Lesson 14
Tables

Overview

Introduction

Frequently, an application needs some sort of table. There are several ways of implementing tables depending on the application requirements. In this chapter we examine some of the more common approaches.

In this section

Following is a list of topics in this lesson:

The Program Counter

Introduction

The most common type of PIC table requires that we have a much better understanding of the program counter than we have needed previously.

A Size Problem

In this section, we are going to depart from our custom of talking solely about the PIC16F84, and talk about some of the larger memory parts. The reason is that some of the handling of the program counter makes more sense when you consider larger PICs. There are some issues that can be ignored for the 16F84, but without seeing what happens when program memory exceeds 2K, some of the behavior seems a little odd.

The PIC program counter is 13 bits wide. On the PIC16F84, only the lower 10 bits are decoded since the F84 has only 1K of program memory. In other words, any address above H'3ff' has its high order bits dropped. On PICs with 2K of program memory, such as the PIC16F628, 11 bits are decoded. On PICs with 8K, all 13 bits are used.

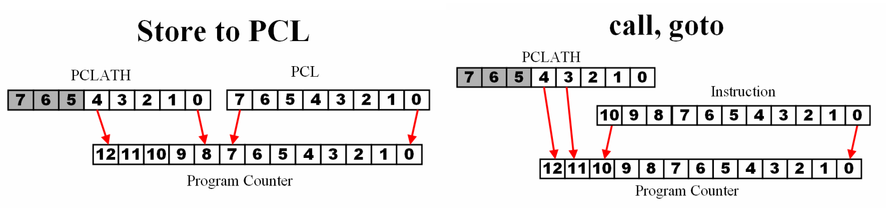
If you look carefully at the instruction formats in the datasheet, you will notice that the `call` and `goto` instructions only have 11 bits available for an address. This means that the target of a `goto` must be within 2K. Obviously, this could be a problem for PICs with more memory.

But there is a more subtle problem. Often, as we will see later in this lesson, we would like to be able to calculate an address to execute based on something in our application. However, the PIC only deals with 8 bit data. While we have seen how to do arithmetic on larger numbers, the program counter presents a special case. As soon as we store the first byte into the program counter, the next instruction will be fetched and executed. However the address used will be one byte of new address and the other byte will be from the old address. Thus we need a technique to write TWO bytes simultanously to the program counter. The solution is using a special register called PCLATH.

PCL and PCLATH

The programmer can change the low 8 bits of the program counter directly by writing to PCL. However the high 8 bits of the program counter are not writeable. There is a special register, PCLATH, (**P**rogram **C**ounter **LAT**ch **H**igh) which contains the high 8 bits of the PC. PCLATH is set to zero on a processor reset, and thereafter can only be changed by the programmer.

Whenever the programmer writes to PCL, the rightmost 5 bits of PCLATH are transferred to the high 5 bits of the program counter. Whenever a `call` or `goto` instruction is executed, bits 3 and 4 of PCLATH are transferred to bits 11 and 12 of the program counter.



Tables in Program Memory

Introduction

For most uses of tables, we would like to program the table into the PIC and then leave it unchanged. Since the file register contents are unknown at power up, that leaves us with program memory or EEPROM. EEPROM is fairly clumsy, as we will see later, so program memory is a nice choice.

RETLW

The PIC includes an instruction, `retlw`, which means return with a literal in W. This allows the programmer to specify the contents of W on return from a subroutine. Although this may seem at first to be a sort of odd instruction, it can be very valuable for implementing a table.

Structure of a table

In general, when we need a table, we have some value where we want to look up some alternate representation. For example, we may have an angle and want to look up its sine, or we may have a letter, and we want to look up its Morse equivalent.

If the table is going to be maintained in the program memory, this implies that we are going to take our index and use it to perform some arithmetic resulting in an address to some code that will give us the desired result.

Since the W is the only convenient register we have for passing results into a subroutine, in the simplest of cases, the calling program will pass the index in via the W register. All that remains for the subroutine, then, is to do arithmetic on that value, and jump to code that will return the right result.

Suppose we look at an extremely simple case:

```
Sub                         ; Multiply by 43
        addwf   PCL,F
        retlw   D’0’
        retlw   D’43’
        retlw   D’86’
        retlw   D’129’
        retlw   D’172’
        retlw   D’215’
Main
        movf    Value,W
        call    Sub
```

Let’s look at what is happening here. In `Main`, we load the W register with a value, which must be between zero and 5. We then call `Sub`. The first instruction in `Sub` adds the contents of the W to the program counter, essentially skipping over as many instructions as the value in W.

Continued on next page

Tables in Program Memory, Continued

Structure of a table (continued)

Suppose that `Value` contained a 3. We would load 3 into the `W` register, then call `Sub`. Remember that when we execute an instruction, first the instruction is read into the CPU. Then the program counter is incremented, and finally the instruction is executed. The call to `Sub` causes the program counter to contain the address of the `addwf PCL,F` instruction so it is fetched into the CPU. Next, the program counter is incremented, so it contains the address of the `retlw D'0'` instruction. We then execute the instruction, which adds 3 to the program counter. This causes the program counter to point to the `retlw D'129'` instruction. On the next cycle, the `retlw D'129'` instruction is fetched. It causes a return to the main program with a 129 in the `W` register.

As long as we were confident that `Value` would never contain a value higher than 5, this snippet would return (43*`Value)` in the `W` register. If you remember our earlier discussion about PCLATH, however, you would recognize that this code will only work if the entire table is in the first 256 words of program memory. There are techniques for dealing with tables longer than 256 entries, but they are rarely needed.

If, however, we wanted to put the table in a different page, we would need to place the high order bits of that page in PCLATH before the `call`, otherwise, the `addwf PCL,F` would give the wrong result.. Perhaps more important, we also need to be sure we reset PCLATH before the next `call` or `goto`.

PCLATH Example

Introduction

In order to get a feel for this PCLATH behavior, we will return to the simulator so we can watch exactly what is happening. To give us the greatest range we will use the PIC16F877 as an example. This part has 8K of program memory, so it can use the entire range of PCLATH.

Setting up the project

In order to switch processors, there are two things we want to do. First, in MPLAB, select the PIC16F877 under the `Configure->Select Device` menu. Secondly, set the processor and include file at the top of the program (Lesson14a).

Besides our normal configuration bit settings, we will also disable the warning about crossing page boundaries so as not to distract us from possible real errors:

```
             processor    pic16f877
             include      p16f877.inc
             __config     _XT_OSC & _WDT_OFF & _PWRTE_ON
             errorlevel   -306
```

Our entire objective in this exercise is to jump around in these page boundaries. The assembler will issue a warning every time we do this, to remind us to fiddle with PCLATH. The `errorlevel -306` disables this warning. Normally, we wouldn't do this, or we would do it right at the point where we knew we had handled PCLATH.

A simple Table

To begin with, we will move our mainline a little way down in memory so that we can have some space to play with later at the start of memory. In this example, we will place the table in page 6. All it takes to call a table is to place something in the W register and do a call. However, because the table is in page 6, we need to load the page number into the PCLATH register:

```
             goto         Start

;      Mainline begins here
             org                 h'80'
Start
             movlw        h'06'        ; Setup PCLATH for a table
             movwf        PCLATH       ; in page 6
             movlw        D'2'         ; Load index into table
             call         Table        ; return with result in W
             nop                       ; Just a chance to look
```

The `org h'80'` statement begins this code 128 (80 hex) locations from the start of the program memory.

Continued on next page

PCLATH Example, Continued

A simple Table (continued)

Now we will put a simple table in page 6:

```
;       Lookup table in page 6
;
;       Return with D'16' times the W register in W

                org         h'600'
Table
                addwf       PCL,F
                retlw       h'00'
                retlw       h'10'
                retlw       h'20'
                retlw       h'30'
                retlw       h'40'
                retlw       h'50'
                retlw       h'60'
```

Notice that this has the same structure as the table earlier. In this case, we could have just as easily done the math with four `rlf` instructions, but this example lets us easily see when we return from the table with the correct result.

Testing the table

If we now assemble the code (after adding an `end` statement of course), we can watch PCLATH get loaded with a 6, then we place the 2 into W and call the `Table` routine at location H'600'. The 2 gets added to the program counter after the first instruction in the `Table` subroutine. Notice that the result is H'603'. This is because the PC actually contains one more than the current location when the instruction is actually executed. The result from the routine ends up being exactly what we expected, H'20'.

The dt directive

The MPASM assembler provides a directive which generates an `retlw` instruction for each byte of data following the directive. We can thus save a little typing by replacing our table with the following:

```
Table
                addwf   PCL,F
                dt      h'00',h'10',h'20',h'30',h'40',h'50',h'60'
```

This is especially handy when we want to store a text string in memory:

```
VrsMsg
                addwf   PCL,F
                dt      "Lesson 14a Ver 1.0"
```

PCLATH and goto/call

Before leaving our investigation of the program counter, let's look at how PCLATH affects the `call` instruction. This behavior is identical to the `goto` instruction.

We will add a simple subroutine that does nothing but return a value we can recognize. We will place copies of that routine at the same offset in several different pages, each returning a different value. This way we can see how the program counter behaves over these long distance calls.

Continued on next page

PCLATH Example, Continued

PCLATH and goto/call (continued)

Add the following code:

```
;       Example subroutine in page 0

        org             h'50'
S0050   movlw           h'00'
        return

;       Subroutine in page 7
        org             h'750'
S0750   movlw           h'07'
        return

;       Subroutine in page 8
        org             h'850'
S0850   movlw           h'08'
        return

;       Subroutine in page 1f
        org             h'1f50'
S1f50   movlw           h'1f'
        return
```

And to call the code, add the following just below the nop:

```
;       Show how calls move around.  Note that PCLATH still
;       contains a 6, but only bits 3 and 4 are used to form
;       the target address of the call

                                ; Set something in W. Even though PCLATH
        movlw           h'ff'   ; contains a 6, call to page 0 works, as
        call            S0050   ; does page 7 because they don't need
        call            S0750   ; bits 3 and 4. But page 8 ends up
        call            S0850   ; in the wrong place as does page 1f
        call            S1f50   ; because bits 3 and 4 are wrong
        nop

;       Lets try those last two calls with PCLATH set correctly
        movlw           h'08'   ; Set up PCLATH to point to
        movwf           PCLATH  ; page 8
        movlw           h'ff'   ; Value in W
        call            S0850   ; Call now goes to correct place and
        nop                     ; returns with 8 in W

        movlw           h'1f'   ; Now set up PCLATH to point to
        movwf           PCLATH  ; page 1f
        movlw           h'ff'   ; Value in W
        call            S1f50   ; Again call ends up in the right place
        nop                     ; and returns with 1f in W
```

After assembling the code, single step through it. Notice that the third and fourth `calls` go to an odd place. They end up going to the routine at the address which would be formed if bits 11 and 12 of the address were clear.

In the final pair of `calls`, we establish the proper value in PCLATH prior to making the `call`, and as a result, the `call` ends up in the right place.

Continued on next page

PCLATH Example, Continued

The pagesel directive

Earlier in the course, we used the `banksel` directive to set the bank bits in the file register. There is a similar directive, the `pagesel` directive, which sets the page bits in the PCLATH register. It is called by simply following the directive `pagesel` with the label we wish to jump to.

Note, however, that the `pagesel` directive only sets bits 3 and 4 of PCLATH. It is therefore interesting only if we are planning a `goto` or `call`. It is not adequate if we wish to do address arithmetic, as in a table. The directive is therefore only useful for processors with more than 2K of program memory.

The lcall and lgoto special instructions

If you have been studying the MPASM Quick Reference, you may have noticed a page entitled “12-Bit/14-Bit Core Special Instruction Mnemonics”. These are not really machine instructions, but rather instructions to the assembler to generate several machine instructions.

Two special mnemonics of interest here are `lcall` and `lgoto`. These set the page bits before executing a `call` or `goto`. This adds to the readability of the source (as do most of the special instructions). However, there is a risk with these two in particular; *they do not reset the page bits on return*. This means that a later `call` or `goto` will end up on the page referenced in the special instruction, rather than on the expected page.

It is important for the programmer to be sure to reset the PCLATH bits after making the jump. This can be done by explicitly setting the bits, by the use of the `pagesel` directive, or by simply remembering to use `lcall` and `lgoto` the next opportunity.

Encoding

Introduction

Up until now, our tables have been fairly simple. Generally, one would use a table when it is difficult or impossible to do the math to translate from the index to the desired result. Sometimes the math is simply too slow. For example, synthesis chips will frequently use a sine table lookup even though it is certainly possible to calculate a sine. However, table lookups are quite fast, and the memory spent on the table may well be a reasonable price to pay for the increased performance.

Non-mathematical codes

Frequently there is a need to look up something that doesn't represent a simple value. The bits stored in memory are only ones and zeroes. They only take on the meaning that we apply to them. There is no rule that says that the byte has to represent a number or a letter. Indeed, there is no rule that says that a byte has to represent only one thing. There are eight bits in a byte, so one could use the byte to represent eight different things with possible true/false results. Or, one could use it to represent two things each with 16 possible outcomes. Or two things, one with 32 possibilities and another with 8. The possibilities are endless, and there is rarely a "best" way.

Imagine, for a moment, that we are developing some sort of instrument, say a counter or LC meter, and we want to output the result in Morse. Since we are only outputting digits, the problem is simplified somewhat. Each digit in Morse consists of five elements. If we allow a one bit to represent a dah, and a zero bit to represent a dit, then we could encode a Morse digit in five bits of a byte:

```
;       Convert a digit, 0-9, to Morse
Table
                addwf           PCL,F
                retlw           B'11111000'     ; 0
                retlw           B'01111000'     ; 1
                retlw           B'00111000'     ; 2
                retlw           B'00011000'     ; 3
                retlw           B'00001000'     ; 4
                retlw           B'00000000'     ; 5
                retlw           B'10000000'     ; 6
                retlw           B'11000000'     ; 7
                retlw           B'11100000'     ; 8
                retlw           B'11110000'     ; 9
```

Notice that we have left the least significant three bits unused. There is no law that says we need to use up all the bits in a byte. However, later we will see how to put those bits to good use.

In this example, we could now output Morse by looking up the digit in the table, then rotating the result left for each of the five elements. After each rotation, if the carry bit was clear, we would send a dit, if set, we would send a dah.

Another Table Example

Introduction

Most of the examples in this lesson are run on the simulator, but let's take a break from that and run one on the hardware. For this example, we will build on Lesson6c.asm. Copy the .asm file to your Lesson 14 folder, and rename the file to Lesson14b.asm. Also, before you build the project, remember to set the processor back to the PIC16F84A in the configuration menu.

A few small tweaks first

If you recall, in Lesson 6, we wrote a program to send the Morse word "TEST" over and over. We hadn't yet learned to program a PIC, so we simply toggled a bit in the File Register to represent our code.

The bit we chose was one that illuminated an LED. However, it would be nice to toggle the keying transistor, so that if we plugged the PIC-EL into a code practice oscillator, we could hear our CW.

In Lesson6c we had a routine to turn on the "transmitter" and another routine to turn it off. Let's begin by modifying the code to toggle the transistor as well as the LED. Remember, the sense of the transistor is opposite that of the LED:

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

We will also need to define KEY, and let's actually toggle the devices now, so instead of using memory for "Output", let's equate Output to PORTB:

```
XMTR    equu            H'02'
KEY     equ             H'07'
Output  equ             PORTB

        cblock          H'20'
;               Output                  ; Output byte to transmitter
```

In this case we have simply commented out the old definition for Output. We could just as well have deleted it entirely.

Finally, we want to initialize the output port:

```
Start
        clrf            Output          ; Initialize output off
        banksel         TRISB
        bcf             TRISB,XMTR
        bcf             TRISB,KEY
        banksel         PORTB
```

When we assemble this code, we will get warning 302 because TRISB is not in bank 0. We can disable this warning with `errorlevel` if we wish. At this point, programming the PIC with this code should cause the center LED to blink out TEST along with the keying transistor.

Continued on next page

Another Table Example, Continued

The new mainline

Our old program wrote out the word TEST. The mainline consisted of calls to subroutines for each of the three different letters. Here, however, we would like to have a routine to send a digit, any digit. So the mainline should look like:

```
;       Main loop here
Loop
                movlw       D'1'
                call        Digit
                movlw       D'2'
                call        Digit
                movlw       D'4'
                call        Digit
                movlw       D'8'
                call        Digit
                call        WordSpace
                goto        Loop
```

We could have chosen any sequence of digits. In fact, to be thorough we should test every digit.

Since we are no longer calling the routines `SendT`, `SendE`, or `SendS`, we can delete them.

Generating the CW

We will use the same Table routine we used in the earlier section. However, let's place the table at location H'300', the last page of the 16F84's memory. We should note that it is customary in PIC16F84 code to place tables in the first page, since this eliminates the need to adjust PCLATH. However, this is not a hard requirement. The only real requirement is that the table be entirely confined to a single page.

To actually generate the Morse from the table:

```
Digit
        movwf       DigToSend       ; Save digit
        movlw       H'5'            ; 5 elements per digit
        movwf       BitCount
        movlw       H'03'           ; Select page with
        movwf       PCLATH          ; digit table
        movf        DigToSend,W     ; Pick up the digit
        call        Table           ; And get it's Morse
        movwf       DigToSend       ; Save it off
        clrf        PCLATH          ; and restore PCLATH
DigLoop
        rlf         DigToSend,F     ; Get next element
        btfsc       STATUS,C        ; Is it a dah?
        goto        SendDah         ; Yes, send a dah
        call        Dit             ; No, send a dit
        goto        EndLoop
SendDah call        Dah
EndLoop
        decfsz      BitCount,F      ; Decrement element count
        goto        DigLoop         ; Not done? Do it again
        call        DahTime         ; Inter-character space
        return
```

This code should be fairly self-explanatory. We do a table lookup to translate the digit to Morse, taking care to handle PCLATH, keep count of the number of elements sent as we rotate the Morse elements into the carry, and depending on the state of the carry, send a dit or a dah. The dit and dah code are unchanged from Lesson 6.

Tables in File Register

Introduction

Since the file register is volatile, we typically wouldn't use it for lookup tables. However, there are many times where we want to process lists of data, or data elements much longer than 8 bits. For example, in a counter, we would want to calculate the ASCII digits that make up the count to send to the display. We would then index through those results to send them to the display. This has the same characteristics as a table, although the use is different.

The INDF and FSR registers

To help us make better use of file register memory, there are two special registers we can use. These two registers work together to allow us to access a calculated location in the file register memory.

The FSR register may be loaded with the address of a file register location. Reading or writing the INDF register won't affect the INDF register at all, but rather will access the file register location whose address is contained in FSR. This is known as "indirect addressing". Both INDF and FSR can be read, written, incremented or tested just like any other register.

Using the file register for a table

With these two registers used together, table lookups in the file register memory are quite simple. One merely stores the index into the FSR register, and reads the result from the INDF register. Since the value stored in the FSR is an address, the programmer may need to add in the starting address of the table before storing the value in FSR. Since file register tables are most often used for storing long data, there would typically be some sort of counter maintained, and the program would probably step through the table, rather than looking up a specific value.

A File Register Example

Introduction

Once again we will turn to the simulator for a simple example. Create Lesson14c and start Lesson14c.asm off with all the normal directives.

Table Storage

To begin with, we need to reserve storage for the lookup table, a counter we will use to keep track of where we are, and a result storage location:

```
cblock          H'20'
        D1                      ; Storage for seven
        D2                      ; element long table
        D3
        D4
        D5
        D6
        D7
        Index                   ; Counter in table
        Target                  ; Result from table
endc
```

Filling the table

When we run the simulator, we would like to have something we can recognize as the “right answer” from the table. In this example, filling the table is fairly tedious. It is the nature of a table in the file register, however, that we must fill it programmatically:

```
                goto            Start
                org             h'80'
Start
                movlw           H'51'           ; Load up the
                movwf           D1              ; table with entries
                movlw           H'52'           ; from 81 to 87
                movwf           D2
                movlw           H'53'
                movwf           D3
                movlw           H'54'
                movwf           D4
                movlw           H'55'
                movwf           D5
                movlw           H'56'
                movwf           D6
                movlw           H'57'
                movwf           D7
```

(The choice of H’80’ as a location to place the mainline was purely arbitrary).

Reading the table

Before actually getting data from the table, we need to load the FSR register and initialize our counter:

```
movlw           D1              ; Point to address of
movwf           FSR             ; first entry
movlw           H'7'            ; Initialize count of
movwf           Index           ; Entries to read
```

Notice that the first `movlw` gets the *address* of D1, since it is a literal move, rather than the contents.

Continued on next page

A File Register Example, Continued

Reading the table (continued)

Now, we can simply loop through the table, fetching the table value, storing it, and moving on to the next entry:

```
Loop
        movf            INDF,W          ; Get value from the table
        movwf           Target          ; and store it
        incf            FSR,F           ; Point to next entry
        decfsz          Index,F         ; Done?
        goto            Loop            ; No, go back and do next
```

In this simple example, we merely stored the result. In an actual application, we would likely have processed the entry in some way. For example, if this were the counter we had talked about in the Lesson14b program, we may well have called our `Digit` routine to send the result out in Morse code, rather than store it in `Target`.

Testing the table

After assembling the program, set a breakpoint just before `movlw D1` and open the file register window so you can see the results. When you run down to the breakpoint notice that locations H’20’ through H’26’ contain H’51’ through H’57’. Stepping through the next two instructions will show the FSR (location H’04’) changed to H’20’. Two more and location H’27’ (`Index`) will be loaded with H’07’, the number of entries in the table.

Now, entering the loop, the first step will load the first table entry into the W register, and the next will store it in `Target` (H’28’). Stepping further will show the FSR incremented, `Index` decremented, and the next time through the loop the second table entry will be processed.

Although this example is very simple, the technique can be applied to many applications.

The EEPROM

Introduction

Most of the PIC16 processors include electrically erasable programmable read only memory, or EEPROM. In the case of the PIC16F84A, there are 64 bytes of EEPROM available.

EEPROM combines features of the file register RAM and the program memory FLASH. Like the file register, it can be read and written under program control. Like the program memory, it retains its contents when power is removed. EEPROM can be written by the assembler as well as the program, so unlike the previous example, the data need not be loaded by the program, if the data is known at the time the chip is programmed.

Reading the EEPROM

Unlike the file register, the EEPROM is accessed through four registers; EEADR, EEDATA, EECON1 and EECON2. In addition, a bit in INTCON is set when a byte has been written. For reading, we need only concern ourselves with EEADR, EEDATA, and EECON1.

Reading data from EEPROM is a three-step process:

- Set the address to be read in EEADR
- Set the RD bit in EECON1
- Read the result from EEDATA

This process is complicated a bit by the fact that EECON1 is in bank 1 in the PIC16F84A.

```
        movf            Location,W      ; Set address in EEPROM
        movwf           EEADR           ; to read
        banksel         EECON1
        bsf             EECON1,RD       ; Cause read to happen
        banksel         EEDATA
        movf            EEDATA,W        ; Grab the data
```

Important note: In other processors, notably the PIC16F628, these registers are in different memory banks, so code for manipulating the EEPROM will not port directly between processors.

Writing EEPROM from MPLAB

MPLAB views the EEPROM as memory starting at address H'2100'. It is important to note that while MPLAB thinks the addresses start at H'2100', the EEPROM addresses start at 0. There are only 64 EEPROM locations in the PIC16F84A, so EEPROM addresses run from H'00' through H'3f'.

MPLAB provides a directive, the `de` directive, to define EEPROM data. The `de` directive works much like the `dt` directive we used earlier; it may be followed by bytes of data in a number of formats:

```
        org                     H'2100'
        de                      H'51'
        de                      "ABC"
```

Notice that while we have the dt to define table data in program memory, and the de for EEPROM, there is no equivalent for the file register since that memory is volatile and its contents will be random on power up.

An EEPROM example

Introduction

Once again we will use the simulator to allow us to see exactly what is going on in our first trip through this code. In this example, we will take the same seven element table we used in the previous example. However, instead of loading the table with a sequence of `movlw`, `movwf` instructions, we will load the table from the assembler.

Reserving file register locations

In this example, we no longer need to reserve the locations for our table in the file register:

```
cblock          H'20'
       Index
       Target
       Location
Endc
```

Initialization

As in the earlier example, we still need to initialize the count, and we also want to initialize the location in EEPROM we are reading:

```
        goto            Start
        org             h'80'
Start
        movlw           H'7'
        movwf           Index
        movlw           0
        movwf           Location
```

Reading the table

Again, we will simply read values from the table and store them in Target. As in the previous example, if this were a real application we likely would have done some processing as we read the values out:

```
Loop
        movf            Location, W     ; Location in EEPROM
        movwf           EEADR           ;
        banksel         EECON1          ; Select bank for EECON1
        bsf             EECON1,RD       ; Initiate read
        banksel         EEDATA          ; Back to bank 0
        movf            EEDATA,W        ; Pick up the data
        movwf           Target          ; and store it off
        incf            Location,F      ; Point to next EEPROM loc
        decfsz          Index,F         ; Count down
        goto            Loop            ; Go do next location
```

Storing the table in EEPROM

Finally, we need to place the values for our table in EEPROM. This is easily done:

```
        org             H'2100'
        de              H'51',H’52’,H’53’,H’54’
        de              H'55',H’56’,H’57’
```

Testing

This program runs a lot like the earlier program. We won’t belabor the testing exercise. Run it and satisfy yourself that there are no surprises.

A Combined Example

Introduction

OK, you know what's coming. You are just dying to read a message out of EEPROM and send it out in Morse. Before we do, let's expand our understanding a bit more.

Encoding yet again

In Lesson14c, we encoded the Morse digits into the left most 5 bits of our table entries. Since most of the Morse characters are 5 elements or less, we could encode most of the Morse alphabet into those 5 bits. The catch is, how could we differentiate between a dit, a dah, and nothing. It seems we need three states per element.

We could use those three left over bits to give us the count of the number of bits to use. Thus, we would encode an A as something like:

```
01xxxxxx   elements (dit, dah)
xxxxx010   count (2)
--------
01xxx010   result
```

where an 'x' represents a don't care bit.

That takes care of the bulk of our alphabet, but there are quite a few characters with six elements, for example, the period and question mark.

We can cheat. Many of those six element characters end in a dah. If we simply use a count of six, the ending dah can be shared with the count, and our code won't really know the difference:

```
010101xx   elements (dit, dah, dit, dah, dit, dah)
xxxxx110   count (6)
--------
01010110   result
```

We still have a problem with some six element characters, and a seven element character ($).

We could further build out our table by using a count of zero as a special flag, to mean that the leftmost bits are a lookup into a table of six element characters, and handle the lone seven element character as a special case (or simply ignore it!)

Now our table logic would look something like:

```
Lookup the letter
Save off the result
Mask off the count
If the count >0
    Rotate the result 'count' times
    Sending a dit or dah as needed
Else
    If the result is >0
        Look up in second table
        If the char is $
            Count is 7
        Else
            Count is 6
        Rotate the result 'count' times
        Sending a dit or dah as needed
    Else
        Send a space
Wait a letter space
```

OK, not terribly simple, but not unmanageable, either.

Continued on next page

A Combined Example, Continued

Reviewing the Code	Rather than talking through the long table and most of the code, it is left for the student to review the code in Lesson14e.asm. Most of the code is simply an extension of Lesson14b. However, there are a few items worth mentioning. When we place a message into memory, it is generally more convenient to do that in ASCII. There are 128 ASCII characters, but many are non-printing, and most do not have Morse representations. To deal with this, we do a little manipulation of the data before looking it up in the table. Still, however, we want to arrange the table so that the order of characters is the same as in ASCII. ASCII characters H'00' through H'1f' are non-printing characters. We subtract H'20' from the character before looking up the character. ASCII characters H'61' through H'7a' are lower-case characters. There are simple ways to uppercase the lowercase characters, but in this program, we have chosen not to do that. Another little tricky bit. In the previous examples we have used a count to tell how far to read in the table. However, here we would like to put a message in the EEPROM, and wouldn't it be nice if we didn't have to count? Instead, we put a label at the end of the message, and stop when our EEPROM address matches the label. The problem is that, although the EEPROM addresses go from H'00' to H'3f', the assembler views them as starting at H'2100', so we need to get rid of the left most 8 bits of the address before we use it. Thus, the cryptic: `EndMsg  equu    $&H'ff'` at the end of the message. What this means is to take the current location ($), AND (&) it with H'ff', and assign that value to EndMsg. The AND operation masks off the high order bits.
Improving the code	Clearly, it is often more readable to use lower case letters, so logic for converting lower case numbers to upper would be a nice addition. A more ambitious enhancement would be to provide for a way to load the EEPROM from the PIC-EL itself. We won't discuss writing to EEPROM, however, for several more lessons.

Wrap Up

Summary	In this lesson, we have examined a number of way to implement tables on the PIC. We have gained a better understanding of the program counter, and seen how to use the PCLATH register to help us deal with addresses. We have used the FSR and INDF registers to step through tables in the file register. And we have seen how to read the EEPROM and how to load its contents from the assembler.
Coming Up	In the next lesson, we will learn how to read the rotary encoder used on the PIC-EL board. This encoder outputs what is known as “gray code”, and is by far the most common type of rotary encoder.