Lesson 18
Display Conversion

Overview

Introduction

In the previous lesson we saw how to display ASCII text to the liquid crystal display. Frequently we want to display the result of a calculation or sensor. However, rarely is this data already in the ASCII format needed for the LCD. Some sort of conversion is frequently needed. In this lesson we will discuss methods of preparing data for display.

In this section

Following is a list of topics in this section:

Hex to ASCII conversion

Introduction

When we are debugging a program, it is helpful to be able to see the results of our program in hexadecimal. Fortunately, this is the simplest form of display conversion, so we will address it first.

Splitting apart the digits

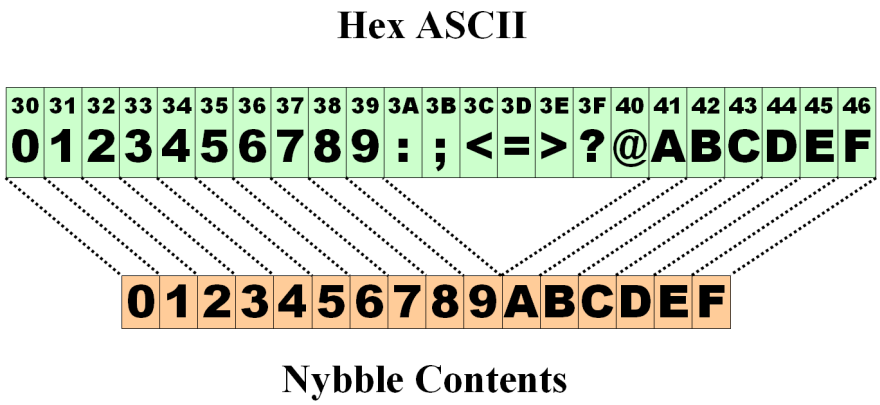
One of the convenient things about hexadecimal is that there are exactly two characters per byte. When we want to display a binary value as hexadecimal, we need to treat the left and right four bits as independent characters. Fortunately, this is relatively simple.

If we have some value in memory, we can pick off the low order nybble by simply `AND`ing it with `H'0f'`. To get the high nybble, we swap the nybbles using the `swapf` instruction, and then `AND` with `H'0f'`.

Converting to ASCII

Each of the nybbles will represent a value from 0 to F. Before sending the value to the LCD, it must be converted to ASCII. This would be fairly simple if the value were only 0 through 9; the ASCII characters '0' through '9' are all in order. To convert a nybble whose value is between 0 and 9 to ASCII, one only needs to add the ASCII representation of '0', or H'30' (since 1 is 1 higher, 2 is 2 higher, etc.)

But if the value is 10, then we want to display the letter 'A', which is not one letter higher on the ASCII table:



So we need to identify whether the value is higher than 9, and add 7 (the difference between ':' and 'A').

Continued on next page

Hex to ASCII conversion, Continued

Committing this to code

Notice that we will take each nybble and treat it the same. Our code, then, will split the byte into two nybbles, and convert each nybble to ASCII. We could write a subroutine that would do the nybble to ASCII conversion and call it twice. But this type of routine is likely to be called within a routine that might be called within a routine, etc. Probably, then, rather than risking a stack overflow, it might be better to create a macro. The code would be duplicated, but only twice.

The binary to ASCII macro

The macro needs to do the following:

- Mask off any high bits thus ensuring the value is within range
- Test whether the result is greater than 9
- If greater, add the difference between the character 'A' and the character '9'
- Add the character '0'

It turns out to be a little shorter if we subtract decimal 10, test the carry, do the necessary `add`s, and then add the 10 back in. Since this is a routine that is likely to get called frequently, it is probably better to save a few cycles. Since we are subtracting a constant from W, we can also shave off a word by adding minus 10 rather than subtracting 10. (The `sublw` instruction subtracts W from a literal and we want to subtract a literal from W, so we would need to save W first. Adding -10 instead eliminates this step. And you remember from high school algebra that adding -10 is the same as subtracting 10. That rule hasn't changed.)

Let's assume our macro takes the nybble to be converted in the W register, and stores the result in a location determined by the macro argument. We'll call that parameter '`digit`' since it will be a hexadecimal digit.

```
;       Macro to convert the low nybble of W to ASCII hex
;       and store the result at the argument 'digit'
ToHex   MACRO       digit
        andlw       H'0f'         ; Mask off excess
        addlw       -D'10'        ; Is it A..F?
        btfsc       STATUS,C
        addlw       A'A'-A'9'-1   ; Yes, add 7
        addlw       A'0'+D'10'    ; Add in '0' plus the 10 we
        movwf       digit         ; subtracted and save.
        ENDM
```

We haven't used macros a lot, but they are a way to save some typing and to ensure we do something the same way every time. However, unlike a subroutine, they don't save any code space.

Notice that rather than simply adding 7, we added `'A'-'9'-1`. This makes it a little more clear what we are doing, rather than just having a 7 appear out of nowhere. The actual arithmetic is done by the assembler, rather than the PIC, so the code generated by MPASM is exactly the same as if we had written a 7.

Continued on next page

Hex to ASCII conversion, Continued

The conversion subroutine

We are going to need some storage; one byte for the input value, and two bytes for the two character result. Let's call these `binary` for the input, and `d1` and `d2` for the two digits. Our subroutine turns out to be pretty simple; we first swap the nybbles in binary placing the result in `W` and call the macro with the argument `d1`. We then load `binary` into the `W` again and call the macro once more with the argument `d2`:

```
        code
ConvHex1
        ; First convert high nybble
        swapf         binary,W      ; Get hi nybble
        ToHex         d1            ; Convert it
        ; Now low nybble
        movf          binary,W      ; Now we don't need the swapf
        ToHex         d2            ; and convert it
        return
```

The test harness

To test this routine we will want a program which calls it with a variety of values. If we start a counter at `H'90'`, we can watch the low digit increment from 0 to 9, then we can check the switch to letters when it goes from 9 to A. When we get to `H'9f'`, and increment one more, we should see the high digit go from 9 to A. This allows us to see most of the error possibilities in a fairly short time.

First, we want to define the starting value, and set aside some space for storage of the three variables we need:

```
#define value H'90'              ; Initial value for test

                udata
binary          res    1         ; Storage for input value
d1              res    1         ; Storage for first digit
d2              res    1         ; Storage for second digit
```

Our main loop simply increments `binary` and calls `ConvHex1`:

```
LoopF
                call   ConvHex1  ; Convert to hex ASCII

                incf   binary,F  ; Try another value
                goto   LoopF
```

Of course, we still need the typical `extern` and `global` statements, the reset vector code, and we need to initialize `binary`.

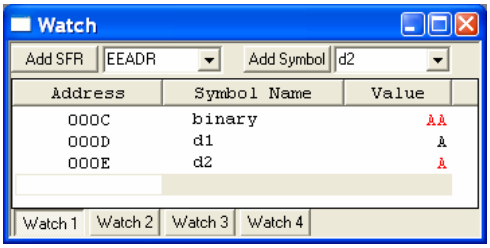
Continued on next page

Hex to ASCII conversion, Continued

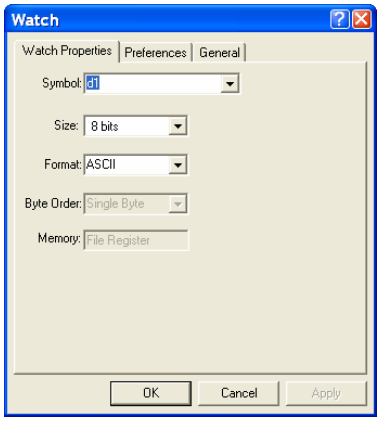
Testing ConvHex1

After we assemble and link the program, we can test it by placing a breakpoint after the call to `ConvHex1`, and repeatedly clicking the Run icon while watching the memory contents.

Using the watch window makes following the changes in `d1` and `d2` easier. Select **View->Watch**, and in the right hand dropdown, select `binary`, and then click "**Add Symbol**". Repeat the process for `d1` and `d2`.



Right-click on `d1` in the watch window, and select '**Properties**'. In the dialog, select **ASCII** from the **Format** drop down. Repeat for `d2`.



Now when we step through the program, we have a convenient list with just the values of interest, in the format we want to see.

Endian-ness

Introduction

Perhaps you have heard the terms “big-endian” or “little-endian”. Proponents of various CPUs have often battled about which is better. Thankfully, with the PIC, we are mostly spared that debate. Unfortunately, we are about to enter a tiny corner where the question happens to rear its ugly head.

Byte order

Whenever we have a byte-addressed processor, like the PIC, we need to use multiple bytes to store larger values. Once we have multiple bytes, we now have a choice about which order to store the bytes in memory. We could place the low value byte in the lowest address. This is called “little-endian” and it how the PC stores data. As a result, file formats that arose mostly on the PC, for example GIF, BMP, RTF or TGA have values stored in the little-endian order. It turns out that a lot of math operations are a little simpler if the little-endian format is used.

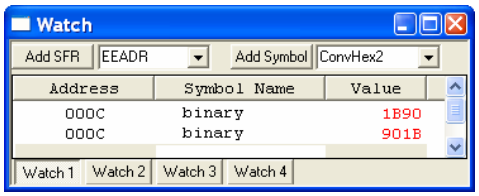
But, when we are looking at a hex dump, having the least significant digits come first is confusing. It seems much more “natural” to have the low value bytes in the highest addresses. This is called “big-endian” and is the format used on the older Motorola-based Mac computers as well as IBM mainframes. File formats that arose on the Mac tend to use the big-endian format. These include MacPaint, Photoshop, and IMG, but interestingly, not QuickTime, which uses the little-endian format.

This is an annoying enough issue that some file formats, like TIFF and XWD, actually have a flag in the file that indicates what byte order was used.

But what about the PIC?

The PIC, at least the 14 bit core PIC16, has no instructions that operate on more than one byte. As a result, the PIC processor really doesn’t have an “endian-ness”. For us, it almost never matters.

When we are using the MPLAB IDE, we sometimes want to look at a value in memory that takes more than one byte, and treat it as a single value. The Watch window, when we choose a format involving more than one byte, can use a little-endian format, or a big-endian format. In the Watch window properties, the **Byte Order** dropdown becomes active if you select a **Size** greater than 8 bits.



We can still choose in our programs to store the bytes in whatever order we prefer. Sometimes one way is easier, sometimes the other. As long as we are aware that the MPLAB IDE display of a multiple byte value might move things around on us, it really isn’t much of an issue.

Multiple-byte Hex to ASCII

Introduction

Frequently our data involves more than a single byte. We could simply call the previous routine twice for 16 bit data, but the routine expects the data to be in particular locations, so we would have to move data in and out of the storage area between calls. If we simply extended ConvHex1 for two bytes, the routine is beginning to get a little gangly. So perhaps we need to modify our approach.

A 16-bit test harness

If we want to test 16 bit data, this will not only require a two byte data value, but four byte result area:

```
                udata
binary          res     2               ; Storage for input
digits          res     4               ; Storage for result
```

In testing a routine which uses a 16 bit input, we should think about an appropriate starting value. It would be helpful interesting things were to happen fairly quickly so we don't need to click on the run icon a few hundred times. Although typically our programs only define 8 bit values, we can actually use the `#define` directive to define larger values:

```
        #define value H'1bf8'           ; Initial value for test
```

Having both bytes of our 16-bit datum in one place is convenient if we want to make changes as we are testing. Of course, if we think about it a moment, this causes us a bit of consternation. If we want to initialize `binary` with this value, we need to pick apart the individual bytes. Fortunately, there are two assembler operators, `high` and `low`, to come to the rescue. (There is also an `upper` operator that gets the high byte of a 24-bit argument).

```
        Start
                movlw           high value      ; Initialize high byte
                movwf           binary          ; of input value, and
                movlw           low value       ; then low byte.
                movwf           binary+1
```

Now the main loop of our test harness is simply a multiple byte increment with a call to our conversion routine:

```
        LoopF
                call            ConvHex2        ; Do the conversion

                incfsz          binary+1,F      ; Increment the low byte
                goto            LoopF           ; If no overflow, do it again
                incf            binary,F        ; otherwise, increment high
                goto            LoopF           ; byte and do it again.
```

Notice that we have chosen the "big-endian" format. In this particular case, there would have been no penalty for the "little-endian" choice.

Continued on next page

Multiple-byte Hex to ASCII, Continued

The conversion routine

Now we need to figure out how to do a multiple-byte conversion. In the single byte example, we used a macro to reduce the amount of code we had to write. Since we only called the macro twice, the actual amount of code wasn't an issue. But now we are going to need to come up with a four character routine. Repeating all that code four times might be getting excessive, so perhaps a subroutine is a better choice. Notice that the price we pay is an additional stack location. While this isn't a problem in our simple test harness, if we use the routine in a more complex application, this might become an issue.

We have another problem, though. While we can pass the nybble to be converted in through the W register, we eventually want to write the result to four different character locations. We can get around this by using indirect addressing. We can use the FSR register to point to a memory location result, and have the "convert a nybble" routine store its result in this location:

```
;       Nybble to convert is in low nybble of W
;       Result stored in location pointed to by FSR
Hexit
                andlw  H'0f'          ; Mask off excess
                iorlw  H'30'          ; Convert to ASCII
                movwf  INDF           ; Test to see
                sublw  A'9'           ; if the result was
                btfsc  STATUS,C       ; 0-9.
                return                ; Yes? done.
                movlw  H'7'           ; No, convert : to A
                addwf  INDF,F         ;
                return
```

We still have to call this routine with each nybble in succession:

```
ConvHex2
        ; First convert high nybble
                movlw  digits         ; Get address of storage area
                movwf  FSR            ; and place in FSR
                swapf  binary,W       ; Get hi nybble
                call   Hexit          ; Convert to hex
        ; Now low nybble
                incf   FSR,F          ; Point to next character
                movf   binary,W       ; Now we don't need the swapf
                call   Hexit          ; Convert low nybble
        ; Now convert high nybble of the next byte
                incf   FSR,F          ; Next character
                swapf  binary+1,W     ; As before
                call   Hexit
        ; Now low nybble
                incf   FSR,F
                movf   binary+1,W
                call   Hexit
                return
```

Obviously, we could make additional optimizations. `Hexit`, for example, could increment FSR, trading 3 increments for one. We would do an unnecessary increment, but we would save two words at the cost of one instruction time, and a slight loss in readability. If we wanted to go beyond two bytes, we might find it useful to do the bytes in a loop, but this would involve additional saving and restoring of FSR.

Continued on next page

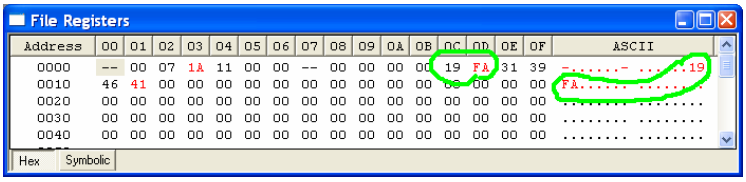
Multiple-byte Hex to ASCII, Continued

Running the test

As in the previous example, the Watch Window can be used to show only the locations of interest. However, we need to set the properties for each of the displayed values. For `binary`, set the **Size** to 16 bits, the **Format** to Hex, and the **Byte Order** to Low:High. For `digits`, set the **Size** to 32 bits, the **Format** to ASCII, and the **Byte Order** to Low:High.

Place a breakpoint after the call to `ConvHex2`. Each time you click on the run button, both values will increment, and should display the same value. As you step through 1BFF the next result should be 1C00.

In the file register hex display, you should see `binary` displayed correctly in the left side of the window, and `digits` correctly on the right.



Decimal Conversion

Introduction	Displaying results in hexadecimal format is interesting when we are debugging, but for real applications, we almost always want to display decimal. Unfortunately, decimal digits don't align nicely on nybble, or even byte, boundaries. We need a different routine.
What is a decimal conversion, anyway?	Let's step back to get a basic understanding of what must be done. First consider converting a single byte to decimal. Since a byte can only hold 256 values, ranging from 0 to 255, so the highest number we will need to display is 255, or three digits. The first digit is nothing more than the value divided by 100. The second is the remainder of that division divided by 10, and the third digit is the remainder of the second division divided by 1. Thus, our divide routine must divide the value by successively smaller decades, saving off the remainder each time for the next decade. Sounds reasonably simple, and at least in concept, can be extended it to any size value.
Divide?	But wait, you say. The PIC doesn't have a divide instruction. The PIC can divide by powers of two by shifting, but 10 and 100 aren't powers of two. How can we perform the necessary division? Once again, step back and ask what is the real problem. What does it mean to divide A by B? It is the number of times we can subtract B from A before A goes negative. When we think of it that way, a solution starts to come to mind. If the quotient was going to be large, it might get quite slow to do a lot of subtractions, but we happen to know that the result will be between 0 and 9, so there will never be a huge number of subtractions. (There are more efficient approaches, but in any case, division is often a slow operation.)
Processing a digit	Since we can do a digit at a time, the approach of using a macro for each digit, as we did in the first hex conversion, might be appealing. This time we have four variables to consider; the initial number, the number to divide by, the resulting ASCII digit, and the remainder. We can reduce this variable count by one by using the same storage location for the number and the remainder, especially since the remainder from the previous division will become the input for the next. In our example we will use a storage location, `work`, for the remainder, and the macro will have two arguments, `Decade` and `Letter`. `DoDigit Macro Decade,Letter`

Continued on next page

Decimal Conversion, Continued

Processing a digit (continued)

Now we need to subtract 'Decade' from 'work' and count how many times we can do that before the result goes negative:

```
Again
        movlw   Decade          ; Subtract Decade from Temp
        subwf   work,W          ;
        btfss   STATUS,C        ; Was there a borrow?
        goto    Alldud          ; Yes, go to next digit
        incf    Letter,F        ; No, increment display
        movwf   work            ; and save off difference
        goto    Again           ; Go back and do it again
Alldud
        movlw   A'0'            ; Convert to ASCII by
        iorwf   Letter,F        ; adding ASCII 0
```

The macro will be used multiple times, so we need to specify that the labels, 'Again' and 'Alldud', are local to the macro so we don't get an error as a result of multiple declarations of the same label:

```
        Local   Again,Alldud
```

Using the macro

Our subroutine will clear out the result area, move the argument to work, and then call the macro with arguments of 100, 10, and 1:

```
        code
ConvBCD1
        ; Clear out result area
        movlw           ' '             ; Will place a blank in
        movwf           digits          ; first spot for display
        clrf            digits+1        ; Clear out the rest
        clrf            digits+2
        clrf            digits+3
        ; Move argument to work (work will be destroyed)
        movf            binary,W
        movwf           work
        ; Convert each digit
        DoDigit         D'100',digits+1
        DoDigit         D'10',digits+2
        DoDigit         D'1',digits+3
        return
```

We will use four characters and fill the first with a blank so we can display the entire value in the watch window as a 32 bit value.

Testing

The test harness can be essentially the same as we used for ConvHex1, although we might choose a different initial value.

Decimal Conversion - another approach

Introduction

In the previous example, the code in the macro was a bit long, and would get fairly unwieldy when we try to extend it to larger values. Perhaps even for the case of a single byte, some sort of looping construct might work a little better.

Keeping our finger in several pages at once

The problem with looping through each of the digits is that there are now several things we need to track. In the previous example, our table of decades was neatly handled by the source lines that called the macros, as was the selection of digit to use for the result.

If we are going to loop, we need to track these things through a memory location. The solution is simple enough; we can use the FSR register to point to the character being calculated. As in the previous example, we can use the same storage location for the dividend and remainder. It seems to make sense to use a table of some sort for the decades:

```
; Table of decades
TABLES code
t1      movf          dindex,W
        addwf         PCL,F
        dt            D'100',D'10',D'1'
```

We need a location, 'dindex', to hold the index into the decades table.

Setting up for the loop

As in the earlier examples, the result area must be cleared before starting:

```
                code
ConvBCD1A
        ; Clear out result area
                clrf          digits
                clrf          digits+1
                clrf          digits+2
```

Now the original argument must be copied to the work area since its value will be destroyed as we go:

```
        ; Move argument to work
                movf          binary,W
                movwf         work
```

Then we need to initialize the index into the decades table, and point the FSR to the location of the first character:

```
                ; Set up for loop
                clrf          dindex ; Point to first divisor
                movlw         digits ; Point to first digit
                movwf         FSR    ; in result area
```

Continued on next page

Decimal Conversion - another approach, Continued

The main conversion loop

For each digit, we will get the divisor from the table, and subtract it from the work variable. If the result was still positive, increment the result and do it again.

```
            ; Next digit
      Again
            call   t1           ; Get divisor
            subwf  work,W        ;
            btfss  STATUS,C      ; Was there a borrow?
            goto   Alldud        ; Yes, go to next digit
            incf   INDF,F        ; No, increment display
            movwf  work          ; and save off difference
            goto   Again         ; Go back and do it again
```

Once the remainder turns negative, we convert the result to ASCII, point the `FSR` at the next digit, determine whether we have processed all the digits, and if not, go do it again:

```
      Alldud
            movlw  A'0'          ; Convert result to ASCII
            iorwf  INDF,F        ; by ORing with ASCII '0'
            incf   dindex,F      ; Set up for next digit
            incf   FSR,F         ;
            movlw  H'3'          ; Check to see if its
            xorwf  dindex,W      ; the last digit
            btfss  STATUS,Z      ; Yes, all done
            goto   Again
```

The looping actually saves us a few instructions compared to the previous example (after the macros have been expanded), but since we are looping, we actually execute a few more instructions, making this example a little slower.

Testing the routine

We can use the same test harness we used for the previous example. However, in this case we chose not to pad the result with a leading blank, so the watch window is a little more clumsy. We could go ahead and add the blank, either in the subroutine, or in the test harness, or we could simply use the file register display to see what is going on.

Multiple byte decimal conversion

Introduction

Now that we have a decimal conversion that indexes through the characters, we can extend the same algorithm to handle larger arguments. Here we will examine a 16 bit argument. 16 bits can have 65,536 different combinations; if we treat the 16 bits as an unsigned integer, then it may have values from 0 to 65535. For this case, then, our input variable will be two bytes, and our output area 5 characters. Most of our intermediate calculations will also need to be two bytes.

The decades table

In the previous example, we needed to divide by 100, 10, and 1. For five digits, we will need to divide by 10K, 1K, 100, 10, and 1. Since we will be doing a double byte divide, our divisor will need to be two bytes, so we will need to make two table entries for each divisor. Once again, we can use the high and low operators to grab the appropriate bytes.

```
; Table of decades - note that each dt will generate two
; retlw instructions, one for each byte of the two byte value.
TABLES code
t1      movf   dindex,W
        addwf  PCL,F
        dt     high D'10000',low D'10000'
        dt     high D'1000',low D'1000'
        dt     high D'100',low D'100'
        dt     high D'10',low D'10'
        dt     high D'1',low D'1'
```

Setting up for the digits loop

The initial steps of this routine are almost identical to the previous, except that we need two of almost everything. The index into the decades table is still one byte. It will go from zero to nine, two for each character since we need to index through both bytes of each divisor.

```
;       Entry point for conversion routine
        code
ConvBCD2
        clrf    digits          ; Clear out the result area
        clrf    digits+1        ;
        clrf    digits+2        ; all
        clrf    digits+3        ; five
        clrf    digits+4        ; digits

        clrf    dindex          ; Index into decades table

        movf    binary,W        ; Move binary into intermediate
        movwf   inter           ;
        movf    binary+1,W      ;
        movwf   inter+1         ;

        movlw   digits          ; Get address of result area
        movwf   FSR             ; and point to it
```

Continued on next page

Multiple byte decimal conversion, Continued

Initialization for each digit

For each digit, we need to pick up both bytes of the divisor and store them in working variables. We can't simply keep the value in W and use it, since it is now a 16-bit number. The same holds for the remainder; we need to save it to a work variable.

```
;      Convert a digit by dividing by divisor from table
ConvL1
       call   t1            ; Pick up divisor high
       movwf  divisor       ; store in divisor
       incf   dindex,F      ; Point to next byte
       call   t1            ; Divisor low
       movwf  divisor+1     ; Store it
       incf   dindex,F      ; Next byte

       movf   inter,W       ; Pick up intermediate high
       movwf  work          ; Move it to work
       movf   inter+1,W     ; Now the low
       movwf  work+1        ;
```

Executing the divide

The division proceeds much like the previous example. However, we now have two bytes to subtract, and we need to take care that, if there is a borrow, the borrow does not cause the remainder to go negative:

```
;      Divide by successive subtraction
ConvL2
       movf   divisor+1,W   ; Pick up low part of divisor
       subwf  work+1,F      ; Subtract low divisor
       btfsc  STATUS,C      ; Borrow?
       goto   noBorrow      ; No
       movf   work,W        ; Check whether borrow will
       btfsc  STATUS,Z      ; cause negative
       goto   doneDig       ; Yes, we're done
       decf   work,F        ; No, do the borrow
noBorrow
       movf   divisor,W     ; Now high part
       subwf  work,F        ;
       btfss  STATUS,C      ; Did it go negative?
       goto   doneDig       ; Yep
       ; We didn't go negative, so save the remainder
       movf   work+1,W      ; Pick up the low byte
       movwf  inter+1       ; save it
       movf   work,W        ; Now the high
       movwf  inter         ;
       incf   INDF,F        ; Count up this subtract
       goto   ConvL2        ; and go do it again
```

Continued on next page

Multiple byte decimal conversion, Continued

Finishing up the digit

Once we have calculated the digit, we can convert it to ASCII, just as we did in the previous example. Then, we point the FSR to the next digit, and see if we have processed all the digits..

```
;       Now have one digit done, convert to ASCII
;       and go do the next
doneDig
        movlw  A'0'          ; Convert digit to ASCII by
        iorwf  INDF,F        ; ORing with ASCII zero
        incf   FSR,F         ; Point to next digit
        movf   dindex,W      ; Pick up the current digit #
        sublw  D'8'          ; and see if we are done
        btfsc  STATUS,C      ;
        goto   ConvL1        ; Nope, go do next digit
        return
```

Testing the routine

We can use the same sort of approach as the previous conversions for testing, except that we have a two byte binary value and a five character buffer for the result. However, there are a number of "boundary conditions" we might want to test. Of course we want to explore carries from one binary byte into the next, as well as carries from one decimal digit to the next. But we also have interesting things happening when a borrow causes the high byte to go negative, so we may want to explore that area as well.

Going Further

Introduction	We have seen how to convert binary to both hexadecimal and decimal ASCII, for one and two byte arguments. All our experiments so far have been on the simulator.
Displaying the result	The source zip file contains a program, L18f, which uses ConvBCD2 to convert a two-byte value and display it on the LCD. The student may want to play with this program and explore what happens at critical places in the calculation. The program goes slow for the first 20 values, after that it speeds up so that the rightmost digit is a blur. Even so, counting over the entire 16 bit range takes quite a bit of time. The student may want to experiment with slowing down the counter at a few critical places.

Larger Values	The algorithm described for two bytes can be easily expanded to any number of bytes. Three bytes is relatively straightforward, as the decades table can still be created using assembler operators:
	
	`dt upper D'10000000',high D'10000000',low D'100000000'`
	
	Handling the intermediate borrows starts to get a little involved, however. A 24 bit number can go as high as 16.7 million.
	
	Going beyond 24 bits will require that the developer manually calculate the bytes of the divisor. This is not too bad since for 32 bits (at least) there are only 10 digits, and the last 8 of them are the same as the 24 bit example except for a leading zero.
	
	`dt H'3b',H'9a',H'ca',H'00' ; 1 billion`
	`dt H'05',H'f5',H'e1',H'00' ; 100 million`
	`dt 0,upper D'10000000',high D'10000000',low D'100000000'`
	`…`
	
	The student is reminded that the "Scientific" view of the Windows calculator offers hexadecimal conversion of rather large numbers.

Efficiency	The algorithms shown are quite inefficient. If the student is interested in faster, shorter, approaches, there are any number of sources for BCD conversions in particular. The approach shown here was chosen primarily for its simplicity. Shorter and faster algorithms are widely available, but most tend to rely on somewhat non-obvious behaviors. The PICList archive is one source for algorithms, as are Microchip's Application Notes 00544d and 00617. If the student wishes to build a library of handy routines, these routines can be understood once, assembled, and locked away in a library to be reused whenever the need arises. For teaching purposes, understandability is paramount. For later applications, though, other considerations are likely to be more important.

Wrap Up

Summary	In this lesson we have examined converting internal representations of numbers into ASCII forms suitable for display on an LCD. We demonstrated single and two byte conversions, and pointed the direction for dealing with larger numbers.
Coming Up	In the next lesson, we will explore using the PIC-EL as an in-circuit programmer for our own circuit. To that end, the student will need to go beyond the PIC-EL hardware and construct a simple circuit that allows us to demonstrate how the student need not be constrained by the 18 pin socket on the PIC-EL. We will use the external circuit to experiment with the analog input and pulse width modulation output of the PIC16F873. We will discuss specifics of the auxiliary board in Lesson 19. There are a lot of ways the experimental circuit could be constructed, and everyone might choose a slightly different approach. We will use the PIC16F873 (**not** the 873A!) The only other "hard part" will be a 1N5711, which is optional. Since there will likely be some variation in different students' circuits, it makes no sense to go out collecting parts until you have read the lesson. But if you should happen across a deal on an 873, you might want to take advantage of it. Just remember, you don't want the 'A' part. The PIC16F873 is part of the family of 873/4/6/7. These parts are basically the same PIC with 4 and 8K and 28 and 40 pins. Another member of the family could be used with only the slightest of changes. So if you should happen to have an 876 lying around, it probably doesn't make sense to buy an 873. The 874/877, being 40 pin parts, have a different pinout, but the pin names are the same, so it isn't a large problem to translate. In Microchip parlance, SP means "skinny-DIP", so the part we will be using is a PIC16F873-20/SP. We will be using a relatively low clock frequency, so a PIC16F873-04/SP will work as well, and will be a little cheaper, but it will give you a little less flexibility to use it in future projects where you might need the clock speed. A PIC16**LF**873 is even better – that part is capable of running at lower voltages, but is a little more expensive. Similarly, parts marked I/SP or E/SP have a better temperature specification. Not usually an issue for amateur applications, and usually a little more expensive, but sometimes you find a deal!