

Lesson 6

Subroutines

Overview

Introduction	Often we have logic we wish to use several places in our programs. The subroutine provides a way to do this effectively. In this section, we will review subroutines, and look at delay timers, which is one example of where we may use subroutines.																			
In this section	Following is a list of topics in this section:																			
	<table><tr><th>Description</th><th>See Page</th></tr><tr><td>Why we want subroutines</td><td>2</td></tr><tr><td>Why we don't want subroutines</td><td>3</td></tr><tr><td>Stack</td><td>4</td></tr><tr><td>Instructions for building subroutines</td><td>5</td></tr><tr><td>Let's try this out</td><td>6</td></tr><tr><td>Using the Subroutine</td><td>8</td></tr><tr><td>Timing Loops</td><td>12</td></tr><tr><td>Wrap Up</td><td>14</td></tr></table>	Description	See Page	Why we want subroutines	2	Why we don't want subroutines	3	Stack	4	Instructions for building subroutines	5	Let's try this out	6	Using the Subroutine	8	Timing Loops	12	Wrap Up	14	
Description	See Page																			
Why we want subroutines	2																			
Why we don't want subroutines	3																			
Stack	4																			
Instructions for building subroutines	5																			
Let's try this out	6																			
Using the Subroutine	8																			
Timing Loops	12																			
Wrap Up	14																			

Why we want subroutines

Introduction	In this section, we will take a look at the advantages of framing our code as subroutines.
Rework	In developing programs, we often find that the same kinds of logic appears over and over again. By placing this logic in a subroutine, we only need to write the code once. It can then be used over and over within our program.
Reliability	Subroutines, when well thought out, are standalone little packages of logic. As such, they can be tested independently, and we can be assured that they do what we expect. We can then use them over and over with confidence, and without the need to constantly debug them.
Readability	Assembler programs tend to get somewhat long and gangly. By breaking our logic into functionally complete modules, we can make our programs a lot easier to understand.
Debugging	By breaking our code into logical modules, we can debug them more or less independently. This greatly improves the debugging process, as it helps us to focus on one small area. It also gives us a degree of confidence that the other parts of the program can be ignored for the time being.
Memory	Since this logic we are re-using appears only once in program memory, subroutines can help reduce the program memory demands of our application.

Why we don't want subroutines

Introduction	While subroutines have their place, there are reasons to avoid their use. Here are just a few.
Complexity	If we are going to make maximum use of subroutines, our logic needs to be made a little more flexible. Depending on the particular situation, this can mean our logic is more complex than it might be, had we used to same algorithm in line.
Performance	<p>Whenever we call a subroutine, we need to have the initial logic we might have put in line, plus, we need to call the subroutine and get back from it. While this overhead isn't great, it is overhead, and it can be an issue in a timing sensitive situation. The total penalty amounts to 4 microseconds with a 4 MHz crystal, or 800 ns with a 20 MHz PIC.</p> <p>The other issue is that the complexity we mentioned above can lead to reduced performance, as well. This depends on our logic, of course, but it can also be a factor in places where we are pressed for cycles.</p>
System Limitations	Some of the real power of subroutines comes from having a subroutine call another subroutine, which may call another subroutine. This nesting of subroutines can have a dramatic impact on the amount of code we need to write. Unfortunately, the PIC has a has a stack only eight elements deep, which means we can only nest our subroutine calls seven or eight deep. While this isn't a real common problem, we do need to be cognizant of just how deeply we are nesting subroutines.

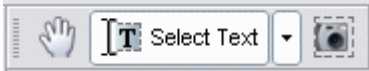
Stack

Introduction	<p>Almost all computers have a special construct called a stack. In some computers, the stack is implemented in the main memory, and there are special pointers to allow for stack specific instructions. In the case of the PIC, the stack is a special hardware memory. Because of the Harvard architecture (remember this from lesson 1?), there are no instructions for directly manipulating the stack. There are instructions which do affect the stack, but we can't actually look at the stack programmatically. (The PIC18 series of microcontrollers do include instructions for directly manipulating the stack.)</p>
What is a stack	<p>A stack is a special memory which is addressed in a special way. Instead of accessing the stack by an address, like other memory, a stack can only be accessed at the "top". The model is very much like a stack of paper. You can "push" data onto the top of the stack, or "pop" data off the top of the stack, much like you would place a piece of paper onto the top of a pile, and remove it from the top.</p> <p>However, unlike a pile of paper, you cannot slide a piece of data into the middle of the stack, nor can you reach in and slide a piece of data out of the middle. You can only access the stack from the top.</p>
The PIC Stack	<p>In the PIC16 parts, the stack is 13 bits wide by 8 words deep. This means that the stack can hold eight program memory locations, and can address the program memory space of any of the PIC16 parts.</p>
Instructions affecting the stack	<p>The PIC16 instruction set contains no instructions for directly manipulating the stack or the stack pointer. However, there are a few instructions which affect the stack. These are: <code>call</code>, <code>return</code>, <code>retlw</code>, and <code>retfie</code>. In addition, an interrupt affects the stack.</p> <p>We will talk about interrupts and the <code>retfie</code> instruction later in the course. In this lesson, we will examine the <code>call</code>, <code>return</code>, and <code>retlw</code> instructions.</p>

Instructions for building subroutines

Introduction	Here we will examine the three instructions that we will use to create subroutines; the <code>call</code> , <code>return</code> , and <code>retlw</code> instructions.
The call instruction	Whenever we want to use a subroutine, we execute a <code>call</code> instruction. The <code>call</code> instruction is just like a <code>goto</code> instruction, except that before jumping, the current program counter is pushed onto the stack. Since the program counter is incremented as soon as an instruction is loaded by the PIC, this value is one higher than the program counter value for the <code>call</code> .
The return instruction	The <code>return</code> instruction does the opposite. The <code>return</code> instruction loads the value at the top of the stack into the program counter. This causes the next instruction to be executed to be the instruction after the original <code>call</code> .
The retlw instruction	Quite often, we would like a subroutine to not only do something, but to return some result to the calling program. This is exactly what the <code>retlw</code> instruction does. This instruction loads a literal value into the working register, and then behaves just like a <code>return</code> instruction.
Putting this together	<p>The <code>call/return</code> pair provides a way to jump out of our main program, do something, and then come back to where we left off. The structure typically looks something like the following:</p> <pre>Do some stuff call MySub Do some other stuff ~ ~ ~ ~ MySub Do some stuff return</pre> <p>Typically, the subroutines are grouped together at the start of the program file, and a <code>goto</code> instruction skips around them to the start of the main logic. There is no rule that says it has to be this way, but there are times when it is convenient for subroutines to be near the front, and keeping all the subroutines together makes for greater readability.</p>

Let's try this out

Introduction	In order to start this new experiment, guess what? We need to create a new project (Lesson6a), and a new assembler file (Lesson6a.asm). Just like we did before.
Our first subroutine	<p>Insert the following code into the assembler file:</p> <pre> processor PIC16F84A include <p16f84a.inc> __config _XT_OSC & _WDT_OFF & _PWRTE_ON cblock H'20' Spot1 ; A variable to play with endc goto Start ; Skip to mainline ; Our subroutine begins here Sub1 incf Spot1,F return ; Here is the start of the mainline Start movlw H'fc' ; Put something in movwf Spot1 ; Spot1 Loop call Sub1 ; Call the subroutine goto Loop ; Do it again end </pre> <p>You should be able to cut and paste the above code, although you may have to clean up the tabs a bit. Use the I-beam tool at the top of your PDF reader:</p> 
Running the Program	<p>After assembling the program, open the Stack window (View->Hardware Stack). Pressing F7 three times should bring you to the <code>call</code> instruction. Notice that the stack window hasn't changed. Also notice that the program counter at the bottom of the window shows pc:0x5. Press F7 once more. A lot happens. The program counter goes to 1, and a 6 gets placed on the stack (5 plus 1). The next step increments a file register location, but the next one returns to the instruction after the call (pc:0x6), and the stack pointer returns to point to 'Empty'.</p> <p>Notice that the stack now contains two 6's. This is actually trash. Whatever is on the stack below the stack pointer (the green arrow) simply doesn't matter. If we do a call, we'll overwrite what is there, and we can't do a return because there is nothing left to pop.</p>

Continued on next page

Let's try this out, Continued

Nesting Subroutines

Let's examine what happens when we call one subroutine from another. Change our subroutine to look like the following:

```
;      New subroutine begins here
Sub2
        incf      Spot1,F
        return
;      Original subroutine begins here
Sub1
        call      Sub2
        return
```

And again, assemble it.

The new code has moved our original `call` instruction down to location 7, so the `call` places an 8 on the stack. The new `call` is at location 3, so a single step puts a 4 at the top of the stack. Notice that everything else is pushed down.

The next step increments `Spot1`, nothing very exciting there, but yet another step executes the `return`, changing the program counter to 4, and removing the 4 from the top of the stack. Clicking again, removes the 8 (which has now risen to the top) and sets the program counter to 8.

We can nest subroutines like this, and make successive `calls`, and the stack mechanism will keep track of where we came from with little worry. However, we can never be more than 8 levels deep or the stack will overflow (and our program will do strange things!)

Using the Subroutine

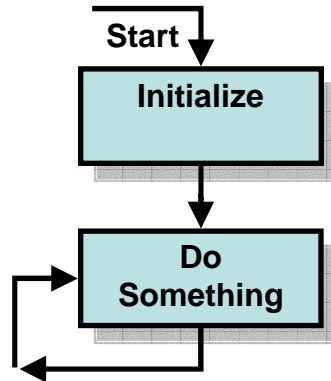
Introduction

We said earlier that subroutines can help us break our program into manageable pieces. Let's do an example.

Begin, as usual, by creating a new project, Lesson6b.

Program Structure

PIC programs are almost always intended to do the same thing over and over again. As a result, our programs almost always look a little like:



So, we can start almost every program out with code that looks something like:

```

processor    pic16f84a
include      <p16f84a.inc>
__config     _XT_OSC & _WDT_OFF & _PWRTE_ON

goto         Start ; Skip to mainline

Start
Loop
goto         Loop
end
  
```

OK, certainly we are going to end up with a `cblock` at the front, too.

Continued on next page

Using the Subroutine, Continued

The main loop

Let's suppose that we want to write a program for the PIC-EL which is going to send the word TEST in Morse code over and over again out the transmitter port. With what we have learned so far, that may seem like a pretty tall order. But by breaking the problem down into small, logical pieces, and breaking those pieces down again and again, we can eventually get to a point where we can envision the code.

We might imagine that our mainline is going to look a bit like the following:

```
Loop
    call    SendT
    call    SendE
    call    SendS
    call    SendT
    call    WordSpace
    goto    Loop
```

Getting from the concept to the code is the second hardest part of developing PIC applications. The subroutine idea can go a long way to helping us.

(The hardest part is coming up with the concept in the first place!)

Sending the letters

OK. Now we need to send the letters. Well, again, if we think about this problem at a high enough level, it's pretty simple:

```
;      Send the letter T
SendT
    call    Dah
    call    LetSpc
    return

;      Send the letter E
SendE
    call    Dit
    call    LetSpc
    return

;      Send the letter S
SendS
    call    Dit
    call    Dit
    call    Dit
    call    LetSpc
    return
```

We've gone down another level, and we've assumed we're going to write a few more routines; one to send a dit, one to send a dah, and one to wait for the period of time we want between letters.

Continued on next page

Using the Subroutine, Continued

Sending the elements

Now we need to figure out how to send the elements that make up the Morse letters. That shouldn't be so tough. Maybe something like:

```
;      Send a Dah
Dah
        call    XmitOn
        call    DahTime
        call    XmitOff
        call    DitTime
        return
;      Send a Dit
Dit
        call    XmitOn
        call    DitTime
        call    XmitOff
        call    DitTime
        return
```

This is starting to get pretty involved, but by applying subroutines, we really haven't done anything particularly hard.

Keying the Transmitter

OK, now it's time for the big disappointment. Here we are, rolling along, but we're not ready, just yet, to talk about manipulating the PIC's I/O pins. So for now, we're simply going to manipulate a single bit in memory instead of the transmitter.

First we need a word to store that bit:

```
        cblock    H'20'
                Output
        endc
```

and we're going to define the particular bit we set:

```
XMTR    equ    H'07'
```

And we need to initialize the `Output` variable:

```
Start
        clrf      Output    ; Init output off
```

And we need routines for turning the transmitter (bit) on and off:

```
;      Turn on the transmitter
XmitOn
        bsf      Output, XMTR
        return
;      Turn off the transmitter
XmitOff
        bcf      Output, XMTR
        return
```

Continued on next page

Using the Subroutine, Continued

Element timing

Now all we have left are subroutines to set the time between our various elements. We want to define all of these from the dit time, so we want some routines like the following:

```
;      Delay a dit time
DitTime
        nop
        return

;      Delay a dah time
DahTime
        call      DITTIME
        call      DITTIME
        call      DITTIME
        return

;      Delay a letter space
LetSpc
        call      DahTime      ; OK, too long
        return

;      Delay a word space
WordSpace
        call      DahTime
        call      DahTime
        return
```

For now, we are making our dit time pretty short ... a call plus a nop plus a return is only 5 cycles, or just over 1 microsecond at 20 MHz. We'll work on that later.

Testing

Now, if you assemble the program and run it, you can watch the single bit in the file register memory (H'20') flash out T E S T in animate mode. (You may want to download the program from amqrp.org rather than typing).

Timing Loops

Introduction

It's an odd thing, for a lot of our programs, the PIC will spend most of it's time wasting time. Here we are going to look at how to waste time!

Make yet another project, Lesson6c, but instead of adding an empty Lesson6c.asm file, copy Lesson6b.asm to Lesson6c.asm and add the filled up Lesson6c.asm to your project.

Counting the hours

Let's see if we can burn enough time with a simple loop. By my calculations, 20 WPM code means about 55 milliseconds per dit. Add a new variable, `L1`, and change `DitTime` to look like the following:

```
DitTime
        movlw      H'00'
        movwf      L1
DitTime1
        decfsz     L1,F
        goto       DitTime1
        return
```

This will spend a lot of time looping around in `DitTime1`. Now, we could calculate how long this loop will take. Each instruction takes 4 cycles at the crystal frequency, unless that instruction changes the program counter, in which case, it will take twice that. As an example, every time through the loop, the `decfsz` instruction will take four cycles except the last time, when it does the skip. On the last loop, the `decfsz` instruction will take 8 cycles.

If we are using a 4 MHz crystal (as we are in the PIC-EL), one instruction, four cycles, conveniently works out to 1 microsecond.

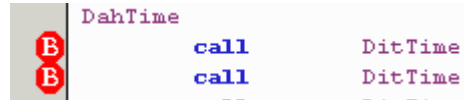
The MPLAB IDE includes a stopwatch feature, however, which allows us to simulate how long a particular piece of code will take.

Continued on next page

Timing Loops, Continued

Stopwatch

Assemble the modified program. In the `DahTime` subroutine, add a breakpoint before each of the `call DitTime` calls.



Now, Select Debugger->Stopwatch from the menu. Check that the processor frequency is shown as 4 MHz. If not, select Debugger->Settings and set the processor to 4 MHz on the Clock tab.

Run the program to the first breakpoint. Click Zero on the stopwatch. Now run the program until the next breakpoint. The stopwatch will measure how long it will take to call the `DitTime` routine and return. 773 microseconds sounds like a long way from 55 milliseconds.

What if we put another loop inside of our first loop? This way we can waste a lot of time a lot of times!

Just after `DitTime1` try something like:

```
        movlw    H'00'
        movwf    L2
DitTime2
        decfsz   L2,F
        goto     DitTime2
```

Running the experiment again gives us something like 197 msec. Too slow, but at least we're on the same planet. If we change our outer loop constant to `H'47'`, though, we will come in to our 20 word per minute time.

In this case, we really only wanted one timer, and everything else was based off of that. In some other cases, we may want a timer to click off, say, a millisecond, and have other timers of 100 msec., 1 second, 1 minute, etc.

Wrap Up

Summary

In this lesson, we have looked at subroutines, and we have studied how the subroutine concept can help us break a fairly complex problem down into small, manageable pieces.

We also took a look at timing loops, and got an idea about how an actual application might look.

Coming Up

In the next lesson, we are going to go back and revisit the status word to see how it can help us overcome the limitation of only having variables with 256 possible values.
