

A Guide to Common PIC Assembly Programming Constructs

Jason Thweatt

0	Preliminary Information	1
0.1	What this guide is, and what it is not	1
0.2	Changes and Editions	1
0.3	Editorial and Acknowledgement Credits	2
1	Initializing a File Register	3
1.1	Initializing a File Register (with a literal)	3
1.2	Initializing a File Register (with a value from a different file register)	3
2	IF-THEN-ELSE statements	4
2.1	IF A > 0	6
2.2	IF A = 0	7
2.3	IF A > B (A and B are stored in file memory)	8
2.4	IF A > X (A is stored in file memory, X is a constant)	9
2.5	IF A = B (A and B are stored in file memory)	9
2.6	IF A = X (A is stored in file memory, X is a constant)	10
3	FOR loops	11
3.1	Executing the FOR loop	11
3.2	Nested-FOR loops	12
4	SWITCH / CASE statements	14
5	Direct addressing vs. Indirect addressing	17
6	GOTO statements vs. CALL statements	19
7	Full Program Code Examples	23
7.1	Direct Addressing Example	23
7.2	Indirect Addressing Example	25
7.3	IF-THEN-ELSE structure	28

0 Preliminary Information

0.1 What this guide is, and what it is not

The purpose of this guide is not to teach you how to program in Assembly. There is no substitute in that regard aside from understanding the instruction set and knowing how to write an algorithm. I am fond of telling students that if you can write an algorithm whose steps are fine enough, that writing an Assembly program becomes a matter of simply translating the steps of your algorithm into the Assembly statements of your processor.

That is perhaps the greatest skill that you can develop in this part of the course, because you will eventually work with a different processor, and a different Assembly standard. But this will only be a matter of learning a new set of Assembly mnemonics, and not one of having to relearn Assembly.

In using this guide, you should also note that while it may give information on how to use the constructs, it is mostly up to you to understand when and why you should be using them. This will require you to have a mastery of the algorithm that implements a given program, and to understand the sometimes-subtle differences that exist between two constructs.

Finally, I am sure that if you have programmed before, you know that programming is at least as much an art as it is a science. The information presented here is not meant to substitute for either your analytic ability or your creativity in designing a particular algorithm. These are tools, and nothing more.

I have tried to present these constructs in a logical order; if construct B uses construct A, then construct A is presented first. As far as usage is concerned, when a label is shown without brackets in a comment:

```
FIRST      equ      0x20      ; FIRST = 0x20
```

then it is the label that has that value. When a label is shown in brackets:

```
MOVLW     FIRST      ; [W] = 0x20
MOVWF     FSR        ; [FSR] = 0x20
```

then it is the contents of the address having that symbolic label as its name that has the value. This convention also applies to numeric values. 0x10 is the literal value hex 10. [0x10] describes the contents of memory address 0x10.

0.2 Changes and Editions

This document will continue to evolve as more people read it, and suggest changes that will make it more appropriate for their needs. Version number **1.5** of this document was first generated on **01 August 2005**. The version number is for editorial purposes only, as I expect this document to be a continually changing draft.

All questions and comments concerning the information presented herein should be forwarded to the author, Jason Thweatt. Professor Thweatt can be reached most easily via e-mail at jthweatt@vt.edu.

All constructive criticism and suggestions for additions and editions are greatly appreciated

0.3 Editorial and Acknowledgement Credits

My greatest thanks go to Bob Lineberry, who helps me to keep this document current by supporting the Computer Engineering Laboratory and its associated website, among other things. Without his tireless efforts, the quality of our department's instruction and instructional support would not be what it is today.

The following people have contributed to this guide by making suggestions on its content, or by finding substantive errors in the content presented. Their contributions are also appreciated.

- ECE 2504 student Mike DiMare found a commenting and logical error in the code segment given in Section 2.4, which led to further corrections in the code segment given in Section 2.3.
- Professor Joseph Tront suggested a clarification concerning the sign value of numbers being compared in the various IF...THEN tests.

1 Initializing a File Register

Sometimes you will want a file register to contain a known value. One example is the use of FSR in indirect addressing. If FSR is an indirect “pointer,” FSR must point to some initial address first. You might know this value beforehand. On the other hand, you might not know the value, but you might know that it is stored in some file memory address. We will address each of these cases in separate sections.

1.1 Initializing a File Register (with a literal)

If the value that you are using to initialize some register is known, you can treat it as a literal. If you can get the literal into the working register (hereafter referred to as W), you can then move it from W into whatever register you are trying to initialize.

Here is an example where FSR is being made to point to file memory address 0x20:

```
MOVLW    0x20        ; [W] = 0x20
MOVWF    FSR         ; [FSR] = 0x20 (through W)
```

We can use a symbolic label to describe the address. Equate the value 0x20 with some label in the label declaration section of the program:

```
FIRST    equ        0x20        ; FIRST = 0x20
```

Now make use of the label in the same fashion as shown above:

```
MOVLW    FIRST      ; [W] = 0x20
MOVWF    FSR         ; [FSR] = 0x20 (through W)
```

1.2 Initializing a File Register (with a value from a different file register)

As mentioned previously, there are times where the initializing value is unknown, but is stored in a known address. This version of the PIC processor cannot do file-memory-to-file-memory transfers. Instead, any operand that is being moved from one file memory address to another must be moved through W.

Suppose that this time, the value that we want to place in FSR is located in file memory address 0x10:

```
MOVF     0x10, 0     ; [W] = [0x10]
MOVWF    FSR         ; [FSR] = [0x10] (through W)
```

Just as before, we can use a symbolic label to refer to the file memory address where the operand can be found.

```
FIRST    equ        0x10        ; FIRST = 0x10
```

Now make use of the label as shown before:

```
MOVF     FIRST, 0    ; [W] = [0x10]
MOVWF    FSR         ; [FSR] = [0x10] (through W)
```

2 IF-THEN-ELSE statements

No single example can demonstrate every instance of an IF-THEN-ELSE statement. For one reason, there are four different PIC assembly instructions that can be used in an IF structure. PIC Assembly performs IF statements through the use of program statements that conditionally skip one line of code. There are four instructions that do this:

INCFSZ f, d

Increment the contents of register f. Store the result in W (if d = 0) or in register f (if d = 1). If the incremented value of f = 0, skip the next line of code.

DECFSZ f, d

Decrement the contents of register f. Store the result in W (if d = 0) or in register f (if d = 1). If the decremented value of f = 0, skip the next line of code.

BTFSC f, b

Test bit b of register f. If bit b of register f is clear (that is, it equals zero), skip the next line of code.

BTFSS f, b

Test bit b of register f. If bit b of register f is set (that is, it equals one), skip the next line of code.

Writing an IF statement requires the programmer to decide how the condition required by the IF can be tested using one of those four instructions. The next two lines of code determine the THEN and the ELSE of the IF statement. In general, it looks something like this:

(Test Instruction)		; This is one of the four instructions shown above. The test ; exemplifies the IF condition being checked.
(ELSE Instruction)		; This instruction is executed if the above test <u>fails</u> and the ; instruction that follows the test instruction <u>is not skipped</u> .
(THEN Instruction)		; This instruction is executed if the above test <u>succeeds</u> and the ; instruction that follows the test <u>is skipped</u> .

Carrying out the THEN and ELSE instructions may require more than one line of code. Since the “skip” instructions only skip one line of code, a GOTO statement is often used to branch to a section of the code where the necessary instructions are carried out. Using a GOTO instruction for both the THEN and ELSE instructions creates two mutually exclusive branches that satisfies the requirements of the THEN and ELSE instructions:

(Test Instruction)		; This is one of the four instructions shown above. The test ; exemplifies the IF condition being checked.
GOTO	ELSE	; ELSE is a routine that satisfies the requirements of the ELSE ; statement.
GOTO	THEN	; THEN is a routine that satisfies the requirements of the THEN ; statement.

Later on in the same program, we would find two routines, ELSE and THEN:

```
ELSE      (instruction)
          (instruction)
          GOTO      PLACE

THEN     (instruction)
          (instruction)
          GOTO      WHEREVER
```

In the above case, the two branches remain separate, as routine ELSE branches to PLACE after carrying out its instructions, while routine THEN branches to WHEREVER. If the two branches eventually rejoin, the two branches can GOTO the same place after carrying out their mutually exclusive instructions, as is shown here:

```
ELSE      (instruction)
          (instruction)
          GOTO      PLACE

THEN     (instruction)
          (instruction)
          GOTO      PLACE
```

Each of these code blocks produces the following general structure **“IF (the test instruction succeeds), THEN (do whatever the THEN instructions are), ELSE (do whatever the ELSE instructions are).”**

Sometimes, a program might not do mutually exclusive things in the IF-THEN-ELSE structure. For example, a program might always do one thing regardless of whether the IF statement is satisfied or not, but it might do something special if the IF statement succeeds (or fails). This can be accomplished by making one of the GOTO statements into a subroutine CALL:

```
(Test Instruction)      ; This is one of the four instructions shown above. The test
                        ; exemplifies the IF condition being checked.

CALL      ELSE          ; ELSE is a routine that satisfies the requirements of the ELSE
                        ; statement.

(instruction)           ; This instruction is executed if the above test succeeds and the
                        ; instruction that follows the test is skipped.

(instruction)           ; Perhaps there is more than one thing that is done as part of the
                        ; THEN case. Perhaps this is just the next line of code.
```

In this specific case, if the test fails, subroutine ELSE is called. That subroutine might look like this:

```
ELSE      (instruction)
          (instruction)
          RETURN
```

The RETURN statement will send the program back to the line of code following the subroutine call. That line happens to be the line that would have been executed had the test succeeded. What we have then is **“IF (the test instruction succeeds), THEN (do whatever the THEN instructions are), ELSE (do whatever is in the ELSE subroutine, and then do whatever the THEN instructions are).”**

The programmer’s analytical ability and creativity now enter the picture, as it is the programmer’s job to determine how the IF condition can be tested, and how the THEN and ELSE instructions are to be carried out. Fortunately, there are any number of conditions in the PIC processor that can be tested and acted upon. We will cover several noteworthy cases here. Since the FOR loop is a specific application of the IF structure, it will be treated separately.

2.1 IF $A > 0$

If we take the contents of a file memory address to be a representation of an 8-bit signed two’s complement number, then we can determine if that number is positive or negative by checking its sign bit. Strictly speaking, this will not differentiate between a positive number and zero, as both would have zero sign bits. However, a test on the most significant bit (bit 7) of a given file memory address will determine whether the number is positive or negative. As in the previous development, we can then do one of two things based on the result of the test.

Suppose that a register called A contains such an 8-bit signed 2’s complement number. We can use one of the two bit test instructions as the IF statement. The two lines that follow define the THEN and ELSE requirements.

BTFSC	A, 7	; Test the sign bit of A. Skip the next instruction of the ; sign bit is 0.
GOTO	NEG	; If this instruction is executed, A_7 is 1. Therefore, [A] is ; negative. Do whatever must be done if [A] is negative.
GOTO	POS	; If this instruction is executed, A_7 is 0. Therefore, [A] is ; positive. Do whatever must be done if [A] is positive.

In this case, the test is **“IF $[A] \geq 0$, THEN (do whatever instructions are found at routine POS), ELSE (do whatever instructions are found at routine NEG).”**

This is not the only test that would work. If we switch from a “skip if clear” to a “skip if set,” changing the order of the THEN and ELSE statements completes the new IF structure:

BTFSS	A, 7	; Test the sign bit of A. Skip the next instruction of the ; sign bit is 0.
GOTO	POS	; If this instruction is executed, A_7 is 0. Therefore, [A] is ; positive. Do whatever must be done if [A] is positive.
GOTO	NEG	; If this instruction is executed, A_7 is 1. Therefore, [A] is ; negative. Do whatever must be done if [A] is negative.

In this case, the test is **“IF $[A] < 0$, THEN (do whatever instructions are found at routine NEG), ELSE (do whatever instructions are found at routine POS).”**

2.2 IF A = 0

No discussion of IF structures in Assembly would be complete without a discussion of the STATUS register. The STATUS register is a file memory register (address 0x03) that is used to keep track of certain other things that occur in the processor's operation. STATUS has two bits that will be especially useful to us. We will discuss one such bit in each of the next two examples.

Bit 2 of the STATUS register is the zero bit, or Z. Z is such that when the result of the execution of a certain set of assembly instructions is identically zero, Z becomes 1. When the result of one of these instructions is anything other than 0, Z becomes 0.

These instructions affect the zero bit: ADDWF, ANDWF, CLRF, CLRW, COMF, DECF, INCF, IORWF, MOVF, SUBWF, XORWF, ADDLW, ANDLW, IORLW, SUBLW, XORLW.

For whatever reason, INCFSZ and DECFSZ do not affect the zero bit. They use a different means to determine whether the register that has been incremented or decremented has become zero.

The easiest way to test (IF A = 0) is through an odd use of a common instruction. First note that the MOVWF instruction doesn't require a data direction bit, as the direction of data flow is implied: MOVWF moves an operand from W to F.

The MOVF instruction does require a data direction bit. F is always the source, and the destination depends on the polarity of the data direction bit. Consider the next two lines of code:

```

MOVWF    A, 0      ; The contents of register A are moved into W.
MOVF     B, 1      ; The contents of register B are moved into register B. (?!)

```

Why would we ever want to move a register's contents back into itself? Notice that that MOVF instruction is one that affects the zero bit. By moving a register's contents into itself, we can set up a testable condition that doesn't change the state of the operands, as is done in the following code block:

```

MOVWF    A, 1      ; The contents of register A are moved into register A. A is
                  ; therefore unchanged.

BTFSS    STATUS, Z ; Moving the contents of register A into itself affects the
                  ; zero bit. We test Z and skip the next instruction if Z = 0.

GOTO     NOTZERO   ; If this instruction is executed, Z = 0. Therefore, [A] is not
                  ; zero. Do whatever needs to be done when [A] ≠ 0.

GOTO     ZERO      ; If this instruction is executed, Z = 1. Therefore, [A] is
                  ; zero. Do whatever needs to be done when [A] = 0.

```

The above program segment runs the test **“IF [A] = 0, THEN (do whatever instructions are found at routine ZERO), ELSE (do whatever instructions are found at routine NOTZERO).”**

We can change the GOTO statements in any fashion described in the section on general IF structures.

2.3 IF $A > B$ (A and B are stored in file memory)

Bit 0 of the STATUS register is the carry bit, or C. C is properly called a carry/not borrow bit. In the general case of addition, if the instruction causes a carry-out of the most significant bit, C becomes 1. If the addition instruction does not cause a carryout, C becomes 0.

In the case of subtraction, the bit polarity reverses. If a subtraction instruction does not require borrowing, C becomes 1. If the subtraction instruction does require borrowing, C becomes 0. When subtracting, think of C as a “magical reservoir of borrow.” If we borrow, the reservoir is depleted ($C = 0$), but if we don’t borrow, the reservoir is preserved ($C = 1$).

These instructions affect the carry bit: ADDWF, RLF, RRF, SUBWF, ADDLW, SUBLW. The two rotate instructions affect the carry bit because the carry bit is involved in the rotate.

We will be using C as a not borrow bit in comparing two numbers. We’re doing this because algebraically, evaluating the statement “Is $A > B$?” is the same as evaluating the statement “Is $A - B > 0$?” We have already seen a test to see if a number is negative, but since the subtract operations affect the carry bit, we can simply test the carry bit right after we subtract.

C is such that if $A - B$ is positive, then borrow did not take place, and therefore $C = 1$. If $A - B$ is negative, then borrowing did take place, and therefore $C = 0$. We will have to account for the order of operands in the instruction, but the convenient thing about using the carry bit is that as long as we treat both numbers as being signed, or both numbers as being unsigned, the technique still works.

In the strictest sense, any comparison involving two quantities A and B should be such that both numbers are positive numbers, or treatable as positive numbers. If the quantities that are being compared are both unsigned, no problem should arise. In the case of quantities that are known to be signed, some care must be taken to ensure that two positive quantities are being compared.

First, let’s compare two numbers that are stored in memory. The SUBWF instruction specifically subtracts W from F, that is, the number in the working register is subtracted from the value in the operand address. So to set up $A - B$, we will have to have B in the working register, and name A as the operand address. Since we don’t want to overwrite A, we will use a data direction bit of zero:

MOVF	B, 0	; [W] = [B]
SUBWF	A, 0	; [W] = [A] - [B]
BTFSS	STATUS, C	; We are testing the carry bit, and we’ll skip the next ; instruction if $C = 1$.
GOTO	ALESSER	; If we execute this instruction, $C = 0$. This means that ; $[A] - [B] < 0$, and [A] is less than [B]. Branch to the code ; block that does whatever is required when $[A] < [B]$.
GOTO	ABIGGER	; If we execute this instruction, $C = 1$. This means that ; $[A] - [B] \geq 0$, and [A] is greater than or equal to [B]. ; Branch to the code block that does whatever is ; required when $[A] \geq [B]$.

The above program segment runs the test “**IF A ≥ B, THEN (do whatever instructions are found at routine ABIGGER), ELSE (do whatever instructions are found at routine ALESSER).**”

Notice that strictly speaking, we aren’t doing “IF A > B,” but “IF A ≥ B.” In another section, we will see a test that can be used to determine whether A = B (or conversely, whether A ≠ B). Performing these tests in series can be used to specifically determine whether A and B are such that A > B, A = B, or A < B.

2.4 IF A > X (A is stored in file memory, X is a constant)

Now let’s compare a number in memory to a constant. The constant can be represented as a literal value, which we can introduce with a literal instruction. The SUBLW instruction subtracts the value in W from a literal and places the result in W.

Since we are performing $L - W$, we note that if $C = 1$ following subtraction, then borrowing didn’t occur. This means that $L - W$ is greater than or equal to zero, and $W \leq L$. If $C = 0$ following subtraction, then borrowing did occur. This means that $L - W$ is less than zero, and $L < W$.

MOVF	A, 0	; [W] = [A]
SUBLW	X	; [W] = X - [A]
BTFSC	STATUS, C	; We are testing the carry bit, and we’ll skip the next ; instruction if C = 0.
GOTO	ALESSER	; If we execute this instruction, C = 1. This means that ; $X - [A] \geq 0$, and [A] is less than or equal to X. Branch to ; the code block that does whatever is required when $[A] \leq X$.
GOTO	ABIGGER	; If we execute this instruction, C = 0. This means that ; $X - [A] < 0$, and [A] is greater than X. Branch to the code ; block that does whatever is required when $[A] > X$.

The above program segment runs the test “**IF A > X, THEN (do whatever instructions are found at routine ABIGGER), ELSE (do whatever instructions are found at routine ALESSER).**”

Because the SUBWF and SUBLW employ two different subtraction operations, the order of operands is switched in this program ($X - A$) relative to the order found in the previous greater than/less than test ($A - B$). To compensate for this, a different bit-test is used in this program (BTFSC versus BTFSS) to determine a slightly different result ($A > X$ versus $A \geq B$).

2.5 IF A = B (A and B are stored in file memory)

Just as we saw previously, a sign bit test cannot differentiate between a positive number and zero. We may have to run an additional test if we care about the difference between A being greater than B, and A being equal to B. Fortunately, we have already seen the means to test for a value being zero. That test would, in a way, combine the last two programming examples: if $A = B$, then $A - B = 0$. SUBWF does affect the zero bit, and we can test for it in a manner similar to the program block that we just wrote:

MOVF	B, 0	; [W] = [B]
SUBWF	A, 0	; [W] = [A] - [B]
BTFSS	STATUS, Z	; We are testing the zero bit, and we'll skip the next instruction if ; Z = 0.
GOTO	NOTEQUAL	; If we execute this instruction, Z = 0. This means that ; [A] - [B] ≠ 0, and [A] is not equal to [B]. Branch to the code ; block that does whatever is required when [A] ≠ [B].
GOTO	EQUAL	; If we execute this instruction, Z = 1. This means that ; [A] - [B] = 0, and [A] is equal to [B]. Branch to the code block ; that does whatever is required when [A] = [B].

The above program segment runs the test “**IF A = B, THEN (do whatever instructions are found at routine EQUAL), ELSE (do whatever instructions are found at routine NOTEQUAL).**”

2.6 IF A = X (A is stored in file memory, X is a constant)

In a similar fashion to what we did previously, we can compare the value of A to some literal X:

MOVF	A, 0	; [W] = [A]
SUBLW	X	; [W] = X - [A]
BTFSS	STATUS, Z	; We are testing the zero bit, and we'll skip the next ; instruction if Z = 0.
GOTO	NOTEQUAL	; If we execute this instruction, Z = 0. This means that ; X - [A] ≠ 0, and [A] is not equal to X. Branch to the code ; block that; does whatever is required when [A] ≠ X.
GOTO	EQUAL	; If we execute this instruction, Z = 1. This means that ; [A] - [B] = 0, and [A] is equal to X. Branch to the code block ; that does whatever is required when [A] = [B].

As a final note, we can change each of these specific comparisons to check for their complements. For example, instead of checking to see if $A \geq B$, we can check to see if $A < B$. We can do this by changing either the order of the mutually exclusive branches, or by changing the polarity of the bit test.

3 FOR loops

As mentioned above, and for reasons that we will see, a FOR loop uses a special application of an IF structure. Specifically, checking for completion of the loop is equivalent to checking for the IF structure **“IF the loop is complete, THEN terminate, ELSE repeat the loop.”**

Ensuring that the FOR loop terminates is a matter of making sure that the same IF structure can be used to validate a condition that could change as a result of any particular execution of the loop.

3.1 Executing the FOR loop

The FOR loop is carried out in two parts. First, a counter must be established whose value is equal to the number of times that the loop must be executed. This can be done with the value initialize that has already been described.

We may know the number of times that the loop should be executed:

```

MOV LW    0x04      ; [W] = 0x04
MOV WF    COUNT    ; [COUNT] = 0x04 (through W). This sets up “FOR COUNT =
                  ; 1 to 4.”

```

If the loop count value has a symbolic label, we can use it as described in section 1:

```

LOOPCNT  equ      0x04      ; LOOP = 0x04

```

And later:

```

MOV LW    LOOPCNT   ; [W] = 0x04
MOV WF    COUNT    ; [COUNT] = 0x04 (through W). This sets up “FOR COUNT =
                  ; 1 to 4.”

```

We may not know the number of times that the loop should be executed, but we may know where that value can be found in file memory:

```

MOV F    0x10, 0    ; [W] = [0x10]. Presumably, the number of times that the loop
                  ; should be executed can be found in address 0x10.
MOV WF    COUNT    ; [COUNT] = [0x10] (through W). This sets up “FOR COUNT
                  ; = 1 to (the value held in register 0x10.)

```

Again, if the register holding the loop count value has a symbolic label, we can use it as described in section 1:

```

LOOPCNT  equ      0x10      ; LOOP = 0x10

```

(The example continues on the next page.)

And later:

```

MOVWF    LOOPCNT, 0    ; [W] = [0x10]. Presumably, the number of times that the
                    ; loop should be executed can be found in address 0x10.

MOVWF    COUNT        ; [COUNT] = [0x10] (through W). This sets up
                    ; "FOR COUNT = 1 to (the value held in register 0x10.)

```

Having initialized the counter, we are now ready to carry out the loop. All of the elements of the loop must be in a block of code that precedes the final check to see if the loop is complete:

```

LOOP      (instruction)
          (instruction)
          (instruction)
          (instruction)

```

At some point, we will reach the end of the block, and we'll need to see if the block should be executed again or not. The operative word here is "if." We can use an IF structure on the counter, as long as we can determine a condition that separates the last time that the block should be run from all others.

If we decrement the counter each time we run the block, then the last time that the block is run, the value will decrement to zero. So if the decremented count value is not zero, we need to run the block again. If the decremented count value is zero, we need to do something else:

```

DECFSZ   COUNT, 1     ; COUNT = COUNT - 1. We use a data direction bit of 1 so that
                    ; the same value is decremented each time.

GOTO     LOOP         ; If COUNT ≠ 0, this line is executed. Since we aren't finished,
                    ; we go back to the top of the loop and execute the block again.

(instruction)         ; If COUNT = 0, this line is executed. Since we are finished, we
                    ; need to do whatever must be done at the end of the loop.

```

The instruction that ends the loop can be almost anything. The program might end at this point, or it might continue. An appropriate "halt" statement might put the program into an infinite loop:

3.2 Nested-FOR loops

It's relatively easy to do nested-FOR loops. It's reasonable to assume that in this case, the code for the outer loop encapsulates the code for the inner loop:

```

LOOP2     (instruction)
          (instruction)
          ...

LOOP1     (instruction)
          (instruction)
          ...

```

LOOP2 is the outer loop, and LOOP1 is the inner loop. Here, something is being done in LOOP2 before LOOP1 is executed. Once the program enters LOOP1, the instructions in the code block for LOOP2 are not done again until LOOP1 is complete.

In the nested-FOR loop, we have to make sure to re-initialize the value of the inner loop counter each time the IF condition of the outer loop counter failed. As we will see, there is more than one place where we could do this.

Here are the “nested” IF-structures that take care of resuming or exiting each of the two loops. The first loop counter must count down to zero before the second loop counter is decremented and checked:

DECFSZ	CNT1, 1	; CNT1 = CNT1 – 1. We use a data direction bit of 1 so that the ; same value is decremented each time.
GOTO	LOOP1	; If CNT1 ≠ 0, this line is executed. Since we aren’t finished ; with the inner loop, we go back to the top of the inner loop and ; execute the block again.
MOVLW	LC1	; If CNT1 = 0, this line is executed. Since we are finished with ; the inner loop, we will first reinitialize CNT1.
MOVWF	CNT1	; [CNT1] = LC1. Presumably LC1 is the number of times the ; inner loop needs to be executed.
DECFSZ	CNT2, 1	; CNT2 = CNT2 – 1. We use a data direction bit of 1 so that the ; same value is decremented each time.
GOTO	LOOP2	; If CNT2 ≠ 0, this line is executed. Since we aren’t finished ; with the outer loop, we go back to the top of the outer loop and ; execute the block again.
(instruction)		; IF CNT2 = 0, this line is executed, we need to do whatever ; must be done at the end of the outer loop.

Each time LOOP2 is executed, we re-execute the code block of LOOP1 however many times LC1 indicates that it should be executed. As shown in a previous section, if the loop counter is contained in an address instead of held as a literal, we can change the above code accordingly. In this example, the inner loop counter is being initialized at the point where the inner loop has been completed.

We could have performed the initialization of the counter used in the inner loop (LOOP1) as part of the code block for the outer loop (LOOP2) instead of doing it in the IF-structure. Since the code block in LOOP2 will only be done each time the code block in LOOP1 is completed, we will be initializing the loop counter for LOOP1 once each time the IF structure for LOOP2 fails, and before LOOP1 is resumed.

LOOP2	MOVLW	LC1
	MOVWF	CNT1
	(instruction)	
	...	
LOOP1	(instruction)	
	...	

4 SWITCH / CASE statements

There may be instances where a single IF-THEN-ELSE structure is not sufficient. It may be the case that a condition can take on one of multiple discrete settings or values. In a microprocessor, it is likely the case that any one of those discrete settings can be mapped to a set of distinct values, so we will confine our discussion to situations involving values.

Suppose that a value contained in a certain register indicates which of a number of mutually exclusive sets of instructions needs to be executed. If there were only two things that had to be done, that is, the value could be mapped to a single bit, then a simple bit test could be used as an element of an IF structure to determine which of the two sets of instructions is to be executed.

If there are more than two sets of instructions, the situation becomes a little more complicated. We could use something along the lines of a binary tree to determine which set of instructions is the correct one. Given n alternatives, we can arrive at the correct alternative after $\lceil \log_2 n \rceil$ bit tests, where $\lceil x \rceil$ indicates “the smallest integer that is greater than or equal to x ”. For example, with eight alternatives numbered 0 through 7 inclusive, we can perform 3 bit tests:

Is the first bit 0?	YES	Is the second bit 0?	YES	Is the third bit 0?	YES	000	
					NO	001	
				NO	Is the third bit 0?	YES	010
						NO	011
	NO	Is the second bit 0?	YES	Is the third bit 0?	YES	100	
					NO	101	
				NO	Is the third bit 0?	YES	110
						NO	111

The code for this binary tree might appear as follows. Note that the labels being used might not represent valid PIC Assembly labels, but are being used for ease of illustration:

```

XXX      BTFSS      X, 2      ; Test the first bit. Skip the next line if the bit equals 1
         GOTO      0XX      ; Go to the next test, knowing that the first bit is zero.
         GOTO      1XX      ; Go to the next test, knowing that the first bit is one.

0XX      BTFSS      X, 1      ; Test the second bit. Skip the next line if the bit equals 1.
         GOTO      00X     ; Go to the next test, knowing that the second bit is zero.
         GOTO      01X     ; Go to the next test, knowing that the second bit is one.

1XX      BTFSS      X, 1      ; Test the second bit. Skip the next line if the bit equals 1.
         GOTO      10X     ; Go to the next test, knowing that the second bit is zero.
         GOTO      11X     ; Go to the next test, knowing that the second bit is one.

00X      BTFSS      X, 0      ; Test the third bit. Skip the next line if the bit equals 1.
         GOTO      000     ; The code is 000. Go to the appropriate instruction.
         GOTO      001     ; The code is 001. Go to the appropriate instruction.

```

(The program continues on the next page.)

01X	BTFSS	X, 0	; Test the third bit. Skip the next line if the bit equals 1.
	GOTO	010	; The code is 010. Go to the appropriate instruction.
	GOTO	011	; The code is 011. Go to the appropriate instruction.
10X	BTFSS	X, 0	; Test the second bit. Skip the next line if the bit equals 1.
	GOTO	100	; The code is 100. Go to the appropriate instruction.
	GOTO	101	; The code is 101. Go to the appropriate instruction.
11X	BTFSS	X, 0	; Test the second bit. Skip the next line if the bit equals 1.
	GOTO	110	; The code is 110. Go to the appropriate instruction.
	GOTO	111	; The code is 111. Go to the appropriate instruction.

Using a binary tree is more efficient than using eight direct comparisons, but we can use a feature of the PIC processor to make the decision as the result of a single test. This method employs a construct called a jump table, which is very much like a CASE statement.

The jump table takes advantage of the fact that we can modify a register in the file memory called the Program Counter. The Program Counter acts as a program memory pointer of sorts, in that it generally points to the address of the next program instruction that is to be executed.

Consider the following code block:

DECFSZ	COUNT, 1	; COUNT = COUNT – 1. We use a data direction bit of 1 ; so that the same value is decremented each time.
GOTO	LOOP	; If COUNT ≠ 0, this line is executed. Since we aren't finished, ; we go back to the top of the loop and execute the block again.
(instruction)		; If COUNT = 0, this line is executed. Since we are finished, we need to do whatever must be done at the end of the loop.

When the DECFSZ instruction is being executed, the Program Counter contains the program memory address of the GOTO instruction. If the DECFSZ instruction happens to be in program memory address 0x06, the Program Counter would contain the value 0x07, which is the address of the GOTO instruction. Remember that instructions are stored in program memory in the sequential order in which they are written. However, they will not necessarily be executed in the order they appear in program memory, since we have statements that can skip instructions, GOTO instructions, or CALL instructions.

As it turns out, there is a file memory address called PCL. PCL contains the **Program Counter's Lowest** eight bits. Since this register is in the file memory, any instruction that can modify a file memory address can modify PCL. At first glance, this might seem dangerous. If it is the Program Counter's job to keep track of the address of the next instruction to be executed, then changing its value *ad hoc* would throw the program into chaos. As such, we hardly ever use PCL as a file memory register operand of an instruction.

The jump table is one instance in which we purposefully modify PCL. From the example developed in the earlier, suppose that some register contains the three-bit value on which we want to base one of eight different mutually exclusive program paths. Consider the following code block:

```

MOVWF      X, 0          ; [W] = [X]
ADDWF     PCL, 1        ; [PCL] = [PCL] + [X] (through W)
GOTO      X0           ; IF [X] = 0
GOTO      X1           ; IF [X] = 1
GOTO      X2           ; IF [X] = 2
GOTO      X3           ; IF [X] = 3
GOTO      X4           ; IF [X] = 4
GOTO      X5           ; IF [X] = 5
GOTO      X6           ; IF [X] = 6
GOTO      X7           ; IF [X] = 7
(instruction 1)          ; This could be anything.
(instruction 2)          ; This could be anything.

```

Just as before, we are assuming that the value of X is in the range that can be represented by three bits. If this assumption is not valid, we may have to operate on the value of X prior to applying it to the jump table.

The key instruction is the ADDWF instruction. After this instruction has been fetched, the Program Counter is pointing to the instruction that immediately follows, GOTO X0. Suppose that the value of X is 0. When that value is added to PCL, the value of PCL does not change. This means that the Program Counter still points to the instruction GOTO X0. Therefore, the instruction GOTO X0 and the program branch to which that instruction jumps are the ones that must correspond to X having a value of 0.

Now suppose that the value of X is 1. When that value of X is added to PCL, the value of PCL is one higher than it was. But that means that it points to the instruction in the next higher address. This instruction is GOTO X1. Therefore, the instruction GOTO X1 and the program branch to which that instruction jumps are the ones that must correspond to X having a value of 1.

In this fashion, we can see that the instruction GOTO X_n , where n is a value from 0 to 7, is the instruction that is reached when $X = n$. As a result, the program must branch to a code block that takes care of case n .

This mode of operation should emphasize why we have to be careful that the value in X is valid. Suppose that X equaled 9. When the value $X = 9$ is added to PCL, we jump to the instruction called (instruction 2) in the code block. This is invalid, as we have jumped over the jump table altogether and presumably landed in the middle of some other routine. Care must be taken to make sure that the index value (the value that we used to determine where we jump) is a value that causes us to land somewhere in the jump table. This has to be done before the index value is actually applied to the jump table.

The manner in which you check the index value depends on the nature of your algorithm, the relative size of the index value, and how vigorously the index value has to be checked – among other things.

5 Direct addressing vs. Indirect addressing

In a normal directly addressed instruction, the operand address names the address where the instruction operand may be found. For example, the instruction:

```
ADDWF    X, 0
```

checks the register whose address is given by the symbolic label X for the operand of the ADDWF instruction. If we had placed the following line in our label-declaration segment:

```
X        equ    0x10
```

then the instruction shown above would add the value contained in address 0x10 to the value in W. Since the data direction bit is 0, the result is stored in W. Had the instruction read as follows:

```
ADDWF    X, 1
```

the result of the instruction would have been stored in address 0x10.

A directly addressed instruction is a single-fetch instruction, since we only have to look in one address to find the operand.

Any instruction that takes a file-memory address as an operand may be executed via indirect addressing. Indirect addressing is best used when the operand address of an instruction varies over multiple executions of the same instruction. You will often use indirectly addressed instructions in this fashion as parts of FOR loops. For example, a loop that is designed to add the elements of an array can be executed using indirect addressing in a loop. The operative part of the loop has the following form:

```
LOOP     ADDWF    INDF, 0
```

INDF is used as the operand register in any indirectly addressed instruction. You may consider INDF to be a directive to the processor to look in the FSR register. FSR can be thought of as an indirect address “pointer.” Upon finding a value in FSR, the processor does not treat this as the operand (as it did in the direct-addressing example), but as the address of the operand. After looking in that address, the true operand is found.

Typically, an indirectly addressed instruction is a double-fetch instruction, since we must first look in one place to find the address of the operand, and then look in that address to find the actual operand.

Using FSR properly is the key to executing any indirectly addressed instruction. In the previous example, we must change the value in FSR so that it “points” to a different address each time. If we are assuming that each of the array elements exist in consecutive file-memory addresses, then each time we execute the ADDWF instruction, we have to update FSR:

```
INCF     FSR, 1    ; FSR = FSR + 1
```

This instruction would have to be carried out at some point prior to the loop being restarted. Here is one way that we might do this:

```

LOOP      ADDWF    INDF, 0
          INCF     FSR, 1
          DECFSZ   COUNT, 1
          GOTO     LOOP
FINISH    GOTO     FINISH

```

This code block doesn't show the process by which FSR and COUNT were initialized. You may assume that it was done in some fashion with that presented in the section on initializing file register. The code block does show a FOR-loop structure that uses COUNT as the FOR-loop counter. Notice how the IF structure operates on whether COUNT equals zero or not.

In addition to the general use of indirect addressing in an operation (ADDWF INDF, SUBWF INDF, IORWF INDF) two specific cases are worth noting. An indirect file memory write will move the contents of W into the file memory register to which FSR points:

```
MOVWF    INDF
```

An indirect file memory read will move the contents of the file memory register to which FSR points into W:

```
MOVF     INDF, 0
```

In both of these cases, as FSR changes, the destination or source of the indirect read or write changes.

In general, **you must be careful with the addressing mode**. When you write an algorithm, you must have a good idea whether a given instruction should be addressed directly or indirectly, as the overall result of the instruction will be different in each case. This is not an error that the assembler will catch, as it is perfectly legitimate to write this:

```
ADDWF    INDF, 0
```

when you meant to specify a certain register, or vice versa.

6 GOTO statements vs. CALL statements

A GOTO statement is an example of an unconditional branch. In executing the GOTO instruction, we jump to the program memory address indicated by the argument of the GOTO statement, and we do this without any intent of eventually returning to the point where the branch occurred. So the statement:

```
GOTO      0x10
```

would jump to program memory address 0x10. The instruction in address 0x10 would be the next one to be carried out.

We have already discussed the Program Counter in this document. To understand how the GOTO statement works, we need to understand how instructions in the PIC processor are carried out. Recall that the Program Counter points to the instruction that will be the next one to be carried out. If the instruction in program memory address 0x06 is being carried out, the Program Counter will usually point to memory address 0x07.

The instruction cycle of the PIC consists of a **fetch** cycle and an **execute** cycle. During the fetch cycle, an instruction is fetched from the program memory and decoded. Decoding the instruction allows the PIC to determine exactly what the instruction needs to do. At the very end of the fetch cycle, the Program Counter is incremented, so that it points to the instruction following the one that has just been fetched. At this point, the processor is already prepared to fetch the next instruction in the program, which it will do in the next fetch cycle.

Instructions like the GOTO instruction change the value of the Program Counter during the execute cycle. During the execute cycle, the instruction that was just fetched is now actually performed. In the case of the GOTO statement, the processor has to be set up so that instead of fetching and executing the instruction that immediately follows the GOTO statement, the instruction in the program memory address represented by the argument of the GOTO statement is fetched and executed. To do this, we have to replace the value currently in the Program Counter with the program memory address represented by the argument of the GOTO statement. But since we don't plan to return to the point of the branch (at least, not without another GOTO statement) we can safely overwrite the value in the Program Counter. In doing this, the next instruction that will be fetched and executed is the one in the address that now occupies the Program Counter – the address that was the argument of the GOTO statement.

Unlike the symbolic labels that we use to name file-memory addresses, it's very difficult to know the address that should be used as the operand of the GOTO statement. (The same is true of the CALL statement.) We can't really be certain what the address of the destination instruction is until we've finished the program. This is why symbolic labels that are used as GOTO (and CALL) operands need only label the line of code that is the destination of the branch.

So this code segment:

```

GOTO      CHECK
(instruction)
(instruction)
...
CHECK    (instruction)

```

will always work, regardless of how many instructions separate the GOTO statement from the destination, or where the instruction labeled by CHECK ends up being stored in program memory. We don't have to declare the value of CHECK in the label declaration segment, as we have been with the labels that we've been using as literals or as file-memory addresses. The assembler will be able to figure out what line of the program CHECK labels, if we take care of one or two other things elsewhere.

A CALL statement is an example of a branch and save return address instruction. This means that even though we are jumping to another point in the program, we are doing it with every expectation that we will eventually return to the point of the branch. Therefore, the address from which we jumped has to be saved.

We should expect that a CALL instruction is executed similarly to a GOTO instruction. During the execute cycle, the program memory address represented by the operand of the CALL statement must be placed into the Program Counter. However, since we plan to return to the program instruction immediately following the CALL statement (the one pointed to by the current value of the Program Counter) we can't simply overwrite it. This is where the stack comes into play. Before moving the CALL statement's operand address into the Program Counter, we first push the current value of the Program Counter onto the stack. Once the return address has been saved, we can overwrite the Program Counter safely.

Eventually, we will arrive at the RETURN statement of the subroutine. Every subroutine CALL must be paired with a RETURN statement that ends the subroutine. When the RETURN is executed, the return address is popped from the stack and returned to the Program Counter. In this fashion, we return to the address of the statement that followed the original subroutine CALL. Since the stack is a last-in-first-out memory device, if a subroutine calls another subroutine, then the last subroutine to have been called is also the first one to have its RETURN statement satisfied. However, since the stack can only hold eight addresses, no more than eight subroutines calls can be executed without an intervening RETURN.

Fortunately, a subroutine need not CALL itself. If a subroutine is a block of code that is to be carried out in a loop, the subroutine can GOTO itself rather than having to CALL itself.

	CALL	SUB
	...	
SUB	ADDWF	INDF, 0
	INCF	FSR, 1
	DECFSZ	COUNT, 1
	GOTO	SUB
	RETURN	

In this case, the subroutine consists of a FOR-loop. FSR and COUNT have been initialized elsewhere. As long as COUNT is non-zero, the FOR-loop expressed in the subroutine continues. This is done with a GOTO that returns to the same line that was called by the main routine. When COUNT = 0, the FOR-loop is complete, and we can RETURN to the main routine.

Even though a subroutine should never CALL itself, a subroutine can always CALL another subroutine.

```

        CALL      SUB1
        ...

SUB1    BTFSC     INDF, 7
        CALL     SUB2.
        ADDWF   INDF, 0      ; if SUB2 is called, the first RETURN is to this line
        INCF   FSR, 1
        DECFSZ  COUNT, 1
        GOTO   SUB
        RETURN

SUB2    COMF     INDF, 1
        INCF   INDF, 1
        RETURN

```

As long as there are no more than eight nested subroutine calls, stack overflow is not a problem.

Having discussed the difference between the CALL and GOTO statements, consider the next two code blocks. The first uses a subroutine CALL:

```

        CALL SUB
        (instruction)
        ...

SUB     (instruction)
        (instruction)
        (instruction)
        RETURN

```

The second uses a pair of GOTO statements:

```

BACK   GOTO SUB
        (instruction)
        ...

SUB     (instruction)
        (instruction)
        (instruction)
        GOTO   BACK

```

Both of these code blocks do the same thing. Both programs branch to SUB from the main routine. When SUB is complete, both programs return to the line following the branch. Why is the first one better than the second?

A GOTO statement can only go to one place – the program memory address referred to by the operand of the GOTO. A RETURN statement always goes back to the line following the subroutine CALL, regardless of where it occurs in a program. This means that the same subroutine can be called from numerous points in the program, and the same RETURN statement will always go back to the right place in the program. The same cannot be said for subroutine “calls” that are done with GOTO statements.

If a code block might have to be executed from numerous points in the main program, it is best organized as a subroutine that can be called from any point in the program, and is capable of returning to the point where it was called with a single RETURN statement.

However, we have also seen many examples where a subroutine CALL and RETURN would be inappropriate. In cases where two mutually exclusive things should happen in a program, two GOTO statements are used to send the program to two different places. Using a subroutine CALL in this instance could set up a situation in which one of the code blocks is executed, and then the other. Sometimes this might be desirable, but it is not always desirable. As with many other aspects of assembly programming, you must know your algorithm well enough to know when a GOTO is appropriate and when a CALL is appropriate.

7 Full Program Code Examples

The following examples of program code have been included to give the reader an idea of how complete programs might look. These programs include elements of the program blocks that we have seen developed in this guide. They also include elements that have not been discussed here, but that you have probably discussed in your classes.

7.1 Direct Addressing Example

```

; *****
;
;       Program Name: Direct Addressing Example
;       J.S. Thweatt
;       06 April 2003
;       2504 Class Software Project 1
; *****
;
; *****
; Microchip-developed include file
;
; This include file is a set of equate statements that allows the use of address and symbolic labels such as
; STATUS for file memory location 0x04, and C for the carry bit of the STATUS register.
; *****

        include P16F84.inc

; *****
; Local equate table
;
; These are user-defined equate statements. They name symbolic variables that are specific to this program.
; *****

w       equ       0           ; DDB = 0 - store the result in w
f       equ       1           ; DDB = 1 - store the result in f

one     equ       0x11        ; The address of the first addend
two     equ       0x12        ; The address of the second addend
sum     equ       0x13        ; The address for sum storage

; *****
; The ORG statement means that the first line of the executable program code after the ORG statement will be
; placed in program memory location 0, and that successive instructions will be placed in the next consecutive
; program memory location.
; *****

        org       0

```

```

; *****
; It's a sad day when there are more comments in a program than there is program in the program, but this is
; supposed to be illustrative. To add the two numbers using direct addressing, we only have to use the proper
; instructions to operate on the addresses where addends are stored.
; *****

        start      clrw          ; We start by setting W = 0.

        addwf      one,w        ; W = W plus the first addend. Note that the result is maintained
                                ; in W, since the data direction bit is 0.

        addwf      two,w        ; W = W plus the second addend. Since we kept the first sum in
                                ; W, W now holds the sum of the addends.

        movwf      sum          ; Move the contents of W into address SUM. The direction of
                                ; data is implied, (w => f) so no data direction bit is needed

finish   goto      finish       ; Since PIC Assembly has no HALT statement, we need an
                                ; "instruction" that will put the processor in a state where no
                                ; changes can possibly happen to the processor's state. An
                                ; infinite loop does the job adequately.

; *****
; The last line of executable code must be followed with the END statement.
; *****

        end

```

7.2 Indirect Addressing Example

```

; *****
;
;   Program Name: Indirect Addressing Example
;   J.S. Thweatt
;   06 April 2003
;   2504 Class Software Project 1
; *****

; *****
; Microchip-developed include file
;
; This include file is a set of equate statements that allows the use of address and symbolic labels such as
; STATUS for file memory location 0x04, and C for the carry bit of the STATUS register.
; *****

        include P16F84.inc

; *****
; Local equate table
;
; These are user-defined equate statements. They name symbolic variables that are specific to this program.
; *****

w          equ          0          ; DDB = 0 - store the result in w
f          equ          1          ; DDB = 1 - store the result in f

n          equ          0x10       ; The number of elements in the array
count      equ          0x11       ; A variable to be used in the FOR loop
sum        equ          0x12       ; The address for sum storage
first      equ          0x20       ; The address of the first array element

; *****
; The ORG statement means that the first line of the executable program code after the ORG statement will be
; placed in program memory location 0, and that successive instructions will be placed in the next consecutive
; program memory location.
; *****

        org            0

; *****
; More comments. To add the array of numbers indirectly, we have to initialize the FSR register, which is the
; PIC processor's "pointer." Note that first is being used as a literal, since the value to which it has been equated
; represents the value of the first address in the array. If memory address 0x13 (first) had *contained* the
; address of the first element of the array, we would have used it as a file location instead of as a literal.
; *****

```

```

start      movlw      first      ; The value 0x20 is placed into W. Remember that FIRST is
          ; being used as a literal here.

          movwf     FSR       ; The value in W is placed into FSR. Now, indirect instructions
          ; are pointing to address 0x20.

          movf      n,w       ; The value in N is being moved to W.

          movwf     count     ; The value in W is being moved to COUNT. Now, COUNT =
          ; N, and we have a copy of N that we can change. This is useful
          ; when we want to leave the original variable value in place.

          clrw      ; Set W = 0.

          call      add       ; A subroutine call. Why not?

          movwf     sum       ; Move the contents of W into address SUM. Since we returned
          ; from the subroutine with the sum of the array elements in W,
          ; we can just move what's in W into the address SUM. Had we
          ; performed other instructions in the subroutine that changed W,
          ; this wouldn't have been so easy.

finish     goto      finish    ; Here's our "HALT" statement again.

; *****
; Here's the subroutine. Operation will continue in this block of code until the RETURN instruction is executed,
; at which point the program will return to the line following the subroutine CALL.
; *****

add        addwf     INDF,w    ; Indirect addressing works as follows:
          ; 1. Look in FSR. The value that you find there is not the
          ; operand, but an address.
          ; 2. Look in the address whose value was found in FSR. The
          ; value that you find there is the operand.

update     incf      FSR,1     ; FSR = FSR + 1. Note that the incremented value of FSR is
          ; returned to address FSR. That way, each successive increment
          ; will operate on the same value. The next indirectly addressed
          ; instruction will end up taking the value in the next consecutive
          ; address.

          decfsz    count,f    ; COUNT = COUNT - 1. Note that the decremented value of
          ; COUNT is returned to address COUNT. That way, each
          ; successive decrement will operate on the same value until it
          ; hits zero.

goto      add        ; Since FSR has already been updated, if the loop is incomplete,
          ; we have to run the summing loop again.

```

```
return                ; If COUNT = 0, this line gets executed.  When COUNT = 0, it
                    ; means that we have finished adding all of the array elements.
                    ; We need to go back to the main routine and store the result
```

```
. *****
;
; The last line of executable code must be followed with the END statement.
. *****
```

```
end
```

7.3 IF-THEN-ELSE structure

```

; *****
;
;   Program Name: Comparison (IF-THEN) Example
;   J.S. Thweatt
;   06 April 2003
;   2504 Class Software Project 1
; *****

; *****
; Microchip-developed include file
;
; This include file is a set of equate statements that allows the use of address and symbolic labels such as
; STATUS for file memory location 0x04, and C for the carry bit of the STATUS register.
; *****

        include P16F84.inc

; *****
; Local equate table
;
; These are user-defined equate statements. They name symbolic variables that are specific to this program.
; *****

w          equ          0          ; DDB = 0 - store the result in w
f          equ          1          ; DDB = 1 - store the result in f

one        equ          0x11       ; The address of the first value
two        equ          0x12       ; The address of the second value

gflag     equ          0x13       ; A "greater than" flag
lflag     equ          0x14       ; A "less than" flag

; *****
; The ORG statement means that the first line of the executable program code after the ORG statement will be
; placed in program memory location 0, and that successive instructions will be placed in the next consecutive
; program memory location.
; *****

        org            0

```

```

; *****
; All IF-THEN statements are comparisons. The condition embodied by the IF either matches the desired
; condition, in which case the THEN portion of the statement is executed, or it doesn't, in which case an ELSE
; portion of the statement is executed, if one exists.
;
; There are many different ways that we can compare values; two useful ones are comparisons of magnitude
; (greater than or less than) and equality. This example will compare magnitude with the help of the CARRY
; bit of the STATUS register. A comparison of equality would work similarly, except that it might use the
; ZERO bit of the status register.
; *****

start      clrw                ; We start by setting W = 0.

           clrf      gflag     ; Both flags are initialized to zero.
           clrf      lflag

           movf      one,w      ; The first value is moved into W.

           subwf     two,w      ; The syntax of this instruction is "Subtract W from F." So here,
                               ; we are performing the operation [two] minus [one]. This is
                               ; easy to get confused, especially considering what happens next.

; *****
; STATUS register bits tell us things about what is happening as a result of the operations that are executed.
; Certain instructions can change certain bits of the STATUS register, and these conditions can be checked by
; looking at the appropriate STATUS register bit.
;
; Suppose that we want to know which of the two values is greater. We can make that judgment on the
; following basis:
;
;           If [two] minus [one] > 0, then [two] > one.
;           If [two] minus [one] < 0, then [two] < one.
;
; Normally, the CARRY bit works by indicating whether the result of an add operation included a carry-out of
; the most significant bit. If C = 0, no carry occurred, and if C = 1, a carry occurred.
;
; In the case of subtraction, this is *reversed*. The CARRY bit now becomes a BORROW bit. But, if C = 1,
; *no borrow* occurred, while if C = 0, a borrow did occur. With all of this in mind, we can make the following
; conclusion.
;
;           If [two] > [one], no borrow occurred (since the result of the subtraction was positive) and B = 1.
;           If [two] < [one], a borrow occurred (since the result of the subtraction was negative) and B = 0.
;
; Since we're working with a STATUS bit that records CARRY and BORROW events, it's worth noting that if
; [two] = [one], then no borrow occurs, and C = 1.
;
; Now, since the subtraction occurred in the previous instruction, all we have to do is test the CARRY bit (in its
; capacity as a BORROW bit) to see what happened.
; *****

```

```

; *****
; All of this means that the IF statement is: If [one] >= [two]
; *****

```

```

    btfsc    STATUS,C

```

```

    goto    tbig0      ; If C is not clear, we don't skip. That means that [two] > [one].
                    ; This is the ELSE.

```

```

    goto    obigt      ; If C is clear, we skip the previous line and execute this one.
                    ; That means that [one] >= [two]. This one is the THEN.

```

```

; *****
; Now, whatever happens, happens, because the routines TBIGO and OBIGT contain the two pieces of code that
; we might want to run for the two separate conditions.
; *****

```

```

tbig0    incf    gflag,1      ; We can make any indication that we want to the flag to show
                    ; that a given condition has been satisfied.

```

```

    goto    finish          ; Note that the conditions are mutually exclusive.

```

```

obigt    incf    lflag,1

```

```

finish   goto    finish      ; Halt execution

```

```

; *****
One last thing worth noting is that the order of the conditions after the bit test indicate which is the THEN and
; which is the ELSE. If you want to switch the condition being tested, just switch the order of the statements, or
; the type of bit test being executed, i.e, we could have also performed a bit-test-set (BTFSS).
; *****

```

```

; *****
; The last line of executable code must be followed with the END statement.
; *****

```

```

end

```