P1 I/O 口，其地址 90H

sfr 关键定后面是一个要定义的名字，可任意选取，但要符合标识符的命名规则，名字最好有一定的含义如 P1 口能用 P1 为名，这样程序会变的好读好多。 等号后面必须是常数，不允许有带运算符的表达式，而且该常数必须在特殊功能寄存器的地址范围之内（80H－FFH），具体可查看附录中的相关表。sfr 是 定义 8 位的特殊功能寄存器而 sfr16 则是用来定义 16 位特殊功能寄存器，如 8052 的 T2 定时器，能定义为：

sfr16 T2 = 0xCC; //这里定义 8052 定时器 2，地址为 T2L=CCH，T2H=CDH

用 sfr16 定义 16 位特殊功能寄存器时，等号后面是它的低位地址，高位地址一定要位于物理低位地址之上。注意的是不能用于定时器 0 和 1 的定义。

sbit 可定义可位寻址对象。如访问特殊功能寄存器中的某位。其实这样应用是经常要用的如要访问 P1 口中的第 2 个引脚 P1.1。我们能照以下的方法去定义：

（1）sbit 位变量名＝位地址

```
sbit P1_1 = 0x91;
```

这样是把位的绝对地址赋给位变量。同 sfr 一样 sbit 的位地址必须位于 80H-FFH 之间。

（2）Sbit 位变量名＝特殊功能寄存器名^位位置

```
sft P1 = 0x90;
sbit P1_1 = P1 ^ 1; //先定义一个特殊功能寄存器名再指定位变量名所在的位置
```

当可寻址位位于特殊功能寄存器中时可采用这种方法

（3）sbit 位变量名＝字节地址^位位置

```
sbit P1_1 = 0x90 ^ 1;
```

这种方法其实和 2 是一样的，只是把特殊功能寄存器的位址直接用常数表示。

在单片机 c 语言存储器类型中供给有一个 bdata 的存储器类型，这个是指可位寻址的数据存储器，位于单片机的可位寻址区中，能将要求可位录址的数据定义为 bdata，如：

```
unsigned char bdata ib; //在可位录址区定义 ucsigned char 类型的变量 ib
int bdata ab [2] ; //在可位寻址区定义数组 ab [2] ，这些也称为可寻址位对象
```

```
sbit ib7=ib^7 //用关键字 sbit 定义位变量来独立访问可寻址位对象的其中一位

sbit ab12=ab [1] ^12;
```

操作符“ ^ ”后面的位位置的最大值取决于指定的基址类型，char0-7，int0-15，long0-31。

下面我们用上一课的电路来实践一下这一课的知识。同样是做一下简单的跑马灯实验，项目名为 RunLED2。程序如下：

```
sfr P1 = 0x90; //这里没有使用预定义文件，

sbit P1_0 = P1 ^ 0; //而是自己定义特殊寄存器

sbit P1_7 = 0x90 ^ 7; //之前我们使用的预定义文件其实就是这个作用

sbit P1_1 = 0x91; //这里分别定义 P1 端口和 P10，P11，P17 引脚

void main (void)

{

unsigned int a;

unsigned char b;

do{

for (a=0;a《50000;a++)

P1_0 = 0; //点亮 P1_0
```

```
for （a=0;a《50000;a++）
P1_7 = 0; //点亮 P1_7
for （b=0;b《255;b++）
{
for （a=0;a《10000;a++）
P1 = b; //用 b 的值来做跑马灯的花样
}
P1 = 255; //熄灭 P1 上的 LED
for （b=0;b《255;b++）
{
for （a=0;a《10000;a++） //P1_1 闪烁
P1_1 = 0;
for （a=0;a《10000;a++）
P1_1 = 1;
}
}while（1）;
```

}

。 Keil c51 指针变量

单片机 c 语言支持一般指针（Generic Pointer）和存储器指针（Memory_Specific Pointer）。

1. 1. 一般指针

一般指针的声明和使用均与标准 C 相同，不过同时还能说明指针的存储类型，例如：

long * state;为一个指向 long 型整数的指针，而 state 本身则依存储模式存放。

char * xdata ptr；ptr 为一个指向 char 数据的指针，而 ptr 本身放于外部 RAM 区，以上的 long，char 等指针指向的数据可存放于任何存储器中。

一般指针本身用 3 个字节存放，分别为存储器类型，高位偏移，低位偏移量。

2. 2. 存储器指针

基于存储器的指针说明时即指定了存贮类型，例如：

char data * str;str 指向 data 区中 char 型数据

int xdata * pow; pow 指向外部 RAM 的 int 型整数。

这种指针存放时，只需一个字节或 2 个字节就够了，因为只需存放偏移量。

3. 3. 指针转换

即指针在上两种类型之间转化：

1 当基于存储器的指针作为一个实参传递给需要一般指针的函数时，指针自动转化。

1 如果不说明外部函数原形，基于存储器的指针自动转化为一般指针，导致错误，因而请用“＃include”说明所有函数原形。

1 能强行改变指针类型。

变量的存储类别

一、static（静态局部）变量。

1、静态局部变量在程序整个运行期间都不会释放内存。

2、对于静态局部变量，是在编译的时候赋初值的，即只赋值一次。如果在程序运行时已经有初值，则以后每次调用的时候不再重新赋值。

3、如果定义局部变量的时候不赋值，则编译的时候自动赋值为 0。而对于自动变量而言，定义的时候不赋值，则是一个不确定的值。

4、虽然静态变量在函数调用结束后仍然存在，但是其他函数不能引用。

二、用 extern 声明外部变量。

用 extern 声明外部变量，是为了扩展外部变量的作用范围。比如一个程序能由多个源程序文件组成。如果一个程序中需要引用另外一个文件中已经定义的外部变量，就需要使用 extern 来声明。

正确的做法是在一个文件中定义外部变量，而在另外一个文件中使用 extern 对该变量作外部变量声明。

一个文件中： int abc;

另外一个文件中： extern abc;

例子：

用 extern 将外部变量的作用域扩展到其他文件：

文件 1：

```
//用 extern 将外部变量的作用域扩展到其他文件中
#include
#include
#include
unsigned int array [10] ;
void fillarray () ;
void init_ser ()
{
SCON=0X50;
TMOD|=0X20;
TH1=0XF3;
TR1=1;
```

```
TI=1;
}
void main ()
{
unsigned int i;
init_ser () ;
fillarray () ;
for (i=0;i《10;i++)
{
printf ( “array [%d] =%d\n”, i, array [i] ) ;
}
for (;;) {;}
}
```

文件 2：

```
extern int array [10] ;
void fillarray ()
```

```
{
unsigned char i;
for（i=0;i《10;i++）
{
array [i] =i;
}
}
```

在单片机 c 语言中变量的空间分配几个方法

1、 data 区空间小，所以只有频繁用到或对运算速度要求很高的变量才放到 data 区内，比如 for 循环中的计数值。

2、 data 区内最好放局部变量。

因为局部变量的空间是能覆盖的某个函数的局部变量空间在退出该函数是就释放，由别的函数的局部变量覆盖），能提高内存利用率。当然静态局部变量除外，其内存使用方式与全局变量相同;

3、 确保你的程序中没有未调用的函数。

在 Keil C 里遇到未调用函数，编译器就将其认为可能是中断函数。函数里用的局部变量的空间是不释放，也就是同全局变量一样处理。这一点 Keil C 做得很愚蠢，但也没办法。

4、 程序中遇到的逻辑标志变量能定义到 bdata 中，能大大降低内存占用空间。

在 51 系列芯片中有 16 个字节位寻址区 bdata，其中能定义 8*16=128 个逻辑变量。定义方法是： bdata bit LedState;但位类型不能用在数组和结构体中。

5、 其他不频繁用到和对运算速度要求不高的变量都放到 xdata 区。

6、如果想节省 data 空间就必须用 large 模式,将未定义内存位置的变量全放到 xdata 区。当然最好对所有变量都要指定内存类型。

7、 当使用到指针时，要指定指针指向的内存类型。

在单片机 c51 语言中未定义指向内存类型的通用指针占用 3 个字节；而指定指向 data 区的指针只占 1 个字节;指定指向 xdata 区的指针占 2 个字节。如指针 p 是指向 data 区，则应定义为： char data *p;。还可指定指针本身的存放内存类型，如：char data * xdata p;。其含义是指针 p 指向 data 区变量，而其本身存放在 xdata 区。

第六课、C51 运算符和表达式

上两课说了常量和变量，先来补充一个用以重新定义数据类型的的语句吧。这个语句就是 typedef，这是个很好用的语句，但我却不常用它，通常我定义变量的数据类型时都是使 用标准的关键字，这样别人能很方便的研读你的程序。如果你是个 DELPHI 编程爱好者或是 DELPHI 程序员，你对变量的定义也许习惯了 DELPHI 的关键字，如 int 类型常会用关键字 Integer 来定义，在用 单片机 c 语言时你还想用回这个的话，你能这样写：

```
typedef int integer;

integer a, b;
```

这两句在编译时,其实是先把 integer 定义为 int,在以后的语句中遇到 integer 就用 int 置换，integer 就等于 int，所以 a，b 也就被定义为 int。typedef 不能直接用来定义变量，它 只是对已有的数据类型作一个名字上的置换，并不是产生一个新的数据类型。下面两句就是一个错误的例子：

```
typedef int integer;

integer = 100;
```

使用 typedef 能有方便程序的移植和简化较长的数据类型定义。用 typedef 还能定义结 构类型，这一点在后面详细解说结构类型时再一并说明。typedef 的语法是

typedef 已有的数据类型 新的数据类型名 运算符就是完成某种特定运算的符号。运算符按其表达式中与运算符的关系可分为单目

运算符，双目运算符和三目运算符。单目就是指需要有一个运算对象，双目就要求有两个运 算对象，三目则要三个运算对象。表达式则是由运算及运算对象所组成的具有特定含义的式 子。C 是一种表达式语言，表达式后面加“；”号就构成了一个表达式语句。

赋值运算符

对于“=”这个符号大家不会陌生的，在 C 中它的功能是给变量赋值，称之为赋值运算 符。它的作用不用多说大家也明白，就是但数据赋给变量。如，x=10;由此可见利用赋值运 算符将一个变量与一个表达式连接起来的式子为赋值表达式，在表达式后面加“；”便构成 了赋值语句。使用“=”的赋值语句格式如下：

变量 = 表达式； 示例如下

```
a = 0xFF; //将常数十六进制数 FF 赋于变量 a

b = c = 33; //同时赋值给变量 b，c d = e; //将变量 e 的值赋于变量 d

f = a+b; //将变量 a+b 的值赋于变量 f 由上面的例子能知道赋值语句的意义就是先
```

计算出“=”右边的表达式的值，然后将得到 的值赋给左边的变量。而且右边的表达式能是一个赋值表达式。

在一些朋友的来信中会出现“==”与“=”这两个符号混淆的错误原码，问为何编译报错，一般就是错在 if （a=x）之类的语句中，错将“=”用为“==”。“==”符号是用来进行相 等关系运算。

算术，增减量运算符

对于 a+b，a/b 这样的表达式大家都很熟悉，用在 C 语言中，+，/，就是算术运算符。单片机 c 语言 中的算术运算符有如下几个，其中只有取正值和取负值运算符是单目运算符，其它则都是双 目运算符：

+ 加或取正值运算符

- 减或取负值运算符

* 乘运算符

/ 除运算符

% 取余运算符 算术表达式的形式：

表达式 1 算术运算符 表达式 2 如：a+b*（10-a）， （x+9）/（y-a）

除法运算符和一般的算术运算规则有所不一样，如是两浮点数相除，其结果为浮点数，如

10.0/20.0 所得值为 0.5，而两个整数相除时，所得值就是整数，如 7/3，值为 2。像别的语 言一样 C 的运算符与有优先级和结合性，同样可用用括号“（）”来改变优先级。这些和我们 小时候学的数学几乎是一样的，也不必过多的说明了。

++ 增量运算符

-- 减量运算符

这两个运算符是 C 语言中特有的一种运算符。在 VB，PASCAL 等都是没有的。作用就是 对运算对象作加 1 和减 1 运算。要注意的是运算对象在符号前或后，其含义都是不一样的，虽 然同是加 1 或减 1。如：I++，++I，I--，--I。

I++（或 I--） 是先使用 I 的值，再执行 I+1（或 I-1）

++I（或--I） 是先执行 I+1（或 I-1），再使用 I 的值。增减量运算符只允许用于变量的运算中，不能用于常数或表达式。 先来做一个实验吧。学习运算符和另外一些知识时，我们还是给我们的实验板加个串行

接口吧。借助电脑转件直观的看单片机的输出结果，如果你用的是成品实验板或仿真器，那你就能跳过这一段了。

在制作电路前我们先来看看要用的 MAX232，这里不去具体讨论它，只要知道它是 TTL 和 RS232 电平相互转换的芯片和基本的引脚接线功能就行了。通常我会用两个小功率晶体管加少量的电路去替换 MAX232，能省一点，效 果也不错 （如有兴趣能查看 网站中的相关资料）。下图就是 MAX232 的基本接线图。

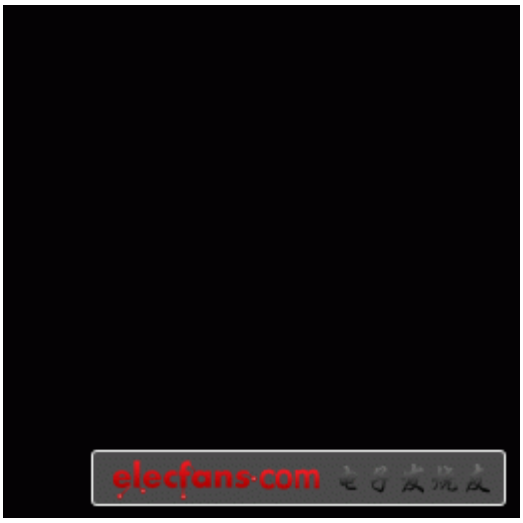


图 6－1 MAX232

在上两课的电路的基础上按图 6－3 加上 MAX232 就能了。串行口座用 DB9 的母头，这样 就能用买来的 PC 串行口延长线进行和电脑相连接，也能直接接到电脑 com 口上。

图 6－2 DB9 接头

图 6－3 加上了 MAX232 的实验电路 做好后，就先用回前面的“Hello World！”程序，用它来和你的电脑说声 Hello！把程序

烧到芯片上，把串行口连接好。嘿嘿，这个时候要打开你的串行口调试软件，没有就赶快到网上 DOWN 一个了。你会用 Windows 的超级终端也行，不过我从不用它。我用 http://emouze.com 的 comdebug，它是个不错的软件，我喜欢它是因为它功能好而且还有“线路状态”功能，这对

我制作小玩意时很有用。串行口号，波特率调好，打开串行口，单片机上电，就能在接收区看 到不断出现的“Hello World！”。一定要先打开软件的串行口，再把单片机上电，不然可能因字符不对齐而看到乱码哦。

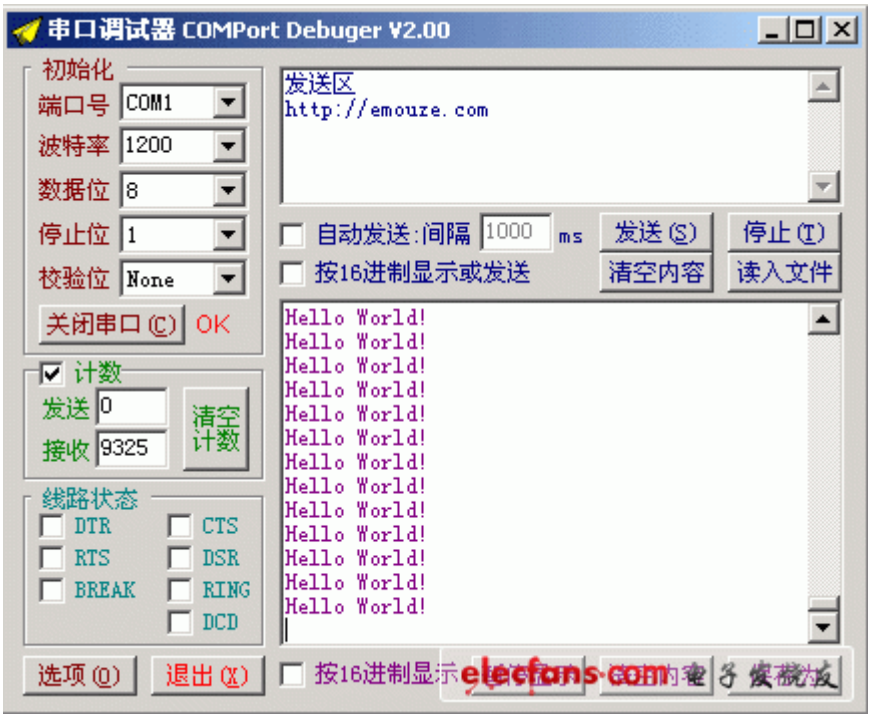


图 6-4 调试结果

第七课、运算符和表达式（关系运算符）

关系运算符，同样我们也并不陌生。单片机 C 语言中有六种关系运算符，这些东西同样是在我们小时候学算术时就已经学习过了的：

> 大于

< 小于

>= 大于等于

<= 小于等于

== 等于

！= 等于

或者你是个非 C 语言 程序员，那么对前四个一定是再熟悉不过的了。而“==”在 VB 或 PASCAL 等中是用“=”，“！=”则是用“not ”。

小学时的数学课就教授过运算符是有优先级别的，计算机的语言也不过是人类语言的一种扩展，这里的运算符同样有着优先级别。前四个具有相同的优先级，后两个也具有相同的优先级，但是前四个的优先级要高于后 2 个的。

当两个表达式用关系运算符连接起来时，这个时候就是关系表达式。关系表达式通常是用来判别某个条件是否满足。要注意的是用关系运算符的运算结果只有 0 和 1 两种，也就是逻辑的真与假，当指定的条件满足时结果为 1，不满足时结果为 0。

表达式 1 关系运算符 表达式 2 如：I<J，I==J，（I=4）》（J=3），J+I》J

借助我们在上一课做好的电路和学习了的相关操作。我们来做一个关系运算符相关的实例程序。为了增加学习的趣味性和生动性，不妨我们来假设在做一个会做算术的机器人，当然真正会思考对话的机器，我想我是做不出来的了，这里的程序只是用来学习关系运算符的基本应用。

```
#include 《AT89X51.H》

#include 《stdio.h》
```

```
void main (void)
{
int x, y;
SCON = 0x50; //串行口方式 1，允许接收 TMOD = 0x20; //定时器 1 定时方式 2
TH1 = 0xE8; //11.0592MHz 1200 波特率 TL1 = 0xE8;
TI = 1;
TR1 = 1; //启动定时器
while (1)
{
printf ("您好！我叫 Robot！我是一个会做算术的机器人！\n"); //显示
printf ("请您输入两个 int, X 和 Y\n"); //显示
scanf ("%d%d", &x, &y); //输入
if (x 《 y)
printf ("X《Y\n"); //当 X 小于 Y 时
else //当 X 不小于 Y 时再作判断
{
```

```
if （x == y）
printf（“X=Y\n”）; //当 X 等于 Y 时
else
printf（“X》Y\n”）; //当 X 大于 Y 时
}
}
}
```

要注意的是，在连接 PC 串行口调试时。发送数字时，发送完一个数字后还要发送一个回车符，以使 scanf 函数确认有数据输入。

逻辑运算符 关系运算符所能反映的是两个表达式之间的大小等于关系，那逻辑运算符则是用于求条件式的逻辑值，用逻辑运算符将关系表达式或逻辑量连接起来就是逻辑表达式 了。也许你会 对为什么“逻辑运算符将关系表达式连接起来就是逻辑表达式了”这一个描述有疑惑的地方。 其实之前说过“要注意的是用关系运算符的运算结果只有 0 和 1 两种，也就是逻辑的真与假”， 换句话说也就是逻辑量，而逻辑运算符就用于对逻辑量运算的表达。逻辑表达式的一般形式 为：

逻辑与：条件式 1 && 条件式 2 逻辑或：条件式 1 || 条件式 2 逻辑非： ！ 条件式 2

图 7-1 演示结果

逻辑与，说白了就是当条件式 1“与”条件式 2 都为真时结果为真（非 0 值），不然为 假（0 值）。也就是说运算会先对条件式 1 进行判断，如果为真（非 0 值），则继续对条件式

2 进行判断，当结果为真时，逻辑运算的结果为真（值为 1），如果结果不为真时，逻辑运算 的结果为假（0 值）。如果在判断条件式 1 时就不为真的话，就不用再判断条件式 2 了，而 直接给出运算结果为假。

逻辑或，是指只要二个运算条件中有一个为真时，运算结果就为真，只有当条件式都不 为真时，逻辑运算结果才为假。

逻辑非则是把逻辑运算结果值取反，也就是说如果两个条件式的运算值为真，进行逻辑 非运算后则结果变为假，条件式运算值为假时最后逻辑结果为真。

同样逻辑运算符也有优先级别，！（逻辑非）→&&（逻辑与）→||（逻辑或），逻辑非的 优先值最高。

!True ||	False	&&	True	
False ||	False	&&	True	//!Ture 先运算得 False
False ||	False			//False && True 运算得 False
False				//最终 False || False 得 False

如有 ！True || False && True

按逻辑运算的优先级别来分析则得到（True 代表真，False 代表假）

下面我们来用程序语言去有表达，如下：

```
#include 《AT89X51.H》

#include 《stdio.h》

void main (void)

{

unsigned char True = 1; //定义
```

```
unsigned char False = 0;

SCON = 0x50; //串行口方式 1，允许接收 TMOD = 0x20; //定时器 1 定时方式 2

TH1 = 0xE8; //11.0592MHz 1200 波特率 TL1 = 0xE8;

TI = 1;

TR1 = 1; //启动定时器

if （! True || False && True）

printf（“True\n”）; //当结果为真时

else

}

printf（“False\n”）; //结果为假时
```

大家能使用以往学习的方法用 keil 或烧到片子上用串行口调试。能更改“！True || False

&& True”这个条件式，以实验不一样算法组合来掌握逻辑运算符的使用方法。

第八课、运算符和表达式（位运算符）

学过汇编的朋友都知道汇编对位的处理能力是很强的，但是单片机 C 语言也能对运算对象进行按位操作，从而使单片机 C 语言也能具有一定的对硬件直接进行操 作的能力。位运算符的作用是按位对变量进行运算，但是并不改变参与运算的变量的值。如果要求按位改变变量的值，则要利用相应的赋值运算。还有就是位运算符 是不能用来对浮点型数据进行操作的。单片机 c 语言中共有 6 种位运算符。位运算一般的表达形式如下：

变量 1 位运算符 变量 2 位运算符也有优先级，从高到低依次是：“~”（按位取反）→“《《”（左移） →“》》”（右

移） →“&”（按位与）→“^”（按位异或）→“|”（按位或）

表 8－1 是位逻辑运算符的真值表，X 表示变量 1，Y 表示变量 2

X	Y	~X	~Y	X&Y	X|Y	X^Y
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	1	0

表 8－1 按位取反，与，或和异或的逻辑真值表

利用以前建立起来的实验板，我们来做个实验验证一下位运算是否真是不改变参与变量 的值，同时学习位运算的表达形式。程序很简单，用 P1 口做运算变量，P1.0-P1.7 对应 P1 变量的最低位到最高位，通过连接在 P1 口上的 LED 我们便能直观看到每个位运算后变量 是否有改变或如何改变。程序如下：

```
#include 《at89x51.h》

void main (void)

{

unsigned int a;

unsigned int b;

unsigned char temp; //临时变量
```

```
P1 = 0xAA; //点亮 D1，D3，D5，D7 P1 口的二进制为 10101010，为 0 时点亮 LED
for （a=0;a《1000;a++）
for （b=0;b《1000;b++）; //延时
temp = P1 & 0x7; //单纯的写 P1|0x7 是没有意义的，因为没有变量被影响，不会被编译
//执行 P1丨0x7 后结果存入 temp，这个时候改变的是 temp，但 P1 不会被影响。
//这个时候 LED 没有变化，仍然是 D1，D3，D5，D7 亮
for （a=0;a《1000;a++）
for （b=0;b《1000;b++）; //延时 P1 = 0xFF; //熄灭 LED
for （a=0;a《1000;a++）
for （b=0;b《1000;b++）; //延时
P1 = 0xAA; //点亮 D1，D3，D5，D7 P1 口的二进制为 10101010，为 0 时点亮 LED
for （a=0;a《1000;a++）
for （b=0;b《1000;b++）; //延时
P1 = P1 & 0x7; //这个时候 LED 会变得只有 D2 灭
//因为之前 P1=0xAA=10101010
//与 0x7 位与 0x7=00000111
```

```
//结果存入 P1 P1=00000010 //位为 0 时点亮 LED，电路看第三课
for （a=0;a《1000;a++）
for （b=0;b《1000;b++）; //延时 P1 = 0xFF; //熄灭 LED
while（1）;
//大家能根据上面的程序去做位或，左移，取反等等。
}
```

复合赋值运算符

复合赋值运算符就是在赋值运算符“=”的前面加上其他运算符。以下是 C 语言中的复 合赋值运算符：

+=	加法赋值	>>=	右移位赋值
-=	减法赋值	&=	逻辑与赋值
*=	乘法赋值	|=	逻辑或赋值
/=	除法赋值		逻辑异或赋值

%= 取模赋值 -= 逻辑非赋值

《《= 左移位赋值 复合运算的一般形式为：

变量 复合赋值运算符 表达式 其含义就是变量与表达式先进行运算符所要求的运算，再把运算结果赋值给参与运算的

变量。其实这是 C 语言中一种简化程序的一种方法，凡是二目运算都能用复合赋值运算符 去简化表达。例如：

a+=56 等价于 a=a+56

y/=x+9 等价于 y=y/（x+9） 很明显采用复合赋值运算符会降低程序的可读性，但这样却能使程序代码简单化，并

能提高编译的效率。对于开始学习 C 语言的朋友在编程时最好还是根据自己的理解力和习惯去使 用程序表达的方式，不要一味追求程序代码的短小。

逗号运算符

如果你有编程的经验，那么对逗号的作用也不会陌生了。如在 VB 中“Dim a，b，c”的逗 号就是把多个变量定义为同一类型的变量，在 C 也一样，如“int a，b，c”，这些例子说明逗 号用于分隔表达式用。但在 C 语言中逗号还是一种特殊的运算符，也就是逗号运算符，能 用它将两个或多个表达式连接起来，形成逗号表达式。逗号表达式的一般形式为：

表达式 1，表达式 2，表达式 3……表达式 n

这样用逗号运算符组成的表达式在程序运行时，是从左到右计算出各个表达式的值，而 整个用逗号运算符组成的表达式的值等于最右边表达式的值，就是“表达式 n”的值。在实 际的应用中，大部分情况下，使用逗号表达式的目的只是为了分别得到名个表达式的值，而 并不一定要得到和使用整个逗号表达式的值。要注意的还有，并不是在程序的任何位置出现 的逗号，都能认为是逗号运算符。如函数中的参数，同类型变量的定义中的逗号只是用来 间隔之用而不是逗号运算符。

条件运算符

上面我们说过单片机 C 语言中有一个三目运算符，它就是“？：”条件运算符，它要求有三个运算对象。它能把三个表达式连接构成一个条件表达式。条件表达式的一般形式如下：

逻辑表达式？ 表达式 1 ： 表达式 2 条件运算符的作用简单来说就是根据逻辑表达式的值选择使用表达式的值。当逻辑表达

式的值为真时（非 0 值）时，整个表达式的值为表达式 1 的值；当逻辑表达式的值为假（值

为 0）时，整个表达式的值为表达式 2 的值。要注意的是条件表达式中逻辑表达式的类型可 以与表达式 1 和表达式 2 的类型不一样。下面是一个逻辑表达式的例子。

如有 a=1，b=2 这个时候我们要求是取 ab 两数中的较小的值放入 min 变量中，也许你会这样 写：

```
if （a《b）

min = a;

else

min = b; //这一段的意思是当 a《b 时 min 的值为 a 的值，不然为 b 的值。
```

用条件运算符去构成条件表达式就变得简单明了了：

min = （a《b）？a ： b 很明显它的结果和含意都和上面的一段程序是一样的，但是代码却比上一段程序少很多，编译的效率也相对要高，但有着和复合赋值表达式一样的缺点就是可读性相 对效差。在实际应 用时根据自己要习惯使用，就我自己来说我喜欢使用较为好读的方式和加上适当的注解，这 样能有助于程序的调试和编写，也便于日后的修改读写。

第九课、C51 运算符和表达式（指针和地址运算符）

在第 3 课我们学习数据类型时，学习过指针类型，知道它是一种存放指向另一个数据的地址的变量类型。指针是单片机 C 语言中一个十分重要的概念，也是学习单片机 C 语 言

中的一个难点。对于指针将会在第九课中做详细的讲解。在这里我们先来了解一下单片机 C 语言中供给的两个专门用于指针和地址的运算符：

* 取内容

& 取地址取内容和地址的一般形式分别为：

变量 = * 指针变量 指针变量 = & 目标变量

取内容运算是将指针变量所指向的目标变量的值赋给左边的变量；取地址运算是将目标变量的地址赋给左边的变量。要注意的是：指针变量中只能存放地址（也就是指针型数据）， 一般情况下不要将非指针类型的数据赋值给一个指针变量。

下面来看一个例子，并用一个图表和实例去简单理解指针的使用方法和含义。

设有两个 unsigned int 变量 ABC 处 CBA 存放在 0x0028，0x002A 中 另有一个指针变量 portA 存放在 0x002C 中 那么我们写这样一段程序去看看*，&的运算结果

```
unsigned int data ABC _at_ 0x0028; unsigned int data CBA _at_ 0x002A; unsigned int data *Port _at_ 0x002C;

#include 《at89x51.h》

#include 《stdio.h》

void main (void)

{

SCON = 0x50; //串行口方式 1，允许接收 TMOD = 0x20; //定时器 1 定时方式 2

TH1 = 0xE8; //11.0592MHz 1200 波特率 TL1 = 0xE8;
```

```
TI = 1;
TR1 = 1; //启动定时器
ABC = 10; //设初值 CBA = 20;
Port = &CBA; //取 CBA 的地址放到指针变量 Port
*Port = 100; //更改指针变量 Port 所指向的地址的内容
printf（“1： CBA=%d\n”，CBA）; //显示此时 CBA 的值
Port = &ABC; //取 ABC 的地址放到指针变量 Port
CBA = *Port; //把当前 Port 所指的地址的内容赋给变量 CBA
printf（“2： CBA=%d\n”，CBA）; //显示此时 CBA 的值
printf（“ ABC=%d\n”，ABC）; //显示 ABC 的值
}
```

程序初始时

值	地址	说明
0x00	0x002DH	
0x00	0x002CH	
0x00	0x002BH	
0x00	0x002AH	
0x0A	0x0029H	
0x00	0x0028H	

执行 ABC = 10;向 ABC 所指的地址 0x28H 写入 10(0xA)，因 ABC 是 int 类型要占用 0x28H 和 0x29H 两个字节的内存空间，低位字节会放入高地址中，所以 0x28H 中放入 0x00，0x29H 中 放入 0xOA

值	地址	说明
0x00	0x002DH	
0x00	0x002CH	
0x00	0x002BH	
0x00	0x002AH	
0x0A	0x0029H	ABC 为 int 类型占用两字节
0x00	0x0028H	

执行 CBA = 20;原理和上一句一样

值	地址	说明
0x00	0x002DH	
0x00	0x002CH	
0x14	0x002BH	CBA 为 int 类型占用两字节
0x00	0x002AH	
0x0A	0x0029H	ABC 为 int 类型占用两字节
0x00	0x0028H	

执行 Port = &CBA; 取 CBA 的首地址放到指针变量 Port

值	地址	说明
0x00	0x002DH	
0x2A	0x002CH	CBA 的首地址存入 Port
0x14	0x002BH	
0x00	0x002AH	
0x0A	0x0029H	
0x00	0x0028H	

*Port = 100; 更改指针变量 Port 所指向的地址的内容

值	地址	说明
0x00	0x002DH	
0x2A	0x002CH	
0x64	0x002BH	Port 指向了 CBA 所在地址 2AH
0x00	0x002AH	并存入 100
0x0A	0x0029H	
0x00	0x0028H	

其它的语句也是一样的道理，大家能用 Keil 的单步执行和打开存储器查看器一看，这样

就更不难理解了。

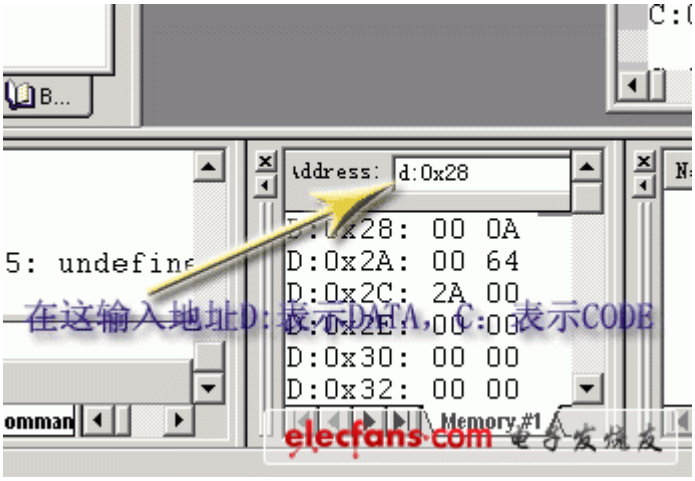


图 9－1 存储器查看窗

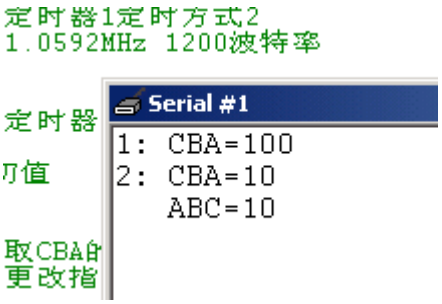


图 9－2 在串行调试窗口的最终结果

sizeof 运算符

看上去这确实是个奇怪的运算符，有点像函数，却又不是。大家看到 size 应该就猜到 是和大小有关的吧？是的，sizeof 是用来求数据类型、变量或是表达式的字节数的一个运 算符，但它并不像“＝”之类运算符那样在程序执行后才能计算出结果，它是直接在编译时 产生结果的。它的语法如下：

sizeof （数据类型）

sizeof （表达式） 下面是两句应用例句，程序大家能试着编写一下。

printf（“char 是多少个字节？ ½ 字节\n”，sizeof（char））；

printf（“long 是多少个字节？ ½ 字节\n”，sizeof（long））；

结果是：

char 是多少个字节？ 1 字节

long 是多少个字节？ 4 字节

强制类型转换运算符 不知你们是否有自己去试着编一些程序，从中是否有遇到一些问题？开始学习时我就遇到过

这样一个问题：两个不一样数据类型的数在相互赋值时会出现不对的值。如下面的一段小程序：

```
void main（void）
{
unsigned char a;
```

```
unsigned int b;

b=100*4;

a=b;

while（1）;

}
```

这段小程序并没有什么实际的应用意义，如果你是细心的朋友定会发现 a 的值是不会等于

100*4 的。是的 a 和 b 一个是 char 类型一个是 int 类型，从以前的学习可知 char 只占一个 字节值最大只能是 255。但编译时为何不出错呢？先来看看这程序的运行情况：

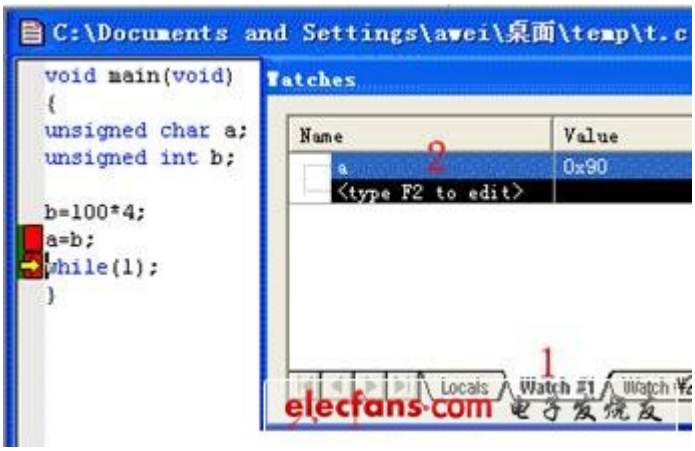


图 9－3 小程序的运行情况

b=100*4 就能得知 b=0x190,这个时候我们能在 Watches 查看 a 的值,对于 watches 窗口我们 在第 5 课时简单学习过，在这个窗口 Locals 页里能查看程序运行中的变量的值，也能

在 watch 页中输入所要查看的变量名对它的值进行查看。做法是按图中 1 的 watch#1（或

watch#2），然后光标移到图中的 2 按 F2 键，这样就能输入变量名了。在这里我们能查看

到 a 的值为 0x90，也就是 b 的低 8 位。这是因为执行了数据类型的隐式转换。隐式转换是 在程序进行编译时由编译器自动去处理完成的。所以有必要了解隐式转换的规则：

1. 变量赋值时发生的隐式转换，“＝”号右边的表达式的数据类型转换成左边变量的数

据类型。就如上面例子中的把 INT 赋值给 CHAR 字符型变量，得到的 CHAR 将会是 INT 的低 8 位。如把浮点数赋值给整形变量，小数部分将丢失。

2. 所有 char 型的操作数转换成 int 型。

3.两个具有不一样数据类型的操作数用运算符连接时,隐式转换会按以下次序进行：如 有一操作数是 float 类型，则另一个操作数也会转换成 float 类型；如果一个操作数为 long 类型，另一个也转换成 long；如果一个操作数是 unsigned 类型，则另一个操作会被转换成 unsigned 类型。

从上面的规则能大概知道有那几种数据类型是能进行隐式转换的。是的，在 单片机 c 语言 中只有 char，int，long 及 float 这几种基本的数据类型能被隐式转换。而其它的数据类型 就只能用到显示转换。要使用强制转换运算符应遵循以下的表达形式：

（类型） 表达式 用显示类型转换来处理不一样类型的数据间运算和赋值是十分方便和方便的，特别对指针

变量赋值是很有用的。看一面一段小程序：

```
#include 《at89x51.h》
#include 《stdio.h》
void main (void)
{
char xdata * XROM;
char a;
int Aa = 0xFB1C;
long Ba = 0x893B7832;
float Ca = 3.4534;
SCON = 0x50; //串行口方式 1，允许接收 TMOD = 0x20; //定时器 1 定时方式 2
TH1 = 0xE8; //11.0592MHz 1200 波特率 TL1 = 0xE8;
TI = 1;
TR1 = 1; //启动定时器
XROM=(char xdata *) 0xB012; //给指针变量赋 XROM 初值
*XROM = 'R'; //给 XROM 指向的绝对地址赋值
```

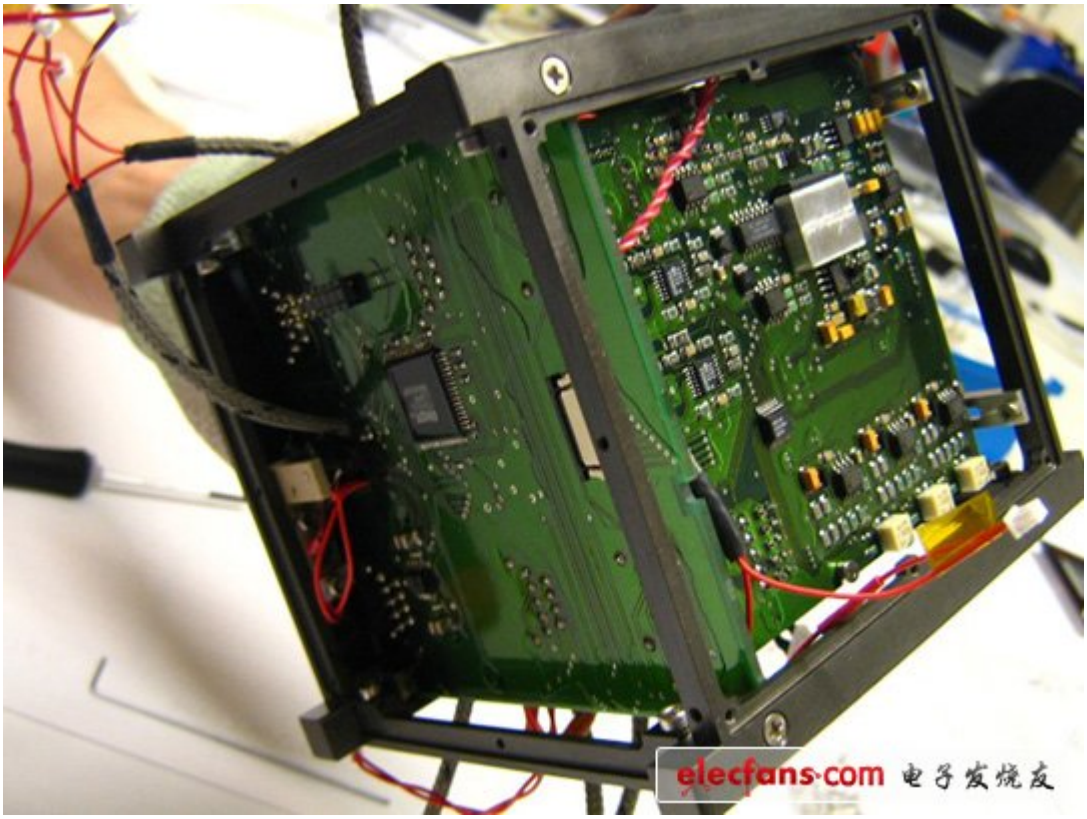
```
    a = *((char xdata *) 0xB012); //等同于 a = *XROM

    printf ("%bx %x %d %c \n", (char) Aa, (int) Ba, (int)Ca, a);//
转换类型并输出

    while(1);

    }
```

程序运行结果：1c 7832 3 R 在上面这段程序中，能很清楚到到各种类型进行强制类型转换的基本使用方法，程序中先

在外部数据存储器 XDATA 中定义了一个字符型指针变量 XROM，当用 XROM=(char xdata *)

0xB012 这一语句时，便把 0xB012 这个地址指针赋于了 XROM，如你用 XROM 则会是非法的，这种方法特别适合于用标识符来存取绝对地址，如在程序前用#define ROM 0xB012 这样的 语句，在程序中就能用上面的方法用 ROM 对绝对地址 0xB012 进行存取操作了。

单片机 C 语言知识点全攻略（三）



电子发烧友网讯：继《单片机学习知识点全攻略》得到广大读者好评，根据有网友提出美中不足的是所用单片机编程语言为汇编，基于此，电子发烧友网再接再厉再次为读者诚挚奉上非常详尽的《单片机 C 语言知识点全攻略》系列单片机 C 语言学习教程，本教程共分为四部分，本文为第三部分，主要知识点如下所示。

参阅相关系列文章，

单片机 C 语言知识点全攻略（一）
单片机 C 语言知识点全攻略（二）

第三部分知识点：

- 第十课 C51 表达式语句及仿真器
- 第十一课 C51 复合语句和条件语句
- 第十二课 C51 开关分支语句
- 第十三课 C51 循环语句
- 第十四课 C51 函数

#p#C51 表达式语句及仿真器#e#

第十课、C51 表达式语句及仿真器

前面学习了大部分的基本语法，以下所要学习的各种基本语句的语法能说是组成程序的灵魂。在前面的课程中的例子里，也简单理解过一些语句的使用方法，能看 出 C 语言是一种结构化的程序设计语言。C 语言供给了相当丰富的程序控制语句。学习掌握这些语句的使用方法也是单片机 C 语言学习中的重点。

表达式语句是最基本的一种语句。不一样的程序设计语言都会有不一样的表达式语句，如 VB 就是在表达式后面加入回车就构成了 VB 的表达式语句，而在 51 单片机的 C 语言中则是加入分号“;”构成表达式语句。举例如下：

```
b = b * 10; Count++;

X = A;Y = B;
```

Page = （a+b）/a-1;

以上的都是合法的表达式语句。在我收到的一些网友的 Email 中，发现很多开始学习的朋友一般在编写调试程序时忽略了分号“;”，造成程序不能被正常的编译。我本人的经验是在遇 到编译错误时先语法是否有误，这在开始学习时一般会因在程序中加入了全角符号、运算符打错 漏掉或没有在后面加“;”。

在 C 语言中有一个特殊的表达式语句，称为空语句，它仅仅是由一个分号“;”组成。有时候为了使语法正确，那么就要求有一个语句，但这个语句又没有实际的运行效果那么这 时就要有一个空语句。说起来就像大家在晚自修的时候用书包占位一样，呵呵。

空语句通常用会以下两种使用方法。

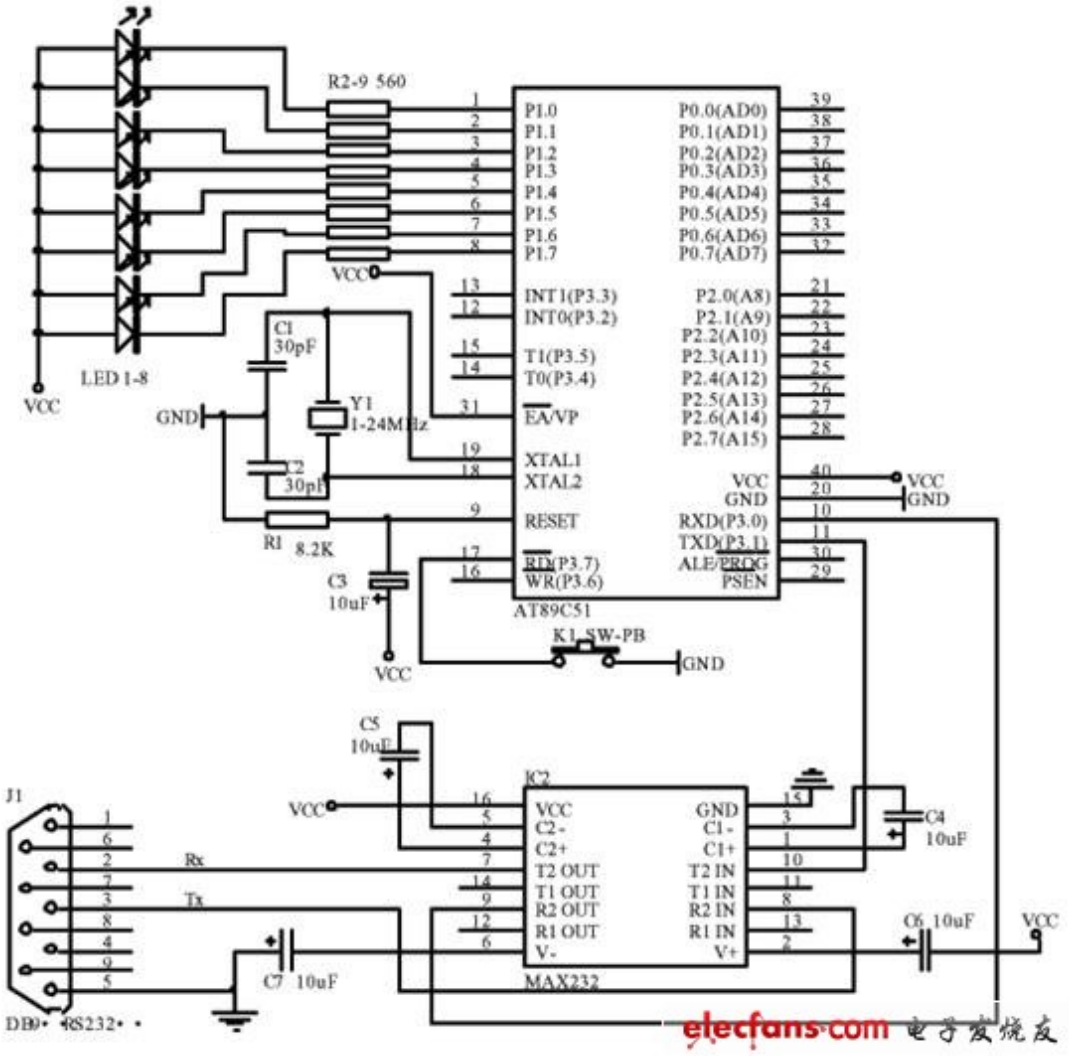
（1）while，for 构成的循环语句后面加一个分号，形成一个不执行其它操作的空循环体。 我会会常常用它来写等待事件发生的程序。大家要注意的是“;”号作为空语句使用时，要 与语句中有效组成部分的分号相区别，如 for （;a《50000;a++）;第一个分号也应该算是 空语句，它会使 a 赋值为 0（但要注意的是如程序前有 a 值，则 a 的初值为 a 的当前值），最后一个分号则使整个语句行成一个空循环。若此时 a=0，那么 for （;a《50000;a++）;就相当

于 for （a=0;a《50000;a++）;我本人习惯是写后面的写法，这样能使人更不难读明白。 （2）在程序中为有关语句供给标号，标记程序执行的位置，使相关语句能跳转到要执行

的位置。这会用在 goto 语句中。

下面的示例程序是简单说明 while 空语句的使用方法。硬件的功能很简单，就是在 P3.7 上 接一个开关，当开关按下时 P1 上的灯会全亮起来。当然实际应用中按钮的功能实现并没有 这么的简单，一般还要进行防抖动处理等。

先在我们的实验板上加一个按钮。电路图如图 10－1。



程序如下：

```
#include 《AT89x51.h》
```

```
void main (void)
{
```

图 10-1 加了按钮的实验电路图

```
unsigned int a;
do
{
P1 = 0xFF; //关闭 P1 上的 LED
while (P3_7); //空语句，等待 P3_7 按下为低电平，低电平时执行下面的语句 P1 = 0; //点亮 LED
for (;a《60000;a++); //这也是空语句的使用方法，注意 a 的初值为当前值
} //这样第一次按下时会有一延时点亮一段时间，以后按多久就亮多久
while (1); //点亮一段时间后关闭再次判断 P3_7，如此循环
}
```

上面的实验电路已加入了 RS232 串行口电路，只要稍微改变一下，就能变为具有仿真功能的 实验电路。这个改变的关键就是把芯片改用 SST89C58，并在芯片中烧入仿真监控程序。SST89C58 同样也是一种 51 架构的单片机，它具有 24K＋8K 的两个程序存储区，能选择其 一做为程序的启动区。只要把一个叫 SOFTICE.HEX 的监控程序用支持 SST89C58 的编程器烧 录到芯片中（使用编程器或用 CA 版的 SST89C58 烧录 SOFTICE 的具体方法和文件能参考 ），就 能把上 面 的电路升级为

MON51 仿真实验器。那么怎么用它和 KEIL 实现联机仿真呢？

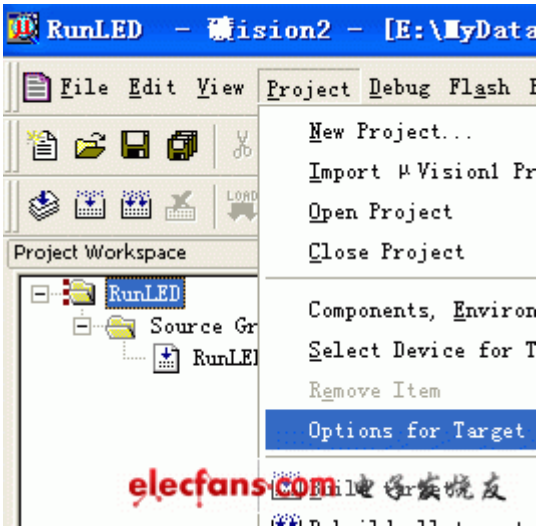


图 10—2 项目设置菜单

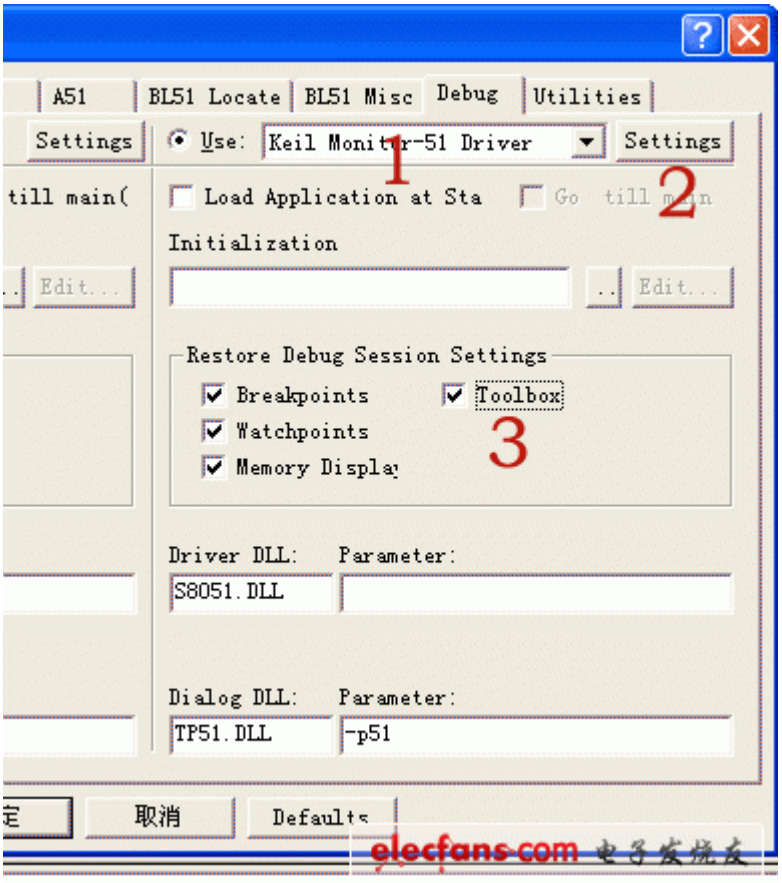


图 10－3 项目设置

首先要在你要仿真的程序项目设置仿真器所使用的驱动，在 Debug 页中选择对应本仿真器的 KeilMon51 驱 动，如图 10 中 1 所示。图 10-3 的 3 是选择在仿真时能使用的工具窗口，如内存显示，断点等等。按 2 进 行图 10-4 中的仿真器设置。设置好串行口号，波特率，晶体震荡器为 11.0592M 时选 38400。Cache Options 为仿真 缓选取后会加快仿真的运行的速度。设好后编译运行程序就能连接仿真器了，连接成功会出现如图 10－

5 的画面。如连接不成功就出现图 10－6 的图，这个时候能先复位电路再按“Try Again”，还不成功连接的话则 应检查软件设置和硬件电路。图 10－5 中 1 是指示仿真器的固件版本为 F-MON51V3.4 版。点击 3 中小红 点位置时为设置和取消断点，点击 2 则运行到下一个断点。图 10－7 则是变量和存储器的查看。仿真器在

软件大概的使用方法和软件仿真相差不多。

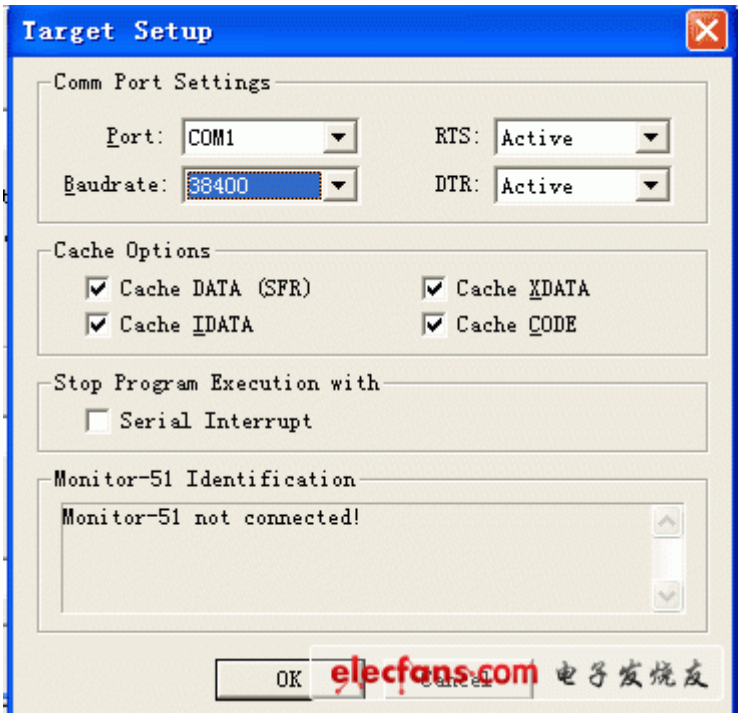


图 10－4 仿真器设置

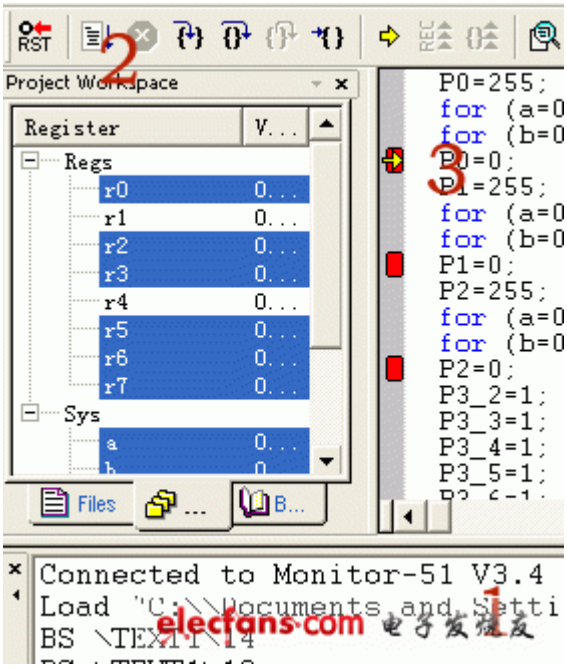


图 10－5 仿真器连接成功

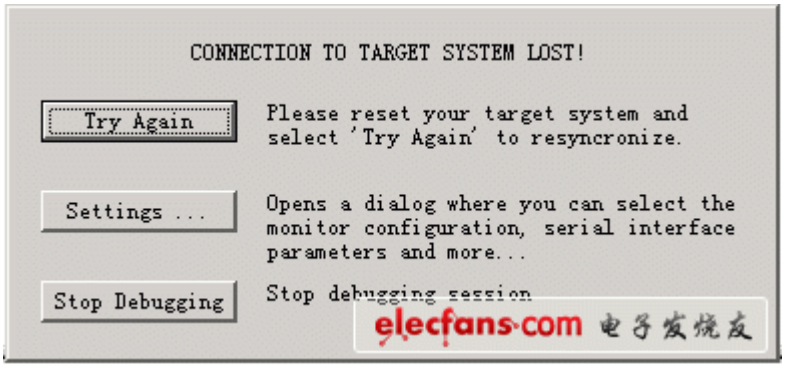


图 10－6 连接不成功提示

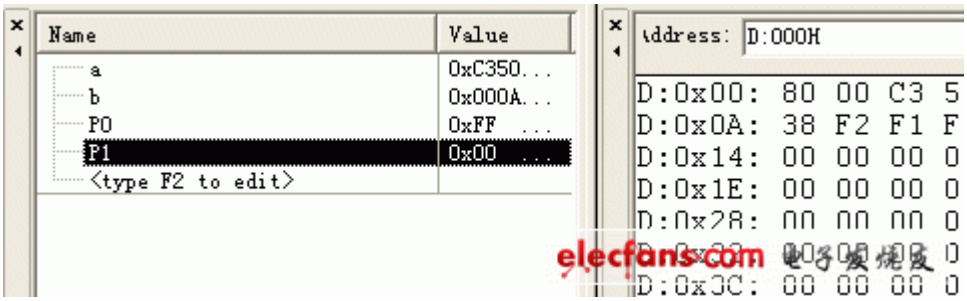


图 10－7 变量及内存查看

#p#C51 复合语句和条件语句#e#

第十一课 C51 复合语句和条件语句

曾经在 BBS 上有朋友问过我{}是什么意思？什么作用？在 C 中是有不少的括号，如{}，［］，（）等，确实会让一些初入门的朋友不解。在 VB 等一些语言中同一个（）号会有不一样的 作用，它能用于组合若干条语句形成功能块，能用做数组的下标等，而在 C 中括号的分 工较为明显，{}号是用于将若干条语句组合在一起形成一种功能块，这种由若干条语句组合 而成的语句就叫复合语句。复合语句之间用{}分隔，而它内部的各条语句还是需要以分号“;” 结束。复合语句是允许嵌套的，也是就是在{}中的{}也是复合语句。复合语句在程序运行时，{}中的各行单语句是依次顺序执行的。单片机 C 语言中能将复合语句视为一条单语句，也就是说 在语法上等同于一条单语句。对于一个函数而言，函数体就是一个复合语句，也许大家会因 此知道复合语句中不单能用可执行语句组成，还能用变量定义语句组成。要注意的是在 复合语句中所定义的变量，称为局部变量，所谓局部变量就是指它的有效范围只在复合语句 中，而函数也算是复合语句，所以函数内定义的变量有效范围也只在函数内部。下面用一段简单的例子简单说明复合语句和局部变量的使用。

```
#include 《at89x51.h》
```

```
#include 《stdio.h》
void main (void)
{
unsigned int a, b, c, d; //这个定义会在整个 main 函数中?
SCON = 0x50; //串行口方式 1，允许接收 TMOD = 0x20; //定时器 1 定时方式 2
TH1 = 0xE8; //11.0592MHz 1200 波特率 TL1 = 0xE8;
TI = 1;
TR1 = 1; //启动定时器
a = 5; b = 6; c = 7;
d = 8; //这会在整个函数有效
printf ("0:  %d, %d, %d, %d\n", a, b, c, d);
{ //复合语句 1
unsigned int a, e; //只在复合语句 1 中有效
a = 10, e = 100;
printf ("1:  %d, %d, %d, %d, %d\n", a, b, c, d, e);
{ //复合语句 2
```

```
unsigned int b, f; //只在复合语句 2 中有效
b = 11, f = 200;
printf (“2: %d, %d, %d, %d, %d, %d\n”, a, b, c, d, e, f) ;
}//复合语句 2 结束
printf (“1: %d, %d, %d, %d, %d\n”, a, b, c, d, e) ;
}//复合语句 1 结束
printf (“0: %d, %d, %d, %d\n”, a, b, c, d) ;
while (1) ;
}
```

运行结果：

0：5，6，7，8

1： 10，6，7，8，100

2： 10，11，7，8，100，200

1： 10，6，7，8，100

0：5，6，7，8 结合以上的说明想想为何结果会是这样。

读完前面的文章大家都会大概对条件语句这个概念有所认识吧？是的，就如学习语文中 的条件语句一样，C 语言也一样是“如果 XX 就 XX”或是“如果 XX 就 XX 不然 XX”。

也就是 当条件符合时就执行语句。条件语句又被称为分支语句，也有人会称为判断语句，其关键字 是由 if 构成，这大众多的高级语言中都是基本相同的。C 语言供给了 3 种形式的条件语句：

1： if （条件表达式） 语句 当条件表达式的结果为真时，就执行语句，不然就跳过。如 if （a==b） a++; 当 a 等于 b 时，a 就加 1

2： if （条件表达式） 语句 1

else 语句 2

当条件表达式成立时，就执行语句 1，不然就执行语句 2 如 if （a==b）

a++;

else

a--;

当 a 等于 b 时，a 加 1，不然 a-1。

3：if （条件表达式 1） 语句 1

else if （条件表达式 2） 语句 2

else if （条件表达式 3） 语句 3

else if （条件表达式 m） 语句 n else 语句 m

这是由 if else 语句组成的嵌套，用来实现多方向条件分支，使用应注意 if 和 else 的配对使用，要是少了一个就会语法出错，记住 else 总是与最临近的 if 相配对。一般条件 语句只会用作单一条件或少数量的分支，如果多数量的分支时则更多的会用到下

一篇中的开 关语句。如果使用条件语句来编写超过 3 个以上的分支程序的话，会使程序变得不是那么清晰易读。

#p#C51 开关分支语句#e#

第十二课 C51 开关分支语句

学习了条件语句，用多个条件语句能实现多方向条件分支，但是能发现使用过多的 条件语句实现多方向分支会使条件语句嵌套过多，程序冗长，这样读起来也很不好读。这个时候 使用开关语句同样能达到处理多分支选择的目的，又能使程序结构清晰。它的语法为下：

```
switch （表达式）
{
case 常量表达式 1： 语句 1; break; case 常量表达式 2： 语句 2; break; case 常量表达式 3： 语句 3; break; case 常量表达式 n： 语句 n; break; default： 语句
}
```

运行中 switch 后面的表达式的值将会做为条件，与 case 后面的各个常量表达式的值相 对比，如果相等时则执行 case 后面的语句，再执行 break（间断语句）语句，跳出 switch 语句。如果 case 后没有和条件相等的值时就执行 default 后的语句。当要求没有符合的条 件时不做任何处理，则能不写 default 语句。

在上面的章节中我们一直在用 printf 这个标准的 C 输出函数做字符的输出，使用它当 然会很方便，但它的功能强大，所占用的存储空间自然也很大，要 1K 左右字节空间，如果 再加上 scanf 输入函数就要达到 2K 左右的字节，这样的话如果要求用 2K 存储空间的芯片时 就无法再使用这两个函数，例如 AT89C2051。在这些小项目中，通常我们只是要求简单的字 符输入输出，这里以笔者发表在本人网站的一个简单的串行口应用实例为例，

一来学习使用开 关语句的使用，二来简单了解 51 芯片串行口基本编程。这个实例是用 PC 串行口通过上位机程序 与由 AT89c51 组成的下位机相通信，实现用 PC 软件控制 AT89c51 芯片的 IO 口，这样也就可 以再通过相关电路实现对设备的控制。为了方便实验，在此所使用的硬件还是用回以上课程 中做好的硬件，以串行口和 PC 连接，用 LED 查看实验的结果。原代码请到在笔者的网站 下载，上面有 单片机 c 语言 下位机源码、PC 上位机源码、电路图等资料。

代码中有多处使 用开关语句的，使用它对不一样的条件做不一样的处理，如在 CSToOut 函数 中根据 CN [1] 来选择输出到那个 IO 口，CN [1] =0 则把 CN [2] 的值送到 P0，CN [1] =1 则送到 P1， 这样的写法比起用 if （CN [1] ==0）这样的判断语句来的清晰明了。当然它们的效果没有太大 的差别（在不考虑编译后的代码执行效率的情况下）。

在这段 代码主要的作用就是通过串行口和上位机软件进行通信，跟据上位机的命令字串， 对指定的 IO 端口进行读写。InitCom 函数，原型为 void InitCom（unsigned char BaudRate）， 其作用为初始化串行口。它的输入参数为一个字节，程序就是用这个参数做为开关语句的选择 参数。如调用 InitCom（6），函数就会把波特率设置为 9600。当然这段代码只使用了一种波特 率，能用更高效率的语句去编写，这里就不多讨论了。

看到这里，你也许会问函数中的 SCON，TCON，TMOD，SCOM 等是代表什么？它们是特殊 功能寄存器。

SBUF 数据缓冲寄存器 这是一个能直接寻址的串行口专用寄存器。有朋友这样问起过“为何在串行口收发中，都只是使用到同一个寄存器 SBUF？而不是收发各用一个寄存器。” 实际上 SBUF 包含了两个独立的寄存器，一个是发送寄存，另一个是接收寄存器，但它们都 共同使用同一个寻址地址－99H。CPU 在读 SBUF 时会指到接收寄存器，在写时会指到发送寄

存器，而且接收寄存器是双缓冲寄存器，这样能避免接收中断没有及时的被响应，数据没

有被取走，下一帧数据已到来，而造成的数据重叠问题。发送器则不需要用到双缓冲，一般 情况下我们在写发送程序时也不必用到发送中断去外理发送数据。操作 SBUF 寄存器的方法 则很简单，只要把这个 99H 地址用关键字 sfr 定义为一个变量就能对其进行读写操作了，

如 sfr SBUF = 0x99;当然你也能用其它的名称。通常在标准的 reg51.h 或 at89x51.h 等 头文件中已对其做了定义，只要用#include 引用就能了。

SCON 串行口控制寄存器 通常在芯片或设备中为了监视或控制接口状态，都会引用到接口控制寄存器。SCON 就是 51 芯片的串行口控制寄存器。它的寻址地址是 98H，是一个 能位寻址的寄存器，作用就是监视和控制 51 芯片串行口的工作状态。51 芯片的串行口能 工作在几个不一样的工作模式下，其工作模式的设置就是使用 SCON 寄存器。它的各个位的具 体定义如下：

（MSB） （LSB） SM0 SM1 SM2 REN TB8 RB8 TI RI

表 8－1 串行口控制寄存器 SCON

SM0、SM1 为串行口工作模式设置位，这样两位能对应进行四种模式的设置。看表 8

－2 串行口工作模式设置。

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
1	0	2	9 位 UART	fosc/32 或 fosc/64
1	1	3	9 位 UART	可变

表 8－2 串行口工作模式设置

在这里只说明最常用的模式 1，其它的模式也就一一略过，有兴趣的朋友能找相关的硬件资料查看。表中的 fosc 代表振荡器的频率，也就是晶体震荡器的频率。UART 为（Universal Asynchronous Receiver）的英文缩写。

SM2 在模式 2、模式 3 中为多处理机通信使能位。在模式 0 中要求该位为 0。

REM 为允许接收位，REM 置 1 时串行口允许接收，置 0 时禁止接收。REM 是由软件置位或 清零。如果在一个电路中接收和发送引脚 P3.0，P3.1 都和上位机相连，在软件上有串行口中断 处理程序，当要求在处理某个子程序时不允许串行口被上位机来的控制字符产生中断，那么可 以在这个子程序的开始处加入 REM=0 来禁止接收，在子程序结束处加入 REM=1 再次打开串行口 接收。大家也能用上面的实际源码加入 REM=0 来进行实验。

TB8 发送数据位 8，在模式 2 和 3 是要发送的第 9 位。该位能用软件根据需要置位或 清除，通常这位在通信协议中做奇偶位，在多处理机通信中这一位则用于表示是地址帧还是 数据帧。

RB8 接收数据位 8，在模式 2 和 3 是已接收数据的第 9 位。该位可能是奇偶位，地址/ 数据标识位。在模式 0 中，RB8 为保留位没有被使用。在模式 1 中，当 SM2=0，RB8 是已接 收数据的停止位。

TI 发送中断标识位。在模式 0，发送完第 8 位数据时，由硬件置位。其它模式中则是在 发送停止位之初，由硬件置位。TI 置位后，申请中断，CPU 响应中断后，发送下一帧数据。 在任何模式下，TI 都必须由软件来清除，也就是说在数据写入到 SBUF 后，硬件发送数据，

中断响应（如中断打开），这个时候 TI=1，表明发送已完成，TI 不会由硬件清除，所以这个时候必须

用软件对其清零。

RI 接收中断标识位。在模式 0，接收第 8 位结束时，由硬件置位。其它模式中则是在接 收停止位的半中间，由硬件置位。RI=1，申请中断，要求 CPU 取走数据。但在模式 1

中，SM2=1 时，当未收到有效的停止位，则不会对 RI 置位。同样 RI 也必须要靠软件清除。

常用的串行口模式 1 是传输 10 个位的，1 位起始位为 0，8 位数据位，低位在先，1 位停止 位为 1。它的波特率是可变的，其速率是取决于定时器 1 或定时器 2 的定时值（溢出速率）。 AT89c51 和 AT89C2051 等 51 系列芯片只有两个定时器，定时器 0 和定时器 1，而定时器 2

是 89C52 系列芯片才有的。

波特率 在使用串行口做通信时，一个很重要的参数就是波特率，只有上下位机的波特率 一样时才能进行正常通信。波特率是指串行端口每秒内能传输的波特位数。有一些开始学习 的朋友认为波特率是指每秒传输的字节数，如标准 9600 会被误认为每秒种能传送 9600 个字节，而实际上它是指每秒能传送 9600 个二进位，而一个字节要 8 个二进位，如用串 口模式 1 来传输那么加上起始位和停止位，每个数据字节就要占用 10 个二进位，9600 波特 率用模式 1 传输时，每秒传输的字节数是 9600÷10＝960 字节。51 芯片的串行口工作模式 0 的波特率是固定的，为 fosc/12，以一个 12M 的晶体震荡器来计算，那么它的波特率能达到 1M。 模式 2 的波特率是固定在 fosc/64 或 fosc/32，具体用那一种就取决于 PCON 寄存器中的 SMOD 位，如 SMOD 为 0，波特率为 focs/64，SMOD 为 1，波特率为 focs/32。模式 1 和模式 3 的波 特率是可变的，取决于定时器 1 或 2（52 芯片）的溢出速率。那么我们怎么去计算这两个模 式的波特率设置时相关的寄存器的值呢？能用以下的公式去计算。

波特率＝（2SMOD÷32）×定时器 1 溢出速率

上式中如设置了 PCON 寄存器中的 SMOD 位为 1 时就能把波特率提升 2 倍。通常会使用 定时器 1 工作在定时器工作模式 2 下，这个时候定时值中的 TL1 做为计数，TH1 做为自动重装值 ， 这个定时模式下，定时器溢出后，TH1 的值会自动装载到 TL1，再次开始计数，这样能不 用软件去干预，使得定时更准确。在这个定时模式 2 下定时器 1 溢出速率的计算公式如下：

溢出速率＝（计数速率）/（256－TH1） 上式中的“计数速率”与所使用的晶体振荡器频率有关，在 51 芯片中定时器启动后会

在每一个机器周期使定时寄存器 TH 的值增加一，一个机器周期等于十二个振荡周期，所以

能得知 51 芯片的计数速率为晶体振荡器频率的 1/12，一个 12M 的晶体震荡器用在 51 芯片上， 那么 51 的计数速率就为 1M。通常用 11.0592M 晶体是为了得到标准的无误差的波特率，那 么为何呢？计算一下就知道了。如我们要得到 9600 的波特率，晶体震荡器为 11.0592M 和 12M，定 时器 1 为模式 2，SMOD 设为 1，分别看看那所要求的 TH1 为何值。代入公式：

11.0592M

9600＝（2÷32）×（（11.0592M/12）/（256-TH1））

TH1＝250 //看看是不是和上面实例中的使用的数值一样？

12M

9600＝（2÷32）×（（12M/12）/（256-TH1）） TH1≈249.49

上面的计算能看出使用 12M 晶体的时候计算出来的 TH1 不为整数，而 TH1 的值只能取

整数，这样它就会有一定的误差存在不能产生精确的 9600 波特率。当然一定的误差是能 在使用中被接受的，就算使用 11.0592M 的晶体振荡器也会因晶体本身所存在的误差使波特

率产生误差，但晶体本身的误差对波特率的影响是十分之小的，能忽略不计。

#p#C51 循环语句#e#

第十三课 C51 循环语句

循环语句是几乎每个程序都会用到的，它的作用就是用来实现需要反复进行多次的操作。如一个 12M 的 51 芯片应用电路中要求实现 1 毫秒的延时，那么就要执行 1000 次空语句 才能达到延时的目的(当然能使用定时器来做，这里就不讨论)，如果是写 1000 条空语 句那是多么麻烦的事情，再者就是要占用很多的存储空间。我们能知道这 1000 条空语句， 无非就是一条空语句重复执行 1000 次，因此我们就能用循环语句去写，这样不但使程序

结构清晰明了，而且使其编译的效率大大的提高。在 C 语言中构成循环控制的语句有 while，do-while，for 和 goto 语句。同样都是起到循环作用，但具体的作用和使用方法又大不一 样。我们具体来看看。

goto 语句

这个语句在很多高级语言中都会有，记得小时候用 BASIC 时就很喜欢用这个语句。它是 一个无条件的转向语句，只要执行到这个语句，程序指针就会跳转到 goto 后的标号所在的 程序段。它的语法如下：

goto 语句标号; 其中的语句标号为一个带冒号的标识符。示例如下

```
void main (void)
{
unsigned char a;
start:  a++;
```

```
if （a==10） goto end;

goto start;

end：;

}
```

上面一段程序只是说明一下 goto 的使用方法，实际编写很少使用这样的手法。这段程序的意思

是在程序开始处用标识符“start：”标识，表示程序这是程序的开始，“end：”标识程序的 结束，标识符的定义应遵循前面所讲的标识符定义原则，不能用 C 的关键字也不能和其它变 量和函数名相同，不然就会出错了。程序执行 a++，a 的值加 1，当 a 等于 10 时程序会跳到 end 标识处结束程序，不然跳回到 start 标识处继续 a++，直到 a 等于 10。上面的示例说明 goto 不但能无条件的转向，而且能和 if 语句构成一个循环结构，这些在 C 程序员的程 序中都不太常见，常见的 goto 语句使用方法是用它来跳出多重循环，不过它只能从内层循环 跳到外层循环，不能从外层循环跳到内层循环。在下面说到 for 循环语句时再略为提一提。为何大多数 C 程序员都不喜欢用 goto 语句？那是因为过多的使用它时会程序结构不清晰，

过多的跳转就使程序又回到了汇编的编程风格，使程序失去了 C 的模块化的优点。

while 语句

while 语句的意思很不难理解，在英语中它的意思是“当…的时候…”，在这里我们可 以理解为“当条件为真的时候就执行后面的语句”，它的语法如下：

```
while （条件表达式） 语句;
```

使用 while 语句时要注意当条件表达式为真时，它才执行后面的语句，执行完后再次回

到 while 执行条件判断，为真时重复执行语句，为假时退出循环体。当条件一开始就为假时， 那么 while 后面的循环体（语句或复合语句）将一次都不执行就退出循环。在调试程序时要

注意 while 的判断条件不能为假而造成的死循环，调试时适当的在 while 处加入断点，也许 会使你的调试工作更加顺利。当然有时会使用到死循环来等待中断或 IO 信号等，如在第一 篇时我们就用了 while（1）来不停的输出“Hello World！”。下面的例子是显示从 1 到 10 的累 加和，读者能修改一下 while 中的条件看看结果会如果，从而体会一下 while 的使用方法。

```
#include 《AT89X51.H》

#include 《stdio.h》

void main (void)

{

unsigned int I = 1;

unsigned int SUM = 0; //设初值

SCON = 0x50; //串行口方式 1，允许接收

TMOD = 0x20; //定时器 1 定时方式 2

TCON = 0x40; //设定时器 1 开始计数

TH1 = 0xE8; //11.0592MHz 1200 波特率

TL1 = 0xE8; TI = 1;
```

```
TR1 = 1; //启动定时器

while（I《=10）

{

SUM = I + SUM; //累加

printf （“%d SUM=%d\n”，I，SUM）；//显示

I++;

}

while（1）；//这句是为了不让程序完后，程序指针继续向下造成程序“跑飞”

}

//最后运行结果是 SUM=55;
```

do while 语句

do while 语句能说是 while 语句的补充，while 是先判断条件是否成立再执行循环体，

而 do while 则是先执行循环体，再根据条件判断是否要退出循环。这样就决定了循环体无 论在任何条件下都会至少被执行一次。它的语法如下：

do 语句 while （条件表达式）

用 do while 怎么写上面那个例程呢？先想一想，再参考下面的程序。

```
#include 《AT89X51.H》
#include 《stdio.h》
void main (void)
{
unsigned int I = 1;
unsigned int SUM = 0; //设初值
SCON = 0x50; //串行口方式 1，允许接收 TMOD = 0x20; //定时器 1 定时方式 2
TCON = 0x40; //设定时器 1 开始计数
TH1 = 0xE8; //11.0592MHz 1200 波特率 TL1 = 0xE8;
TI = 1;
TR1 = 1; //启动定时器
do
{
SUM = I + SUM; //累加
printf （“%d SUM=%d\n”，I，SUM）; //显示 I++;
}
```

```
while（I《=10）；

while（1）；

}
```

在上面的程序看来 do while 语句和 while 语句似乎没有什么两样，但在实际的应用中要注

意任何 do while 的循环体一定会被执行一次。如把上面两个程序中 I 的初值设为 11，那么 前一个程序不会得到显示结果，而后一个程序则会得到 SUM=11。

#p#单片机 C 语言之 for 语句#e#

for 语句

在明确循环次数的情况下，for 语句比以上说的循环语句都要方便简单。它的语法如下： for （[初值设定表达式]；[循环条件表达式]；[条件更新表达式]） 语句 中括号中的表达式是可选的，这样 for 语句的变化就会很多样了。for 语句的执行：先

代入初值，再判断条件是否为真，条件满足时执行循环体并更新条件，再判断条件是否为 真……直到条件为假时，退出循环。下面的例子所要实现的是和上二个例子一样的，对照着 看不难理解几个循环语句的差异。

```
#include 《AT89X51.H》

#include 《stdio.h》

void main（void）

{
```

```
unsigned int I;

unsigned int SUM = 0; //设初值

SCON = 0x50; //串行口方式 1，允许接收 TMOD = 0x20; //定时器 1 定时方式 2

TCON = 0x40; //设定时器 1 开始计数

TH1 = 0xE8; //11.0592MHz 1200 波特率 TL1 = 0xE8;

TI = 1;

TR1 = 1; //启动定时器

for （I=1; I《=10; I++） //这里能设初始值，所以变量定义时能不设

{

SUM = I + SUM; //累加

printf （“%d SUM=%d\n”，I，SUM）; //显示

}

while（1）;

}
```

如果我们把程序中的 for 改成 for（;I《=10;I++）这样条件的初值会变成当前 I 变量的

值。如果改成 for（;;）会怎么样呢？试试看。

continue 语句

continue 语句是用于中断的语句，通常使用在循环中，它的作用是结束本次循环，跳过循环体中没有执行的语句，跳转到下一次循环周期。语法为：

continue;

continue 同时也是一个无条件跳转语句，但功能和前面说到的 break 语句有所不一样， continue 执行后不是跳出循环，而是跳到循环的开始并执行下一次的循环。在上面的例子 中的循环体加入 if （I==5） continue;看看什么结果？

return 语句

return 语句是返回语句，不属于循环语句，是要学习的最后一个语句所以一并写下了。返回语句是用于结束函数的执行，返回到调用函数时的位置。语法有二种：

return （表达式）；

return; 语法中因带有表达式，返回时先计算表达式，再返回表达式的值。不带表达式则返回的

值不确定。

下面是一个同样是计算 1－10 的累加，所不一样是的用了函数的方式。

```
#include 《AT89X51.H》

#include 《stdio.h》

int Count（void）; //声明函数

void main（void）
```

```
{
unsigned int temp;
SCON = 0x50; //串行口方式 1，允许接收 TMOD = 0x20; //定时器 1 定时方式 2
TCON = 0x40; //设定时器 1 开始计数
TH1 = 0xE8; //11.0592MHz 1200 波特率 TL1 = 0xE8;
TI = 1;
TR1 = 1; //启动定时器
temp = Count（）；
printf （“1-10 SUM=%d\n”，temp）; //显示
while（1）；
}
int Count（void）
{
unsigned int I， SUM;
for （I=1; I《=10; I++）
{
```

```
    SUM = I + SUM; //累加
    }
    return （SUM）;
    }
```

#p#C51 函数#e#

第十四课 C51 函数

上一篇的最后一个例子中有用到函数，其实一直出现在例子中的 main（）也算是一个函数，只不过它比较特殊，编译时以它做为程序的开始段。有了函数 C 语言就有了模块化的优 点，一般功能较多的程序，会在编写程序时把每项单独的功能分成数个子程序模块，每个子 程序就能用函数来实现。函数还能被反复的调用，因此一些常用的函数能做成函数库 以供在编写程序时直接调用，从而更好的实现模块化的设计，大大提高编程工作的效率。
一. 函数定义

通常 C 语言的编译器会自带标准的函数库，这些都是一些常用的函数，Keil uv 中也不 例外。标准函数已由编译器软件商编写定义，使用者直接调用就能了，而无需定义。但是 标准的函数不足以满足使用者的特殊要求，因此 C 语言允许使用者根据需要编写特定功能的 函数，要调用它必须要先对其进行定义。定义的模式如下：

```
函数类型 函数名称（形式参数表）
 {
函数体
}
```

函数类型是说明所定义函数返回值的类型。返回值其实就是一个变量，只要按变量

类型来定义函数类型就行了。如函数不需要返回值函数类型能写作“void”表示该函数没 有返回值。注意的是函数体返回值的类型一定要和函数类型一致，不然会造成错误。函数名 称的定义在遵循 C 语言变量命名规则的同时，不能在同一程序中定义同名的函数这将会造成 编译错误(同一程序中是允许有同名变量的,因为变量有全局和局部变量之分)。形式参数 是指调用函数时要传入到函数体内参与运算的变量，它能有一个、几个或没有，当不需要 形式参数也就是无参函数，括号内能为空或写入“void”表示，但括号不能少。函数体中 能包含有局部变量的定义和程序语句,如函数要返回运算值则要使用 return 语句进行返 回。在函数的｛｝号中也能什么也不写，这就成了空函数，在一个程序项目中能写一些 空函数，在以后的修改和升级中能方便的在这些空函数中进行功能扩充。

二．函数的调用

函数定义好以后，要被其它函数调 用了才能被执行。C 语言的函数是能相互调用的，但在调用函数前，必须对函数的类型进行说明，就算是标准库函数也不例外。标准库函数的 说明会被按功能分别写在不一样的头文件中，使用时只要在文件最前面用#include 预处理语 句引入相应的头文件。如前面一直有使用的 printf 函数说明就是放在文件名为 stdio.h 的 头文件中。调用就是指一个函数体中引用另一个已定义的函数来实现所需要的功能，这个时候函 数体称为主调用函数，函数体中所引用的函数称为被调用函数。一个函数体中能调用数个 其它的函数,这些被调用的函数同样也能调用其它函数,也能嵌套调用。笔者本人认为 主函数只是相对于被调用函数而言。在 c51 语言中有一个函数是不能被其它函数所调用的， 它就是 main 主函数。调用函数的一般形式如下：

函数名 （实际参数表） “函数名”就是指被调用的函数。实际参数表能为零或多个参数，多个参数时要用逗

号隔开，每个参数的类型、位置应与函数定义时所的形式参数一一对应，它的作用就是把参 数传到被调用函数中的形式参数，如果类型不对应就会产生一些错误。调用的函数是无参函 数时不写参数，但不能省后面的括号。

在以前的一些例子我们也能看不一样的调用方式：

1．函数语句

如 printf （“Hello World! \n”）；这是在 我们的第一个程序中出现的，它以“Hello

World! \n”为参数调用 printf 这个库函数。在这里函数调用被看作了一条语句。

2．函数参数 “函数参数”这种方式是指被调用函数的返回值当作另一个被调用函数的实际参

数，如 temp=StrToInt（CharB（16））;CharB 的返回值作为 StrToInt 函数的实际参数传递。

3．函数表达式

而在上一篇的例子中有 temp = Count（）;这样一句，这个时候函数的调用作为一个运算 对象出现在表达式中，能称为函数表达式。例子中 Count（）返回一个 int 类型的返回 值直接赋值给 temp。注意的是这种调用方式要求被调用的函数能返回一个同类型的值， 不然会出现不可预料的错误。

前面说到调用函数前要对被调用的函数进行说明。标准库函数只要用#include 引入已写好说明的头文件，在程序就能直接调用函数了。如调用的是自定义的函数则要用如下形式编写函数类型说明

类型标识符 函数的名称（形式参数表）；这样的说明方式是用在被调函数定义和主调函数是在同一文件中。你也能把这些写到

文件名.h 的文件中用#include “文件名.h”引入。如果被调函数的定义和主调函数不是在同 一文件中的，则要用如下的方式进行说明，说明被调函数的定义在同一项目的不

一样文件之上， 其实库函数的头文件也是如此说明库函数的，如果说明的函数也能称为外部函数。

extern 类型标识符 函数的名称(形式参数表)；函数的定义和说明是完全不一样的，在编译的角度上看函数的定义是把函数编译存放在

ROM 的某一段地址上，而函数说明是告诉编译器要在程序中使用那些函数并确定函数的地 址。如果在同一文件中被调函数的定义在主调函数之前，这个时候能不用说明函数类型。也就 是说在 main 函数之前定义的函数，在程序中就能不用写函数类型说明了。能在一个函 数体调用另一个函数（嵌套调用），但不允许在一个函数定义中定义另一个函数。还要注意 的是函数定义和说明中的“类型、形参表、名称”等都要相一致。

三．中断函数 中断服务函数是编写单片机应用程序不可缺少的。中断服务函数只有在中断源请求响应

中断时才会被执行，这在处理突发事件和实时控制是十分有效的。例如：电路中一个按钮， 要求按钮后 LED 点亮，这个按钮何时会被按下是不可预知的，为了要捕获这个按钮的事件， 通常会有三种方法，一是用循环语句不断的对按钮进行查询，二是用定时中断在间隔时间内 扫描按钮，三是用外部中断服务函数对按钮进行捕获。在这个应用中只有单一的按钮功能， 那么第一种方式就能胜任了，程序也很简单，但是它会不停的在对按钮进行查询浪费了

CPU 的时间。实际应用中一般都会还有其它的功能要求同时实现，这个时候能根据需要选用第 二或第三种方式，第三种方式占用的 CPU 时间最少，只有在有按钮事件发生时，中断服务函 数才会被执行，其余的时间则是执行其它的任务。

如果你学习过汇编语言的话，刚开始写汇编的中断应用程序时，你一定会为出入堆栈的 问题而困扰过。单片机 c 语言 语言扩展了函数的定义使它能直接编写中断服务函数，你能不必考 虑出入堆栈的问题，从而提高了工作的效率。扩展的关键字是 interrupt，它是函数定义时 的一个选项，只要在一个函数定义后面加上这个选项，那么这个函数就变成了中断服务函数。

在后面还能加上一个选项 using，这个选项是指定选用 51 芯片内部 4 组工作寄存器中的

那个组。开始学习者能不必去做工作寄存器设定，而由编译器自动选择，避免产生不必要的错 误。定义中断服务函数时能用如下的形式。

函数类型 函数名 （形式参数） interrupt n [using n]

interrupt 关键字是不可缺少的，由它告诉编译器该函数是中断服务函数，并由后面的

n 指明所使用的中断号。n 的取值范围为 0－31，但具体的中断号要取决于芯片的型号，像 AT89c51 实际上就使用 0－4 号中断。每个中断号都对应一个中断向量，具体地址为 8n+3， 中断源响应后处理器会跳转到中断向量所处的地址执行程序，编译器会在这地址上产生一个 无条件跳转语句，转到中断服务函数所在的地址执行程序。下表是 51 芯片的中断向量和中 断号。

中断号	中断源	中断向量
0	外部中断 0	0003H
1	定时器/计数器 0	000BH
2	外部中断 1	0013H
3	定时器/计数器 1	001BH
4	串行口	0023H

表 9－1 AT89c51 芯片中断号和中断向量

使用中断服务函数时应注意：中断函数不能直接调用中断函数;不能通过形参传速参数; 在中断函数中调用其它函数，两者所使用的寄存器组应相同。限于篇幅其它与函数相关的知 识这里不能一一加以说明，如变量的传递、存储，局部变量、全部变量等，有兴趣的朋友可 以访问笔者的网站 阅读更多相关文章。

下面是简 单的例子。首先要在前面做好的实验电路中加多一个按钮，接在 P3.2(12 引脚外 部中断 INT0）和地线之间。把编译好后的程序烧录到芯片后，当接在 P3.2 引脚的按钮接下 时，中断服务函数 Int0Demo 就会被执行，把 P3 当前的状态反映到 P1，如按钮接下后 P3.7

（之前有在这脚装过一按钮）为低，这个时候 P1.7 上的 LED 就会熄灭。放开 P3.2 上的按钮后，

P1LED 状态保持先前按下 P3.2 时 P3 的状态。

```
#include 《at89x51.h》

unsigned char P3State（void）; //函数的说明，中断函数不用说明

void main（void）

{

IT0 = 0; //设外部中断 0 为低电平触发

EX0 = 1; //允许响应外部中断 0

EA = 1; //总中断开关

while（1）;

}

//外部中断 0 演示，使用 2 号寄存器组

void Int0Demo（void） interrupt 0 using 2
```

```
{
unsigned int Temp; //定义局部变量
P1 = ~P3State（）; //调用函数取得 p2 的状态反相后并赋给 P1
for （Temp=0; Temp《50; Temp++）; //延时 这里只是演示局部变量的使用
}
//用于返回 P3 的状态，演示函数的使用
unsigned char P3State（void）
{
unsigned char Temp;
Temp = P3; //读取 P3 的引脚状态并保存在变量 Temp 中
//这样只有一句语句实在没必要做成函数，这里只是学习函数的基本使用方法
return Temp;
}
```

单片机 C 语言知识点全攻略（完结篇）

电子发烧友网讯：继《单片机学习知识点全攻略》得到广大读者好评，根据有网友提出美中不足的是所用单片机编程语言为汇编，基于此，电子发烧友网再接再厉再次为读者诚挚奉上非常详尽的《单片机 C 语言知识点全攻略》系列单片机 C 语言学习教程，本教程共分为四部分，本文为第四部分，也是完结篇，主要知识点如下所示。

参阅相关系列文章，

单片机 C 语言知识点全攻略（一）
单片机 C 语言知识点全攻略（二）
单片机 C 语言知识点全攻略（三）

第四部分（完结篇）知识点：

- 第十五课 C51 数组的使用
- 第十六课 C51 指针的使用
- 第十七课 C51 结构、联合和枚举的使用

- 附录（运算符优先级和结合性等）

#p#C51 数组的使用#e#

第十五课、C51 数组的使用

前面的文章中，都是介绍单个数据变量的使用，在“走马灯”等的例子中略有使用到数组，不难看出，数组不过就是同一类型变量的有序集合。形象的能这样去理 解，就像一个 学校在操场上排队，每一个级代表一个数据类型，每一个班级为一个数组，每一个学生就是 数组中的一个数据。数据中的每个数据都能用唯一的下标来确定其位置，下标能是一维 或多维的。就如在学校的方队中要找一个学生，这个学生在 I 年级 H 班 X 组 Y 号的，那么 能把这个学生看做在 I 类型的 H 数组中（X，Y）下标位置中。数组和普通变量一样，要求先定义了才能使用，下面是定义一维或多维数组的方式：

数据类型	数组名	[常量表达式]；
数据类型	数组名	[常量表达式 1]..... [常量表达式 N]；

“数据类型”是指数组中的各数据单元的类型，每个数组中的数据单元只能是同一数据

类型。“数组名”是整个数组的标识，命名方法和变量命名方法是一样的。在编译时系统会 根据数组大小和类型为变量分配空间，数组名能说就是所分配空间的首地址的标识。“常 量表达式”是表示数组的长度和维数，它必须用“［］”括起，括号里的数不能是变量只能是 常量。

unsigned int xcount ［10］；//定义无符号整形数组，有 10 个数据单元

char inputstring [5]; //定义字符形数组，有 5 个数据单元

float outnum [10], [10];//定义浮点型数组，有 100 个数据单元

在 C 语言中数组的下标是从 0 开始的而不是从 1 开始，如一个具有 10 个数据单元的数

组 count，它的下标就是从 count [0] 到 count [9]，引用单个元素就是数组名加下标，如 count [1] 就是引用 count 数组中的第 2 个元素，如果错用了 count [10] 就会有错误出现了。还有一点要 注意的就是在程序中只能逐个引用数组中的元素，不能一次引用整个数组，但是字符型的数 组就能一次引用整个数组。

数组也是能赋初值的。 在上面介绍的定义方式只适用于定义在内存 DATA 存储器使用的内存，有的时候我们需要把一些数据表存放在数组中，通常这些数据是不用在程序中改 变数值的，这个时候就要把这些数据在程序编写时就赋给数组变量。因为 51 芯片的片内 RAM 很有限，通常会把 RAM 分给参与运算的变量或数组，而那些程序中不变数据则应存放在片 内的 CODE 存储区，以节省宝贵的 RAM。赋初值的方式如下：

数据类型 [存储器类型] 数组名 [常量表达式] = {常量表达式};

数据类型 [存储器类型] 数组名 [常量表达式 1]。..。.. [常量表达式 N] ={{ 常量表达 式}。..{常量表达式 N}};

在定义并为数组赋初值时，开始学习的朋友一般会搞错初值个数和数组长度的关系，而致使 编译出错。初值个数必须小于或等于数组长度，不指定数组长度则会在编译时由实际的初值 个数自动设置。

unsigned char LEDNUM [2] ={12, 35}; //一维数组赋初值

int Key [2] [3] ={{1, 2, 4}, {2, 2, 1}}; //二维数组赋初值

unsigned char IOStr [] ={3, 5, 2, 5, 3}; //没有指定数组长度，编译器自动设置

unsigned char code skydata [] ={0x02, 0x34, 0x22, 0x32, 0x21, 0x12}; //数据保存在 code 区

下面的一个简单例子是对数组中的数据进行排序，使用的是冒泡法，一来了解数组的使 用，二来掌握基本的排序算法。冒泡排序算法是一种基本的排序算法，它每次顺序取数组中 的两个数，并按需要按其大小排列，在下一次循环中则取下一次的一个数和数组中下一个数 进行排序，直到数组中的数据全部排序完成。

```
#include 《AT89X51.H》

#include 《stdio.h》

void taxisfun （int taxis2 [] ）

{

unsigned char TempCycA, TempCycB, Temp;

for （TempCycA=0; TempCycA《=8; TempCycA++）

for （TempCycB=0; TempCycB《=8-TempCycA; TempCycB++）

{//TempCycB《8-TempCycA 比用 TempCycB《=8 少用很多循环

if （taxis2 [TempCycB+1] 》taxis2 [TempCycB] ） //当后一个数大于前一个 数

{

Temp = taxis2 [TempCycB] ; //前后 2 数交换

taxis2 [TempCycB] = taxis2 [TempCycB+1] ;
```

```
taxis2 [TempCycB+1] = Temp; //因函数参数是数组名调用形
参的变动影响实参
}
}
}
void main (void)
{
int taxis [] = {113, 5, 22, 12, 32, 233, 1, 21, 129, 3};
char Text1 [] = {“source data: ”}; //“源数据”
char Text2 [] = {“sorted data: ”}; //“排序后数据”
unsigned char TempCyc;
SCON = 0x50; //串行口方式 1，允许接收
TMOD = 0x20; //定时器 1 定时方式 2
TCON = 0x40; //设定时器 1 开始计数
TH1 = 0xE8; //11.0592MHz 1200 波特率
TL1 = 0xE8; TI = 1;
```

```
TR1 = 1; //启动定时器
printf ( “%s\n”, Text1) ; //字符数组的整体引用
for (TempCyc=0; TempCyc《10; TempCyc++)
printf ( “%d ”, taxis [TempCyc] ) ;
printf ( “\n----------\n” ) ;
taxisfun (taxis) ; //以实际参数数组名 taxis 做参数被函数调用
printf ( “%s\n”, Text2) ;
for (TempCyc=0; TempCyc《10; TempCyc++) //调用后 taxis 会被改变
printf ( “%d ”, taxis [TempCyc] ) ;
while (1) ;
}
```

例子中能看出，数组同样能作为函数的参数进行传递。数组做参数时是用数组名进行传递的，一个数组的数组名表示该数组的首地址，在用数组名作为函数的调用 参数时，它的传递方式是采用了地址传递，就是将实际参数数组的首地址传递给函数中的形式参数数组，这个时候实际参数数组和形式参数数组实际上是使用了同一段内存单元，当形式参数数组在 函数体中改变了元素的值，同时也会影响到实际参数数组，因为它们是存放在同一个地址的。 上面的例子同时还使用到字符数组。字符数组中每一个数据都是一个字符，这样一个一 维的字符数组就组成了一个字符串，在 C 语言中字符串是以字符数组来表达处理的。为了 能测定字符串的长度，C 语言中规定以‘\o’来做为字符串的结束标识，编译时会自动在字 符串的最后加入一个‘\o’，那么要注意的是如果用一个数组要保存一个长度为 10 字节的字 符串则要求这个数组至少能保存 11 个元素。‘\o’是转义字符，它的

含义是空字符，它的 ASCII 码为 00H，也就是说当每一个字符串都是以数据 00H 结束的，在程序中操作字符数 据组时要注意这一点。字符数组除了能对数组中单个元素进行访问，还能访问整个数组， 其实整个访问字符数组就是把数组名传到函数中，数组名是一个指向数据存放空间的地址指 针，函数根据这个指针和‘/o’就能完整的操作这个字符数组。对于这一段所说的，能 参看下面一例 1602LCD 显示模块的驱动演示例子进行理解。这里要注意就是能用单个字

符数组元素来进行运算，但不能用整个数组来做运算，因为数组名是指针而不是数据。

```
/*=========================================================

使用 1602 液晶显示的实验例子 明浩 2004/2/27

=========================================================

SMC1602A（16*2）模拟口线接线方式 连接线图：

-----------------------------------------------

|LCM-----51 | LCM-----51 | LCM------51 |

-------------------------------------------|

|DB0-----P1.0 | DB4-----P1.4 | RW-------P2.0 |

|DB1-----P1.1 | DB5-----P1.5 | RS-------P2.1 |

|DB2-----P1.2 | DB6-----P1.6 | E--------P2.2 |

|DB3-----P1.3 | DB7-----P1.7 | VLCD 接 1K 电阻到 GND|

-----------------------------------------------
```

```
    [注：AT89S51 使用 12M 晶体震荡器]

    ================================================================*/

    #define LCM_RW P2_0 //定义引脚

    #define LCM_RS P2_1

    #define LCM_E P2_2

    #define LCM_Data P1

    #define Busy 0x80 //用于检测 LCM 状态字中的 Busy 标识

    #include 《at89x51.h》

    void WriteDataLCM（unsigned char WDLCM）；

    void WriteCommandLCM（unsigned char WCLCM，BuysC）；

    unsigned char ReadDataLCM（void）；unsigned char ReadStatusLCM（void）；void
LCMInit（void）；

    void DisplayOneChar（unsigned char X， unsigned char Y， unsigned char DData）；

    void DisplayListChar（unsigned char X， unsigned char Y， unsigned char code
*DData）；

    void Delay5Ms（void）；

    void Delay400Ms（void）；
```

```
unsigned char code cdle_net [] = {“www.51hei.com”};

unsigned char code email [] = {“pnzwzw@51hei.com”};

void main (void)

{

Delay400Ms ( ) ; //启动等待，等 LCM 讲入工作状态

LCMInit ( ) ; //LCM 初始化

Delay5Ms ( ) ; //延时片刻（可不要）

DisplayListChar(0, 0, cdle_net); DisplayListChar(0, 1, email); ReadDataLCM
( ) ;//测试用句无意义 while (1) ;

}

//写数据

void WriteDataLCM (unsigned char WDLCM)

{

ReadStatusLCM ( ) ; //检测忙 LCM_Data = WDLCM; LCM_RS = 1;

LCM_RW = 0;

LCM_E = 0; //若晶体震荡器速度太高能在这后加小的延时

LCM_E = 0; //延时
```

```
    LCM_E = 1;
    }
    //写指令
    void WriteCommandLCM（unsigned char WCLCM，BuysC） //BuysC 为 0 时忽略忙检
测
    {
    if （BuysC） ReadStatusLCM（）；//根据需要检测忙
    LCM_Data = WCLCM; LCM_RS = 0; LCM_RW = 0;
    LCM_E = 0;
    LCM_E = 0; LCM_E = 1;
    }
    //读数据
    unsigned char ReadDataLCM（void）
    {
    LCM_RS = 1; LCM_RW = 1; LCM_E = 0; LCM_E = 0; LCM_E = 1; return（LCM_Data）；
    }
    //读状态
```

```
unsigned char ReadStatusLCM（void）

{

LCM_Data = 0xFF; LCM_RS = 0; LCM_RW = 1; LCM_E = 0; LCM_E = 0; LCM_E = 1;

while （LCM_Data & Busy）; //检测忙信号

return（LCM_Data）;

}

void LCMInit（void） //LCM 初始化

{

LCM_Data = 0;

WriteCommandLCM（0x38，0）; //三次显示模式设置，不检测忙信号

Delay5Ms（）;WriteCommandLCM（0x38，0）;Delay5Ms（）;WriteCommandLCM（0x38，
0）; Delay5Ms（）;

WriteCommandLCM（0x38，1）; //显示模式设置，开始要求每次检测忙信号

 WriteCommandLCM（0x08，1）; //关闭显示 WriteCommandLCM（0x01，1）; //显示
清屏 WriteCommandLCM（0x06，1）;// 显示光标移动设置 WriteCommandLCM（0x0C，1）;
// 显示开及光标设置

}

//按指定位置显示一个字符
```

```
void DisplayOneChar（unsigned char X， unsigned char Y， unsigned char DData）
{
Y &= 0x1;
X &= 0xF; //限制 X 不能大于 15，Y 不能大于 1
if （Y） X |= 0x40; //当要显示第二行时地址码+0x40; X |= 0x80; //算出指令码
WriteCommandLCM（X， 0）; //这里不检测忙信号，发送地址码
WriteDataLCM（DData）;
}
//按指定位置显示一串字符
void DisplayListChar（unsigned char X， unsigned char Y， unsigned char code
*DData）
{
unsigned char ListLength;
ListLength = 0; Y &= 0x1;
X &= 0xF; //限制 X 不能大于 15，Y 不能大于 1
while （DData [ListLength] 》0x20） //若到达字串尾则退出
{
```

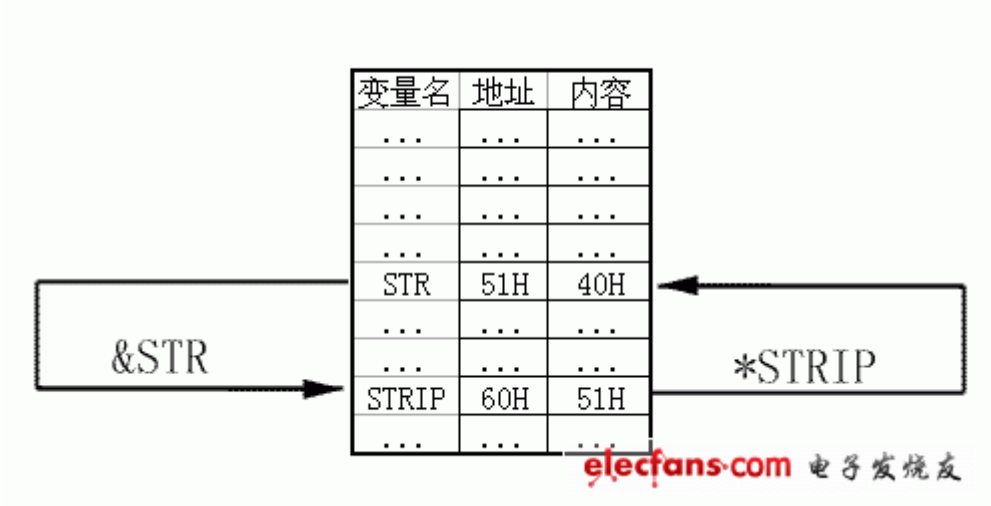
```
if （X 《= 0xF） //X 坐标应小于 0xF
{
DisplayOneChar（X， Y， DData [ListLength] ）; //显示单个字符
ListLength++; X++;
}
}
}
//5ms 延时
void Delay5Ms（void）
{
unsigned int TempCyc = 5552;
while（TempCyc--）;
}
//400ms 延时
void Delay400Ms（void）
{
```

```
unsigned char TempCycA = 5; unsigned int TempCycB; while (TempCycA--)
{
TempCycB=7269;
while (TempCycB--) ;
};
}
```

#p#C51 指针的使用#e#

第十六课、C51 指针的使用

指针就是指变量或数据所在的存储区地址。如一个字符型的变量 STR 存放在内存单元 DATA 区的 51H 这个地址中，那么 DATA 区的 51H 地址就是变量 STR 的指针。在 C 语言中 指针是一个很重要的概念，正确有效的使用指针类型的数据，能更有效的表达复杂的数据 结构，能更有效的使用数组或变量，能方便直接的处理内存或其它存储区。指针之所以 能这么有效的操作数据，是因为无论程序的指令、常量、变量或特殊寄存器都要存放在内 存单元或相应的存储区中，这些存储区是按字节来划分的，每一个存储单元都能用唯一的 编号去读或写数据，这个编号就是常说的存储单元的地址，而读写这个编号的动作就叫做寻 址，通过寻址就能访问到存储区中的任一个能访问的单元，而这个功能是变量或数组等 是不可能代替的。C 语言也因此引入了指针类型的数据类型，专门用来确定其他类型数据的 地址。用一个变量来存放另一个变量的地址，那么用来存放变量地址的变量称为“指针变量”。 如用变量 STRIP 来存放文章开头的 STR 变量的地址 51H，变量 STRIP 就是指针变量。下面 用一个图表来说明变量的指针和指针变量两个不一样的概念。



变量的指针就是变量的地址，用取地址运算符‘&’取得赋给指针变量。&STR 就是把变量 STR 的地址取得。用语句 STRIP = &STR 就能把所取得的 STR 指针存放在 STRIP 指针变量中。STRIP 的值就变为 51H。可见指针变量的内容是另一个变量的地址，地址所属的 变量称为指针变量所指向的变量。

要访问变量 STR 除了能用‘STR’这个变量名来访问之外，还能用变量地址来访 问。方法是先用&STR 取变量地址并赋于 STRIP 指针变量，然后就能用*STRIP 来对 STR 进行访问了。‘*’是指针运算符，用它能取得指针变量所指向的地址的值。在上图中指针 变量 STRIP 所指向的地址是 51H，而 51H 中的值是 40H，那么*STRIP 所得的值就是 40H。使用指针变量之前也和使用其它类型的变量那样要求先定义变量，而且形式也相类似，

一般的形式如下：

数据类型 ［存储器类型］ * 变量名;

unsigned char xdata *pi //指针会占用二字节，指针自身存放在编译器默认存储区，指

向 xdata 存储区的 char 类型

unsigned char xdata * data pi; //除指针自身指定在 data 区，其它同上

int * pi; //定义为一般指针，指针自身存放在编译器默认存储区，占三个字节 在定义形式中“数据类型”是指所定义的指针变量所指向的变量的类型。“存储器类型”

是编译器编译时的一种扩展标识，它是可选的。在没有“存储器类型”选项时，则定义为一

般指针，如有“存储器类型”选项时则定义为基于存储器的指针。限于 51 芯片的寻址范围，

指针变量最大的值为 0xFFFF，这样就决定了一般指针在内存会占用 3 个字节，第一字节存 放该指针存储器类型编码，后两个则存放该指针的高低位址。而基于存储器的指针因为不用 识别存储器类型所以会占一或二个字节，idata，data，pdata 存储器指针占一个字节，code，xdata 则会占二个字节。由上可知，明确的定义指针，能节省存储器的开销，这在严格要求程序 体积的项目中很有用处。

指针的使用方法很多，限于篇幅以上只能对它做一些基础的介绍。下面用在讲述常量时 的例程改动一下，用以说明指针的基本使用方法。

#include 《AT89X51.H》 //预处理文件里面定义了特殊寄存器的名称如 P1 口定义为 P1

void main (void)

{

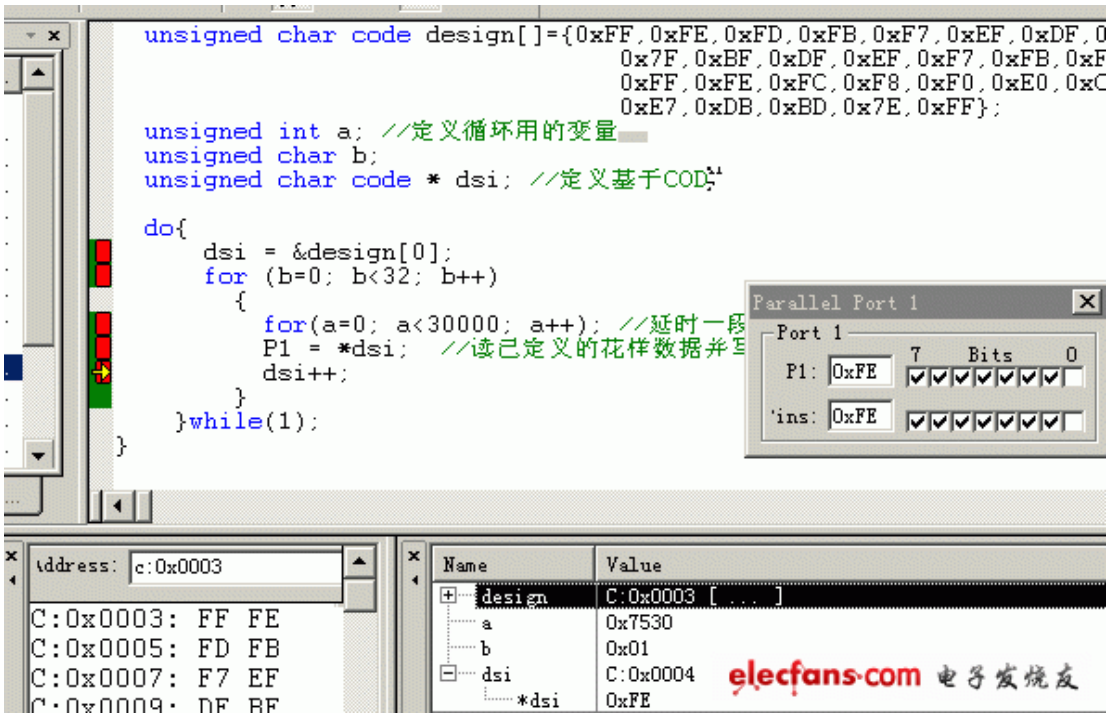
//定义花样数据，数据存放在片内 CODE 区中

```
    unsigned char code design [] ={0xFF, 0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF,
0x7F,

    0x7F, 0xBF, 0xDF, 0xEF, 0xF7, 0xFB, 0xFD, 0xFE, 0xFF,

    0xFF, 0xFE, 0xFC, 0xF8, 0xF0, 0xE0, 0xC0, 0x80, 0x0,

    0xE7, 0xDB, 0xBD, 0x7E, 0xFF};

    unsigned int a; //定义循环用的变量

    unsigned char b;

    unsigned char code * dsi; //定义基于 CODE 区的指针

    do{

    dsi = &design [0] ; //取得数组第一个单元的地址

    for  (b=0; b《32; b++)

    {

    }

    }while (1) ;

    }

    for (a=0; a《30000; a++) ; //延时一段时间

    P1 = *dsi; //从指针指向的地址取数据到 P1 口
```

dsi++; //指针加一，

为了能清楚的了解指针的工作原理，能使用 keil uv2 的软件仿真器查看各变量和存储器的

值。编译程序并执行，然后打开变量窗口，如图。用单步执行，就能查到到指针的变量。 如图中所示的是程序中循环执行到第二次，这个时候指针 dsi 指向 c:0x0004 这个地址，这个地址 的值是 0xFE。在存储器窗口则能察看各地址单元的值。使用这种方法不但在学习时能 帮助更好的了解语法或程序的工作，而且在实际使用中更能让你更快更准确的编写程序或解 决程序中的问题。



#p#C51 结构、联合和枚举的使用#e#

第十七课、C51 结构、联合和枚举的使用

前面的文章中介绍了 C 语言的基本数据类型，为了更有效的处理更复杂的数据，C 语言引入了构造类型的数据类型。构造类型就是将一批各种类型的数据放在一起形成一种特殊 类型的数据。之前讨论过的数组也算是一种构造类型的数据，单片机 c 语言 中的构造类型还有结构、 枚举和联合。

结构

结构是一种数据的集合体，它能按需要将不一样类型的变量组合在一起，整个集合体用 一个结构变量名表示，组成这个集合体的各个变量称为结构成员。理解结构的概念，能用 班级和学生的关系去理解。班级名称就相当于结构变量名，它代表所有同学的集合，而每个 同学就是这个结构中的成员。使用结构变量时，要先定义结构类型。一般定义格式如下：

```
struct 结构名 {结构元素表};

例子：struct FileInfo

{

unsigned char FileName [4] ; unsigned long Date; unsigned int Size;

}
```

上面的例子中定义了一个简单的文件信息结构类型，它可用于定义用于简单的单片机文 件信息，结构中有三个元素，分别用于操作文件名、日期、大小。因为结构中的每个数据成 员能使用不一样的数据类型，所以要对每个数据成员进行数据类型定义。定义好一个

结构类 型后，能按下面的格式进行定义结构变量，要注意的是只有结构变量才能参与程序的执 行，结构类型只是用于说明结构变量是属于那一种结构。

struct 结构名 结构变量名 1，结构变量名 2……结构变量 N； 例子：struct FileInfo NewFileInfo， OleFileInfo;

通过上面的定义 NewFileInfo 和 OleFileInfo 都是 FileInfo 结构，都具有一个字符型数组 一个长整型和一个整形数据。定义结构类型只是给出了这个结构的组织形式，它不会占用存 储空间，也就说结构名是不能进行赋值和运算等操作的。结构变量则是结构中的具体成员， 会占用空间，能对每个成员进行操作。

结构是允许嵌套的，也就是说在定义结构类型时，结构的元素能由另一个结构构成。如：

```
struct clock
{
unsigned char sec， min， hour;
}
struct date
{
unsigned int year;
unsigned char month， day;
struct clock Time; //这是结构嵌套
}
```

struct date NowDate; //定义 data 结构变量名为 NowDate

开始学习的朋友看到这可能会发问：“各个数据元素要如何引用、赋值呢？”使用结构变量 时是通过对它的结构元素的引用来实现的。引用的方法是使用存取结构元素成员运算符“。” 来连接结构名和元素名，格式如下：

结构变量名。结构元素

要存取上例结构变量中的月份时，就要写成 NowDate..year。而嵌套的结构，在引用元 素时就要使用多个成员运算符，一级一级连接到最低级的结构元素。要注意的是在 单片机 c 语言 中 只能对最低级的结构元素进行访问，而不可能对整个结构进行操作。操作例子：

NowDate.year = 2005;

NowDate.month = OleMonth+ 2; //月份数据在旧的基础上加 2

NowDate.Time.min++; //分针加 1，嵌套时只能引用最低一级元素 一个结构变量中元素的名字能和程序中其他地方使用的变量同名，因为元素是属于它所在 的结构中，使用时要用成员运算符指定。

结构类型的定义还能有如下的两种格式。

struct

{

结构元素表

} 结构变量名 1，结构变量名 2……结构变量名 N;

例：struct

{

unsigned char FileName [4] ; unsigned long Date; unsigned int Size;

} NewFileInfo， OleFileInfo;

这一种定义方式定义没有使用结构名，称为无名结构。通常会用于程序中只有几个确定 的结构变量的场合，不能在其它结构中嵌套。

另一种定义方式如下：

struct 结构名

{

结构元素表

} 结构变量名 1，结构变量名 2……结构变量名 N;

例：struct FileInfo

{

unsigned char FileName [4] ; unsigned long Date; unsigned int Size;

} NewFileInfo， OleFileInfo;

使用结构名能便于阅读程序和便于以后要在定义其它结构中使用。 枚举

在程序中经常要用到一些变量去做程序中的判断标志。如经常要用一个字符或整型变量

去储存 1 和 0 做判断条件真假的标志，但我们也许会疏忽这个变量只有当等于 0 或 1 才是有

效的，而将它赋上别的值，而使程序出错或变的混乱。这个时候能使用枚举数据类型去定义变 量，限制错误赋值。枚举数据类型就是把某些整型常量的集合用一个名字表示，其中的整型 常量就是这种枚举类型变量的可取的合法值。枚举类型的二种定义格式如下：

enum 枚举名 {枚举值列表} 变量列表;

例 enum TFFlag {False， True} TFF;

enum 枚举名 {枚举值列表};

emum 枚举名 变量列表;

例 enum Week {Sun， Mon， Tue， Wed， Thu， Fri， Sat};

enum Week OldWeek， NewWeek;

看了上面的例子，你也许有一个地方想不通，那就是为什么枚举值不用贬值就能使 用？那是因为在枚举列表中，每一项名称代表一个整数值，在默认的情况下，编译器会自动 为每一项赋值，第一项赋值为 0，第二项为 1…。..如 Week 中的 Sun 为 0，Fri 为 5。C 语言也 允许对各项值做初始化赋值，要注意的是在对某项值初始化后，它的后续的各项值也随之递 增。如：

enum Week {Mon=1， Tue， Wed， Thu， Fri， Sat， Sun};

上例的枚举就使 Week 值从 1 到 7，这样会更符合我们的习惯。使用枚举就如变量一样， 但在程序中不能为其赋值。

联合

联合同样是 C 语言中的构造类型的数据结构。它和结构类型一样能包含不一样类型的 数据元素，所不一样的是联合的数据元素都是从同一个数据地址开始存放。结构变量占用的内 存大小是该结构中数据元素所占内存数的总和，而联合变量所占用内存大小只是该联合中最 长的元素所占用的内存大小。如在结构中定义了一个 int 和一个 char，那么结构变量就会占

用 3 个字节的内存，而在联合中同样定义一个 int 和一个 char，联合变量只会占用 2 个字节。 这种能充分利用内存空间的技术叫‘内存覆盖技术’，它能使不一样的变量分时的使用同一 个内存空间。使用联合变量时要注意它的数据元素只能是分时使用，而不能同时使用。举个 简单的例子，程序先为联合中的 int 赋值 1000，后来又为 char 赋值 10，那么这个时候就不能引用

int 了，不然程序会出错，起作用的是最后一次赋值的元素，而上一次赋值的元素就失效了。 使用中还要注意定义联合变量时不能对它的值初始化、能使用指向联合变量的指针对其操 作、联合变量不能作为函数的参数进行传递，数组和结构能出现在联合中。

联合类型变量的定义方法和结构的定义方法差不多，只要把关键字 struct 换用 union 就 能了。联合变量的引用方法除也是使用‘。’成员运算符。

下面就用一个综合的例子说明三种类型的简单使用。

```
#include 《AT89X51.H》

#include 《stdio.h》

void main (void)

{

enum TF {

False, True} State; //定义一个枚举，使程序更易读
```

```
union File { //联合中包含一数组和结构，
unsigned char Str [11] ; //整个联合共用 11 个字节内存
struct FN {
unsigned char Name [6] , EName [5] ;} FileName;
} MyFile;
unsigned char Temp;
SCON = 0x50; //串行口方式 1，允许接收
TMOD = 0x20; //定时器 1 定时方式 2
TCON = 0x40; //设定时器 1 开始计数
TH1 = 0xE8; //11.0592MHz 1200 波特率
TL1 = 0xE8; TI = 1;
TR1 = 1; //启动定时器
State = True; //这里演示 State 只能赋为 False，True 两个值，其它无效
//State = 3;这样是错误的
printf （“Input File Name 5Byte： \n”）;
scanf（“%s”， MyFile.FileName.Name）; //保存 5 字节字符串要 6 个字节
```

```
printf （“Input File ExtendName 4Byte： \n”）；
scanf（“%s”， MyFile.FileName.EName）；
if （State == True）
{
printf （“File Name ： ”）；
for （Temp=0; Temp《12; Temp++）
printf （“%c”， MyFile.Str［Temp］）； //这里列出所有的字节
printf （“\n Name ： ”）；
printf （“%s”， MyFile.FileName.Name）；
printf （“\n ExtendName ： ”）；
printf （“%s”， MyFile.FileName.EName）；
}
while（1）；
}
```

图 17-1 所示是运行的结果，A 中所示是说明例程中联合中的数组和结构占用的是同一段地址的内存空间，而结构中的两数组是各占两段不一样内存空间。

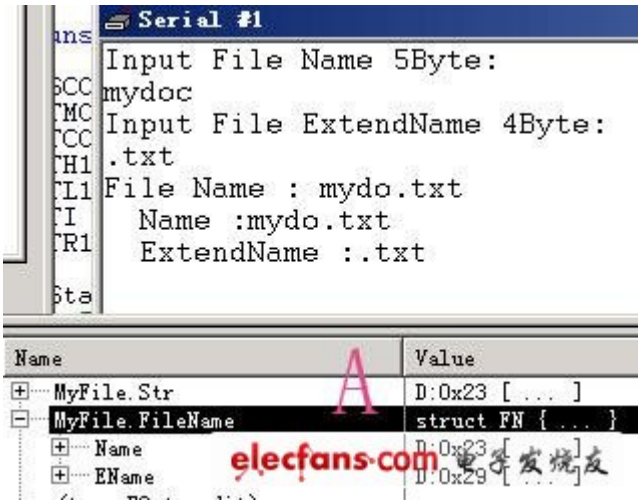


图 17-1

在此简单的单片机 C 语言教程就结束了，请关注电子发烧友网后续技术报道。

#p#附录(运算符优先级和结合性#e#

第十八课、附录(运算符优先级和结合性等

附表 1－2　C51 编译器的扩展关键字

关键字	用　途	说　明

auto	存储种类说明	用以说明局部变量，缺省值为此
break	程序语句	退出最内层循环
case	程序语句	Switch 语句中的选择项
char	数据类型说明	单字节整型数或字符型数据
const	存储类型说明	在程序执行过程中不可更改的常量值
continue	程序语句	转向下一次循环
default	程序语句	Switch 语句中的失败选择项
do	程序语句	构成 do..while 循环结构
double	数据类型说明	双精度浮点数
else	程序语句	构成 if..else 选择结构
enum	数据类型说明	枚举
extern	存储种类说明	在其他程序模块中说明了的全局变量
flost	数据类型说明	单精度浮点数
for	程序语句	构成 for 循环结构
goto	程序语句	构成 goto 转移结构
if	程序语句	构成 if..else 选择结构
int	数据类型说明	基本整型数
long	数据类型说明	长整型数
register	存储种类说明	使用 CPU 内部寄存的变量
return	程序语句	函数返回
short	数据类型说明	短整型数
signed	数据类型说明	有符号数，二进制数据的最高位为符号位
sizeof	运算符	计算表达式或数据类型的字节数
static	存储种类说明	静态变量
struct	数据类型说明	结构类型数据
swicth	程序语句	构成 switch 选择结构
typedef	数据类型说明	重新进行数据类型定义

union	数据类型说明	联合类型数据
unsigned	数据类型说明	无符号数数据
void	数据类型说明	无类型数据
volatile	数据类型说明	该变量在程序执行中可被隐含地改变
while	程序语句	构成 while 和 do..while 循环结构

附表 1－1　ANSIC 标准关键字

关键字	用　途	说　明
bit	位标量声明	声明一个位标量或位类型的函数
sbit	位标量声明	声明一个可位寻址变量
Sfr	特殊功能寄存器声明	声明一个特殊功能寄存器
Sfr16	特殊功能寄存器声明	声明一个 16 位的特殊功能寄存器
data	存储器类型说明	直接寻址的内部数据存储器
bdata	存储器类型说明	可位寻址的内部数据存储器
idata	存储器类型说明	间接寻址的内部数据存储器
pdata	存储器类型说明	分页寻址的外部数据存储器
xdata	存储器类型说明	外部数据存储器
code	存储器类型说明	程序存储器
interrupt	中断函数说明	定义一个中断函数
reentrant	再入函数说明	定义一个再入函数
using	寄存器组定义	定义芯片的工作寄存器

附录二　AT89C51 特殊功能寄存器列表（适用于同一架构的芯片）
带*号的特殊功能寄存器都是可以位寻址的寄存器

符　号	地　址	注　释
*ACC	E0H	累加器
*B	F0H	乘法寄存器
*PSW	D0H	程序状态字
SP	81H	堆栈指针
DPL	82H	数据存储器指针低 8 位
DPH	83H	数据存储器指针高 8 位
*IE	A8H	中断允许控制器
*IP	D8H	中断优先控制器
*P0	80H	端口 0
*P1	90H	端口 1
*P2	A0H	端口 2
*P3	B0H	端口 3
PCON	87H	电源控制及波特率选择
*SCON	98H	串行口控制器
SBUF	99H	串行数据缓冲器
*TCON	88H	定时器控制
TMOD	89H	定时器方式选择
TL0	8AH	定时器 0 低 8 位
TL1	8BH	定时器 1 低 8 位
TH0	8CH	定时器 0 低 8 位
TH1	8DH	定时器 1 高 8 位

附录三　运算符优先级和结合性

级　别	类　别	名　称	运算符	结合性
1	强制转换、数组、结构、联合	强制类型转换	( )	右结合
		下标	[ ]	
		存取结构或联合成员	->或.	
2	逻　辑	逻辑非	!	左结合
	字　位	按位取反	~	
	增　量	加一	++	
	减　量	减一	--	
	指　针	取地址	&	
		取内容	*	
	算　术	单目减	-	
	长度计算	长度计算	sizeof	
3	算　术	乘	*	右结合
		除	/	
		取模	%	
4	算术和指针运算	加	+	
		减	-	
5	字　位	左移	<<	
		右移	>>	
6	关系	大于等于	>=	
		大于	>	
		小于等于	<=	
		小于	<	
7		恒等于	==	
		不等于	!=	

8	字　位	按位与	&	
9		按位异或	^	
10		按位或	|	
11	逻　辑	逻辑与	&&	左结合
12		逻辑或	||	
13	条　件	条件运算	?:	
14	赋　值	赋值	=	
		复合赋值	0p=	
15	逗　号	逗号运算	,	右结合

欢迎使用电子发烧友网为您提供的贴心服务，现在正为百万注册用户提供优质服务

数十万份技术资料免费下载：

http://www.elecfans.com/soft/

最新最全电子设计资料

http://www.elecfans.com/article/

海量参考电子设计电路图

http://www.elecfans.com/article/88/131/

电子发烧友社区 - 国内规模最大的电子工程师网上交流社区

http://bbs.elecfans.com/

电子行业资讯每日更新：

欢迎访问 电子发烧友网http://www.elecfans.com/
每日最新电子技术资料，设计电路图，行业资讯，百万工程师交流平台，上百万份资料下载基地

http://www.elecfans.com/article/90/

电子百科：电子工程师每天都需要查阅的百科全书

http://www.elecfans.com/baike/