A High-Power PIC Macro Library

by Karl Lunt

The PIC microcontrollers pack a lot of power and speed in a tiny, inexpensive package, but the hobbyist is somewhat limited when it comes to writing software for them. The BASIC interpreter built into the Stamps is slow, the few high-level compilers can be expensive if you're on your own dime, and I've always found the assembler too weird to use.

But PIC assembly language runs so fast, and can be so small, that I wasn't willing to give up on using it. This article describes an extensive library of PIC macros that you can add to your own assembly language projects. Including this library with your PIC source file gives you the power of programming structures such as FOR-NEXT loops and greatly simplifies your code, all with minimal code bloat. Best of all, the library works with the standard Microchip MPASM assembler.

Using the library

You can find the library file — macros.asm — on my web site at www.seanet.com/~karllunt. I've included comments in the file to describe how to invoke each of the many macros. I also describe one or two problems that you'll need to watch for on the older PICs, such as the PIC 16c54.

Just copy the file into your working PIC directory. Next, edit your assembly language source file and add the following line somewhere near the top of the file:

```
include  ".\macros.asm"
```

The best place to put this line is immediately following the statement where you include the equates specific to the PIC chip you're using. For example, if your assembly source file targets the PIC 12c508, the first few lines of your source file might look like:

```
include  ".\p12c508.inc"
include  ".\macros.asm"
```

Note that the macro library assumes that you are using the Microchip equates file for your target MCU, and relies on some common register definitions. If you don't use the Microchip equates file, you will need to provide the required register definitions yourself.

With this change made, you can now invoke any of the supplied macros. Let's take a look at what is available in the macro library.

BEQ and BNE

These are simple, and I use them virtually everywhere. As implemented in the macro library, they check the state of the Z bit in the status register and branch accordingly. You use them immediately after performing some operation that alters the Z bit.

The BEQ macro is of the form:

```
beq      foo      ; if Z is set, branch
                  ; to foo
```

where foo is a label somewhere in your source file. If the Z bit is set when this macro is executed, control jumps to label foo. Otherwise, control continues with the next instruction in line.

The BNE macro is nearly identical:

```
bne      foo      ; if Z is clear, branch
                  ; to foo
```

where foo is a label somewhere in your source file. If the Z bit is clear when this macro is executed, control jumps to label foo. Otherwise, control continues with the next instruction in line.

FOR-NEXT

The FOR-NEXT family of macros provides an iterated looping structure. This structure gives you a simple way to perform a task a given number of times. The basic form of the FOR macro is:

```
for      var, begl, endl
```

where var is a variable you've chosen to use as the index, begl is a literal for the starting value of the index, and endl is a literal for the ending value of the index.

When the FOR macro first executes, it writes the literal value begl to the index variable var. At the top of the FOR loop, the macro tests the current value in var against the literal value endl; if they match, control exits the looping structure at the associated NEXT macro. If they don't match, control continues with the next statement in sequence. This macro makes it easy to write counted loops. If you need to issue a certain amount of pulses, you could use code such as:

```
for   n, 0, 60     ; need to issue 60
                   ; pulses
bsf   portb,1      ; raise a line
bcf   portb,1      ; drop a line
next  n            ; end of loop
```

Note that FOR-NEXT structures — like all of the macro structures in this library — can be nested as deep as you like. This allows you to do some fancy loops:

```
for      x, 0, 10     ; do 10 times
for      y, 30, 50    ; step across y
bsf      portb,3      ; raise a line
bcf      portb,3      ; drop a line
next     y            ; end of y loop
movlw    100          ; need to delay
                      ; now
call     delay        ; do the delay
next     x            ; end of x loop
```

As you can see, the macros save you from having to muck about with the W register and the flag bits. They also take care of all the labels and goto opcodes used in such operations. Thus, you get to focus more on what you want the program to do, and spend less time on how it gets done.

Note that the macro library assumes that you are using the Microchip equates file for your target MCU, and relies on some common register definitions.

The FOR macro uses a pair of literals; one for the starting value of the index and one for the ending value. Sometimes, however, you need to use a variable to hold the ending value. In such cases, you can use the FORF (for-flag) version of this macro:

```
forf     var, begl, endf
```

where var is a variable you've chosen to use as the index, begl is a literal for the starting value of the index, and endf is a variable that holds the ending value of the index.

This macro works just like the basic FOR macro above, with one important difference. When this macro tests the index variable to see if the loop should end, it uses the value in the variable endf. This means your code could modify the endf variable, resulting in a FOR-NEXT loop that runs a variable number of iterations.

A typical use of this structure might be:

```
forf   n, 0, steps   ; need to issue
                     ; some pulses
bsf    portb,1       ; raise a line
bcf    portb,1       ; drop a line
next   n             ; end of loop
```

where the actual number of pulses to issue isn't known when you write the assembler source, but is held in the variable steps when the program runs.

Both versions of the FOR macro above end with a NEXT macro that refers to the same variable name. The basic NEXT macro looks like this:

```
next     var
```

where var is the same index variable used in the matching FOR or FORF macro.

It is important that you match the FOR or FORF index variable with the corresponding NEXT index variable. The NEXT macro adds one to the index variable and loops back to the matching FOR macro so the variable can be tested. If the variables don't match, the FOR index variable will never change and the loop will never end.

In some cases, you need to add more than one to the index variable at the NEXT macro. This is similar to BASIC's FOR-NEXT-STEP structure. I've included the NEXTL macro to provide that capability. This macro looks like:

```
nextl    var, incl
```

where var is the same index variable used in the matching FOR or FORF macro, and incl is a literal that is added to the index variable. For example, the following loop steps through several odd integers:

```
for      n, 1, 31  ; start of loop,
                   ; n = 1
nextl    n, 2      ; end of loop, add
                   ; 2 to n each time
```

NOTE: The NEXTL macro contains an addlw instruction, one of the newer PIC opcodes. Thus, older devices — such as the 16c54 — cannot execute this macro. If you assemble a source file for the 16c54 or similar processors, and the assembler detects this instruction, it will issue an assembler error.

To complete the package, the NEXTF macro works just like the NEXTL macro, but it instead adds the value in a variable to the index variable:

```
nextf    var, incf
```

where var is the same index variable used in the matching FOR or FORF macro, and incf is a variable whose contents are added to the index variable. This lets you create loops that execute a variable number of times:

```
for      n, 1, 31  ; start of loop,
                   ;n = 1
nextf    n, steps  ; end of loop, add
                   ; steps to n each
                   ; time
```

REPEAT structures

In some cases, you need to create

Reprinted from July 1999 Nuts & Volts Magazine. All rights reserved. No duplication permitted without permission from T & L Publications, Inc.

A High-Power PIC Macro Library

a conditional loop. That is, you need a structure that loops until a specific condition exists or ceases to exist. For that, I've built the REPEAT series of structures. The REPEAT macro marks the beginning of the structure, and is of the form:

```
repeat
```

Exactly what type of structure you create depends on what macro you use to match up with this REPEAT macro. The simplest such structure is the unconditional loop, built from the REPEAT and ALWAYS macros. Here, control endlessly executes the code between the REPEAT and its matching ALWAYS macro. For example:

```
repeat              ; start an endless
                    ; loop
bsf       portb,1   ; raise a line
bcf       portb,1   ; drop a line
always              ; do forever
```

You can build a conditional loop using the UNTILEQ macro with the REPEAT macro. This combination yields:

```
repeat
untileq
```

Now control executes the code between the REPEAT and UNTILEQ macros until the Z bit in the status register is set when the UNTILEQ macro executes. The instructions just prior to the UNTILEQ should perform some operation to test for the ending condition, with the ending condition signaled by setting the Z bit. For example:

```
repeat              ; start a loop
movfw     portb     ; read a port
andlw     0x20      ; leave only bit 5
untileq             ; loop until bit 5 is
                    ;low
```

Here, the Z bit will be set when the W register is 0 after executing the AND operation; that is, when bit 5 of portb is 0.

The macro library provides the matching and opposite function with the UNTILNE macro. It works just like the UNTILEQ macro, except that control loops until the Z bit is cleared when the UNTILNE macro executes. The above example could be rewritten as:

```
repeat              ; start a loop
movfw     portb     ; read a port
andlw     0x20      ; leave only bit 5
untilne             ; loop until bit 5 is
                    ; high
```

Now the loop repeats until bit 5 of portb goes high. This condition leaves the Z bit cleared when the UNTILNE macro executes, and control leaves the loop.

SELECT-CASE structures

The SELECT-CASE structure acts as a large switch table, allowing your code to take one of several paths, based on the value of a selector variable. The general format of a SELECT-CASE structure looks like this:

```
select
case
 .
 .
endcase
case
 .
 .
endcase
 .
endselect
```

This general description shows how a SELECT statement marks the beginning of the structure. If the selector variable holds the value called for in the first CASE statement, then code between that CASE statement and its matching ENDCASE statement is executed. If not, then the selector variable is tested against each successive CASE value in turn.

Exactly what type of structure you create depends on what macro you use to match up with this REPEAT macro.

If the selector variable does not match any CASE value, the code immediately following the last ENDCASE statement, if any, is executed as a default. After executing any CASE block of code, control passes immediately to the matching ENDSELECT statement, skipping over any intervening CASE blocks.

This creates a very powerful structure, ideally suited for many robotic applications, where functions to be performed depend on the value in some global state variable. As implemented in the macro library, the SELECT macro is simply:

```
select    var
```

where var is the selector variable. When the SELECT macro executes, the underlying code copies the value in variable var into the PIC's W register.

The CASE macro actually tests the contents of the W register; the macro looks like this:

```
case              lit
```

where lit is a literal value used for testing. When the CASE macro executes, it compares the literal value lit against the contents of the W register. If they match, the code immediately following the CASE statement is executed. If, however, they don't match, control passes to just below the matching ENDCASE statement.

Note that regardless of whether the test passes or not, the contents of the W register are preserved. Thus, code following the CASE statement may rely on the W register containing the literal value called out in the CASE statement.

The ENDCASE macro marks the end of a CASE-ENDCASE structure. This macro looks like:

```
endcase
```

There must be a matching ENDCASE macro for each and every CASE macro. When code inside a CASE-ENDCASE block hits the ENDCASE macro, control passes immediately to the ENDSELECT macro that closes out the current SELECT-ENDSELECT structure. The ENDSELECT macro marks the end of a SELECT-ENDSELECT structure. This macro looks like:

```
endselect
```

There must be a matching ENDSELECT macro for each and every SELECT macro. If any code exists between the final ENDCASE macro and an ENDSELECT macro, that code is treated as a default case. This means that any time a selector value fails all CASE tests, the code after the final ENDCASE is executed.

An example should make all of this clearer:

```
select    foo       ; use foo as the
                    ; selector variable
case      5         ; if foo = 5...
bsf       portb, 1  ; raise a line
endcase             ; all done
case      8         ; if foo = 8...
bcf       portb, 1  ; drop the line
endcase             ; all done
incf      count     ; default, not 5 or
                    ; 8, count it
endselect           ; end of select
                    ; structure
```

Here, I've used the variable foo as the selector value. The SELECT macro copies foo into the W register. The first CASE statement compares W to the literal 5. If they match, the code sets bit 1 of portb, then jumps to the ENDSELECT macro. If they don't match, the second CASE statement compares W to the literal 8. If they match, the code clears bit 1 of portb, then jumps to the ENDSELECT macro. If they don't match, then foo contained neither 5 nor 8.

In this case, the default code following the last ENDCASE is executed, incrementing the variable count. Finally, control falls through to the ENDSELECT macro.

The macro library contains an alternate form of the CASE macro, useful when you need to test the selector value against another variable, rather than a literal. The CASEF macro looks like:

```
casef     var
```

where var is a variable whose contents are compared against the W register. This means you can write CASE structures such that the test values aren't known at assembly time, but are created in the code at run time.

WAITWHILE and WAITUNTIL

These macros create very small, very fast loops that block, or wait, until a specific condition exists or ceases to exist. They are ideal for monitoring one or more I/O lines for an input condition. The WAITWHILE macro looks like:

```
waitwhile         addr, andl, xorl
```

where addr is the port register to monitor, andl is a literal value used as an AND mask, and xorl is a literal value used as an exclusive-or (XOR) mask.

This macro creates a small loop that reads the value in the port register at address addr, ANDs that value with the literal andl, then XORs the result with the literal xorl. This sequence of operations repeats for so long as the final result is non-zero. Control does not leave the WAITWHILE macro until the final result is zero. At that time, control passes to the statement following the WAITWHILE macro.

The use of the andl as an AND mask should seem obvious. It isolates only those bits in the port register value that match bits in the andl literal. A mask value of $0f, for example, lets your code test the lower four bits of an address without caring what happens to the upper four bits.

The XOR mask may not seem so obvious. This effectively inverts the state of selected bits, allowing your code to test for active-low inputs. For example, suppose bit 2 of the port is active-low, and your code needs to wait while that bit is 0. Using an XOR mask with bit 2 set ($04) inverts that bit, yielding a logic 1 when the input is a logic 0. An example might help:

```
waitwhile         portb, 0x02, 0x02
```

This WAITWHILE macro reads the value in register portb, ANDs that value with $02 to leave only bit 1 intact, then XORs that value with $02 to invert the state of bit 1. If the resulting value is non-zero, control repeats the WAITWHILE macro. Otherwise, control falls through to the statement following the WAITWHILE.

This macro library makes heavy use of the powerful macro operators built into the Microchip MPASM assembler.

The opposite of WAITWHILE is WAITUNTIL, which loops until a certain condition is non-zero. The WAITUNTIL macro looks like:

```
waituntil         addr, andl, xorl
```

Reprinted from July 1999 Nuts & Volts Magazine. All rights reserved. No duplication permitted without permission from T & L Publications, Inc.

A High-Power PIC Macro Library

This macro creates a small loop that reads the value in the port register at address addr, ANDs that value with the literal andl, then XORs the result with the literal xorl. This sequence of operations repeats until the final result is non-zero. Control does not leave the WAITUNTIL macro until the final result is non-zero. At that time, control passes to the statement following the WAITUNTIL macro.

Note that if you use a value of 0 for the xorl literal in either of the above macros, the macro does not generate any PIC code for the XOR operation. XORing a value with 0 leaves that value unchanged, so there is no point in generating code for that operation.

POLL and ENDPOLL

The WAITWHILE and WAITUNTIL macros create fast blocking loops, but your code cannot perform any operations inside the loops. The POLL-ENDPOLL structure lets you perform functions inside an I/O polling loop. The POLL macro looks like:

```
poll      port, andl, xorl
```

where port is the address of a port register to monitor, andl is a literal value used as an AND mask, and xorl is a literal value used as an XOR mask.

This macro acts just like the front end of the WAITWHILE or WAITUNTIL macros. It reads the value at the address port, ANDs that value with the literal andl, then XORs the result with the literal xorl. If the resulting value is true (non-zero), control falls through to the next instruction in sequence. If, however, the resulting value is false (zero), control jumps to just below the matching ENDPOLL macro.

Each POLL macro must be paired with an ENDPOLL macro. The ENDPOLL macro looks like:

```
endpoll
```

When control reaches the ENDPOLL macro, it returns automatically to the previous matching POLL macro.

The use of the andl as an AND mask should seem obvious.

Thus, the POLL-ENDPOLL structure lets your code monitor, or poll, a set of I/O lines for a specific condition. If that condition occurs, your code can take appropriate action, as defined inside the POLL-ENDPOLL structure. For example:

```
poll      portb, 0x80, 0x80
incf      count
endpoll
```

Here, the POLL macro tests for a low on bit 7 of portb. If that bit is low, then the variable count is incremented. If, however, that bit is high, control passes directly to the ENDPOLL macro and count is not incremented.

Under the hood

This macro library makes heavy use of the powerful macro operators built into the Microchip MPASM assembler. The following paragraphs will walk you through the design of one of the macros, so you can see how I built it. You can then apply these techniques to create your own macros.

Here is the code specific to the FOR macro. I have added line numbers for reference only; they do not appear in the macro source file:

```
1.          variable  _forknt=0
2.          variable  _nxtknt=0

3. for      macro     var,begl,endl
4.          movlw     begl
5.          movwf     var
6. _for#v(_forknt)
7.          movlw     endl
8.          subwf     var,w
9.          beq       _next#v(_forknt)
10. _forknt   set     _forknt+1
11. _nxtknt   set     _forknt
12. endm
```

Lines 1 and 2 define two assembler variables that will be used by both the FOR and NEXT macros. Note that these are NOT variables used by your PIC program when it runs. These variables exist only while MPASM assembles your source, and they will only be used by MPASM. I intentionally use a leading underscore on all of my MPASM variable names, to avoid conflicts with PIC variables that you might declare in your program.

Line 3 declares the format of the FOR macro, as required by MPASM. Here you can see that the FOR macro requires three arguments, and you can see the names that they will be given throughout the FOR macro. MPASM is smart enough to know that a macro argument, such as begl, is different from a variable or equate that you have declared elsewhere in your source file.

Lines 4 and 5 copy begl, the literal value used as a starting index, to the index variable var. This code is executed once, when control enters the FOR macro at run-time.

Line 6 shows one of the powerful features of the MPASM macro operators. This line creates an assembler label composed of the characters "_for" followed by the characters for the current value of the assembler variable _forknt. Thus, if _forknt holds the value 3, line 6 will assemble as:

```
_for3
```

Note how the #v() macro operator reads the value of an assembler variable and adds that value to the end of a label. For more details on using the #v() macro operator, consult the MPASM Assembler User's Guide from Microchip.

Lines 7 through 9 test the current value of var against the ending literal value endl. If var matches endl, then the PIC's Z-bit will be set. The beq macro at line 9 will either pass control outside the FOR-NEXT loop if the Z-bit is set, or allow control to fall through to the next line of code if the Z-bit is clear. Note how line 9 again uses the #v() macro operator to build up the label used for the beq target. Here, the label consists of the string "_next" followed by the value in the assembler variable _forknt. Thus, if _forknt holds the value 4, then line 9 will assemble as:

```
beq       _next4
```

Line 10 increments the value in assembler variable _forknt, so the labels created at the next use of the FOR macro will differ from those just created. This ability to modify the values in the assembler variables is another powerful feature in the MPASM macro utility, and is essential to the proper functioning of this library.

Similarly, line 11 changes the value in assembler variable _nxtknt. The FOR macro doesn't use this variable in creating any labels, but it must perform the bookkeeping so the matching NEXT macro does create the correct label. Remember that the NEXT macro must generate a branch back to the top of the FOR macro, at the label in line 6. This adjustment of the assembler variable _nxtknt ensures that the correct branch label will be created.

Finally, line 12 contains the endm psuedo-op, used to indicate the end of a macro definition. This explanation is still pretty theoretical. It may not be clear yet how all of this comes together when the assembler actually processes a FOR macro. Let's finish up with a specific example. Assume that at some point in the assembly of your program, assembler variables _forknt and _nxtknt both hold the value 3. Your program now contains the FOR macro:

```
for       foo, 4, 20
```

Given the above, MPASM will create the following assembler source lines for your macro:

I've presented a suite of powerful PIC macros that you can add to your own assembler source files.

```
4.          movlw     4
5.          movwf     foo
6. _for3
7.          movlw     20
8.          subwf     foo,w
9.          beq       _next3
10. _forknt   set     _forknt+1
11. _nxtknt   set     _forknt
```

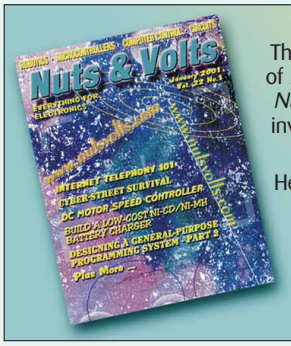
I have eliminated all the setup lines from the above FOR macro explanation, leaving only those actually used by MPASM when it processes your FOR macro. Note that the beq opcode in line 9 is itself a macro, so MPASM will expand it as well. I have left the beq unexpanded for clarity.

This example should make clear the power behind the MPASM macro utilities and this macro library. You don't need to match up the labels in the FOR macro with the labels in the corresponding NEXT macro. You don't have to worry that you might have used the ending literal in the FOR initialization, rather than the starting literal. All you have to do is write the FOR macro and match it up with a NEXT macro, and you're done.

That's a wrap

I've presented a suite of powerful PIC macros that you can add to your own assembler source files. These macros provide most of the common programming structures, such as FOR-NEXT and REPEAT-ALWAYS. Additionally, I've added structures such as WAITWHILE and POLL-ENDPOLL that simplify writing embedded control code, where monitoring I/O lines occurs frequently.

I hope you'll look on these macros as a beginning, not an end. Feel free to expand what I've done here. By studying my use of the assembler variables and the #v() macro operator, you should be able to create complex macros of your own. And, as your use of control structure macros increases, writing PIC assembly language programs should become simpler and less frustrating. **NV**



Shameless Plug

This article was reprinted from the July 1999 issue of *Nuts & Volts Magazine*. If you aren't familiar with *Nuts & Volts* and are interested in electronics, we invite you to stop by our website and see what it's all about. (Even if you have seen it, stop by anyway.)

Heck, we'll even send you a sample copy for free if you've never seen it before. And of course, you can also subscribe for only $19.00.

What a deal!

The details are on our site.

www.nutsvolts.com

Reprinted from July 1999 Nuts & Volts Magazine. All rights reserved. No duplication permitted without permission from T & L Publications, Inc.