Introduction to PIC Programming

Baseline Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 6: Assembler Directives and Macros

As the programs presented in these tutorials are becoming longer, it's appropriate to take a look at some of the facilities that MPASM (the Microchip PIC assembler) provides to simplify the process of writing and maintaining code.

This lesson covers:

- Arithmetic and bitwise operators
- Text substitution with `#define`
- Defining constants with `equ` or `constant`
- Conditional assembly using `if` / `else` / `endif`, `ifdef` and `ifndef`
- Outputting warning and error messages
- Assembler macros

Each of these topics is illustrated by making use of it in code from previous tutorials in this series.

Arithmetic Operators

MPASM supports the following arithmetic operators:

negate	`-`
multiply	`*`
divide	`/`
modulus	`%`
add	`+`
subtract	`-`

Precedence is in the traditional order, as above.

For example, 2 + 3 * 4 = 2 + 12 = 14.

To change the order of precedence, use parentheses: `(` and `)`.

For example, (2 + 3) * 4 = 5 * 4 = 20.

*Note: These calculations take place during the assembly process, before any code is generated. They are used to calculate constant values which will be included in the code to be assembled. They **do not** generate any PIC instructions.*

These arithmetic operators are useful in showing how a value has been derived, making it easier to understand the code and to make changes.

For example, consider this code from lesson 2:

```
        ; delay 500ms
        movlw   .244            ; outer loop: 244 x (1023 + 1023 + 3) + 2
        movwf   dc2             ;   = 499,958 cycles
        clrf    dc1             ; inner loop: 256 x 4 - 1
dly1    nop                     ; inner loop 1 = 1023 cycles
        decfsz  dc1,f
        goto    dly1
dly2    nop                     ; inner loop 2 = 1023 cycles
        decfsz  dc1,f
        goto    dly2
        decfsz  dc2,f
        goto    dly1
```

Where does the value of 244 come from? It is the number of outer loop iterations needed to make 500 ms.

To make this clearer, we could change the comments to:

```
        ; delay 500ms
        movlw   .244            ; outer loop: #iterations =
        movwf   dc2             ;   500ms/(1023+1023+3)us/loop = 244
```

Or, instead of writing the constant '244' directly, write it as an expression:

```
        ; delay 500ms
        movlw   .500000/(.1023+.1023+.3) ; number of outer loop iterations
        movwf   dc2                      ; for 500ms delay
```

If you're using mainly decimal values in your expressions, as here, you may wish to change the default radix to decimal, to avoid having to add a '.' before each decimal value. As discussed in lesson 2, that's not necessarily a good idea; if your code assumes that some particular default radix has been set, you need to be very careful if you copy that code into another program, which may have a different default radix. But, if you're prepared to take the risk, add the '`radix`' directive near the start of the program. For example:

```
        radix   dec
```

The valid radix values are 'hex' for hexadecimal (base 16), 'dec' for decimal (base 10) and 'oct' for octal (base 8). The default radix is hex.

With the default radix set to decimal, this code fragment can be written as:

```
        ; delay 500ms
        movlw   500000/(1023+1023+3) ; # outer loop iterations for 500ms
        movwf   dc2
```

Defining Constants

Programs often contain values which may need to be tuned or changed later, particularly during development. When a change needs to be made, finding these values in the code can difficult. And making changes can be error-prone if the same value (or another value derived from the value being changed) occurs more than once in the code.

To make the code more maintainable, each constant value should be defined only once, near the start of the program, where it is easy to find and change.

A good example is the reaction timer developed in lesson 5, where "success" was defined as pressing a pushbutton less than 200 ms after a LED was lit. But what if, during testing, we found that 200 ms is unrealistically short? Or too long?

To change this maximum reaction time, you'd need to find and then modify this fragment of code:

```
        ; check elapsed time
btn_dn  movlw   .25             ; if time < 200ms (25x8ms)
        subwf   cnt8ms,w
        btfss   STATUS,C
        bsf     GPIO,1          ;   turn on success LED
```

To make this easier to maintain, we could define the maximum reaction time as a constant, at the start of the program.

This can be done using the '`equ`' (short for "equate") directive, as follows:

```
MAXRT   equ     .200            ; Maximum reaction time in ms
```

Alternatively, you could use the '`constant`' directive:

```
    constant MAXRT=.200         ; Maximum reaction time in ms
```

The two directives are equivalent. Which you choose to use is simply a matter of style.

'`equ`' is more commonly found in assemblers, and perhaps because it is more familiar, most people use it.

Personally, I prefer to use '`constant`', mainly because I like to think of any symbol placed on the left hand edge (column 1) of the assembler source as being a label for a program or data register address, and I prefer to differentiate between address labels and constants to be used in expressions. But it's purely your choice.

However you define this constant, it can be referred to later in your code, for example:

```
        ; check elapsed time
btn_dn  movlw   MAXRT/8         ; if time < max reaction time (8ms/count)
        subwf   cnt8ms,w
        btfss   STATUS,C
        bsf     GPIO,1          ;   turn on success LED
```

Note how constants can be usefully included in arithmetic expressions. In this way, the constant can be defined simply in terms of real-world quantities (e.g. ms), making it readily apparent how to change it to a new value (e.g. 300 ms), while arithmetic expressions are used to convert that into a quantity that matches the program's logic. And if that logic changes later (say, counting by 16 ms instead of 8 ms increments), then only the arithmetic expression needs to change; the constant can remain defined in the same way.

Text Substitution

As discussed above, the ability to define numeric constants is very useful. It is also very useful to be able to define "text constants", where a text *string* is substituted into the assembler source code.

Text substitution is commonly used to refer to I/O pins by a descriptive label. This makes your code more readable, and easier to update if pin assignments change later.

Why would pin assignments change? Whether you design your own printed circuit boards, or layout your circuit on prototyping board, swapping pins around can often simplify the physical circuit layout. That's one of the great advantages of designing with microcontrollers; as you layout your design, you can go back and modify the code to simplify that layout, perhaps repeating that process a number of times.

For example, consider again the reaction timer from lesson 5. The I/O pins were assigned as follows:

```
;   Pin assignments:                                                *
;       GP1 - success LED                                           *
;       GP2 - start LED                                             *
;       GP3 - pushbutton                                            *
```

These assignments are completely arbitrary; the LEDs could be on any pin other than GP3 (which is input only), while the pushbutton could be on any unused pin.

One way of defining these pins would be to use numeric constants:

```
    constant nSTART=2                ; Start LED
    constant nSUCCESS=1              ; Success LED
    constant nBUTTON=3               ; pushbutton
```

(The 'n' prefix used here indicates that these are numeric constants; this is simply a convention, and you can choose whatever naming style works for you.)

They would then be referenced in the code, as follows:

```
        bsf     GPIO,nSTART          ; turn on start LED

w_tmr0  btfss   GPIO,nBUTTON         ; check for button press (low)

        bsf     GPIO,nSUCCESS        ;   turn on success LED
```

A significant problem with this approach is that larger PICs (i.e. most of them!) have more than one port. Instead of GPIO, larger PICs have ports named PORTA, PORTB, PORTC and so on. What if you moved an input or output from PORTA to PORTC? The above approach, using numeric constants, wouldn't work, because you'd have to go through your code and change all the PORTA references to PORTC.

This problem can be solved using text substitution, using the '#define' directive, as follows:

```
    #define START       GPIO,2       ; Start LED
    #define SUCCESS     GPIO,1       ; Success LED
    #define BUTTON      GPIO,3       ; pushbutton
```

These definitions are then referenced later in the code, as shown:

```
        bsf     START                ; turn on start LED

w_tmr0  btfss   BUTTON               ; check for button press (low)

        bsf     SUCCESS              ;   turn on success LED
```

Note that there are no longer any references to GPIO in the main body of the code. If you later move this code to a PIC with more ports, you only need to update the definitions at the start. Of course, you also need to modify the corresponding port initialisation code, such as 'tris' instructions, normally located at the start of the program.

Bitwise Operators

We've seen that operations on binary values are fundamental to PIC microcontrollers: setting and clearing individual bits, flipping bits, testing the status of bits and rotating the bits in registers. Many configuration options are specified as a collection of bits (or flags) which are assembled into a byte or word; for example, the '__CONFIG' directive and the 'option' instruction.

To facilitate operations on bits, MPASM provides the following bitwise operators:

compliment	~
left shift	<<
right shift	>>
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|

Precedence is in the order listed above.

As for arithmetic operators, parentheses are used to change the order of precedence: '(' and ')'.

We've seen an example of the bitwise AND operator being used in every program so far:

```
                ; int reset, no code protect, no watchdog, 4MHz int clock
    __CONFIG    _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC
```

These symbols are defined in the 'p12F509.inc' include file as follows:

```
;==========================================================================
;
;       Configuration Bits
;
;==========================================================================

_MCLRE_ON                    EQU     H'0FFF'
_MCLRE_OFF                   EQU     H'0FEF'
_CP_ON                       EQU     H'0FF7'
_CP_OFF                      EQU     H'0FFF'
_WDT_ON                      EQU     H'0FFF'
_WDT_OFF                     EQU     H'0FFB'
_LP_OSC                      EQU     H'0FFC'
_XT_OSC                      EQU     H'0FFD'
_IntRC_OSC                   EQU     H'0FFE'
_ExtRC_OSC                   EQU     H'0FFF'
```

The '`equ`' directive is described above; you can see that these are simply symbols for numeric constants.

In binary, the values in the '`__CONFIG`' directive above are:

```
      _MCLRE_OFF         H'0FEF'  =  1111 1110 1111
      _CP_OFF            H'0FFF'  =  1111 1111 1111
      _WDT_OFF           H'0FFB'  =  1111 1111 1011
      _IntRC_OSC         H'0FFE'  =  1111 1111 1110
                                     --------------
```

ANDing these together gives: `1111 1110 1010`

So the directive above is equivalent to:

```
    __CONFIG    b'111111101010'
```

For each of these configuration bit symbols, where a bit in the definition is '0', it has the effect of setting the corresponding bit in the configuration word to '0', because either '0' or '1' ANDed with '0' equals '0'.

The 12-bit configuration word in the PIC12F508/509 is as shown:

Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
-	-	-	-	-	-	-	MCLRE	$\overline{\text{CP}}$	WDTE	FOSC1	FOSC0

These configuration options were described briefly in lesson 1. Recapping:

MCLRE enables the external processor reset, or "master clear", on pin 4. Clearing it allows GP3 to be used as an input.

$\overline{\text{CP}}$ disables code protection. Clearing $\overline{\text{CP}}$ protects your code from being read by PIC programmers.

WDTE enables the watchdog timer, which is used to reset the processor if it crashes. For more details, see lesson 7. Clearing WDTE to disables the watchdog timer.

The FOSC bits set the clock, or oscillator, configuration; FOSC<1:0> = 10 specifies the internal RC oscillator. The other oscillator configurations are described in lesson 7.

Given this, to configure the PIC12F509 for internal reset (GP3 as an input), no code protection, no watchdog timer and the internal RC oscillator, the lower five bits of the configuration word must be set to 01010.

That's the same pattern of bits produced by the `__CONFIG` directive, above (the value of the upper seven bits is irrelevant, as they are not used), showing that ANDing together the symbols in the Microchip-provided include file gives the correct result. Using the symbols is simpler, and safer; it's easy to mistype a long binary value, leading to a difficult-to-debug processor configuration error. If you mistype a symbol, the assembler will tell you, making it easy to correct the mistake.

It is also useful (clearer and less error-prone) to be able to use symbols instead of binary numbers when setting bits in special-function registers, such as OPTION.

The bits in the OPTION register are defined in the 'p12F509.inc' include file as follows:

```
NOT_GPWU                     EQU     H'0007'
NOT_GPPU                     EQU     H'0006'
T0CS                         EQU     H'0005'
T0SE                         EQU     H'0004'
PSA                          EQU     H'0003'
PS2                          EQU     H'0002'
PS1                          EQU     H'0001'
PS0                          EQU     H'0000'
```

Unlike the definitions used for the configuration bits, these symbols define a bit *position*, not a pattern. It tells us, for example, that T0CS is bit 5. To set only bit 5, normally we'd use something like:

```
    movlw   b'00100000'         ; select counter mode: TOCS=1
    option                      ; (set bit 5 in OPTION)
```

But note that the binary constant `b'00100000'` is a '1' shifted right five times, and so can be represented by the expression "`1<<5`", using the binary right-shift operator '<<'.

That means that the following code can be used instead of the above:

```
    movlw   1<<T0CS             ; select counter mode: TOCS=1
    option
```

This works because the symbol '`T0CS`' is defined in the include file to be equal to '5'.

This code is clearer, needing fewer comments, and it is harder to make a mistake, as mistyping a symbol is likely to be picked up by the assembler, while mistyping a binary constant (say getting the '1' in the wrong position) is likely to be missed.

Typically a number of bits need to be set at the same time. To do this, simply bitwise-OR the expressions together. For example, the crystal-based LED flasher code from lesson 5 included:

```
    movlw   b'11110110'     ; configure Timer0:
            ; --1-----          counter mode (T0CS = 1)
            ; ----0---          prescaler assigned to Timer0 (PSA = 0)
            ; -----110          prescale = 128 (PS = 110)
    option                  ;   -> increment at 256 Hz with 32.768 kHz input
```

This can be replaced by:

```
        ; configure Timer0
        movlw   1<<T0CS|0<<PSA|b'110'
                                  ; counter mode (T0CS = 1)
                                  ; prescaler assigned to Timer0 (PSA = 0)
                                  ; prescale = 128 (PS = 110)
        option                    ; -> increment at 256 Hz with 32.768 kHz input
```

Including '`0<<PSA`' in the expression does nothing, since a zero right-shifted any number of times is still zero, and ORing zero into any expression has no effect. But it makes it explicit that we are clearing PSA.

In this application, we don't care what the $\overline{\text{GPWU}}$, $\overline{\text{GPPU}}$ and T0SE bits are set to, so they are not included in the expression. And where a bit field (such as PS<2:0>) is most clearly expressed as a binary pattern, it can be ORed into the expression as a binary constant, as shown.

Macros

We've seen in lesson 3 that, if we wish to reuse the same piece of code a number of times in a program, it often makes sense to place that code into a subroutine and to call the subroutine from the main program.

But that's not always appropriate, or even possible. The subroutine call and return is an overhead that takes some time; only four instruction cycles, but in timing-critical pieces of code, it may not be justifiable. A more significant problem is that the baseline PICs have only two stack registers, meaning that you must be very careful when nesting subroutine calls, or else the stack will overflow and your subroutine won't return to the right place. It's usually not worth using up a stack level, just to avoid repeating a short piece of code.

Another problem with subroutines is that, as we saw in lesson 3, to pass parameters to them, you need to load the parameters into registers – an overhead that leads to longer code, perhaps negating the space-saving advantage of using a subroutine, for small pieces of code. And loading parameters into registers, before calling a subroutine, isn't very readable. It would be nicer to be able to simply list the parameters on a single line, as part of the subroutine call.

Macros address these problems, and are often appropriate where a subroutine is not. A *macro* is a sequence of instructions that is inserted (or *expanded*) into the source code by the assembler, prior to assembly.

Note: The purpose of a macro is to make the source code more compact; unlike a subroutine, it ***does not*** *make the resultant object code any smaller. The instructions within a macro are expanded into the source code, every time the macro is called.*

Here's a simple example. Lesson 3 introduced a '`delay10`' subroutine, which took as a parameter in W a number of multiples of 10ms to delay. So to delay for 200 ms, we had:

```
        movlw   .20             ; delay 20 x 10ms = 200ms
        call    delay10
```

This was used in a program which flashed a LED with a 20% duty cycle: on for 200 ms, then off for 800 ms. Rewritten a little from the code presented in lesson 3, the main loop looks like this:

```
loop
        bsf     FLASH           ; turn on LED
        movlw   .20             ; stay on for 0.2s:
        pagesel delay10
        call    delay10         ;   delay 20 x 10ms = 200ms
        bcf     FLASH           ; turn off LED
        movlw   .80             ; stay off for 0.8s:
        call    delay10         ;   delay 80 x 10ms = 800ms
        pagesel $
        goto    loop            ; repeat forever
```

It would be nice to be able to simply write something like '`DelayMS 200`' for a 200 ms delay. We can do that by defining a macro, as follows:

```
DelayMS MACRO   ms                  ; delay time in ms
        movlw   ms/.10              ; divide by 10 to pass to delay10 routine
        pagesel delay10
        call    delay10
        pagesel $
        ENDM
```

This defines a macro called ‘DelayMS’, which takes a single parameter: ‘ms’, the delay time in milliseconds. Parameters are referred to within the macro in the same way as any other symbol, and can be used in expressions, as shown.

A macro definition consists of a label (the macro’s name), the ‘MACRO’ directive, and a comma-separated list of symbols, or *arguments*, used to pass parameters to the macro, all on one line. It is followed by a sequence of instructions and/or assembler directives, finishing with the ‘ENDM’ directive.

When the source code is assembled, the macro’s instruction sequence is inserted into the code, with the arguments replaced by the parameters that were passed to the macro.

That may sound complex, but using a macro is easy. Having defined the ‘DelayMS’ macro, as above, it can be called from the main loop, as follows:

```
loop
        bsf     FLASH               ; turn on LED
        DelayMS .200                ; stay on for 200ms
        bcf     FLASH               ; turn off LED
        DelayMS .800                ; stay off for 800ms
        goto    loop                ; repeat forever
```

We could describe this ‘DelayMS’ macro as a *wrapper*, making the ‘delay10’ subroutine easier to use.

Note that the pagesel directives have been included as part of the macro, first to select the correct page for the ‘delay10’ subroutine, and then to select the current page again after the subroutine call. That makes the macro transparent to use; there is no need for pagesel directives before or after calling it.

As a more complex example, consider the debounce code presented in lesson 5:

```
wait_dn clrf    TMR0            ; reset timer
chk_dn  btfsc   GPIO,3          ; check for button press (GP3 low)
        goto    wait_dn         ;   continue to reset timer until button down
        movf    TMR0,w          ; has 10ms debounce time elapsed?
        xorlw   .157            ;   (157=10ms/64us)
        btfss   STATUS,Z        ; if not, continue checking button
        goto    chk_dn
```

If you had a number of buttons to debounce in your application, you would want to use code very similar to this, multiple times. But since there is no way of passing a reference to the pin to debounce (e.g. ‘GPIO,3’) as a parameter to a subroutine, it’s necessary to use a macro to achieve this.

For example, a debounce macro could be defined as follows:

```
; Debounce switch on given input port,pin
; Waits for switch to be 'high' continuously for 10ms
;
; Uses:     TMR0        Assumes: TMR0 running at 256us/tick
;
DbnceHi MACRO  port,pin
    local      start,wait,DEBOUNCE
    variable   DEBOUNCE=.10*.1000/.256  ; debounce count = 10ms/(256us/tick)

        pagesel $               ; select current page for gotos
start   clrf   TMR0            ; button down so reset timer (counts "up" time)
wait    btfss  port,pin        ; wait for switch to go high (=1)
        goto   start
        movf   TMR0,w          ; has switch has been up continuously for
        xorlw  DEBOUNCE        ;   debounce time?
        btfss  STATUS,Z        ; if not, keep checking that it is still up
        goto   wait
        ENDM
```

There are a few things to note about this macro definition, starting with the comments. As with subroutines, you'll eventually build up a library of useful macros, which you might keep together in an include file, such as 'stdmacros.inc' (which you would reference using the `#include` directive, instead of copying the macros into your code.) When documenting a macro, it's important to note any resources (such as timers) used by the macro, and any initialisation that has to have been done before the macro is called.

The macro is called '`DbnceHi`' instead of '`DbnceUp`' because it's waiting for a pin to be consistently high. For some switches, that will correspond to "up", but not in every case. Using terms such as "high" instead of "up" is more general, and thus more reusable.

The '`local`' directive declares symbols (address labels and variables) which are only used within the macro. If you call a macro more than once, you must declare any address labels within the macro as "local", or else the assembler will complain that you have used the same label more than once. Declaring macro labels as local also means that you don't need to worry about whether those labels are used within the main body of code. A good example is '`start`' in the definition above. There is a good chance that there will be a '`start`' label in the main program, but that doesn't matter, as the *scope* of a label declared to be "local" is limited to the macro it is defined in.

The '`variable`' directive is very similar to the '`constant`' directive, introduced earlier. The only difference is that the symbol it defines can be updated later. Unlike a constant, the value of a variable can be changed after it has been defined. Other than that, they can be used interchangeably.

In this case, the symbol '`DEBOUNCE`' is being defined as a variable, but is used as a constant. It is never updated, being used to make it easy to change the debounce period from 10 ms if required, without having to find the relevant instruction within the body of the macro (and note the way that an arithmetic expression has been used, to make it easy to see how to set the debounce to some other number of milliseconds).

So why define '`DEBOUNCE`' as a variable, instead of a constant? If it was defined as a constant, there would potentially be a conflict if there was another constant called '`DEBOUNCE`' defined somewhere else in the program. But surely declaring it to be "local" would avoid that problem? Unfortunately, the '`local`' directive only applies to labels and variables, not constants. And that's why '`DEBOUNCE`' is declared as a "local variable". Its scope is limited to the macro and will not affect anything outside it. You can't do that with constants.

Finally, note that the macro begins with a '`pagesel $`' directive. That is placed there because we cannot assume that the page selection bits are set to the current page when the macro is called. If the current page was not selected, the '`goto`' commands within the macro body would fail; they would jump to a different page. That illustrates another difference between macros and subroutines: when a subroutine is called, the page the subroutine is on must have been selected (or else it couldn't have been called successfully), so any '`goto`' commands within the subroutine will work. You can't safely make that assumption for macros.

Complete program

The following program demonstrates how this "debounce" macro is used in practice.

It is based on the "toggle an LED" program included in lesson 5, but the press of the pushbutton is not debounced, only the release. It is not normally necessary to debounce both actions – although you may have to think about it a little to see why!

Using the macro doesn't make the code any shorter, but the main loop is much simpler:

```
;************************************************************************
;   Description:    Lesson 6, example 5                                 *
;                   Toggles LED when button is pressed                  *
;                                                                       *
;   Demonstrates use of macro defining Timer0-based debounce routine    *
;************************************************************************

    list        p=12F509
    #include    <p12F509.inc>
```

```
                ; int reset, no code protect, no watchdog, 4MHz int clock
    __CONFIG    _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC

; pin assignments
    constant    nLED=1              ; indicator LED on GP1
    #define     BUTTON  GPIO,3      ; pushbutton on GP3

;***** MACROS

; Debounce switch on given input port,pin
; Waits for switch to be 'high' continuously for 10ms
;
; Uses:     TMR0        Assumes: TMR0 running at 256us/tick
;
DbnceHi MACRO   port,pin
    local       start,wait,DEBOUNCE
    variable    DEBOUNCE=.10*.1000/.256    ; debounce count = 10ms/(256us/tick)

        pagesel $               ; select current page for gotos
start   clrf    TMR0            ; button down so reset timer (counts "up" time)
wait    btfss   port,pin        ; wait for switch to go high (=1)
        goto    start
        movf    TMR0,w          ; has switch has been up continuously for
        xorlw   DEBOUNCE        ;   debounce time?
        btfss   STATUS,Z        ; if not, keep checking that it is still up
        goto    wait
        ENDM

;***** VARIABLE DEFINITIONS
        UDATA_SHR
sGPIO   res 1                   ; shadow copy of GPIO

;**********************************************************************
RESET   CODE    0x000           ; effective reset vector
        movwf   OSCCAL          ; update OSCCAL with factory cal value


;***** MAIN PROGRAM

;***** Initialisation
start
        movlw   ~(1<<nLED)      ; configure LED pin (only) as output
        tris    GPIO
        movlw   b'11010111'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----111          prescale = 256 (PS = 111)
        option                  ;   -> increment every 256 us

        clrf    GPIO            ; start with LED off
        clrf    sGPIO           ;   update shadow

;***** Main loop
loop
wait_dn btfsc   BUTTON         ; wait for button press (=0)
        goto    wait_dn

        movf    sGPIO,w         ; toggle LED
        xorlw   1<<nLED         ;   using shadow register
        movwf   sGPIO
        movwf   GPIO            ;   write to port
        DbnceHi BUTTON          ; wait until button released

        goto    loop            ; repeat forever

        END
```

Conditional Assembly

We've seen how the processor include files, such as 'p12F509.inc', define a number of symbols that allow you to refer to registers and flags by name, instead of numeric value.

While looking at the 'p12F509.inc' file, you may have noticed these lines:

```
    IFNDEF __12F509
        MESSG "Processor-header file mismatch.  Verify selected processor."
    ENDIF
```

This is an example of *conditional assembly*, where the actions performed by the assembler (outputting messages and generating code) depend on whether specific conditions are met.

When the processor type is specified by the '`list p=`' directive, or selected in MPLAB, a symbol specifying the processor is defined; for the PIC12F509, the symbol is '`__12F509`'. This is useful because the assembler can be made to perform different actions depending on which processor symbol has been defined.

In this case, the idea is to check that the correct processor include file is being used. If you include the include file for the wrong processor, you'll almost certainly have problems. This code checks for that.

The '`IFNDEF`' directive instructs the assembler to assemble the following block of code if the specified symbol *has not* been defined.

The '`ENDIF`' directive marks the end of the block of conditionally-assembled code.

In this case, everything between '`IFNDEF`' and '`ENDIF`' is assembled if the symbol '`__12F509`' has not been defined. And that will only be true if a processor other than the PIC12F509 has been selected.

The '`MESSG`' directive tells the assembler to print the specified message in the MPLAB output window. This message is only informational; it's useful for providing information about the assembly process or for issuing warnings that do not necessarily mean that assembly has to stop.

So, this code tests that the correct processor has been selected and, if not, warns the user about the mismatch.

Similar to '`IFNDEF`', there is also an '`IFDEF`' directive which instructs the assembler to assemble a block of code if the specified symbol *has* been defined.

A common use of '`IFDEF`' is when debugging, perhaps to disable parts of the program while it is being debugged. Or you might want to use a different processor configuration, say with code protection and the watchdog timer enabled. For example:

```
;***** Configuration
    #define DEBUG

    ifdef DEBUG
                    ; int reset, no code protect, no watchdog, 4MHz int clock
        __CONFIG    _MCLRE_OFF & _CP_OFF & _WDT_OFF & _IntRC_OSC
    else
                    ; int reset, code protect on, watchdog on, 4MHz int clock
        __CONFIG    _MCLRE_OFF & _CP_ON & _WDT_ON & _IntRC_OSC
    endif
```

If the ‘DEBUG’ symbol has been defined (it doesn’t have to be set equal to anything, just defined), the first __CONFIG directive is assembled, turning off code protection and the watchdog timer.

The ‘ELSE’ directive marks the beginning of an alternative block of code, to be assembled if the previous conditional block was not selected for assembly.

That is, if the ‘DEBUG’ symbol has *not* been defined, the second __CONFIG directive is assembled, turning on code protection and the watchdog timer.

When you have finished debugging, you can either comment out the ‘#define DEBUG’ directive, or change ‘DEBUG’ to another symbol, such as ‘RELEASE’. The debugging code will now no longer be assembled.

In many cases, simply testing whether a symbol exists is not enough. You may want the assembler to assemble different sections of code and/or issue different messages, depending on the value of a symbol, or of an expression containing perhaps a number of symbols.

As an example, suppose your code is used to support a number of hardware configurations, or revisions. At some point the printed circuit board may have been revised, requiring different pin assignments. In that case, you could use a block of code similar to:

```
;***** Configuration
    constant    REV='A'             ; hardware revision

; pin assignments
    IF REV=='A'                         ; pin assignments for REV A:
        #define LED         GPIO,1      ;   indicator LED on GP1
        #define BUTTON      GPIO,3      ;   pushbutton on GP3
    ENDIF
    IF REV=='B'                         ; pin assignments for REV B:
        #define LED         GPIO,2      ;   indicator LED on GP2
        #define BUTTON      GPIO,5      ;   pushbutton on GP5
    ENDIF
    IF REV!='A' && REV!='B'
        ERROR "Revision must be 'A' or 'B'"
    ENDIF
```

This code allows for two hardware revisions, selected by setting the constant ‘REV’ equal to ‘A’ or ‘B’.

The ‘IF *expr*’ directive instructs the assembler to assemble the following block of code if the expression *expr* is true. Normally a *logical expression* (such as a test for equality) is used with the ‘IF’ directive, but arithmetic expressions can also be used, in which case an expression that evaluates to zero is considered to be logically false, while any non-zero value is considered to be logically true.

MPASM supports the following logical operators:

not (logical compliment)	!
greater than or equal to	>=
greater than	>
less than	<
less than or equal to	<=
equal to	==
not equal to	!=
logical AND	&&
logical OR	||

Precedence is in the order shown.

As for arithmetic and bitwise operators, parentheses are used to change the order of precedence: '`(`' and '`)`'.

Note that the test for equality is two equals signs; '==', not '='.

In the code above, setting '`REV`' to 'A' means that the first pair of `#define` directives will be executed, while setting '`REV`' to 'B' executes the second pair. But what if '`REV`' was set to something other than 'A' or 'B'? Then neither set of pin assignments would be selected and the symbols '`LED`' and '`BUTTON`' would be left undefined. The rest of the code would not assemble correctly, so it is best to check for that error condition.

This error condition can be tested for, using the more complex logical expression:

```
        REV!='A' && REV!='B'
```

Incidentally, this can be rewritten equivalently[1] as:

```
        !(REV=='A' || REV=='B')
```

You can of course use whichever form seems clearest to you.

The '`ERROR`' directive does essentially the same thing as '`MESSG`', but instead of printing the specified message and continuing, '`ERROR`' will make the progress bar that appears during assembly turn red, and the assembly process will halt.

The '`IF`' directive is also very useful for checking that macros have been called correctly, particularly for macros which may be reused in other programs.

For example, consider the delay macro defined earlier:

```
DelayMS MACRO   ms                  ; delay time in ms
        movlw   ms/.10              ; divide by 10 to pass to delay10 routine
        call    delay10
        ENDM
```

The maximum delay allowed is 2.55 s, because all the registers, including W, are 8-bit and so can only hold numbers up to 255. If you try calling '`DelayMS`' with an argument greater than 2550, the assembler will warn you about "Argument out of range", but it will carry on anyway, using the least significant 8 bits of '`ms/.10`'. That's not a desirable behaviour. It would be better if the assembler reported an error and halted, if the macro is called with an argument that is out of range.

That can be done as follows:

```
DelayMS MACRO   ms                  ; delay time in ms
    IF ms>.2550
        ERROR "Maximum delay time is 2550ms"
    ENDIF
        movlw   ms/.10              ; divide by 10 to pass to delay10 routine
        call    delay10
        ENDM
```

By testing that parameters are within allowed ranges like this, you can make your code more robust.

[1] This equivalence is known as De Morgan's theorem.

MPASM offers many more advanced facilities that can make your life as a PIC assembler programmer easier, but that's enough for now. Other MPASM directives will be introduced in future lessons, as appropriate.

The next lesson covers the 12F508/9's sleep mode, watchdog timer, and clock (oscillator) options.