PIC16F87X Tutorial by Example

Document History.

Dec 25, '00 – Original document in html format.

Jan 5, '01 – Converted to pdf format. Added routines related to data EEPROM (EEPROM_1.C, FIRST_TM and EE_SAVE) ,use of Timer 0 (TMR0_1.c and count.c), use of a CCP module for input capture (capture_1.c and capture_2.c) and for output compare (out_cmp1, out_cmp2.c and out_cmp3.c).

Jan 21, '01. Issue 1A. Unions, bit fields, use of a potentiometer in conjunction with EEPROM for calibration, SPI master using bit-bang. Use of the SSP module as an SPI Master. Interfaces with Microchip 25LC640 EEPROM, TI TLC2543 11-channel 12-bit A/D, Microchip MCP3208 8-channel 12-bit A/D and MAX7219 LED Driver.

Mar 12, '01. Issue 1B. Continues discussion of SPI devices including Atmel AT45 series EEPROM and Dallas DS1305 Real Time Clock. Philips I2C master using bit bang and using the SSP module including interfaces with Microchip 24LC256 EEPROM, Philips PCF8574 8-bit IO Expander, Dallas DS1803 Dual Potentiometer, Maxim Dual D/A, Dallas DS1307 RTC, Dallas DS1624 Thermometer and EEPROM and Philips PCF8583 Real Time Clock and Event Counter.

April 9, '01. Issue 1C. Dallas 1-wire interface including DS18S20 Thermometer. Use of the hardware USART for sending and receiving characters. Use of the PIC16F877 as an I2C Slave and SPI Slave. Additional routines for the PIC16F628 including SFR definitions, flashing an LED and use of the hardware UART.

Introduction

This is a "tutorial by example" developed for those who have purchased our Serial MPLAB PIC16F87X Development Package. All of the C routines are in a separate zipped file. This is an ongoing project and I will add to this and send an updated copy in about two weeks.

Although all of this material is copyright protected, feel free to use the material for your personal use or to use the routines in developing products. But, please do not make this narrative or the routines public.

PIC16F87X Data Sheet

It is strongly suggested that you download the 200 page "data sheet" for the PIC16F877 from the Microchip web site. I usually print out these manuals and take them to a copy center to have them make a back-to-back copy and bind it in some manner.

Use of the CCS PCM Compiler

All routines in this discussion were developed for the CCS PCM compiler ($99.00). I have used many C compilers and find that I keep returning to this inexpensive compiler. All routines were tested and debugged using the same hardware you have as detailed in Figures 1 - 6.

Special Function Register and Bits

In using the CCS compiler, I avoid the blind use of the various built-in functions provided by CCS; e.g., #use RS232, #use I2C, etc as I have no idea as to how these are implemented and what PIC resources are used. One need only visit the CCS User Exchange to see the confusion.

Rather, I use a header file (defs_877.h) which defines each special function register (SFR) byte and each bit within these and then use the "data sheet" to develop my own utilities. This approach is close to assembly language programming without the aggravation of keeping track of which SFR contains each bit and keeping track of the register banks. The defs_877.h file was prepared from the register file map and special function register summary in Section 2 of the "data sheet".

One exception to avoiding blindly using the CCS #use routines is I do use the #int feature to implement interrupt service routines.

Snippets of defs_f877.h;

```
#byte TMR0 = 0x01
#byte PCL = 0x02
#byte STATUS = 0x03
#byte FSR = 0x04
#byte PORTA = 0x05
#byte PORTB = 0x06
#byte PORTC = 0x07
#byte PORTD = 0x08
...
#bit portd5 = PORTD.5
#bit portd4 = PORTD.4
#bit portd3 = PORTD.3
#bit portd2 = PORTD.2
#bit portd1 = PORTD.1
#bit portd0 = PORTD.0
```

Note that I have identified bytes using uppercase letters and bits using lower case.

Thus, an entire byte may be used;

```
TRISD = 0x00;       // make all bits outputs
PORTD = 0x05;       // output 0000 0101

TRISD = 0xff;       // make all bits outputs
x = PORTD;          // read PORTD or a single bit;

trisd4 = 0;         // make bit 4 an output
portd4 = 1;

trisd7 = 1;         // make bit 7 an input
x = portd7;         // read bit 7
```

Use of upper and lower case designations requires that you use the #case directive which causes the compiler to distinguish between upper and lower case letters.

(This has a side effect that causes problems when using some of the CCS header files where CCS has been careless in observing case. For example they may have a call to "TOUPPER" in a .h file when the function is named "toupper". Simply correct CCS's code to be lower case when you encounter this type of error when compiling.)

I started programming with a PIC16F84 several years ago and there is one inconsistency in "defs_877" that I have been hesitant to correct as doing so would require that I update scores of files. The individual bits in ports A through E are defined using the following format;

```
porta0      // bit 0 of PORTA
rb0         // bit 0 of PORTB - note that this is
            // inconsistent with other ports
portc0      // bit 1 of PORTC
portd0      // bit 0 of PORTD
porte0      // bit 0 of PORTE
```

Program FLASH1.C. (See Figure #4).

Program FLASH1.C continually flashes an LED on portd4 on and off five times with a three second pause between each sequence.

Note that PORTD may be used as a Parallel Slave Port or as a general purpose IO port by setting the pspmode to either a one or zero. In this routine, PORTD is used for general purpose IO and thus;

```
pspmode = 0;
```

Thus illustrates the beauty of C. For someone programming in assembly, they must remember that this bit is bit 4 in the TRISE register which is located in RAM bank 1. Thus, the corresponding assembly would be;

```
BCF STATUS, RP1         ; RAM bank 1
BSF STATUS, RP0
BCF TRISE, 4            ; clear pspmode bit
```

When using a bit as an input or output, the corresponding bit in the TRISD register must be set to a "one" or "zero". I remember this as a "1" looks like an "i" and a "0" as an "o". In this case, PORTD, bit 4 is made an output;

```
trisd4 = 0;     // make bit 4 an output
```

Routine FLASH1.C uses a short loop timing routine written in assembly to implement delay_10us() and routine delay_ms() simply calls this routine 100 times for each ms. Note that the these routines are intended for operation using a 4.0 MHz clock where each instruction is executed in 1 us. They are not absolutely accurate as I failed to take into account the overhead associated with setting the loop and the call to delay_10us but, they are useful in applications where absolute time is not all that important. I can't really tell they difference between an LED being on for 500 or 500.060 ms.

```
// FLASH1.C
//
// Continually flashes an LED on PORTD.4 in bursts of five flashes.
//
//
// Although this was written for a 4.0 MHz clock, the hex file may be
```

```
// used with a target processor having 8.0, 10.0 or 20.0 MHz clock.
// Note that the time delays will be 2, 2.5 and 5 times faster.
//
// copyright, Peter H. Anderson, Baltimore, MD, Dec 14, '00
//

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>

void flash(byte num_flashes);
void delay_10us(byte t);
void delay_ms(long t);

void main(void)
{
   while(1)
   {
     pspmode = 0;    // make PORTD general purpose IO
     flash(5);
     delay_ms(3000);
   }
}

void flash(byte num_flashes)
{
   byte n;
   for (n=0; n<num_flashes; n++)
   {
     trisd4 = 0;        // be sure bit is an output
     portd4 = 1;
     delay_ms(500);
     portd4 = 0;
     delay_ms(500);
   }
}

void delay_10us(byte t)
// provides delay of t * 10 usecs (4.0 MHz clock)
{
#asm
      BCF STATUS, RP0
DELAY_10US_1:
      CLRWDT
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      DECFSZ t, F
      GOTO DELAY_10US_1
#endasm
}
```

```
void delay_ms(long t)   // delays t millisecs (4.0 MHz clock)
{
   do
   {
     delay_10us(100);
   } while(--t);
}
```

Program FLASH2.C.

This routine is precisely the same as FLASH1.C except that the timing routines have been declared in lcd_out.h and they are implemented in lcd_out.c.

The CCS compiler does not support the ability to compile each of several modules to .obj files and then link these to a single executable (.hex) file. However, you can put routines that are commonly used and thoroughly debugged in a separate file and simply #include the files at the appropriate point.

File lcd_out.c is a collection of the two timing routines plus a number of other routines to permit you to display text on the LCD panel. However, the CCS compiler will not compile a routine, which is not used, and thus no program memory is wasted. Surprisingly, this is not true of all compilers.

```
// FLASH2.C
//
// Same as FLASH1.C except that the timing routines are located in
// lcd_out.h and lcd_out.c
//
// Continually flashes an LED on PORTD.4 in bursts of five flashes.
//
// This is intended as a demo routine in presenting the various
// features of the Serial In Circuit Debugger.
//
// Although this was written for a 4.0 MHz clock, the hex file may be
// used with a target processor having 8.0, 10.0 or 20.0 MHz clock.
// Note that the time delays will be 2, 2.5 and 5 times faster.
//
// copyright, Peter H. Anderson, Baltimore, MD, Dec 14, '00

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>

void flash(byte num_flashes);

void main(void)
{
   while(1)
   {
     pspmode = 0;    // make PORTD general purpose IO
     flash(5);
```

```
     delay_ms(3000);
   }
}

void flash(byte num_flashes)
{
   byte n;
   for (n=0; n<num_flashes; n++)
   {
     trisd4 = 0;        // be sure bit is an output
     portd4 = 1;
     delay_ms(500);
     portd4 = 0;
     delay_ms(500);
   }
}

#include <lcd_out.c>
```

Program DIAL_1.C

This program illustrates a telephone dialer that might be used in a remote monitor or alarm.

When the pushbutton on PORTB.0 goes to ground, the processor operates an LED (dial pulse relay) on PORTD.4. Following a brief delay to assure dial tone is probably present, the processor dials the telephone number, waits for a party to answer and then sends the quantity in the form of zips (or beeps) using a speaker on PORTD.0. For example, the quantity 103 is sent as one beep, followed by ten beeps, followed by three beeps. This is repeated three times and the processor then hangs up.

The momentary push button might in fact be a timer or alarm detector.

Note that the telephone number is stored as a constant array;

```
    const byte tel_num[20];
```

The advantage of using a "const" array is that the array is implemented in program memory and initialized when programming the PIC. With the CCS compiler, a const array cannot be passed to a function. However, I have never found this to be a serious obstacle.

In function dial_tel_num(), each digit is fetched from the constant array and the digit is passed to function dial_digit() until the "end of number" indicator (0x0f) is encountered.

In function dial_digit(), the digit is pulsed out at 10 pulses per second with a 63 percent break. Note that when the digit is zero, the number of pulses sent is ten.

On completion of dialing the telephone number, and a brief delay, the quantity is sent using zip tones. In this example, I used a temperature of 103 degrees. In function send_quan(), the hundreds, tens and units are passed in turn to function zips(). Function zips() calls function zip() the specified number of times with a 200 ms delay between each beep. Note that if the quantity is zero, 10 beeps are sent.

Function zip() repeatedly brings PORTD.0 high and low with two one ms delays which results in a tone of nominally 500 Hz. This is repeated duration / 2 times.

```
// Program DIAL_1.C
//
// Dials the telephone number 1-800-555-1212 and sends data T_F using
// 200 ms zips of nominally 500 Hz.  The send data sequence is repeated
// three times and the processor then hangs up.
//
// LED (simulating dial pulse relay) on PORTD.4.  Speaker through 47
// uFd on PORTD.0.  Pushbutton on input PORTB.0.
//
// copyright, Peter H. Anderson, Baltimore, MD, Dec, '00

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>

void dial_tel_num(void);
void dial_digit(byte num);
void send_quan(byte q);
void zips(byte x);
void zip(byte duration);

void main(void)
{

   byte T_F = 103, n;

   pspmode = 0;

   portd4 = 0;
   trisd4 = 0;             // dial pulse relay

   trisd0 = 0;             // speaker

   trisb0 = 1;             // pushbutton is an input
   not_rbpu = 0;           // enable internal pullups

   while(1)
   {

      while(rb0)           // loop until pushbutton depressed
      {
      }
      portd4 = 1;          // go off hook
      delay_ms(1000);      // wait for dial tone

      dial_tel_num();

      delay_ms(1000);      // wait for answer

      for (n=0; n<3; n++)  // send the quantity T_F three time
      {
         send_quan(T_F);
         delay_ms(1500);
      }
```

```
      portd4 = 0;          // back on-hook
   }

}


void dial_tel_num(void)
{
    const byte tel_num[20] = {1, 8, 0, 0, 5, 5, 5, 1, 2, 1, 2, 0x0f};
    byte n;

    for (n=0; n<20; n++)      // up to 20 digits
    {
        if (tel_num[n] == 0x0f)  // if no more digits
        {
            break;
        }

        else
        {
            dial_digit(tel_num[n]);
        }
        delay_ms(500);  // inter digit delay
    }
}

void dial_digit(byte num)
{
    byte n;
    for (n=0; n<num; n++)
    {
       portd4 = 0;   // 63 percent break at 10 pulses per second
       delay_ms(63);
       portd4 = 1;
       delay_ms(37);
    }
}

void send_quan(byte q)
{
    byte x;
    if (q > 99)         // if three digits
    {
       x = q/100;
       zips(x);         // sned the hundreds
       delay_ms(500);
       q = q % 100;     // strip off the remainder
    }
    x = q / 10;
    zips(x);            // send the tens
    delay_ms(500);
    x = q % 10;
    zips(x);            // units
}

void zips(byte x)
```

```
{
    byte n;
    if (x == 0)
    {
        x = 10;
    }
    for (n=0; n0; n--) // duration/2 * 2 ms
   {
       portd0 = 1;
       delay_10us(100); // 1 ms
       portd0 = 0;
       delay_10us(100);
   }
}

#include
```

Using the LCD.

A 20X4 DMC20434 LCD along with a 74HC595 shift register was included with the full development package (Figures #5 and #6). The software routines to support this circuitry are contained in lcd_out.c. Note that this uses Port E, bits 0, 1 and 2. The idea in using these bits was that aside from A/D converter inputs, they serve no function other than general purpose IO.

A description of the various routines is included in lcd_out.c but for your convenience it also appears below;

```
// Program LCD_OUT.C
//
// This collection of routines provides an interface with a 20X4 Optrex
// DMC20434 LCD using a 74HC595 Shift Register to permit the display
// of text.  This uses PIC outputs PORTE2::PORTE0.
//
// Also provides delay_10us() and delay_ms() timing routines which
// are implemented using looping.  Note that although these routines
// were developed for 4.0 MHz (1 usec per instruction cycle) they may
// be used with other clock frequencies by modifying delay_10us.
//
// Routine lcd_init() places the LCD in a 4-bit transfer mode, selects
// the 5X8 font, blinking block cursor, clears the LCD and places the
// cursor in the upper left.
//
// Routine lcd_char(byte c) displays ASCII value c on the LCD.  Note
// that this permits the use of printf statements;
//
//      printf(lcd_char, "T=%f", T_F).
//
// Routine lcd_dec_byte() displays a quantity  with a specified number
// of digits. Routine lcd_hex_byte() displays a byte in two digit hex
// format.
//
// Routine lcd_str() outputs the string. In many applications, these
// may be used in place of printf statements.
//
// Routine lcd_clr() clears the LCD and locates the cursor at the upper
// left.  lcd_clr_line() clears the specified line and places the
```

```
// cursor at the beginning of that line.  Lines are numbered 0, 1, 2, and 3.
//
// Routine lcd_cmd_byte() may be used to send a command to the lcd.
//
// Routine lcd_cursor_pos() places the cursor on the specified line
// (0-3) at the specified position (0 - 19).
//
// The other routines are used to implement the above.
//
//   lcd_data_nibble() - used to implement lcd_char.  Outputs the
//   specified nibble.
//
//   lcd_cmd_nibble() - used to implement lcd_cmd_byte.  The difference
//   between lcd_data_nibble and lcd_cmd_nibble is that with data, LCD
//   input RS is at a logic one.
//
//   lcd_shift_out() - used to implement the nibble functions.
//
//   num_to_char() - converts a digit to its ASCII equivalent.
//
// copyright, Peter H. Anderson, Baltimore, MD, Dec, '00
```

Program LCD_TST.C.

This routine is intended to illustrate most of the features contained in lcd_out.c.

Note that the LCD must be initialized by a call to routine lcd_init(). Note that the ADCON1 register (See Section 11 of the Data Sheet) must be configured such that PORTE2::0 are not configured as A/D inputs. In the lcd_init() routine, I opted for configuration 2/1. lcd_init() also places the LCD in a 4-bit transfer mode, sets the font and cursor type and homes the cursor to the upper left.

This routine displays byte variable q in decimal with leading zero suppression using lcd_byte() and in tow digit hexadecimal using lcd_hex(). These are both displayed on the same line with a separation using routine lcd_cursor_pos().

Note that the standard printf may also be used in conjunction with lcd_char;

```
    printf(lcd_char, "%d    %x", q, q)
```

The routine also illustrates the display of a float using the standard printf %f format specifier and presents an alternate technique. Although the second appears more cumbersome, you may wish to tinker with each and verify that a printf using the "%f" format specifier uses a good deal of program memory.

```
// Program LCD_TST.C
//
// Illustrates how to display variables and text on LCD using
// LCD_OUT.C.
//
// Copyright, Peter H. Anderson, Baltimore, MD, Dec, '00

#case

#device PIC16F877 *=16 ICD=TRUE
```

```
#include <defs_877.h>
#include <lcd_out.h>

void main(void)
{
    byte q, T_F_whole, T_F_fract;
    float T_F;
    long temp;

    pcfg3 = 0; pcfg3 = 1; pcfg2 = 0; pcfg0 = 0;
            // configure A/D for 3/0 operation
            // this is necessary to use PORTE2::0 for the LCD
    lcd_init();
    q = 0;

    while(1)
    {
       lcd_clr_line(0);       // beginning of line 0
       lcd_dec_byte(q, 3);
       lcd_cursor_pos(0, 10); // line 0, position 10
       lcd_hex_byte(q);

       lcd_clr_line(1);       // advance to line 1
       printf(lcd_char, "   Hello World   ");

       T_F = 76.6  + 0.015 * ((float) (q));
       lcd_clr_line(2);
       printf(lcd_char, "T_F = %f", T_F); // print a float

       lcd_clr_line(3);       // to last line
       printf(lcd_char, "T_F = ");

       temp = (long)(10.0 * T_F);   // separate T_F into two bytes
       T_F_whole = (byte)(temp/10);
       T_F_fract = (byte)(temp%10);

       if (T_F_whole > 99)    // leading zero suppression
       {
          lcd_dec_byte(T_F_whole, 3);
       }
       else if (T_F_whole > 9)
       {
          lcd_dec_byte(T_F_whole, 2);
       }
       else
       {
          lcd_dec_byte(T_F_whole, 1);
       }

       lcd_char('.');
       lcd_dec_byte(T_F_fract, 1);

       ++q;       // dummy up a new value of q

       delay_ms(1000);
   }
```

```
}

#include <lcd_out.c>
```

Program FONT.C

This program continually increments byte n and displays the value in decimal, hexadecimal and as a character. The intent is to illustrate the LCD characters assigned to each value.

```
// Program FONT.C
//
// Sequentially outputs ASCII characters to LCD
//
// copyright, Peter H. Anderson, Baltimore, MD, Dec, '00


#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>

void main(void)
{
    byte n;


    pcfg3 = 0; pcfg3 = 1; pcfg2 = 0; pcfg0 = 0;
            // configure A/D for 3/0 operation
            // this is necessary to use PORTE2::0 for the LCD
    lcd_init();

    for (n=0; ; n++)          // byte rolls over from 0xff to 00
    {

       lcd_clr_line(0);       // beginning of line 0
       printf(lcd_char, "%u  %x   %c", n, n, n);
       delay_ms(2000);
    }

}

#include <lcd_out.c>
```

Program TOGGLE_1.C

This program toggles the state of an LED on PORTD.4 when a pushbutton on PORTB.0/INT is depressed. It uses the external interrupt feature of the PIC (See Section 12 of the PIC16F877 Data Sheet).

Note that weak pullup resistors are enabled;

```
    not_rbpu = 0;
```

The edge that causes the external interrupt is defined to be the negative going edge;

```
    intedg = 0;
```

The not_rbpu and intedg bits are in the OPTION register and are discussed in Section 2 of the PIC16F87X Data Sheet.

Interrupts are discussed in Section 12.

```
// Program TOGGLE_1.C
//
// Reverses the state of an LED on PORTD.4 when pushbutton on input PORTB.0 is
// momentarily depressed.  Also, continually outputs to the LCD.
//
// Note that there is a problem with switch bounce where an even number of
// bounces will cause an even number of toggles and thus the LED will not appear
// to change
//
// copyright, Peter H. Anderson, Baltimore, MD, Dec, '00

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>

void main(void)
{
    byte n;

    pspmode = 0;  // PORTD as general purpose IO

    portd4 = 0;   // be sure LED is off
    trisd4 = 0;   // make it an output

    trisb0 = 1;   // make an input (not really neccessary)

    not_rbpu = 0; // enable weak pullup resistors on PORTB
    intedg = 0;   // interrupt on falling edge

    intf = 0;     // kill any unwanted interrupt
    inte = 1;     // enable external interrupt

    gie = 1;      // enable all interrupts

    pcfg3 = 0; pcfg3 = 1; pcfg2 = 0; pcfg0 = 0;
            // configure A/D for 3/0 operation
            // this is necessary to use PORTE2::0 for the LCD
    lcd_init();

    for (n=0; ; n++)    // continually
    {
       lcd_clr_line(0);       // beginning of line 0
       printf(lcd_char, "%u  %x   %c", n, n, n);
```

```
       delay_ms(2000);
    }

}

#int_ext ext_int_handler(void)
{
    portd4 = !portd4;   // invert the state of output
}

#int_default default_int_handler(void)
{
}

#include <lcd_out.c>
```

Analog to Digital Conversion.

See Section 11 of the PIC16F87X Data Sheet.

Program AD_1.C

Program AD_1.C sets up the A/D converters for a 3/0 configuration (pcfg bits), right justified result (adfm), internal RC clock (adcs1 and adcs0), measurement on channel 0 (chs2, chs1, chs0), turns on the A/D (adon) and initiates a conversion by setting bit adgo. The routine then loops until bit adgo goes to zero.

The A/D result is then displayed on the LCD. The angle of the potentiometer is then calculated and then displayed.

There is a natural inclination to fetch the result as;

```
    ad_val = ADRESH << 8 | ADRESL;        // wrong
```

However, note that ADRESH is a byte and thus, after shifting it eight bits to the left, the result of the first term will be zero.

An alternative is;

```
    long high_byte;
    ...
    high_byte = ADRESH;
    ad_val = high_byte << 8 | ADRESH;
```

In the following, I opted not to introduce the extra variable high_byte and simply used ad_val;

```
    ad_val = ADRESH;
    ad_val = ad_val << 8 | ADRESL;

// Program AD_1.C
//
// Illustrates the use of the A/D using polling of the adgo
// bit.  Continually measures voltage on potentiometer on AN0
```

```
// and displays A/D value and angle.
//
//
// copyright, Peter H. Anderson, Baltimore, MD, Dec, '00

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h> // LCD and delay routines

main()
{
   long ad_val;
   float angle;

   pcfg3 = 0; pcfg2 = 1; pcfg1 = 0; pcfg0 = 0;
   // config A/D for 3/0

   lcd_init();

   adfm = 1;   // right justified
   adcs1 = 1; adcs0 = 1; // internal RC

   adon=1;  // turn on the A/D
   chs2=0;  chs1=0;   chs0=0;

   delay_10us(10);          // a brief delay

   while(1)
   {
      adgo = 1;
      while(adgo)    ;  // poll adgo until zero
      ad_val = ADRESH;
      ad_val = ad_val << 8 | ADRESL;
      angle = (float) ad_val * 270.0 / 1024.0;
      lcd_clr_line(0);
      printf(lcd_char, "%ld", ad_val);
      lcd_clr_line(1);
      printf(lcd_char, "Angle = %2.1f", angle);
      delay_ms(3000);   // three second delay
   }
}

#include <lcd_out.c>
```

Program AD_2.C

Program AD_2.C is functionally the same as AD_1.C except that the processor is placed in the sleep mode while the A/D conversion is being performed;

```
    adgo = 1;       // start the conversion
#asm
    CLRWDT
```

```
    SLEEP
#endasm
    // a/d conversion is complete
```

The advantage is that the switching noise associated with the processor is minimized during the A/D conversion. Note that when using this implementation, the internal RC oscillator must be used.

At the recent PIC Workshop we were also using CCP1 to PWM a motor on CCP1/RC2. I expected that the PWM would cease during the time the processor was in the sleep mode. I was surprised to find that the PWM did not come on after the sleep mode was exited. (I assume that simply turning timer2 on again would have resolved this problem; tmr2on = 1).

In another application, we were rapidly switching between A/D 0 and A/D 1 and not leaving sufficient time for the sample and hold circuit to "capture" a valid sample. Thus, when changing channels, allow a delay prior to beginning the conversion.

One point that has bitten me dozens of times is that an A/D interrupt is propagated only if bit peie is set. See Section 12.10 of the 16F87X Data Sheet.

There is one snippet that involves disabling the general interrupt enable in the following code which may appear confusing;

```
    while(gie)
    {
       gie = 0;
    }
```

There is a very subtle point here. Assume the code had been written as;

```
#asm
    CLRWDT
    SLEEP
#endasm
    gie = 0;
    // subsequent instructions
```

Although this routine is not a good example, assume that an interrupt occurs just as the processor begins to execute the gie=0. The processor will complete executing the current instruction and program flow will transfer to the interrupt service routine. However, on return from the ISR the internal architecture of the PIC is such that the gie bit will be a logic one. Thus, the processor will continue on executing subsequent instructions with gie set to one.

As noted, this routine is not a good example, but this is a bug which is very hard to find and thus I have made it a habit to always turn off interrupts by continually setting gie to zero until it is actually at zero.

```
// Program AD_2.C
//
// Illustrates the use of the A/D using interrupts.  Continually measures
// voltage on potentiometer on AN0 and displays A/D value and angle.
//
//
```

```
// copyright, Peter H. Anderson, Baltimore, MD, Dec, '00

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h> // LCD and delay routines

main()
{

   long ad_val;
   float angle;

   pcfg3 = 0; pcfg2 = 1; pcfg1 = 0; pcfg0 = 0;
                        // config A/D for 3/0

   lcd_init();

   adfm = 1;            // right justified
   adcs1 = 1; adcs0 = 1; // internal RC

   adon=1;              // turn on the A/D
   chs2=0;  chs1=0;   chs0=0;
   delay_10us(10);

   while(1)
   {
      adif = 0;         // kill any previous interrupt - just to be sure
      adie = 1;         // enable A/D interrupt
      peie = 1;         // enable peripheral interrupts
      gie = 1;
      adgo = 1;
#asm
      CLRWDT
      SLEEP
#endasm
      while(gie)        // be sure gie is off
      {
          gie = 0;      // turn of interrupts
      }
      ad_val = ADRESH;
      ad_val = ad_val << 8 | ADRESL;
      angle = (float) ad_val * 270.0 / 1024.0;
      lcd_clr_line(0);
      printf(lcd_char, "%ld", ad_val);
      lcd_clr_line(1);
      printf(lcd_char, "Angle = %2.1f", angle);
      delay_ms(3000);   // three second delay
   }
}

#int_ad ad_int_handler(void)
{
}
```

```
#int_default default_int_handler(void)
{
}

#include <lcd_out.c>
```

Program TOGGLE_2.C

This routine combines aspects of routines TOGGLE_1.C and AD_2.C. The program continually loops with an A/D conversion being performed nominally every three seconds with the LED on PORTD.0 being toggled each time the pushbutton on PORTB.0 is depressed.

Note that the external interrupt is momentarily disabled during the brief time the A/D conversion is being performed.

Prior to enabling an interrupt, I usually clear the corresponding flag bit;

```
  intf = 0;      // kill flag
  inte = 1;      // and enable external interrupt
```

```
// Program TOGGLE_2.C
//
// Illustrates the use of the A/D using interrupts.  Continually measures
// voltage on potentiometer on AN0 and displays A/D value and angle.
//
// Also toggles LED on PORTD.0 when pushbutton on PORTB.0 is depressed.
//
// copyright, Peter H. Anderson, Baltimore, MD, Dec, '00

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h> // LCD and delay routines

main()
{

   long ad_val;
   float angle;

   pcfg3 = 0; pcfg2 = 1; pcfg1 = 0; pcfg0 = 0;
   // config A/D for 3/0

   lcd_init();

   adfm = 1;             // right justified
   adcs1 = 1; adcs0 = 1; // internal RC

   adon=1;  // turn on the A/D
   chs2=0;  chs1=0;   chs0=0;
   delay_10us(10);   // brief delay to allow capture

   not_rbpu = 0;     // internal pullup enabled
```

```
   intedg = 0;       // negative going transition

   trisb0 = 1;

   pspmode = 0;

   portd4 = 0;       // start with LED off
   trisd4 = 0;

   gie = 1;

   while(1)
   {
      inte = 0;      // disable external interrupt

      adif = 0;      // kill any previous interrupt
      adie = 1;      // enable A/D interrupt
      peie = 1;      // enable peripherals

      adgo = 1;
#asm
      CLRWDT
      SLEEP
#endasm
      adie = 0;      // disable A/D interrupts
      intf = 0;
      inte = 1;      // and enable external interrupt

      ad_val = ADRESH;
      ad_val = ad_val << 8 | ADRESL;
      angle = (float) ad_val * 270.0 / 1024.0;
      lcd_clr_line(0);
      printf(lcd_char, "%ld", ad_val);
      lcd_clr_line(1);
      printf(lcd_char, "Angle = %2.1f", angle);
      delay_ms(3000);   // three second delay
   }
}

#int_ad ad_int_handler(void)
{
}

#int_ext external_int_handler(void)
{
    portd4 = !portd4;
}

#int_default default_int_handler(void)
{
}

#include <lcd_out.c>
```

Program PWM_1.C

Note that the use of the CCP modules is discussed in Section 8 of the PIC16F87X data sheet. Operation of Timer 2 is discussed in Section 7.

This routine illustrates the use of the CCP modules for generating PWM. The PIC16F87X family all have two CCP modules and both may be configured for PWM, both using the same period.

Both use 8-bit Timer 2 as a time base which is clocked by the PIC's clock; fosc/4. This may be prescaled to 1:1, 1:4 or 1:16 using bits t2ckps1 and t2ckps0. This routine uses 1:1 and thus Timer 2 has a periodicity of 256 usecs (about 4.0 kHz) when using a 4.0 MHz clock.

Both use the period register PR2 which controls the periodicity of Timer 2. Thus, if PR2 is set to 0x3f (63), Timer2 increments from zero to 63 and then rolls over to zero. Thus, the periodicity is 64 usecs (about 16 kHz) when using a 4.0 MHz clock.

CCPR1L and CCPR2L are associated with the duty of the CCP1 and CCP2 modules, respectively. Thus, if CCPR1L is set to 63 and PR2 is set to 255, Timer 2 will count from 0 to 63 (64 usecs) and during this time, the CCP1 output will be high and from 64 to 255 (192 usecs) the CCP1 output will be low. Thus, the duty cycle will be 25 percent.

Most of the terminology makes sense. Timer 2 and PR2 are associated with both modules and CCPR1L and CCPR2L are associated with the CCP1 and the CCP2 modules, respectively. The thing that doesn't make sense is that the CCP1 output is PORTC.2 and the CCP2 output is PORTC.1. It took me a good deal of time to decipher this.

In this routine, the Timer 2 prescale is set to 1:1 using the t2ckps1 and t2ckps0 bits. I don't believe the post scale feature affects the CCP in the PWM mode, but I set them to 1:1 by clearing the toutps3, toutps2, toutps1 and toutps0 bits. Timer 2 is turned on using the tmr2on bit.

The PWM mode is selected by setting bits ccp1m3 and ccp1m2.

PORTC.2 is configured as an output.

Changing the duty cycle is then simply a matter of modifying CCPR1L. In this routine, the duty is decreased toward zero when the push button on PORTB.0 is open (logic one) and increased toward 255 when the push button is closed to ground.

I opted to increase or decrease the duty in steps of five which leads to the subtle point that when working with an unsigned char, all values other than zero are greater than zero. That is, there is no minus. Consider the following that might be used when decreasing the duty;

```
if (duty > 0)       // wrong
{
   duty = duty - 5;
}
```

If duty is 3, the new duty is calculated as -2, which in reality is 254 and of course the next time the expression is evaluated, duty will be greater than 0. That is, it will be 254 and not -2. Thus, in the following, note that I go through a bit of trickery to assure that when decreasing the duty, I don't roll past 0 and when increasing the duty, that I don't roll past 0xff (255).