3.2.3 MPASM 的伪指令

我们在第一章中已经详细介绍了中档 PIC 单片机的 35 条指令，源程序的编写主要就是用这些基本的指令实现你的控制任务。但为了增加源程序的可读性和可维护性，我们引入了伪指令的概念。伪指令本身不会产生可执行的汇编指令，但它们可以帮组“管理”你编写的程序，其实用性和必要性绝不亚于 35 条正真的汇编指令。我们在此着重介绍最常用的几种伪指令。

- **#include 或 include**

#include 伪指令的作用是把另外一个文件的内容全部包含复制到本伪指令所在的位置。被包含复制的文件可以是任何形式的文本文件，当然文件中的内容和语法结构必须是 MPASM 能够识别的。最经常被“include”的是针对 PIC 单片机内部特殊功能寄存器定义的包含头文件，在 MPLAB 安装后它们全部放在路径“C:\Program Files\MPLAB IDE\MCHIP_Tools”下，每一个型号的 PIC 单片机都有一个对应的预定义包含头文件，扩展名是“.inc”。除了一些符号预定义文件，你也可以把现有的其它程序文件作为一个代码模块直接“包含”进来作为自己程序的一部分。见例 3-01。

```
#include <p16f877a.inc>     ;把预定义的 PIC16F877A 寄存器符号包含到此处
#include "math.asm"         ;把现有的程序文件包含进来作为自己代码的一部分
```

例 3-01

请注意被包含文件的引用方式。一种是<>尖括号引用，这种引用意味着让编译器去默认的路径下寻找该文件，MPASM 默认的寄存器预定义文件存放路径即为上面提及的 MPLAB 安装后的目录；另一种是""双引号引用，这种引用方式的意思是指示编译器从引号中指定的全程文件路径下寻找该文件。例 3-01 中"math.asm"没有指定路径，即意味着在当前项目路径下寻找 math.asm 文件。如果编译器找不到被包含的文件，将会有错误信息告知。

请在你的源程序中尽量用 MPLAB 标准头文件定义的寄存器符号。一来这些被定义的寄存器符号和芯片数据手册上的描述一一对应，理解起来即直观又容易；二来如果用你自己定义符号就缺乏一个大家能一起交流的标准平台，其他人要解读你的代码时将费时费力。故例 3-01 中的首行#include 包含引用伪指令可以说是 PIC 单片机程序编写时的标准必备。

- **list**

list 伪指令可以设定程序编译时的一些信息，例如所选单片机的型号，编译时选择的缺省数制等。例如：

```
list p=16f877a, r=DEC          ;单片机型号为 PIC16F877A，无特别指明的数字为十进制数
```

例 3-02

如果程序开发时使用项目管理的模式，则所有 list 伪指令可以描述的参数项都可以在项目的设定选项中通过对话框的形式设定并保存。在此只需对 list 伪指令稍作了解即可。

- **__config**

此伪指令的重要作用是把芯片的配置字设定在源程序中，请参阅 2.5 节的详细说明。建议大家尽量用此伪指令把芯片的配置字写在程序中。

- **__idlocs**

PIC 单片机中有一处非常特殊的标记单元。它独立于任何其它存储器，唯一的作用就是作为一个标记。此标记值无法用软件读到，读取和写入的方法只有通过编程器实现。此标记值没有读保护，你可以利用它存放程序的版本或日期等信息。如果需要，则可以用伪指令__idloc 在程序中定义具体的值。

```
__idloc 0x1234                    ;设定芯片的标记值为 0x1234，注意前面有两个下划线符
```

例 3-03

和__config 伪指令定义的配置字一样，用__idloc 定义的芯片标记值在最后也会存放在 HEX 文件中，这就要求编程器能够解析它。

- **errorlevel**

errorlevel 的用途是控制编译信息的输出显示。编译器在编译你的源程序时会提供很多信息，有些信息是你必须要处理的，例如错误信息（Error），只要有错误信息存在，你的程序将永远无法完成编译；有些可能只需要关注，例如警告信息(Warning)；也有一些可能你根本就不感兴趣，它们只是一些提示信息（Message）而已。注意出现警告和提示信息时将不会中止编译器的编译工作，你的程序将被编译并最终产生 HEX 文件。图 3-14 中显示了一个程序编译后的各种信息实例，其中既有错误信息，也有警告和提示信息。我们可以用 errorlevel 伪指令来控制输出信息的级别，或刻意关闭／打开一些提示信息。

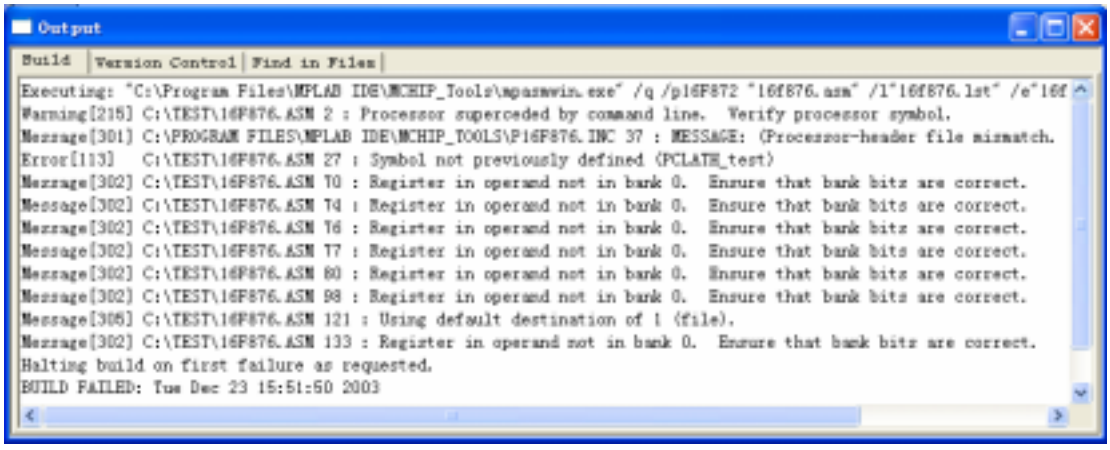


图 3-14

编译信息的输出显示级别有三种，分别是 0、1 和 2。级别 0 代表显示所有信息，包括各种错误、警告和提示信息，如图 3-14 所示；级别 1 代表显示错误和警告信息，忽略提示信息；级别 3 代表只显示错误信息而忽略警告和提示信息。在任何一个大的级别上还可以对某些信息单独设定显示或关闭。每个信息都有一个识别标号，见图 3-14 中信息项“ [] ”中的数字，打开或关闭某类信息只需在 errorlevel 伪指令中引用信息识别标号，并在其前面用“ + ”或“ - ”号，即代表打开或关闭这一类信息，例如：

```
errorlevel 0, -302, -305          ;显示所有信息，但不需要 302 和 305 这两类提示信息
errorlevel 1, +305                ;显示错误和警告信息，但同时还要关注 305 类的提示信息
```

例 3-04

- **#define / #undefine**

#define 的作用是定义常数符号，即用一个符号变量替换另一个符号串或变量。被替换的可以是任意字母数字组成的符号但替换者本身不能是一个纯数字。例如：

```
#define DELAY_TIME  1000        ;定义常数符号，即用 DELAY_TIME 符号代替 1000
#define KEY1        PORTB,7     ;用 KEY1 符号代替端口 PORTB 的第 7 引脚
```

例 3-05

用#define 伪指令定义符号后，可使程序中的变量或指令变得更具实际意义，也使程序变得更易维护。指令“ btfss PORTB,7 ”和“ btfss KEY1 ”在事先用了例 3-05 中的#define 后编译的结果是一样的，但明显地后者看起来更容易理解，一看就知道这是在测试编号为 KEY1 的一个按键。而且如果你的硬件设计改动了 KEY1 所接的单片机引脚，只要改动这一处#define 重新定义引脚位置，程序的其它部分无需任何修改，再编译一次即可得到更新后的软件代码。一个好的编程习惯是事先把一些代表实际意义的变量、单片机的输入输出引脚在硬件电路中的实际功能等用#define 伪指令定义成简单直观的符号名字，然后在程序中直接用其符号名字而不用简单机械的数字形式。替换的工作由编译器在编译时自动完成。它会先扫描你的源程序代码，把事先#define 的符号名改回成被替换的字符串，然后再继续编译生产机器码。

- **equ**

equ 顾名思义是“ 等于 ”的意思，其作用和#define 伪指令有点类似，也是用一个符号名字替换其它数字变量，但它只能替换立即数。如果要替换一个符号名字，则此符号名必须事先用#define 或 equ 伪指令已经定义替换了一个立即数。例如：

```
#define MyCount   0x70          ;定义 MyCount 符号替换立即数 0x70
w_temp equ  0x20                ;符号名 w_temp 等于 0x20
count1 equ  MyCount             ;符号名 count1 等同于 MyCount
                                ;如果 MyCount 没有事先定义则会产生一个错误
```

例 3-06

在绝对定位的编程模式中 equ 被经常用于定义用户自己的变量，即用一个符号名代替一个固定的存储单元地址，上例 3-06 中的 w_temp 定义即属于此类。用 equ 方式定义的符号在汇编后可以生成相关的调试信息，可以通过各种变量观察的方式显示此符号所代表的内存地址处的数据内容，但用#define 方式定义的符号则不能产生调试信息。要注意 equ 伪指令本身并没有限定所定义的一定是一个变量地址，它只是一个简单的符号和数字替换而已，其意义必须和具体的指令结合才能确定，如下例 3-07 中对符号 w_temp 的理解。

```
w_temp equ  0x20                ;符号名 w_temp 等于 0x20

movlw   0x55                    ;W=0x55
movwf   w_temp                  ;把 W 的值送给变量 w_temp，（0x20 单元内容＝0x55）
movf    w_temp, w               ;把 w_temp 单元内容送 W，（W=0x55）
movwf   FSR                     ;把 W 的内容送 FSR，（FSR=0x55）
movlw   w_temp                  ;把 w_temp 所代表的立即数即地址值送给 W，（W=0x20）
movwf   FSR                     ;让 FSR 指针指向 w_temp，（FSR=0x20 而不是 0x55）
```

例 3-07

- **cblock / endc**

用 equ 伪指令可以给一个符号变量分配一个地址。但在一个程序设计过程中往往需要定义很多变量，你当然可以给每一个变量逐个用 equ 的方法分配一个地址空间。但如果变量很多，这样做就显得非常麻烦，你必须自己安排每个变量的地址，小心不能出现地址重叠；若要在已定义分配好的变量间插入新的变量，那就必须重新逐个安排随后变量的地址等等。cblock/endc 伪指令可以轻松解决有很多变量定义的场合出现的这些问题，我们把它叫作变量块连续定义。具体用法如下：

cblock 伪指令声明变量块的起始地址，endc 伪指令声明变量块定义结束，cblock/endc 中间可以插入任意多的变量声明。其地址编排由编译器自动计算：第一个变量地址分配从起始地址开始，然后按所声明变量保留的字节数自动分配后面变量的地址，变量所需保留的字节数用“ : ”加后面的数字表示，如果只有一个字节“ :1 ”可以省略不写。以例 3-08 来说明：

```
                cblock 0x20             ;变量定义起始地址为 0x20
w_temp                                  ;w_temp 地址为 0x20，占一个字节
status_temp                             ;status_temp 地址为 0x21，占一个字节
buffer:8                                ;buffer 的起始地址为 0x22，并保留 8 个字节单元
var1                                    ;var1 的地址为 0x2a，占一个字节
var2                                    ;var2 的地址为 0x2b，占一个字节
                endc                    ;结束变量连续定义
```

例 3-08

用 cblock 方式定义的变量和用 equ 方式定义的变量一样在汇编后可以生成相关的调试信息，可以通过各种变量观察的方式显示此符号所代表的内存地址和其中的数据内容，所以实际编程时一般无需关心计算每个变量的具体地址。程序员要注意的用这种方式连续定义很多变量时不要让变量块跨越所处 bank 的边界。你可以在 cblock 中随意插入新定义的变量，或通过改变起始地址的方式使变量块整个挪到其它内存地址处，地址的更新由编译器代劳。

- **org**

org 用以定义程序代码的起始地址，通过此伪指令你可以把程序定位到任何可用的程序空间，它实现的是程序代码绝对定位，如例 3-09：

```
            org         0x0000          ;定义复位入口地址，以下指令从地址 0x0000 开始
            goto        main            ;

            org         0x0004          ;定义中断入口地址，以下指令从地址 0x0004 开始
            movwf       w_temp          ;保存 w
            ;...                        ;其它中断服务代码

            org         0x0800          ;定义 page1 的起始地址，以下指令代码放在 page1
Sub1        return
```

例 3-09

只要你认为代码需要确定放在某一特定地址处，在程序的任何地方都可以用 org 伪指令重新定义存放的起始地址，且地址顺序可以任意编排。但要注意的是若干个确定起始地址的代码块不能相互重叠，否则编译器会报错，无法得到正确结果。若用可重定位方式开发指令

代码时一般不能用 org 伪指令绝对定位代码。

- **dt**

dt 的作用是定义表格数据。在第一章介绍基本汇编指令时已经提到，PIC 单片机实现表格定义的最基本指令是“ retlw xx ”，表格中的每一个字节数据都以指令“ retlw ”的形式出现。若表格较大，就需要很多“ retlw ”指令，比较麻烦，可读性也差。这时我们可以用此“ dt ”伪指令替代“ retlw ”实现很多数据的表格定义。如例 3-10：

```
Table       addwf       PCL,f             ;PC 相对寻址查表
            dt          0                 ;retlw 0
            dt          1, 2, '3'         ;retlw 1
                                          ;retlw 2
                                          ;retlw 0x33 ('3'的 ASCII 码)
            dt          "ABC"             ;retlw  'A'
                                          ;retlw  'B'
                                          ;retlw  'C'
```

例 3-10

- **de**

de 伪指令可以让你在源程序中定义片内 EEPROM 的初值。毫无疑问，该条伪指令只适用于那些内含 EEPROM 数据存储器的单片机，例如：PIC16F87x、PIC16F62x 等等。在中档 PIC 单片机中，除了 PIC16F7x 系列外，其它 Flash 型的单片机都有片上 EEPROM，只是字节数多少的问题。你可以编写代码在程序运行时来设定片内 EEPROM 数据区的初值，但此 EEPROM 区还可以在芯片编程烧写时通过编程器对其设定初值。对编程器而言 EEPROM 数据区是程序空间的延伸，它有个特别的编程起始地址 0x2100。基于这一前提，我们可以在源程序中利用“ org ”和“ de ”伪指令定义片内 EEPROM 数据的初值，这样最后得到的 HEX 文件被烧入到单片机内后，EEPROM 区就同时被特定数据所初始化。看例 3-11 的实例

```
            org         0x2100            ;特殊的程序空间起始地址
                                          ;编程器能识别此地址作为 EEPROM 数据区的起始地址
            de          0, 1, 2, 3        ;EEPROM 地址单元[0]=0, [1]=1, [2]=2, [3]=3
            de          "ABCD"            ;[4]=0x41, [5]=0x42, [6]=0x43, [7]=0x44
```

例 3-11

按例 3-11 所示的定义，芯片完成编程烧入后，其内部 EEPROM 区从 0x00 单元开始被分别初始化成 0x00、0x01、0x02、0x03、0x41、0x42、0x43、0x44。其它未被初始化的 EEPROM 单元全部是 0xff。

要注意并不是所有的编程工具都能支持此法定义的 EEPROM 初始值烧入。能直接挂接在 MPLAB 环境下的 Microchip 原厂或兼容的编程工具都可以支持“ de ”伪指令定义的 EEPROM 初值烧入，但其它第三方生产的编程工具就不一定，使用前请咨询编程器的生产厂商。

- **fill**

fill 伪指令可以实现对程序空间连续自动填充某一特定的指令数据，被填充的可以是一个立即数（实际肯定代表某一条指令），也可以是一条形象的汇编指令。基本上在一个设计

中都有一些程序空间没有写上具体的指令编码（空白处），在单片机正常运行时这些地方的指令是不会被执行到的。但在有干扰的情况下程序跑飞正好落在这些非法指令处时，就有必要设置软件陷阱捕捉这些非法跳转，让程序恢复正常运行。如果要程序员一个一个地址去分析哪里有空的指令单元然后又用特殊指令一条一条填入，这是根本行不通的。fill 伪指令在这时就派上用场了。

```
    fill     0x0000, 5                ;从当前地址处连续 5 个程序字填成 0x0000(NOP 指令)
    fill     (goto $), NEXT_BLOCK-$   ;从当前地址开始到标号 NEXT_BLOCK 前所有程序空间填上
                                      ;goto $ （死循环）指令
    org      0x0800
NEXT_BLOCK
```

例 3-12

请大家特别注意上例 3-12 中第二行 fill 伪指令的用法。在你自己的程序中也可以用同样的方法把所有未用到的程序空间填上“ goto $ ”这样一条死循环的指令。一旦单片机执行过程中非法跳到这些指令处时指令运行就将被“ 俘获 ”，停在那里直到看门狗复位，然后程序从头开始。这是软件陷阱的最基本处理方法。若填充指令“ goto 0x0000 ”直接跳转到复位地址处可能会有问题，因为 goto 指令执行时必须和 PCLATH 寄存器配合（跨页跳转的问题），若 PCLATH[4:3]不为 00 就不能跳到复位地址 0x0000 处。在程序跑飞非法跳转到设定的陷阱处时你又怎能保证 PCLATH 中的页面设定为正好指向第 0 页？

- **end**

end 伪指令告诉汇编编译器编译工作到此为止，end 后面所有的信息，不管正确与否，一概不管。绝大多数情形下你的程序的最后一行应该是“ end ”。无论如何，end 必须出现在程序中，不然编译器会报错，无法进行编译工作。

3.2.4 MPASM 内的直接运算符

为了使所编的程序理解更直观，维护更方便，MPASM 汇编器允许你在程序的编写过程中直接以数学表达式的形式在指令中实现一些数字运算的功能。千万不要误解成 MPASM 可以替你生成数学运算的指令，那可是其它编译器（例如 C 编译器）才能完成的工作。这里讲的数字运算前提是所有参与运算的操作数全部是明明白白的立即数，如果是符号名字则必须事先用#define 或 equ 伪指令明确定义了的。整个运算过程是由编译器在扫描你的源程序时进行的，运算结果也只能是一个确定的立即数。我们将在这里介绍几种非常有用的运算符。

- **取当前指令的地址值：$**

你可以在写程序时给一条指令前加上一个标号，然后直接引用该标而得到此程序字的地址。如果你的程序经常需要用到指令的当前地址或附近的地址值，这样的标号就需要写很多且不能重复。用“ $ ”运算符让汇编器替你计算当前指令所处的位置将有效地减轻你的这份工作量。见例 3-12 和 3-13。

```
;用语句标号得到指令地址
Here        goto      Here              ;跳转到当前地址，程序进入死循环
Delay       decfsz    count, f          ;计数器减 1 并判 0
            goto      Delay             ;跳转到上一行重复循环
;用$运算符得到指令地址而无需定义任何语句标号
```

```
goto      $                 ;跳转到当前地址，程序进入死循环
decfsz    count, f          ;计数器减 1 并判 0
goto      $-1               ;跳转到（当前地址－1）处，即上一行，重复循环
```

例 3-13

- **取 16 位立即数的高低字节：high 和 low**

一个 16 位的立即数在 8 位单片机中必须被拆解成高 8 位一个字节（高字节）和低 8 位一个字节（低字节）才能用指令一条条处理，类似的处理在对两字节变量赋立即数初值和基于 PC 相对跳转查表前设定 PCLATH 寄存器时经常碰到。MPASM 提供了 high 和 low 两个运算符分别计算一个立即数的高字节和低字节。我们看例 3-14 的代码实例：

```
;两字节变量赋立即数初值
#define   DELAY_TIME    .1000              ;定义一个常数立即数
          movlw         low(DELAY_TIME)    ;取立即数的低字节值，经编译器计算将得到 0xe8
          movwf         count              ;赋给变量的低字节
          movlw         high(DELAY_TIME)   ;取立即数的高字节值，经编译器计算将得到 0x03
          movwf         count+1            ;赋给变量的高字节
;查表前设定 PCLATH 寄存器。关于 PC 相对跳转的概念详见 1.5.2 节
          movlw         high(Table)        ;取查找表入口地址的高字节值
          movwf         PCLATH             ;设定 PCLATH 寄存器
          movf          index,w            ;取查表索引值
          call          Table              ;调用查表子程序
```

例 3-14

- **加减乘除：+ - * /**

实际上前面的很多代码范例中都已经说明了“ + ”、“ - ”运算符的使用方法。“ * ”和“ / ”的运算也类似。看下面例 3-15 计算异步串行通讯波特率常数的方法。

```
;高速异步通信波特率 BPS＝Fosc/(16＊(X+1))
;故，波特率常数 X = Fosc/(BPS*16) – 1
#define   BPS           .9600                ;定义工作波特率
#define   Fosc          .4000000             ;定义单片机工作振荡频率 4MHz
          ;...                               ;其它代码
          movlw         Fosc/(BPS*.16) – 1   ;编译器计算得到.25（10 进制 25）
          movwf         SPBRG                ;设定波特率定时寄存器
```

例 3-15

程序中用了统一的计算公式后，在调试时只要简单地改变前面的#define 语句定义新的波特率或振荡频率值，然后重新编译一次程序即实现了波特率设定代码的更新，非常方便。

- **移位运算：>> 和 <<**

“ >> ”运算符把一个立即数算术右移若干位（高位补 0），“ << ”运算符把一个立即数算术左移若干位（低位补 0）。

```
#define   xxx           0x55

          movlw         xxx>>1               ;W=0x2a
```

```
movlw       xxx<<2              ;W=0x54
movlw       1<<7                ;W=0x80
```

例 3-16

- **立即数逻辑运算： & | ^**

“ & ”运算符把一个立即数和另外一个立即数相“ 与 ”;“ | ”运算符把一个立即数和另外一个立即数相“ 或 ”;“ ^ ”运算符把一个立即数和另外一个立即数相“ 异或 ”。例 3-17 的代码利用异或运算符“ ^ ”实现类似于 C 语言“ switch-case ”功能的汇编代码指令，注意例中的 VAL1、VAL2、VAL3 等判别值都是事先已经定义的立即数而不是 RAM 中的变量。

```
;利用异或运算实现类似于 C 语言的 switch-case 语句
        movf        switchVal, w        ;取分支判断值. switch (W)
        xorlw       VAL1                ;W=W ^ VAL1
        btfsc       STATUS, Z           ;判 0 标志
        goto        Case_VAL1           ;case VAL1: （原始 W=VAL1）
        xorlw       VAL1^VAL2           ;W=(W^VAL1)^(VAL1^VAL2) = W^VAL2
        btfsc       STATUS, Z           ;判 0 标志
        goto        Case_VAL2           ;case VAL2: （原始 W=VAL2）
        xorlw       VAL2^VAL3           ;W=(W^VAL2)^(VAL2^VAL3) = W^VAL3
        btfsc       STATUS, Z           ;判 0 标志
        goto        Case_VAL3           ;case VAL3: （原始 W=VAL3）
        ;...                            ;其它 case 情况判别
```

例 3-17

3.2.5 MPASM 的宏指令

引入宏指令的目的也是为了增强程序的可读性和易维护性。和伪指令不同的是，伪指令所起的只是辅助性的作用，其本身不会直接产生真正的机器码；但宏指令是真正的指令，它实际上是若干条基本汇编指令的集合。为了编程方便，MPASM 已经内含了一些非常好用的宏指令，用户也可以自己编写任意形式的宏指令。

3.2.5.1 MPASM 内含的宏指令

MPASM 内含的宏指令就象扩充了的标准汇编指令一样，其名字已作为 MPLAB 的关键词而被保留。虽然经过编译器编译后最终将变成真正的汇编指令机器码，但某些宏指令的转换过程还是有其独到之处。

- **banksel**

banksel 和下面的 pagesel 宏指令可以说是所有宏指令中最好用最有用的了。banksel 可以帮助你非常方便地实现寄存器 bank 的设定。你只需在 banksel 后给它一个变量名或地址，编译器会自动按照变量地址所在的 bank，自动生成设定 STATUS 寄存器 RP1:RP0 位的指令。更聪明的是，编译器知道你所选的芯片最多有几个 bank，它将用最少的指令完成 bank 设定。例如：

```
;芯片选择 PIC16F874A，RAM 共有 2 个 bank
```

```
        banksel     TRISC               ;设定 TRISC 所在的 bank (TRISC 在 bank1)
        ;编译后的机器码
        bsf         STATUS, RP0         ;只生成 1 条汇编代码

;芯片选择 PIC16F877A，RAM 共有 4 个 bank
        banksel     TRISC               ;设定 TRISC 所在的 bank (TRISC 在 bank1)
        ;编译后的机器码
        bsf         STATUS, RP0         ;生成 2 条汇编代码
        bcf         STATUS, RP1         ;
```

例 3-18

同样的一条“ banksel TRISC ”指令，针对不同的芯片编译器生成的汇编代码可能不同。两个 bank 的芯片只要用到 RP0 一位即可实现 bank 选择，banksel 宏指令会转换成一条汇编指令；四个 bank 的芯片则必须用 RP1:RP0 两位一起实现 bank 选择，故一条 banksel 宏指令将转换成两条汇编指令。用 banksel 的好处是显而易见的，你无需太多关心你准备操作的寄存器落在哪个 bank 内，编译器会知道这个寄存器的实际地址，然后替你生成相关的汇编代码以正确设定 bank 位；需要时你可以随意移动变量的定义地址而无需修改其它代码，只需重新编译一次即可；另外，如果你用代码可重定位方式进行软件开发时，在写指令之时根本就无法知道自己定义的变量最后会落在哪个 bank 中，想自己设定具体的 bank 都不行。此时，只有用 banksel 宏指令让编译器连接器一起在连接定位后再“ 自动填入 ”相关的 bank 位设定指令。

- **bankisel**

和 banksel 类似，不过它对付的是用于寄存器相对寻址的 STATUS 寄存器中的 IRP 位。它也会用最少的代码实现 IRP 位的设定。如果是只有两个 bank 的芯片，用 bankisel 将不产生任何指令！在代码可重定位开发方式下，对可重定位的变量作相对寻址需要设定 IRP 位时，也只能用 bankisel 交由编译器连接器来替你实现。

```
;芯片选择 PIC16F877A，RAM 共有 4 个 bank
            cblock      0x120
buffer:8                                ;从地址 0x120 起定义 8 字节的数据区
            endc

            bankisel    buffer          ;用 bankisel 自动设定 IRP 位
            movlw       low(buffer)     ;取 buffer 的地址（只有低 8 位）
            movwf       FSR             ;送给 FSR
            ;编译后的机器码
            bsf         STATUS, 7       ;真正的设定 IRP 的汇编代码
            movlw       0x20
            movwf       FSR
```

例 3-19

- **pagesel**

pagesel 可以帮助你设定程序的页面。使用方式和 banksel 相似，只是它改变的是 PCLATH[4:3]两位。该宏指令也同样将用最少的代码实现程序页面设定：程序空间不超过 2K

字（只有 1 页）的将不产生任何汇编代码；程序空间不超过 4K 字（最多 2 页）的芯片将只生成一条设定 PCLATH[3]的汇编代码；只有超过 4K 字（最多 4 页）的芯片才会生成两条代码。同样，pagesel 在代码可重定位的开发模式下也是不可或缺的。

```
;芯片选择 PIC16F877A，RAM 共有 4 个页面
            org         0x0100                  ;在第 0 页内
main        pagesel     sub1                    ;用宏指令设定被调用子程序的页面
            call        sub1                    ;随后调用该子程序
            pagesel     $                       ;用宏指令设定当前地址的页面
            goto        main                    ;循环
            org         0x0800                  ;第 1 页起始
sub1        return                              ;子程序返回

            ;编译后的机器码（main 部分）
main        bsf         PCLATH, 3               ;设定 sub1 所在的页面
            bcf         PCLATH, 4
            call        sub1
            bcf         PCLATH, 3               ;设定当前指令所在的页面
            bcf         PCLATH, 4
            goto        main
```

例 3-20

- **clrc/setc**

clrc/setc 针对的是状态寄存器 STATUS 中的进位标志位。

```
clrc    等同于   bcf  STATUS, C    ;C=0
setc    等同于   bsf  STATUS, C    ;C=1
```

- **clrz/setz**

clrz/setz 针对的是状态寄存器 STATUS 中的零标志位。

```
clrz    等同于   bcf  STATUS, Z    ;Z=0
setz    等同于   bsf  STATUS, Z    ;Z=1
```

- **clrdc/setdc**

clrdc/setdc 针对的是状态寄存器 STATUS 中的半字节进位标志位。

```
clrdc   等同于   bcf  STATUS, DC   ;DC=0
setdc   等同于   bsf  STATUS, DC   ;DC=1
```

- **skpc/skpnc**

skpc/skpnc 是判状态寄存器 STATUS 中的进位标志位，若条件满足则程序跳过下一条指令。

```
skpc   等同于 btfss  STATUS, C     ;若 C=1 则程序跳过下一条指令
skpnc  等同于 btfsc  STATUS, C     ;若 C=0 则程序跳过下一条指令
```

- **skpz/skpnz**

skpz/skpnz 是判状态寄存器 STATUS 中的零标志位，若条件满足则程序跳过下一条指令。

```
skpz   等同于 btfss  STATUS, Z     ;若 Z=1 则程序跳过下一条指令
skpnz  等同于 btfsc  STATUS, Z     ;若 Z=0 则程序跳过下一条指令
```

- **skpdc/skpndc**

skpdc/skpndc 是判状态寄存器 STATUS 中的半字节进位标志位，若条件满足则程序跳过下一条指令。

```
skpdc   等同于 btfss  STATUS, DC     ;若 DC=1 则程序跳过下一条指令
skpndc  等同于 btfsc  STATUS, DC     ;若 DC=0 则程序跳过下一条指令
```

- **bc/bnc**

bc/bnc 宏指令的作用有点象 51 单片机的“ jc/jnc ”指令。它判别状态寄存器 STATUS 中的进位标志位，按进位标志实现程序的分支跳转。如例 3-21。

```
            movlw       0x31            ;W=0x31
            addwf       sum,f           ;sum = sum+W
            bc          Carry1          ;如果发生进位就跳转到 Carry1 处执行
            nop                         ;如果没有进位则继续执行 bc 的下一条指令
            ;...
Carry1      nop

            bc          XXX             ;如果 C=1 就跳转到标号 XXX，否则程序执行 bc 的下一条
            等同于
            btfsc       STATUS,C
            goto        XXX

            bnc         YYY             ;如果 C=0 就跳转到标号 YYY，否则程序执行 bnc 的下一条
            等同于
            btfss       STATUS,C
            goto        YYY
```

例 3-21

请不要被 bc/bnc 这样“一条”指令所迷惑，它实际上是由两条汇编指令组成，且用到了“ goto ”实现跳转，故在用此宏指令前注意页面的设定。

- **bz/bnz**

同 bc/bnc 一样，只不过判别的是状态寄存器 STATUS 中的零标志位。

```
            movlw       0x55            ;W=0x55
            xorwf       flag,w          ;flag = 0x55 ?
            bz          Match           ;Z=1, flag=0x55, 跳转到 Match 处执行
            nop                         ;Z=0,继续执行 bz 的下一条指令
            ;...
Match       nop
```

```
bz          XXX             ;如果 Z=1 就跳转到标号 XXX，否则程序执行 bz 的下一条
```

等同于

```
btfsc       STATUS,Z
goto        XXX
```

```
bnz         YYY             ;如果 Z=0 就跳转到标号 YYY，否则程序执行 bnz 的下一条
```

等同于

```
btfss       STATUS,Z
goto        YYY
```

例 3-22

- **bdc/bndc**

同上，判别的是状态寄存器 STATUS 中的半字节进位标志位。

```
bdc         XXX             ;如果 DC=1 就跳转到标号 XXX，否则程序执行 bdc 的下一条
```

等同于

```
btfsc       STATUS,DC
goto        XXX
```

```
bndc        YYY             ;如果 DC=0 就跳转到标号 YYY，否则程序执行 bndc 的下一条
```

等同于

```
btfss       STATUS,DC
goto        YYY
```

例 3-23

3.2.5.2 用户自定义宏

除了 MPASM 内带的宏指令外，按实际开发的需要和个人的习惯，程序员可以自己定义任意形式的宏指令。大量使用定义合理的宏指令可以使程序的可读性大大提高，也更容易移植。

自己定义宏指令时须遵循一些语法规则。宏指令的定义由“ 宏指令名“ 开始，后跟关键词“ macro ”，其后可以带若干宏参数，也可以不跟任何宏参数；然后从下一行起开始写基本的汇编指令或已被认可的其它宏指令（宏嵌套）;指令可以是任意多行，最后以关键词“ endm ”结束整个宏定义，例如：

```
;定义宏指令实现一个字（两字节数）加 1 的功能
IncWord     macro       wordVal         ;IncWord 是宏指令名， wordVal 是宏参数
                                        ;下面为宏的实体（实际的汇编指令）
            incf        wordVal,f       ；对字的低字节加 1，如果结果为 0 则为字节计数溢出
            skpnz                       ;如果没有溢出（上面指令结果不为 0）就跳过下一条指令
            incf        wordVal+1,f     ;若低字节加 1 后溢出，则字的高字节加 1
            endm                        ;结束宏指令定义
```

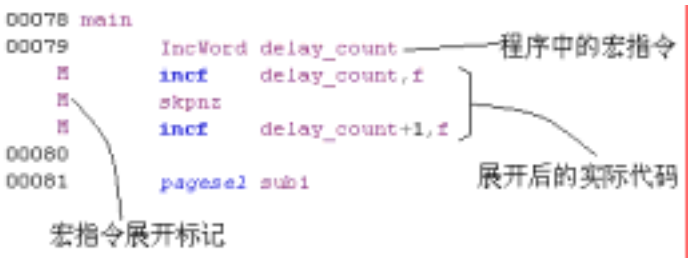
```
;程序中对宏指令的引用
            cblock      0x20
counter:2                               ;定义一个两字节的字变量
            endc

Loop
            IncWord     counter         ;用宏指令实现变量 counter 每次循环加 1
                                        ;编译器会把这“ 一条 ”宏指令展开成原定义的 3 条汇编指令
                                        ;并用实际的 counter 符号替换宏参数 wordVal
            goto        Loop            ;跳转重复循环。注意一条宏指令展开后可能是多条汇编指令
                                        ;故此处用“ $-? ”时要特别小心。
```

例 3-24

使用宏指令时几个问题需要注意。

㈠ 宏指令不同于子程序调用指令。编译器在编译你的代码时会用原宏定义中的若干条汇编指令代替程序中的“ 一条 ”宏指令插入到此宏指令位置处（图 3-15）。若程序中有很多地方用了同样的宏指令，那么相同的汇编指令集也会被复制成同样多份，它不能节省代码长度。而子程序调用只有一条指令，若一个子程序在程序中被多处调用，增加的只是调用指令“ call ”而子程序只有一个，它可以减少代码长度。宏指令最有用的是集成少量且非常相关的代码实现一个特定任务，例如 3-24 中的字变量加 1 这样的功能。你可以安自己的习惯和项目的需要设计这样的宏指令，甚至可以建一个宏指令库头文件，以后程序开发时直接用#include 包含进你的程序即可使用。



源程序编译后的 lst 列表文件（局部）

图 3-15

㈡ 虽然宏指令定义中允许使用语句标号以便给“ goto ”指令引用，但最好不要这样做。因为若此宏指令将被多处使用的话，相同的宏定义会被重复复制，其中的语句标号也会一样复制，这就使得程序有“ 标号重复定义 ”的语法错误而无法成功编译。所以在宏指令中需要用 goto 指令跳转时尽量使用“ $ ”配合“ + ”、“ - ”运算，例如：“ goto $-3 ”、“ goto $+2 ”等等；或者使用宏参数给 goto 指令一个特定的语句标号。例如 3-25 的宏定义：

```
;定义宏指令实现寄存器和立即数比较大小
;若寄存器值 >=立即数 则程序跳到某一位置
FL_JGE      macro       fileReg, litVal, jumpTo
                        ;fileReg 为寄存器， litVal 为立即数， jumpTo 为跳转的语句标号
            movlw       litVal    &   ；把立即数送给 W （确保 0x00~0xff）
                        0xff
            subwf       fileReg, w    ;计算 寄存器－W
            skpnc                     ;若 C=0 即， 寄存器值<W，程序跳过下一行继续运行
            goto        jumpTo        ;若 寄存器值>=W，则跳到指定标号处继续运行
```

```
            endm                    ;结束宏指令定义

;程序中对宏指令的引用
val1        equ         0x20        ;定义一个寄存器变量
            ;使用宏指令
            FL_JGE      val1, .100, Val1_Over
                        ;变量 val1 和立即数 100 比较，
                        ;如果 val1>=100 则程序跳到 Val1_Over 处运行
            nop         ;若 val1<.100，则程序执行这条语句
            ;...        ;其它代码
Val1_Over   nop         ;若 val1>=.100，则程序将跳到这里继续运行
```

例 3-25

㈢ 程序仿真调试时对用户自定义的宏指令和 MPASM 自己提供的内部宏指令支持度不尽相同。在用户自定义的宏指令处无法设置断点，但 MPASM 自己提供的内部宏指令没有此限制；在程序单步运行时用户自定义的宏指令无法“ 一步 ”执行完毕，调试器会单步跟踪进入宏定义体一步一步执行，MPASM 内部宏指令可以“ 一步 ”执行完毕。