51单片机

串口通讯

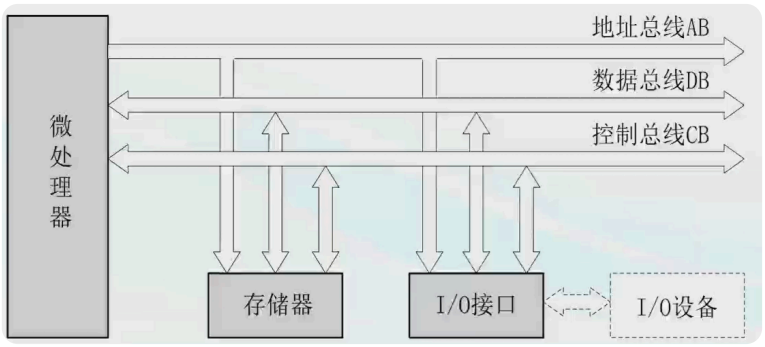
- 基本概念
- 通信类型
- 工作原理
- 如何使用51的UART模块

硬件概念

抽象思想

三大结构：微处理器、存储器、I/O接口。这三者由三大总线（地址、数据、控制）联系起来。

注意：这些都是**抽象概念**，下图也是一个抽象示意图，意思是它们是最笼统、广泛的一个叫法，仅仅表示一个精神而已，具体怎么实现按下不表



具体实现架构

单片机有八大结构，目的就是把刚才的思想实现出来：

1. CPU（运算+控制）
2. 数据存储器
3. 程序存储器
4. 特殊功能寄存器（SFR）

5. P0、P1、P2、P3口
6. 串行口
7. 定时计数器
8. 中断系统

酷炫名词之---**内部集成的外部设备**：指串行口、定时计数器、中断系统这些，它们相对于别的几个关键结构而言都是附加的“外部设备”，但是一般也要用，所以直接“集成地做好再芯片里了”，因而得名

细化---CPU

CPU由运算器和控制器组成

1. 运算器由运算单元ALU、累加器A、寄存器B、程序状态寄存器PSW构成，这样，“运算并且记录运算结果”的任务就能做到了

2. 控制器由PC、DPTR、SP、IR等组成。PC是一直指向下一个待操作指令的指针，DPTR是指向数据的指针；前者是程序空间里的导游，后者是数据空间里的导游。其他的几个先不管

细化---存储器

Princeton结构是程序与数据在一起存放的，Harvard结构则是分开的，51就是Harvard结构

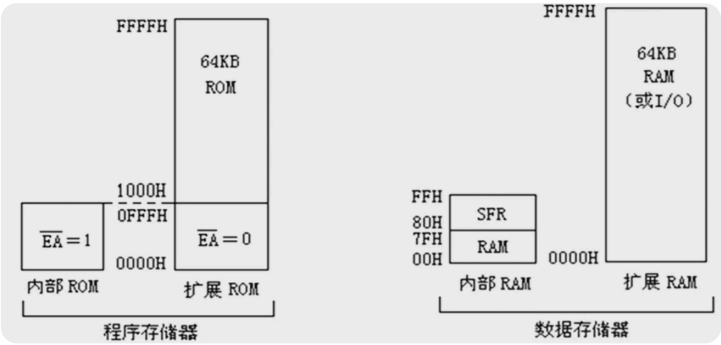
51的存储器可以在**物理结构**上分为5块：

1. 片内ROM程序存储器
2. 片外ROM程序存储器
3. 片内RAM数据存储器
4. 片外RAM数据存储器
5. （特殊功能寄存器）

片内就是“自带的、天生就有”的意思，片外就是“扩展”的意思。片内程序、片内数据、特

殊功能这三者都是本来就制作好在内的，另两个是额外加的

之所以把第五个打括号了，是因为SFR在物理上是和数据存储器合在一起的（见下图右边）



至于大小，比如上图，仅针对程序存储器而言，片内的是0-4KB，剩下的都是片外（扩展ROM上），一共64KB，EA（External access）如其名就是用来指示要存“内”还是“外”的。

而数据存储器比较复杂，片内256B（128+128），片外有64KB

在用户**使用逻辑**上来讲，分为这几块：

1. 64KB程序存储器地址空间（片内外统一编址）
2. 256B的片内数据存储器的地址空间
3. 64KB片外数据存储器的地址空间

这个256B其实分为两部分：数据存储器（低128单元）+特殊功能寄存器（高128单元）。但在叫名字的时候呢又统一叫数据存储器（把SFR扔掉了，还是挺绕的）

而现在搞到的都是增强型51（52），那么数据存储器是128+128=256了，也就是说这个高位的128也可以用来存数据（叫通用数据存储器），这时候这高位的128和SFR的128就**重叠**了。重叠了怎么访问不同的内容呢？其访问方式不同，是后话了

总之，这个128B在逻辑上两者是共用的，物理上两者却是分开的

汇编基础

注意

汇编中`;`并不是像C语言里标注一个语句结束的，而是用来写注释的，汇编的分号相当于`/* */`

尤为注意操作对象，比如`ADD`这一操作，后面跟的肯定是`A`（作为destination）。汇编中的操作都是和硬件相联系，因此不能随心所欲。如想把某两个地方的数据加起来，直接`ADD`是做不到的，只有分步走，先放到`A`里，再相加。而且顺序也很重要，`ADD R1, A`也是违法的，因为这样的话destination是R1而非A，还是错了

总之，具体格式怎么样，还请参阅指令清单，并不是像别的语言一样那么灵活

总体流程：人类书写一些英文（助记符。即汇编的代码），写好后由汇编器进行翻译，翻译好的东西（一些01数码）就写入ROM中，执行时就是按照ROM里的这些信息来操作

Directives

译为“伪指令”，实际上看英文更明显，这是一种“指导意见”，而非指令

那些加减运算或是转移数据的操作是实际的指令，它们是真正在执行阶段工作的。但是directives是给翻译官汇编器看的，汇编器起“把人写的英文代码翻译成ROM里面的数码”的翻译作用，而directives告诉它到底该怎么翻译：

`ORG` 指示初始的地址，比如`ORG 0002H`，表示翻译官需要把翻译的内容写到0002H去，依

次往后。相当于告诉考生“请从试卷的第某某页开始书写答案”

`EQU` 用于把某个constant用一个记号替代（同**宏**）比如写`HELLO EQU 25`，之后就可以用`#HELLO`代表25。这相当于“笔者以下把‘离散时间傅里叶变换’简写为DTFT，望读者周知”一样，并不是真正的什么操作指令

`END` 用于指示**翻译**结束，告诉汇编器，END后面的几行都没必要翻译写进ROM了。因此，程序实际的执行并不是到END停下来，END与程序的停止不相关，只与汇编器翻译的停止相关

程序的运行与ROM

当写入`MOV R5,#25H ;load 25H into R5`后，烧入ROM的如下图，其中0000存的`7D`代表的是“把operand移入R5”这个操作本身（opcode），而第二个地址0001中的25就是代表操作数（operand），下面的也同理。

Program 2-1: ROM Contents

Address	Code
0000	7D
0001	25
0002	7F
0003	34
0004	74
0005	00
0006	2D
0007	2F
0008	24
0009	12
000A	80
000B	FE

比如看看`7F`这个就猜出，他其实是与“把operand移入R5”这个操作差两位的“把operand移入R7”这一操作的opcode，而下面的34H就是该操作的operand。

以上图为例：初始接通电源时，PC指针（program counter）指向0000（不论ROM多大，都是从全零开始）位置。在完成第一次操作后，PC+=2（把25H移入R5后PC直接指向0002），这是因为该指令是2字节的指令（2-byte instruction）……之后又是一些2字节指

令，直到0006，是一个1字节指令，运行之后PC+=1……如此下去，直到没有为止

PC指针是导引的作用，但它本身只是16 bits的寄存器而已，所以ROM上限就是2^{16}bits = 64K bytes这么大，因为即使有更多也没用，PC指不到更大的地方

数据处理与RAM

8051中的所有数据都是存在8 bits的寄存器中的，因此要处理一个更高位数的数字，就需要人类自己动脑解决。所以就有了CY，OV等等指示8位运算有无溢出的东西，只有通过它们才能联系起两个8 bits寄存器，使它们的计算相互关联起来，一同处理更大数字的运算

SFR中的PSW，就是用来指示状态用的寄存器，一共8bits，其中有四位就是CY (carry)、AC(auxiliary carry)、P (parity)、OV (overflow)，这些东西不去干预，它自己就会随着程序的进行而更新，反映出状态信息。不过，也可以人为写些指令去设置它们的值。另外两位是RS0、RS1（下面马上就会登场），还有两位是可以用户定义的（PSW.1与PSW.5）

CY	AC	F0	RS1	RS0	OV	--	P

CY	PSW.7	Carry flag.
AC	PSW.6	Auxiliary carry flag.
F0	PSW.5	Available to the user for general purpose.
RS1	PSW.4	Register Bank selector bit 1.
RS0	PSW.3	Register Bank selector bit 0.
OV	PSW.2	Overflow flag.
--	PSW.1	User-definable bit.
P	PSW.0	Parity flag. Set/cleared by hardware each instuction cycle to indicate an odd/even number of 1 bits in the accumulator.

RS1	RS0	Register Bank	Address
0	0	0	00H - 07H
0	1	1	08H - 0FH
1	0	2	10H - 17H
1	1	3	18H - 1FH

Figure 2-4. Bits of the PSW Register

下面是RAM结构图。最下面有4个Register Bank（00~1F），注意这个标号指示的是byte，比如，Bank0（00~07）一共8个bytes，每一个bytes都是一个8 bits寄存器，这样8个8 bits寄存器合成一组，就是一个Bank。

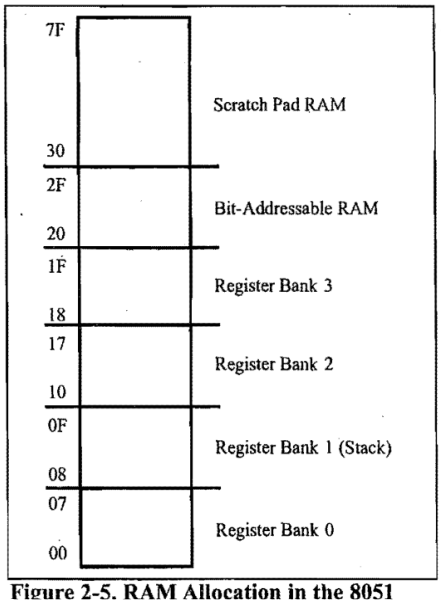


Figure 2-5. RAM Allocation in the 8051

比如写`MOV R4,#99H`就是把99H加载到Bank 0中的R4这个寄存器（注：`R0`到`R7`指Register Bank 0（默认的Bank）中的R0~R7这8个寄存器）。也可以用直接寻址的方式来书写，即`MOV 04,#99H`，两者是一个意思。那想要用其他Bank里的寄存器不就很不方便吗？什么时候`R0`到`R7`能指别的Bank里的寄存器呢？这时候就要依靠PSW了！刚才的RS0和RS1就是用来切换Bank用的：

Table 2-2: PSW Bits Bank Selection

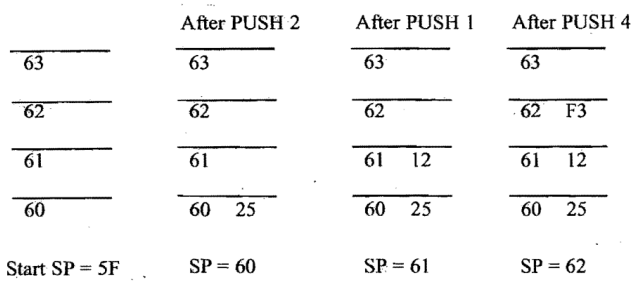
	RS1 (PSW.4)	RS0 (PSW.3)
Bank 0	0	0
Bank 1	0	1
Bank 2	1	0
Bank 3	1	1

实际上，刚才之所以用的是默认的Bank中的寄存器，就是因为PSW.3与PSW.4都是0的缘故。而命令`SETB PSW.4`可以让PSW.4变1，`CLR PSW.4`则是让它变0，在这些命令后，再写`MOV R4,#99H`一句，就会有不一样的效果！

RAM中还有一块是stack，`POP`和`PUSH`就是在这进行的：一开始栈顶指针（stack pointer）SP=07，`PUSH 6`这一句后，就会让SP=08，再把现在R6中的数据写入08位置的寄存器中，再来`PUSH 1`，就会让SP=09，再把现在R1中的数据写入09位置的寄存器中，再来`POP 2`，就会把现在所指的位置中的内容（目前是09中的内容）存出到R2中，然后指针下移SP=08…… 注意：stack的范围是08 ~ 1F，因为20开始就是位寻址区啦，如果你真想把栈做大，也不是没有办法，可以用30 ~ 7F的这些位置，不过你得自己把SP的内容改过去，如下例：

```
        MOV SP,#5FH  ;let 60H be the first location
        MOV R2,#25H
        MOV R1,#12H
        MOV R4,#0F3H
        PUSH 2
        PUSH 1
        PUSH 4
```

配套的流程图：



当然这个例子不仅仅是扩大用的，更是避免冲突用的，比如又想使用Bank1的`R0`到`R7`，又想使用栈操作，但它俩本是同一块地方（见上面的RAM图），所以像这样把SP直接调到上面，就避免了两者共用的冲突

跳转

用一个标签和一句跳转相配合，就能做到循环往复，比如`DJNZ`，用于把其对象减一，然后，若此时不为0就跳转到标签

```
        MOV A,#0       ;clear A
        MOV R2,#10     ;load counter R2=10
AGAIN:  ADD A,#03      ;add 03 to ACC
        DJNZ R2,AGAIN ;repeat until R2=0(10 times)
        MOV R5,A       ;save A in R5
```

该例中的`AGAIN`就是一个标签，而`R2`在执行`DJNZ`一句时每执行一次就减一，减一后不为0，下一步就跳回`AGAIN`处……直到10次以后，把R2减到0了，才会不跳转，然后执行后一句`MOV R5,A`。总体效果就是，把R2当成一个计数器用，一共执行`ADD A,#03`十次

```
        MOV R3,#10      ; R3=10, the outer loop count
NEXT:   MOV R2,#70      ; R2=70, the inner loop count
AGAIN:  ···             ; manipulation
        DJNZ R2,AGAIN   ;repeat it 70 times (inner loop)
        DJNZ R3,NEXT
```

上例展示了一个内外循环，外层由R3负责，内层由R2负责，这样可以循环700次，是单一寄存器（8 bits）远远达不到的循环次数

jump相关的指令操作还有很多（JZ,JNC等等），各个情况自己去看吧，注意所有**条件转移**都是短跳转，短跳转意味着目标地址是在当前PC的基础上$[-128, 127]$这一区间内，不能到更远的地方去。如果写一个`SJMP OVER`（`OVER`是一个标签）那么录入ROM中的就是80??，其中80表示操作码，??代表一个**相对地址**，比如??=03就代表在PC不考虑跳转的移动结束后，还要**多后移**3 bytes，诸如此类。比如`HERE: SJMP HERE`这一行代码，录入ROM中的就是80FE，FE表示在正常后移的基础上再多移动FE个，实际上会溢出，溢出后合效果相当于移动-2 bytes，会又倒退到本身这个位置，实现死循环，PC待在这里来回无穷往复

前面已经说了ROM的上限可有64K bytes，如果真想跳转到任意一个地方，用长跳转LJMP就行，因为长跳转是一个3 bytes命令（烧进ROM后占据3 bytes），其中后两个bytes合起来表示一个16位的地址。

CALL指令

`LCALL`：三字节，第一个字节为opcode，第二第三合并，可以指示16 bits，即ROM的任何地方，因此是long call。

`ACALL`：两字节，只能转到2K bytes之内的sub-routine。范围小但和LCALL相比省了ROM里的一个字节

在CALL这句执行时，处理器自动把这一条指令的后一个指令地址保存到栈中（记录回家

的地址），然后PC再跳到sub-routine去运行，直到遇到`RET`时，重新把栈里的地址取出来，这样PC就能回去了

端口 Port0~3

概况

端口0要接上拉电阻使用，1、2、3端口则不需要

四个端口都是**输入输出**双功能的，比如，想要用端口2轮番输出01010101和10101010，可以写：

```
        MOV A,#55H
BACK:   MOV P2,A
        ACALL DELAY
        CPL A          ;complernent reg A
        SJMP BACK
```

（注意这个CPL恰好把55H变成AAH）上述代码中`DELAY`的具体部分省略了（反正是一个写好的延时功能）

但是想要改成输入功能，即：读取来自端口2的高低电平信息，那就要把端口2先改为输入模式，输入**全1**以实现转变为输入模式：

```
;Get a byte from P2 and send it to P1
        MOV A,#0FFH     ;A=FFH
        MOV P2, A       ;make P2 an input port by writing all 1s to it
BACK:   MOV A,P2        ;get data from P2
        MOV P1,A        ;send it to Port 1
        SJMP BACK       ;keep doing that
```

该段代码实现把Port2作为一个输入端口，然后实时的把Port2中读到的，传递给Port1并且

输出

端口3则是有一些特殊的功能，比如interrupts和串口通信

输入输出切换?

刚才讲到“全1”切换到输入状态，这听起来有些玄乎，可能会问：切换到输入状态后又该怎么切换回输出呢？“全1”这个信号是什么神秘指令嘛?

其实，这些问题根本不是问题，而且这里面有两种情况需要甄别。这需从底层解释，请看下面的电路图：

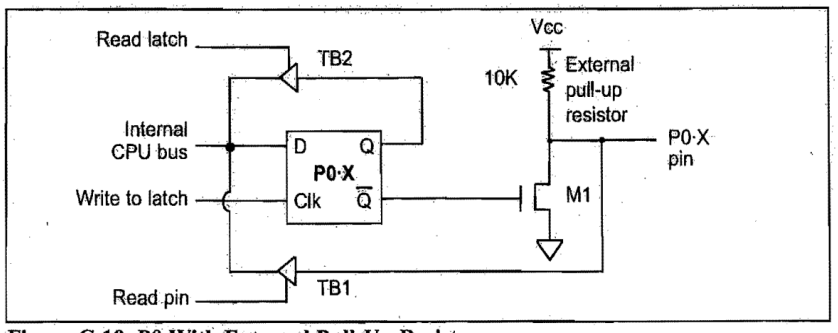


Figure C-19. P0 With External Pull-Up Resistor

首先，TB1和TB2是Tri-state buffer，是单入单出的。以TB1为例，当Read pin引脚（使能引脚）为1时，TB1右边（input）会传递到左边（output）

在图中见，TB1和TB2各有一个使能引脚，其命名与功能相符：TB1是用来读取端口（pin处）外来信息的，而TB2是用来读D型锁存器（latch）里的内容的（即Q）。“读取”的含义就是电平信息经过Tri-state buffer后，进入内部总线（internal CPU bus）到单片机里面去

所以，这个**读取对象**是因情况而异的，下面分两种情况：

Read pin：比如`MOV A,Pl`、`CJNE A,Pl,TARGET`、`JNB Pl.2,TARGET`这些指令，会让Read pin使能，读的对象是P0X引脚pin处的值。而“切换到输入状态”这句话是指在这个情况下：

1. 当锁存的状态为$Q=0$时，相应的$\bar{Q}=1$，这将开启M1这个门。一旦M1开启，Vcc经过M1直连Ground，导致10k电阻下方的那个节点就是地，恒为低电平。这样不论P0X具体是高电平还是低电平，该节点处电压，亦即传递到内部总线处的值，恒为低电平0，因此不能读取来自P0X处的信息，不是“输入状态”

2. 当锁存的状态为$Q=1$时，相应的$\bar{Q}=0$，这将关闭M1这个门。此时，P0X若是高电平，内部总线就是高电平，P0X若是低电平，内部总线就是低电平，此为正确的“输入状态”

所以，在执行这些指令前，为保险起见，先把要用的那个引脚对应的Q置个1比较好！一般来说，都是某端口所有引脚一起充当输入，那就干脆置“全1”好了

Read latch：像`ANL P1,A`、`JBC P1.1, TARGET`、`CPL P1.2`、`INC P1`等等指令，都会让Read latch使能，读的是锁存住的Q值。当然，这些代码比如`CPL P1.2`还包括改写Q这一步骤，因此叫做read-modify-write指令。总之，这和外部的P0X处的值没啥关系了，注意和刚才的Read pin区分！

那么，输出状态也很好理解了，一旦修改Q的值：比如为1，那么M1门关上，Vcc直接连到P0X，输出高电平；比如为0，那么M1门开，地直连P0X，输出低电平。所以，根本不存在什么“切换回输出模式”的问题

> *仍对于该内容有困惑，参阅 THE 8051 MICROCONTROLLER AND EMBEDDED SYSTEMS 的105页以及577页*

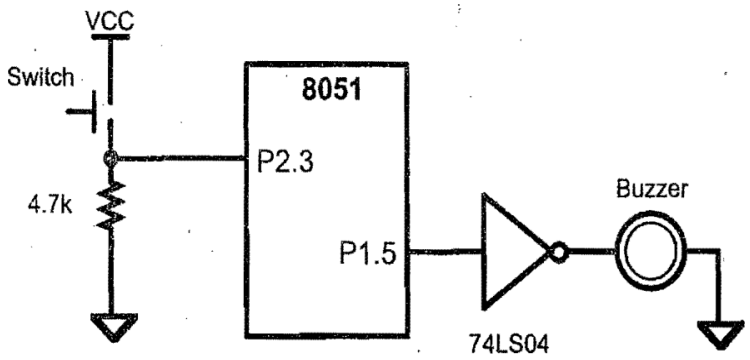
位操作

51的特点之一就是可以进行端口的逐位操作，比如下面就是在某一位上生成占空比66%的方波，而其他位不会受干扰：

```
BACK:   SETB P1.3       ;Set port 1 bit 3 high
        LCALL DELAY
        LCALL DELAY
        CLR P1.3
        LCALL DELAY
        SJMP BACK
```

也略去了`DELAY`具体的内容

下例是用来监视烤箱是否过热，如果过热就让蜂鸣器叫起来。其中P2.3口连着烤箱电路，如果高电瓶意味着过热；而P1.5连着蜂鸣器




```
HERE:   JNB P2.3,HERE
        SETB P1.5
        CLR P1.5
        SJMP HERE
```

下面的实例和刚才说的“切换到输入状态”有关：P1.7与一个开关相连，开关=1，向P2口发送Y，开关=0，向P2口发送N

```
        SETB P1.7
AGAIN:  MOV C,P1.7
        JC OVER          ;jump if Cy = 1
        MOV P2,#'N'
OVER:   SJMP AGAIN
        MOV P2,#'Y'
        SJMP AGAIN
```

上面的第一句就很重要了，否则万一P1.7是0，就没法把开关的状态传到单片机中。另外，这里把值移入了Carry位C中

Q&A for C51

Q: “可位寻址”是什么意思?

A: 比如有一串位10101110，这八个值是有名字的，比如其中某一个0代表的是“TR1=0”，如果是**可寻址**的，那么在程序中写“TR1=1”这样的语句是合法的，即允许直接单独操控其中某一位；而**不可寻址**的情况下这样的语句就是违法的。

Q: sbit和bit的区别?

A: bit可以理解成和int、double一样的东西，只不过是只有一个位那么大，写`bit a = 1;`就是在可寻址的空间里新开了一个位来放这个变量，存储1

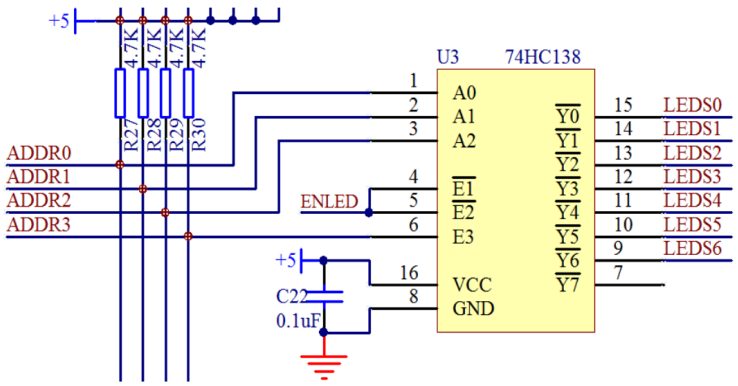
sbit就不一样了，它不能新开一个位来放东西。书写`sbit LED = P0^0;`只是给这个P0^0取一个名字LED而已。不论取不取名，它其实早就自己有一个确切的位地址了

Q: SFR?
A: 看了前面汇编就很清楚了，每一个小的基本功能存储器都存于SFR中，实现各种指令的

寄存器，如ADD对应的A寄存器，还有经常用的R0，R1等等，和各种功能直接相关的寄存器，统一放在SFR中

点亮小灯为例

74HC138



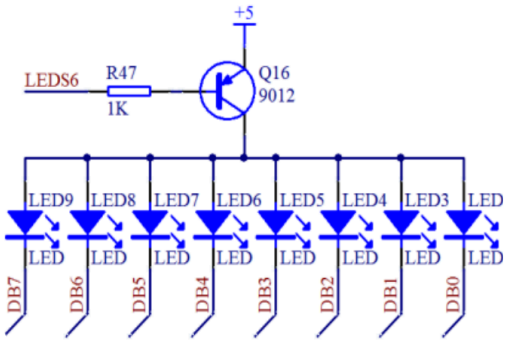
这是一个译码器，简言之，左边一堆是输入，右边一堆是输出。功能就是将3个口ADDR0,ADDR1,ADDR2的输入（二进制编码的形式）转换为右边8个口的one-hot输出形式：

- **A2,A1,A0→Y0~~Y7**
- **000→0 1 1 1 1 1 1 1**
- **001→1 0 1 1 1 1 1 1**
- **010→1 1 0 1 1 1 1 1**
- **011→1 1 1 0 1 1 1 1**
- **100→1 1 1 1 0 1 1 1**
- **101→1 1 1 1 1 0 1 1**
- **110→1 1 1 1 1 1 0 1**
- **111→1 1 1 1 1 1 1 0**

那ADDR3是干嘛的呢？是和ENLED一起来**使能**用的，要让ADDR3=1,ENLED=0，才能正常工作

三极管控制

三极管当成开关，像下图。当LEDS6口接到低电平（0信号）后，emitter与base之间有压差，开启三极管，可以点亮下面的LED；当接到高电平（1信号）后，base的电压过高，工作在截止区，因此不会点亮LED

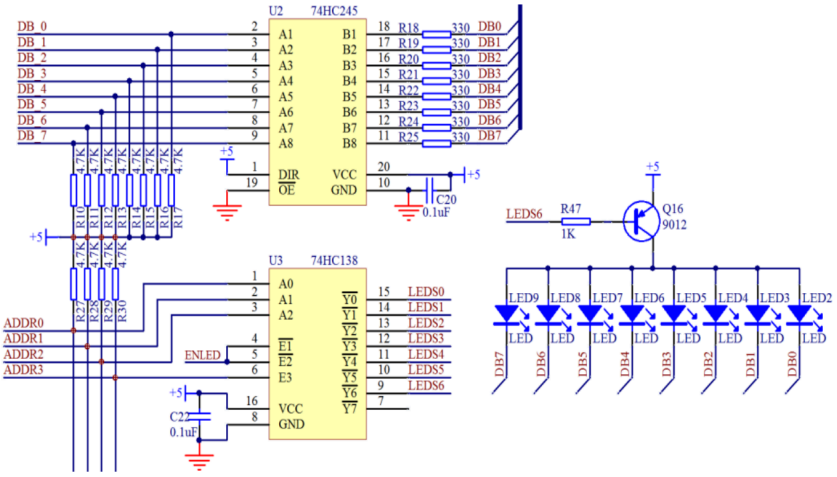


但这样不就8个小灯同亮同灭了吗？我想让他们一个一个亮啊！别忘了还有DB引脚，如果再控制一下DB引脚的电平高低，就可以做到分别控制8个小灯了，比如亮最右边的灯，只要在三极管开启时DB0=0，别的DB1~DB7=1就行了，因此再加上一个模块来控制DB，这个模块就是：

P0口

单片机的P0口是一个“有八个触角的接口”，分别是P0^0 到 P0^7，把这八个触角分别接到刚才的DB0到DB7，就可以通过单片机输出的触角处的高低电平影响DB处的高低电平，从而控制LED灯的亮灭

点亮小灯的整体框架



（图中标的名字一样的线，就是接在一起的）灯是接在LEDS6这个口上的，想点灯只要操作74HC138的LEDS6口（Y6）就行了，如何让Y6为0，而其他的Yn全为1呢？查查译码器的表，看输入的A0,A1,A2分别是什么的时候会有11111101的输出就好了，该A0,A1,A2组成的编码就是要用的打开三极管开关的密钥

打开三极管外，还要控制DBn，选择最右边的灯亮，那么让单片机输出的P0口为P0^0 =0,P0^1 到P0^7 =1即可（就是DB0=0，DB1~DB7=1）！其中74HC245可以当做是透明的不用看（起到缓冲作用，没有逻辑运算作用）

定时器

定时器（timer）也是在SFR中有的（T0与T1两个）

名称	描述	SFR 地址	复位值
TH0	定时器 0 高字节	0x8C	0x00
TL0	定时器 0 低字节	0x8A	0x00
TH1	定时器 1 高字节	0x8D	0x00
TL1	定时器 1 低字节	0x8B	0x00

此外，还有TCON，这个寄存器来控制定时器，这个寄存器一共8 bits，每一位都有对应的含义：

位	符号	描述
7	TF1	定时器 1 溢出标志。一旦定时器 1 发生溢出时硬件置 1。清零有两种方式：软件清零，或者进入定时器中断时硬件清零。
6	TR1	定时器 1 运行控制位。软件置位/清零来进行启动/停止定时器。
5	TF0	定时器 0 溢出标志。一旦定时器 0 发生溢出时硬件置 1。清零有两种方式：软件清零，或者进入定时器中断时硬件清零。
4	TR0	定时器 0 运行控制位。软件置位/清零来进行启动/停止定时器。
3	IE1	外部中断部分，与定时器无关，暂且不看。
2	IT1	
1	IE0	
0	IT0	

“硬件置1”的意思就是人不用去管，随着运行自动会更新，就像进位标志C（Carrry）一样，一旦进位它自动就变1了，不过它只是一个“指示灯”而已，不会施加额外影响。“软件清零”则是指人也有方法去干扰它的变化，以实现某种目的，就像可以人工写`CLR C`一句来清零进位指示符一样，这就是“软件清零”。

可以看到，这里的TF0和TF1就对应着以前汇编中经常用的的进位指示C

还有TMOD，定时器模式寄存器，有两种重要的mode：

1. 两个寄存器THn和TLn合在一起，16 bits的计时器
2. （8位自动重装）只用TLn计时，溢出后THn重装到TLn中。（n取0或1）

第二个模式中，THn全程袖手旁观，不会变化，但是一旦TLn溢出，就把THn中的内容装入TLn中去。THn相当于是一个基准，TLn计时的范围就是自THn的数值开始直到溢出。

这样，就可以推写算时间的程序：首先，已知晶振频率，其倒数就是时钟周期，又因为12个时钟周期等于一个机器周期，由此可以算出机器周期是多长时间，之后，比如时钟是从0运行到65536溢出，那每一次“从清零到溢出”就是65536个机器周期那么长时间，我们可以靠“溢出几次”来推算出过去了多久，如何知道“溢出几次”呢？我们不断观察TF，它一旦变1后，就马上软件清零TF，然后把`cnt++`，这样`cnt`这个变量就存储了时间信息。

中断

中断的流程（ISR是interrupt service routine，中断服务程序的简写）：

1. 结束当前的指令，并且将PC本该指向的下一个地址入栈
2. 保存目前各个中断的状态（存在内部某处）
3. 跳转到内存中固定的一块地方，称做**中断向量表**（interrupt vector table）处，得到ISR的地址
4. PC跳转到ISR处开始依次执行，直到遇到`RETI`指令
5. （准备回家）让栈顶两个出栈，代表该回去的地址，然后PC回到该地方，继续原来的进程

可以看出，如果你在执行中断程序ISR时，调用了堆栈，而且没有让“出栈数=入栈数”，那就捣毁了栈顶的内容，这样PC就回不到原进程那里了

中断向量表是这样的：

中断 函数编号	中断名称	中断 标志位	中断 使能位	中断 向量地址	默认 优先级
0	外部中断 0	IE0	EX0	0x0003	1(最高)
1	T0 中断	TF0	ET0	0x000B	2
2	外部中断 1	IE1	EX1	0x0013	3
3	T1 中断	TF1	ET1	0x001B	4
4	UART 中断	TI/RI	ES	0x0023	5
5	T2 中断	TF2/EXF2	ET2	0x002B	6

解释一下：

1. 这个表里其实还漏了0x0000位置的Reset，也就是ROM“醒来”（wake-up）后回到的位

置

2. 这个表里，中断使能位是用来“使能”这个中断的，是这个中断的闸门；中断标志位是“触发”中断的，就像光敏开关一样

看看其地址，这就是为什么写汇编时，开头要这样：

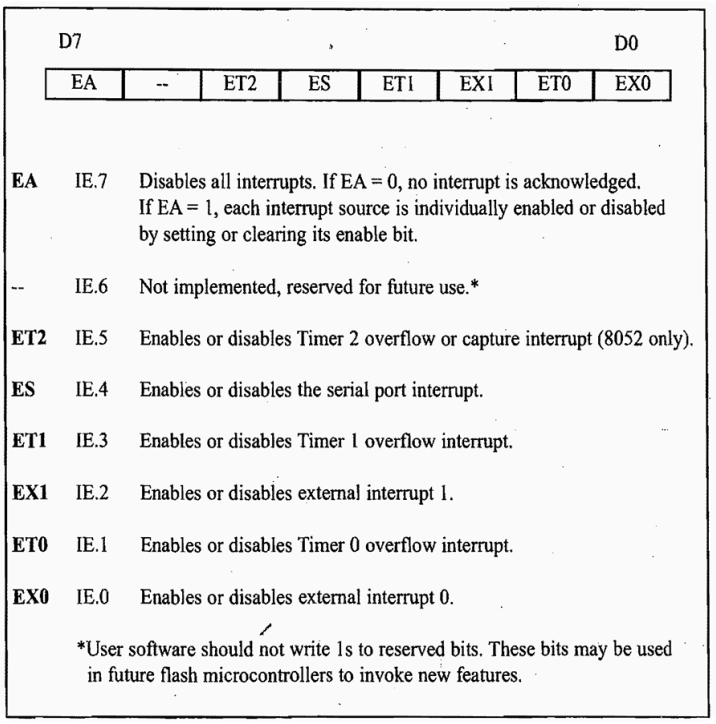
```
        ORG 000H
        LJMP MAIN       ;bypass interrupt vector table
        ORG 0030H
MAIN:
        ...
```

就是为了把PC引走，跳过中断向量表

让中断开启

Reset后，所有这些中断都会自动被disable，这就是为什么之前还不懂中断时，我们让时钟T0溢出了，也不会招致中断程序的运行，因为每次重启后中断仍处于disable态。

要启动某种中断，要打开该中断的闸门，这个总闸门就是**中断使能位**，在叫IE（interrupt enable）的一个寄存器上，IE如下：



D7							D0
EA	--	ET2	ES	ET1	EX1	ET0	EX0

EA	IE.7	Disables all interrupts. If EA = 0, no interrupt is acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
--	IE.6	Not implemented, reserved for future use.*
ET2	IE.5	Enables or disables Timer 2 overflow or capture interrupt (8052 only).
ES	IE.4	Enables or disables the serial port interrupt.
ET1	IE.3	Enables or disables Timer 1 overflow interrupt.
EX1	IE.2	Enables or disables external interrupt 1.
ET0	IE.1	Enables or disables Timer 0 overflow interrupt.
EX0	IE.0	Enables or disables external interrupt 0.

*User software should not write 1s to reserved bits. These bits may be used in future flash microcontrollers to invoke new features.

其中EA位就是总闸门（enable all），它为0时一票否决，没有任何中断会启用，它为1时，再轮到各个中断对应的闸门决定enable与否

T0中断的例子

实现功能：不断从P0口得到8 bits的信息并且发送到P1口，同时，还要在P2.1口生成一个方波信号，以200us为周期

```
        ORG 0000H
        LJMP MAIN

;ISR for Timer0 to generate square wave:
        ORG 000BH       ;Timer 0 interrupt vector table
        CPL P2.1        ;generate square wave
        RETI

        ORG 0030H       ;after vector table space
MAIN:   MOV TMOD,#02H   ;Timer 0, mode 2 (auto-reload)
        MOV P0,#0FFH    ;make P0 an input port
        MOV TH0,#-92    ;TH0=A4H for -92
        MOV IE,#82H     ;i.e. 10000010, enable Timer 0
        SETB TR0        ;Start Timer 0
```

```
BACK:   MOV A,P0
        MOV P1,A
        SJMP BACK

        END
```

解释：

1. 给IE寄存器置#82H的原因是，这相当于10000010，即：给EA置1，给ET0位1，其他都是0。总效果是：启动中断的总闸门EA，再给ET0使能，这样，Timer 0这种中断的中断标志位（TF0）就得到权力，能起到决定作用了，它一旦溢出，中断标志位变1，就会启动中断，然后跳转到000BH开始执行，直到RETI为止。相反，如果ET0和EA中的任何一个没有使能，都不会启动中断。

2. 给TH0加载-92的原因是，要控制T0的溢出时间。别忘记了我们用的是mode 2，也就是“自动重装载”模式，即只用TL0计时，一旦TL0溢出后，把TH0的内容加载进去。现在，每次加载的都是-92，意味着经过92个机器周期后就会溢出然后触发中断，而92个机器周期恰好是100us（这个根据具体晶振自己去算，当晶振11.0592MHz时，100us是92个机器周期：$100 \times 10^{-6} \div (12 \div 11.0592 \times 10^{6})$）

3. 注意，在TF0进位标志位变1后，我们不用手动去把它置为0，因为8051会在中断触发后自动清零标志位TF0，是中断时新有的一步

4. 注意，这个例子里把中断的程序ISR都写在000BH位置的后面，是因为恰好这个例子里的ISR非常短，就一个CPL而已。但长一点的中断程序就不能这样做了，看看中断向量表就知道，0013H的位置就有外部中断1的内容了，短短000BH到0013H这8 bytes的位置里，写不下太长的。所以一般都会这样写：

```
        ORG 000BH
        LJMP ISR_T0     ;jump to ISR_T0
```

然后在ISR_T0子程序的那一段末尾写`RETI`，和用法和CALL差不多

串口通讯

基本概念

1. 并行通信：就像有8个车道同时过去8辆车一样，占用8根线

2. 串行通信：就如一条车道，一次过一辆车，一个0xFE这样一个字节的数据要传输过去的话，发送方式就是0-1-1-1-1-1-1-1-1，一位一位的发送出去的，发送8次才送完一个字节

以下是串口传输与并行传输的比较，串口通讯使得传输更远更简单（远距离的并行传输的话，线不好搞，容易打架），一条线就够了

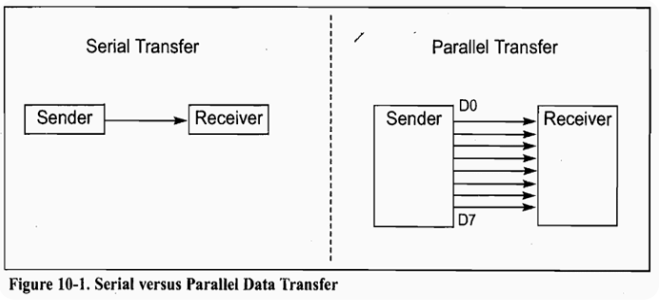


Figure 10-1. Serial versus Parallel Data Transfer

什么是modem（调制解调器）？

为了把0101串和电话的声调相互转换，就需要modem来modulate（调制）与demodulate（解调）

什么是UART?

universal asynchronous receiver-transmitter，是通用异步收发器

TXD是串行发送引脚（transmit），RXD是串行接收引脚（receive）

CH340?

之前在烧写程序的时候，已经接触过这个CH340，但它到底是啥？其实CH340T是一个USB转串口芯片，实现的是“USB通信协议和标准UART串行通信协议的转化”，所以我们才能用USB线去和单片机通讯

通信类型

单工通信、半双工通信、全双工通信

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

2. 半双工通信：数据可以在双方之间相互传播，但同一时刻只能其中一方发给另外一方，比如对讲机就是典型的半双工。

3. 全双工通信：发送数据的同时也能够接收数据，两者同步进行，就如同打电话，自己说话的同时也可以听到对方的声音。

工作原理

串口控制寄存器SCON

表 11-1　SCON——串行控制寄存器的位分配（地址 0x98、可位寻址）

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

表 11-2　SCON——串行控制寄存器的位描述

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

模式1最常用，即第一位是起始位，低电平；最后一位停止位，是高电平。如下是一个完整的按照模式1来的8 bits数据的发送，一共其实发了10位

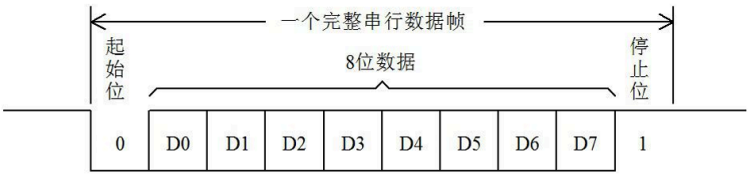


图 11-2　串口数据发送示意图

传输前，双方约定好速度（波特率），这样你传我读不会出错。然后，传出方将要传的打包好（前头加一个低电平起始位，后头加高电平停止位），送递出去。接收方聚精会神一直观察电平的变化，一旦出现一个低电平，它就知道信号将要来了，等一会会，开始“读一下”，就是D0位，然后再空一会，“读一下”，就读到D1位……直到读了8个位后，确认是高电平停止位，然后结束读取，弥留之际（停止位时）RI就会自动变1以标志之

实际上，51内置的串口模块的的采集数据的方式是比较保险，刚才的“读一下”实际是读好几下---采集16次。然后，把中间的三次取出来，这三次投票表决，其中两次如果是高电平，那么就认定这一位数据是1，如果两次是低电平，那么就认定这一位是0（之所以采中间的，是因为电平比较稳定）

如何使用51的UART模块

注意，UART模块配套的Timer是Timer1（或定时器T2），mode1下必须用Timer1的模式2（自动重装载）。意思是说，51已经内部弄好了，如果我们想要用串口通信，只要打开Timer 1，设置到合适的模式，然后SCON也设置匹配，那么，开启Timer 1后，**单片机自己就会开始准备接收或传输**

接收信号:

就像Timer一旦启动后，不用软件调控，它自己会计数递增，如果溢出了会以硬件置TF为1来告诉你一个道理。这里，一旦配置好后，它就会随时准备接受信息，一旦接受完一个，就会硬件置RI为1来告诉你。

就像你在阳台上放好了一个空水盆准备接雨水来浇花，只要放好盆子（配置好了），它会自动存储雨水。如果它盛满了，那么就会停在那里等着，（对应RI=1后停止接受）；如果这时你把这盆水拿去浇花（对应软件清零RI），然后盆子又可以拿来接受下一轮的雨水（开始等待下一个起始位的到来）……这样的方便之处在于，它会“自动”工作，即使中间很长一段时间都是晴天也没事，等哪一次下雨了，它仍会立即接水，等你想浇花了，观察一下有没有满（检查RI是否为1）就行了，中间的时候你不用操心了

1. 设置TMOD寄存器，加载#20H，代表使用Timer1的模式2（默认的）
2. 根据波特率计算一下TH1要装入的值
3. 设置SCON寄存器，加载#50H，代表使用串口通信的模式1（8位数据，十位包裹）
4. 启动Timer1：SETB TR1
5. CLR RI以清零RI
6. 一直监视RI标志位JNB RI xxx（因为一旦传输好了它会变1）
7. RI变1后，将SBUF中的内容能够存到安全的地方

8. 即将开始接收下一位，跳回步骤5

用串口接收传入数据并且在P1上呈现出来的例子：

```
        MOV TMOD,#20H   ;set Timer 1's mode
        MOV TH1,#-6     ;set baud rate
        MOV SCON,#50H   ;mode, 8 bit 1 stop, REN enable
        SETB TR1        ;start TR1
HERE:   JNB RI,HERE     ;wait until RI=1, indicating receiving data
        MOV A,SBUF      ;save the data
        MOV P1,A
        CLR RI          ;ready to receive
        LJMP HERE
```

传输信号：

同理，只不过这里是“自动会传输，一旦传好了，就硬件置TI为1来标志之”，流程为：

1. 设置TMOD寄存器，加载#20H，代表使用Timer1的模式2（默认的）

2. 根据波特率计算一下TH1要装入的值

3. 设置SCON寄存器，加载#50H，代表使用串口通信的模式1（8位数据，十位包裹）

4. 启动Timer1：SETB TR1

5. CLR TI以清零TI

6. 把要传走的首个数据写到SBUF里

7. 一直监视TI标志位JNB TI xxx（因为一旦传输好了它会变1）

8. 即将开始传输下一位，跳回步骤5

用4800的波特率传输字符'A'的例子：

```
        MOV TMOD,#20H    ;set Timer 1's mode
        MOV TH1,#-6      ;set baud rate
        MOV SCON,#50H    ;mode, 8 bit 1 stop, REN enable
        SETB TR1         ;start TR1
AGAIN:  MOV SBUFF,#"A"   ;load data
HERE:   JNB TI,HERE      ;wait until TI=1, indicating finishing transmitting the last one
        CLR TI           ;clear the flag
        LJMP AGAIN
```

实现不断传输"Y""E""S"的程序：

```
        MOV TMOD,#20H           ;set Timer 1's mode
        MOV TH1,#-3             ;set baud rate
        MOV SCON,#50H
        SETB TR1
AGAIN:  MOV A,#"Y"
        ACALL TRANS
        MOV A,#"E"
        ACALL TRANS
        MOV A,#"S"
        ACALL TRANS
        LJMP AGAIN

TRANS:  MOV SBUF,A
HERE:   JNB TI,HERE
        CLR TI
        RET
```

简单解释一下，`JNB TI,HERE`就是在等它传输完，因为`TI=0`时是还未完成，因此如果还未完成就仍然不执行后续内容，跳转回HERE原地踏步。

总结来讲，TI和RI的作用就是告诉你它是否准备好开启下一轮传输（或接收），比如，TI没变1，你就急于下一个8 bits的传输，那么SBUF里，上一轮未传完的东西就被你覆写掉

而未传送走。同样，为了开启新一轮传送，请先把变1的TI清0，这样才合法。