



## 法宝级的 EMC 单片机编程技巧集锦

### EM78 系列单芯片 - 提升软件效率的小程序

#### EM78 系列单芯片 - 提升软件效率的小程序

笔者闲暇时总喜欢一个人窝在房里拿烙铁，焊电路板，在网络上游走，看到喜欢的 DIY 也一定仔细端详，即使按图施工也可以得到不少的乐趣，相信酷爱此道的人应该也不少，除了喜欢看看别人的作品，也可以互相比较一下看谁用的零件少，谁提供的功能强，谁的速度最快，所以经常很容易就搜集到一些不错的电路，日子久了就像堆积木一样，可以一个方块一个方块的拿来用，吾人戏称为积木设计法。将许多有用的电路组合在一起，又是一个新的东西。这种方式的确又快又经济，符合现代人快餐的观念。不仅是硬件可以像堆积木一样的收集起来，软件当然也可以适用于积木法则，于是在不少有心人的努力之下，笔者也收集了 EM78 系列单芯片一些很好的链接库，所以说麻雀虽小，五脏俱全。也因为这些链接库极具参考价值，笔者不忍独享，故决定将紊乱的笔记重新整理后

公开出来，与热爱此系列单芯片的朋友们一同分享。

EM78XXX 单芯片自从问世以来已经陆续推出十余种不同等级的单芯片，小到 8Pin 的 78P152，大到 100Pin OTP 的 78P860，其汇编语言指令都是一样的，仅有 57 个，所以反复练习几次就能熟悉指令的用法。汇编语言用在 I/O 控制非常容易，也有很高的效率，所以坊间的书籍大部份以讨论控制为主，显少专门探讨软件技巧的篇幅，其实老手都知道，关于芯片之控制往往用到时再去翻一翻 DATA BOOK，注意一下 TIMING，然后准备一部示波器，三两下就可以搞定。反倒是算法用的好不好会大大影响产品的稳定度，所以有经验的程序设计师通常都有自己的一套葵花秘笈，所以要提升自己的功力最好的方式除了多练习之外，看看别人的程序也会使你进步很快。

#### BCD 转换成 Binary

由于 EM78XXX 是 8 位的微控器，因此为了节省内存，我们的范例仅以一个 BYTE 存放两位 BCD 数为例，数字的范围在 0~99 之间，转换后的结果放在 ACC，如果您需要更多的位数，相信您在看完之后应该不难自行修改才是。

#### 程序一

这个范例程序共花费 13 个指令 CYCLE，需要两个变量空间，执行后会影响到原 BCD 的内容。

```
MOV A,BCD
MOV TMP,A
MOV A,@0x0F
AND TMP,A
SWAP BCD
AND BCD,A
BC PSW,0
RLC BCD ; *2
MOV A,BCD
ADD TMP,A
RLC BCD
RLCA BCD ; *8
ADD A,TMP
```

#### 说明

在程序一中所采用的方式应该算是最多人知道的方式，也是一种最直觉的方法，先将 BCD 个位数保存起来，因为十位数必须要乘以 10，所以利用移位的技巧乘以 10 再加上个位数，所得的答案放入 ACC。

#### 程序二

在程序一的缺点，就是在执行程序以后，原本 BCD 的内容已经在移位的过程中被破坏掉了，为了改善这项缺失，我们换一种方式看看。下面这个程序，我们企图改善前面的缺失，共花费 11 个指令 CYCLE，仍需要两个变量空间，但是执行后不会破坏原来 BCD 的内容。



```
SWAPA BCD
MOV TMP,A
MOV A,@0x0F
AND BCD,A
AND TMP,A
BC PSW,0
RLCA TMP
SWAP TMP
RRC TMP
ADD A,TMP
ADD A,BCD
```

### 程序三

对于程序二的结果我们仍然不满意，似乎稍嫌复杂，虽然速度有所改善，但在内存的分配上仍有余地，所以我们再改善成程序三的类型。转换过程只花费 10 个指令 CYCLE，而且只需要一个变量空间，执行之后也不会改变原来 BCD 的内容。

```
MOV A,@0x0f
AND A,BCD
JBC BCD,4
ADD A,@10
JBC BCD,5
ADD A,@20
JBC BCD,6
ADD A,@40
JBC BCD,7
ADD A,@80
```

### 说明

看过以上三个范例，您是否觉得程序三最简洁而且容易了解？写程序的确是一项极具挑战性的工作，而且还可以找到很多灵感及乐趣，想不到吧！

### Binary 转换成 BCD 码

下面的范例程序会将存放在 ACC 内的二进制数转换成两位 BCD 码 ( Compacted BCD Code )，可转换最大的 BCD 码是 99。

```
CLR BCD
DIGIT_HI:
ADD A,@256-10
JBS PSW,FC
JMP DIGIT_LO
INC BCD
JMP DIGIT_HI
DIGIT_LO:
ADD A,@10
SWAP BCD
OR BCD,A
```

### 减法的陷阱

EM78 系列汇编语言的减法指令是 SUB，使用这个指令时您得特别注意，因为 ACC 永远都是减数，不可为被减数。SUB 指令的语法有以下三种：

```
SUB A,R (R-A  A)
SUB R,A (R-A  R)
```



SUB A,K(K-A A)

也就是说如果我们想计算 A-2 的值，如果写成：

SUB A,@2

其实是执行 2-A，解决方法如下：

ADD A,@256-2 或

ADD A,@254

交换两组缓存器的内容

如果你觉得要交换两组内存的内容一定要借用第三组变量，那么您可以参考以下的方式，只是用了一些数学技巧就变得又快又简单。

MOV A,REG1

SUB A,REG2

ADD REG1,A

SUB REG2,A

原理说明

A=REG1

A=REG2-REG1

REG1=REG1+A

=REG1+(REG2-REG1)

=REG2

REG2=REG2-(REG2-REG1)

=REG1

若 X>Y 就交换...

延续上一个例子，此法用应用在 Bubble Sort 特别管用。

MOV A,X

SUB A,Y

JBC PSW,FC

JMP NO\_CHANGE

ADD X,A

SUB Y,A

2 补码

2 补码加法经常代替减法，传统上的做法是先取 1 补码，然后加 1。

COM REG

INC REG

或是可以利用另一种方式求得，所不同的是第二种方式会影响 PSW 缓存器。

ADD A,REG

SUB A,REG

如果您所要求的数已经放在 ACC 里面，那只要一行就能解决了。

SUB A,@0

旋转字节运算

在 8051 指令中位左旋有 RLC 与 RL 两种指令区分，RLC 在 ACC 左旋时会连带将 CY 一并旋转，而 RL 只会将 ACC 的 MSB 旋入 LSB。EM78XXX 指令只有 RLC，那么要如何才能做到不带 CY 旋转呢？答案是旋转两次：

RLCA REG1

RLC REG1

如图 1 所示，第一次位旋转并没有真正改变 REG1 的内容，目的是将 REG1 的 MSB 先放入 FC，第二次位旋转才将刚刚放在 FC 内的 MSB 旋入 LSB。同理，两个 BYTES 不经 FC 的位旋转也是相同的原理。

RLCA HI\_BYTE



RLC LO\_BYTE

RLC HI\_BYTE

范围判断

写程序免不了会碰到 IF.THEN. 的场合，有些人觉得 EM78XXX 的条件判断式太过繁琐，所以笔者也将它们整理归纳一下。条件判断式可分为开放区间条件式与封闭区间条件式来讨论，以图 2 来表示。

开放条件式是以 N 点为出发点，当待测值大于 N 或是小于等于 N 时的条件判断，以 C 的语法描述如下：

```
if(number>n)
... /* number 大于 N */
else
... /* number 小于等于 N */
```

EM78XXX 汇编语言写法如下：

```
MOV A,@N+1
SUB A,Number
JBC PSW,FC
JMP LABEL_1; 大于 N
JMP LABEL_2; 小于等于 N
```

封闭式条件判断是指待测值 N 是否在 X 与 Y 的范围之内，若以 C 的语法描述：

```
if((number>=x) && (number<=y))
.... /* in range */
else
.... /* False */
```

如何以 EM78 汇编语言做到呢？一般做法是以减法后的 PSW 做条件判断，程序如下：

```
MOV A,@2
SUB A,number
JBS PSW,FC
JMP FALSE
MOV A,@y+1
SUB A,SI
JBC PSW,FC
JMP FALSE
IN_RANGE:
```

; ....

FALSE:

; ....

这个 IF 条件式要花费 8 个指令 Cycle，还不算太复杂。但是还有个更简洁的方法，以下用加法后的 PSW(R3) 做条件判断，一共只要 5 行就清洁溜溜了。

```
MOV A,Number
ADD A,@255-y
ADD A,@y-x+1
JBC PSW,FC
JMP IN_RANGE
```

FALSE:

; ....

IN\_RANGE:

; ....



## 说明

关键就在前三行， $x$  表示条件式的下限值， $y$  表示条件式的上限值，可以看得出仍是利用 CY 旗标制造的特效，不但精简而且有点小聪明，许多老手都爱用，这也是他们口袋里的秘密武器之一。如果您觉得不错，不妨也收入锦囊中，尔后就可以依样画葫芦了。

### ACC 与缓存器内容交换

这裡我们要介绍一种快速的逻辑算法，只需要 3 个指令 CYCLE，就可以将 ACC 的内容与缓存器的内容交换，不拖泥带水，Very cute!

```
XOR Number,A
```

```
XOR A,Number
```

```
XOR Number,A
```

请读者自行在纸上推算一次，就知道答案了。

### 交换多组缓存器内容

利用上面介绍的方法，可以推广到多组缓存器交换的例子，下面的程序将 5 组 DATA 内容移位，第一笔缓存器的数据传到第二笔缓存器内，第二缓存器的数据再传送到第三笔缓存器内，依此类推，最后一笔数据则传给第一个缓存器，形成一种字节数据旋转。

```
MOV A,@5
```

```
MOV COUNT,A
```

```
MOV A,@DATA1
```

```
MOV RSR,A
```

```
MOV A,DATA5
```

```
NEXT:
```

```
XOR INDIR,A
```

```
XOR A,INDIR
```

```
XOR INDIR,A
```

```
INC RSR
```

```
DJZ COUNT
```

```
JMP NEXT
```

计算 MOD 2N

假如你刚好需要计算  $ACC \bmod X$ ，且  $X$  刚好是 2 的  $N$  次方，使用  $ACC \text{ AND } (X-1)$  是最快的方法了。例如要判断 YEAR 是否为闰年，有个简单的方法，可以排除一些非闰年的条件，只要不能被 4 整除者就不是闰年。所以可以用  $YEAR \text{ AND } 3$  解决。

```
MOV A,@4-1
```

```
AND A,YEAR
```

```
JBS PSW,FZ
```

```
JMP NOLEAP
```

### 清除一段连续的内存

对于连续一段内存做读写最好的方式就是使用间接寻址法，但是要注意在一些如 M78447/811/860 等高阶 MCU，内存 20H~ 3FH 又可以分成 4 组 BANK，如果之前没有切换到正确的 BANK 会造成读写错误。下面的范例程序会将 BANK1 内的 32 个 BYTES 全部清为 0。

```
INDIR == 0x00
```

```
RSR == 0x04
```

```
COUNT == 0x10
```

```
REG == 0x20
```

```
BANK1 == 0x40
```

```
BANK2 == 0x80
```

```
BANK3 == 0xC0
```

```
MOV A,@32
```

```
MOV COUNT,A
```

```
MOV A,@REG|BANK1
```



```
MOV RSR,A
```

```
NEXT:
```

```
CLR INDIR
```

```
INC RSR
```

```
DJZ COUNT
```

```
JMP NEXT
```

计算一个 BYTE 中有多少个"1"

这个小程序可以检查出在某个 BYTE 中共有几个 1，在某些算法的过程可能会用得到，计算的结果放在 ACC。

```
RRCA DATA
```

```
AND A,@0x55
```

```
SUB DATA,A
```

```
MOV A,DATA
```

```
AND A,@0x33
```

```
ADD DATA,A
```

```
RRC DATA
```

```
ADD DATA,A
```

```
RRC DATA
```

```
SWAPA DATA
```

```
ADD A,DATA
```

```
AND A,@0x0F
```

节省 NOP 指令的方法

您还在为程序挤不下伤脑筋吗？NOP 指令有时候在延迟指令时间很有用，假如你有连续两个 NOP 指令可以用 JMP 到下一个指令的方式代替，因为这样可以减少一个指令 BYTE，又可以达到相同的效果。

例如：

```
NOP
```

```
NOP
```

可以写成：

```
JMP NEXT_INST
```

```
NEXT_INST:
```

```
;
```

因为一个 NOP 花费一个指令 Cycle，但是一个 JMP 指令就需要 2 个指令 Cycle，虽然有时候会抱怨 JMP 指令会多花一点时间，但是想不到它也有如此妙用吧。

LABEL 太多？

写汇编语言最令人伤脑筋的问题之一就是程序中到处是 label，这有两个坏处，第一就是不小心就会造成 label 重复的问题，第二就是想不出适当的 label 名称。如果您已经为 label 的命名问题肠枯思竭，给您提供一个小方法，程序中如果用「\$」可以表示目前 PC 的地址，依此推论「\$+2」表示 PC+2，「\$-4」表示 PC-4，看看底下的例子您立刻就明白：

```
MOV R,R
```

```
JBS PSW,FZ
```

```
JMP $+2
```

```
JMP $-4
```

```
;
```

不过也要给您一个建议，label 有个重要的意义就是具有批注的功能，特别是针对一些懒的写批注的人格外重要。所以这个方法仅适合使用在重复性很高的程序片断。

SWITCH...CASE 叙述

在程序设计的过程中，免不了常常会碰到多重选项的问题，利用 EM78XXX 的查表指令试试看，所以 TBL 除了当作一般查表指令



以外，还可以当作多重条件判断之用。

```
MOV A,CASE
TBL
JMP EVENT1 ; CASE=0
JMP EVENT2 ; CASE=1
JMP EVENT3 ; CASE=2
JMP EVENT4 ; CASE=3
```

#### 多字节的递增及递减运算

因为 EM78XXX 是 8 位的单芯片，如果要执行 8 位以上的计算，必须将多个字节看成是一个变量，以下我们举例说明如何将一组 24 位的变量，做到递增及递减运算。

#### 递增(Increment)

```
MOV A,@1
ADD INT24,A
JBC STATUS,FC
ADD INT24+1,A
JBC STATUS,FC
ADD INT24+2,A
```

#### 递减(Decrement)

```
MOV A,@1
SUB INT24,A
JBS STATUS,FC
SUB INT24+1,A
JBS STATUS,FC
SUB INT24+2,A
```

#### 判断多字节变量是否为零

使用简单的逻辑运算指令，将多字节 OR 在一起，然后依据 Z 旗标就可以判断此多字节变量是否为零了。

```
MOV A,INT24
OR A,INT24+1
OR A,INT24+2
JBS PSW,FZ
```

....

#### 复制某些位

有时候我们需要将一些特定的几个位由某个寄存器复制给另一组寄存器，由于并非完全复制寄存器的内容，所以会多了一些抽取位的步骤，现在我们找到一个方法，只要四个步骤就可以将指定的位复制到另一组寄存器里面，举例说明，假设位复制前 (SOURCE)=44H，(TARGET)=5AH，如果我们希望将 SOURCE 的 BIT0~BIT2 复制到 TARGET，则执行程序后(SOURCE)=44H，(TARGET)=5CH。

```
MOV A,SOURCE
XOR A,TARGET
AND A,@0000111B
XOR TARGET,A
```

无论您希望复制哪几个 BIT，只要将第三行程序 MASK 所需的位即可。

#### 奇偶位对调

以下这段程序是根据 Dmitry Kiryashov 的算法设计，假设原本 ACC 内所有位的排列顺序为 abcdefgh，交换后 ACC 顺序变成 badcfegh，程序只有五行，颇耐人寻味。

```
MOV REG,A
```



```
AND A,@0x55
```

```
ADD REG,A
```

```
RRC REG
```

```
ADD A,REG
```

中断程序不需保留 ACC 及 PSW 的方法

中断程序一定要保留 ACC 及 PSW 吗？那倒未必！特别是如果您使用的是 EM78P152/156 之类的迷你级的 MCU，RAM SIZE 都特别小，如果您只需要让 TCC 中断做简单的计数工作，只要小心使用指令，就可以避免中断程序会破坏到 ACC 及 PSW。原因是有些指令并不会对 PSW 产生影响，有些指令不需要经过 ACC。首先设定好预除器，并且让 TCC Free Run。下面的例子完全没用到 ACC 及 PSW。

```
ORG 0
```

```
JMP INIT
```

```
ORG 8
```

```
TCCINT:
```

```
BC RF,TCIF ;清除中断旗标
```

```
INC COUNTER
```

```
RETI
```

Multiple Task 管理与状态机

Multiple Task 就是将 CPU 时间平均分配（也可以是不平均分配）给多个 Task，所以在程序中会有一个时间管理者，依照指定的时间对指定的 Task 服务，没有分配到时间的 Task 必需等候时间到来才能执行。

```
TCCINT:
```

```
MOV R10,A
```

```
SWAP R10
```

```
SWAPA PSW
```

```
MOV R11,A
```

```
INC TASK
```

```
MOV A,@4
```

```
SUB A,TASK
```

```
JBS PSW,FC
```

```
JMP ENDINT
```

```
CLR TASK
```

```
ENDINT:
```

```
BC ISR,TCIF
```

```
SWAPA R11
```

```
MOV PSW,A
```

```
SWAPA R10
```

```
RETI
```

```
;-----
```

```
MAIN:
```

```
MOV A,@0x21
```

```
CONTW
```

```
CLR TCC
```

```
CLR ISR
```

```
MOV A,@0x01
```

```
IOW IOCF
```

```
CLR TASK
```



```
START:
MOV A,TASK
TBL
JMP TASK0
JMP TASK1
JMP TASK2
JMP TASK3
JMP TASK4
;-----
TASK0:
; ....
JMP START
TASK1:
; ....
JMP START
TASK2:
; ....
JMP START
TASK3:
; ....
JMP START
TASK4:
; ....
JMP START
```

上面这个程序将 TCC 规划为 62.5ms 中断一次(系统震荡选用 32.768KHz)，所以 Task 每 62.5ms 会切换到下一个 Task，也就是说每个 Task 都能够平均分享 CPU 的时间，这就是分时多任务的原理。至于中断程序部分不是必须的，可一情况决定是否要由 TCC 安排时间的管理。状态机(State Machine)是根据目前所在的 State 所产生的条件，来决定下一个状态，所以程序原理和上面这个例子大同小异，所不同的是，我们应该把标示为 TASKn 的 Label 视为一个单独的 State，然后根据某些条件将最后面的 JMP 转移到另外一个 State。在这里时间控制也不一定要用到，视需求决定。例如：

```
TASK1:
MOV A,INPUT
JBS PSW,FZ
JMP TASK2
JMP TASK3
```

#### 说明

如果 INPUT=0 的话，将由目前所在的 TASK1 转移到 TASK3 执行，否则状态转移到 TASK2。

#### 后记

戏法人人会变，只是巧妙各有不同，希望笔者提供的这些小技巧对于喜欢玩单芯片的读者能够有所帮助，我们不仅只是强调硬件应该节省，在软件技巧上也应该多发展一些好的算法，如此才能双管齐下，对症下药。吾人期盼藉此抛砖引玉能激发您更多的创意，写出更精简的程序，也期盼您的指教。

竭诚欢迎所有喜爱 EM78x 系列单芯片的朋友来信和我们一起讨论。