PIC16F87X Tutorial by Example

Document History.

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).

```
// Program PWM_1.C
//
// Illustrates use of CCP1 to implement 8-bit PWM on RC2/CCP1.
//
// When pushbutton is open (released), duty cycle decrease to zero.  When
// pushbutton is depressed, duty cycle slowly increases to the maximum of
// 255.
//
//
// 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()
{

   byte duty;

   not_rbpu = 0;  // enable weak pullups
   trisb0 = 1;

   PR2 = 0xff;    // period set to max of 256 usecs  - about 4 kHz
   duty = 0x00;
   CCPR1L = duty;  // duty initially set to zero

   // configure CCP1 for PWM operation
   ccp1m3 = 1;  ccp1m2 = 1;

   // Timer 2 post scale set to 1:1
   toutps3 = 0;  toutps2 = 0; toutps1 = 0; toutps0 = 0;

   // Timer 2 prescale set to 1:1
   t2ckps1 = 0; t2ckps0 = 0;

   tmr2on = 1; // turn on timer #2

   portc2 = 0;
   trisc2 = 0; // make PORTC.2 an output 0

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

   lcd_init();

   while(1)
   {
       if (rb0)   // go down
       {
           if (duty < 5)
           {
```

```
                duty = 0;
            }
            else
            {
                duty = duty - 5;
            }
        }
        else // increase duty
        {
            if (duty > (0xff - 5))
            {
                duty = 0xff;   // max
            }
            else
            {
                duty = duty + 5;
            }
        }
        CCPR1L = duty;
        lcd_cursor_pos(0, 0);
        lcd_hex_byte(duty);
        delay_ms(25);
    }
}

#include
```

Program PWM_2.C

This program continually reads the value of the potentiometer on A/D channel 0 and sets the duty cycle of the PWM on output PORTC.2. The duty cycle is displayed on the LCD.

Note that the A/D result is right justified (adfm = 0). Thus, the 8-bit duty cycle is simply a matter of reading ADRESH.

I used this in a design for a landfill in South Carolina to control the speed of leachate pumps. Prior to that time, they were using resistors in series with the motor winding.

```
// Program PWM_2.C
//
// Varies PWM duty using potentiometer on A/D Ch0 and outputs
// the value of "duty" to LCD.
//
// Uses 8-bit PWM.  The period is 1/256 us or about 4KHz.
//
// 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()
```

```
{

   byte duty;

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

   lcd_init();

   // set up A/D converter
   adfm = 0;   // left justified - high 8 bits in ADRESH
   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

   // Configure CCP1

   PR2 = 0xff;    // period set to max of 256 usecs  - about 4 kHz
   duty = 0x00;
   CCPR1L = duty;  // duty initially set to zero

   // configure CCP1 for PWM operation
   ccp1m3 = 1;  ccp1m2 = 1;

   // Timer 2 post scale set to 1:1
   toutps3 = 0;  toutps2 = 0; toutps1 = 0; toutps0 = 0;

   // Timer 2 prescale set to 1:1
   t2ckps1 = 0; t2ckps0 = 0;

   tmr2on = 1; // turn on timer #2

   portc2 = 0;
   trisc2 = 0; // make PORTC.2 an output 0

   while(1)
   {
      adgo = 1;
      while(adgo)    ;  // poll adgo until zero
      duty = ADRESH;
      CCPR1L = duty;
      lcd_cursor_pos(0, 0);
      printf(lcd_char, "%x", duty);
      delay_ms(25);
   }
}

#include <lcd_out.c>
```

Program PWM_3.C

This routine differs from the above only in that it provides for 10-bit resolution. CCPR1L is the upper eight bits and bits ccp1x and ccp1y are lower the lower two bits of the duty.

The upper eight bits of the period is determined by PR2 and the lower two bits by the prescale value of timer 2 (bits t2ckps1 and t2ckps0). The fact that the lower two bits are also used to control the clock rate of timer 2 confuses me as it would seem as if setting them both to 1 to achieve a period of 0x3ff has the curious effect of slowing the PWM by a factor of 16.

Therefore, in the following, I opted to keep the prescale bits at logic 0 and thus, the period is 0x3fc. (binary 11 1111 1100).

In the following, the A/D conversion format was left justified which permitted CCPR1L to be simply updated by ADRESH and bits ccp1x and ccp1y by the most significant bits of ADRESL using a two byte structure DUTY as an intermediate variable. In fact, in displaying the duty, I was forced to perform some shift operations and put the two bytes together. However, the routine does illustrate the use of a simple structure.

Note that although I did visually verify the operation of this routine, I did not look at the period on a scope.

```
// Program PWM_3.C
//
// Varies PWM duty using potentiometer on A/D Ch0 and outputs
// the value of "duty" to LCD.
//
// Uses 10-bit PWM.  The period is 1/256 us.
//
// 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()
{

   struct DUTY
   {
       byte hi8;  // high 8 bits of duty cycle
       byte lo2;  // low 2 bits in the highest two bits
   };

   struct DUTY duty;
   long duty_l;


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

   lcd_init();

   // set up A/D converter
   adfm = 0;   // right justified -
               // high 8 bits in ADRESH, lowest two bits in high bits of ADRESL
   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

   // Configure CCP1
   duty.hi8 = 0x00;
   duty.lo2 = 0x00;

   PR2 = 0xff;    // period set to max of 256 * 4 usecs  - about 1 kHz

   CCPR1L = duty.hi8;      // duty initially set to zero
   ccp1x = 0;  ccp1y = 0;  // low 8-bits of duty

   // configure CCP1 for PWM operation
   ccp1m3 = 1;  ccp1m2 = 1;

   // Timer 2 post scale set to 1:1
   toutps3 = 0;  toutps2 = 0; toutps1 = 0; toutps0 = 0;

   // Timer 2 prescale set to 1:1
   t2ckps1 = 0; t2ckps0 = 0;

   tmr2on = 1; // turn on timer #2

   portc2 = 0;
   trisc2 = 0; // make PORTC.2 an output 0

   while(1)
   {
      adgo = 1;
      while(adgo)    ;  // poll adgo until zero
      duty.hi8 = ADRESH;
      duty.lo2 = ADRESL;
      CCPR1L = duty.hi8;      // high 8-bits

      ccp1x = 0;  ccp1y =0;   // low 2 bits
      if (duty.lo2 & 0x80)
      {
          ccp1x = 1;
      }
      if (duty.lo2 & 0x40)
      {
          ccp1y = 1;
      }
      lcd_cursor_pos(0, 0);
      duty_l = ((long) (duty.hi8)) * 4 + (duty.lo2 >> 6);
      printf(lcd_char, "%lx", duty_l);
      delay_ms(25);
   }
}

#include <lcd_out.c>
```

Program TIMER2_1.C

This routine uses timer 2 to generate a 500 Hz tone on the speaker using interrupts which frees the processor to perform other tasks at the same time.

When the push button on input PORTB.0 is depressed, a 500 Hz tone is continually generated on PORTD.0 and A/D conversions of Channel 0 are performed and displayed on the LCD.

Note that the timer 2 prescale value was set to 1:4 and the PR2 register to 250, thus causing an interrupt each millisecond (for a 4.0 MHz clock). Note that the post scale is set to 1:1.

```
// Program TIMER2_1.C
//
// Generates nominal 500 Hz tone on PORTD.0 and performs continual A/D
// conversions on Channel 0 when push button on PORTB.0 is depressed.
//
// Illustrates use of TIMER2.
//
// 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()
{

   byte duty;
   long ad_val;

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

   lcd_init();

   not_rbpu = 0;  // enable weak pullups
   trisb0 = 1;

   pspmode = 0;
   portd0 = 0;    // make speaker an ouput 0
   trisd0 = 0;

   // Set up timer2
   PR2 = 250;    // period set to 250 * 4 usecs  = 1 ms

   // Timer 2 post scale set to 1:1
   toutps3 = 0;  toutps2 = 0; toutps1 = 0; toutps0 = 0;

   // Timer 2 prescale set to 1:4
   t2ckps1 = 0; t2ckps0 = 1;

   // set up A/D
   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)
   {
       if (!rb0)
       {
           tmr2ie = 1;  // enable interrupts
           peie = 1;
           tmr2on = 1;  // and turn on timer 2
           gie = 1;

           adgo = 1;
           while(adgo)     ;
           ad_val = ADRESH;
           ad_val = ad_val << 8 | ADRESL;

           lcd_cursor_pos(0, 0);
           printf(lcd_char, "%ld  ", ad_val);
           delay_ms(25);
       }
       else // do nothing
       {
           while(gie)
           {
               gie = 0;
           }
           tmr2ie = 0;
           tmr2on = 0;
       }

   }
}

#int_timer2 timer2_int_handler(void)
{
    portd0 = !portd0;
}

#include <lcd_out.c>
```

Program TIMER1_1.C

Program TIMER1_1.C illustrates the use of the 16-bit Timer 1 in conjunction with a 32.768 kHz crystal to time for one second periods. At the end of each second, the elapsed time in seconds and in hour, minute, second format is displayed and the speaker is beeped for nominally 200 ms.

The use of Timer 1 is discussed in Section 6 of the PIC16F87X Data Sheet. In this routine, the external oscillator circuitry is enabled (bit t1oscen) and the external clock is selected (bit tmr1cs). The prescale is set to 1:1 by setting the t1ckps1 and t1ckps0 bits to zero. The timer is turned on by setting the tmr1on bit. The interrupt is enabled by setting the tmr1ie bit.

Bytes TMR1H and TMR1L are preloaded to 0x8000 such that after 0x8000 (32,678) transitions the counter rolls over and generates an interrupt. Note that on interrupt, only the high byte TMR1H is loaded with 0x80. The low byte (TMR1L) is not set to zero as some time has elapsed in processing the interrupt. That is, if TMR1L has incremented to 17 by the time the the interrupt is serviced, resetting it to zero would have the effect of having a period of;

```
    (32768 + 17) / 32768 seconds
```

The interrupt service routine communicates with the main() using global variable timer1_int_occ. This is set to TRUE only in the interrupt service routine. Thus, main()continually tests this variable and on finding it to be TRUE, performs the required tasks and clears the variable to FALSE.

When finding tmr1_int_occ to be true, the program increments and displays the elapsed time and also increments and displays the time in hour, minute, second format by one second. This is implemented by passing a structure of type TM to function increment_time. Note that with the CCS compiler, structures must be passed by reference.

In addition, the speaker is beeped, function blip_tone(), for nominally 200 ms by configuring Timer 2 for an interrupt each millisecond, calling function delay_ms and then turning Timer 2 off and disabling the tmr2ie. Note that the actual duration of the tone will be somewhat longer than 200 ms due the overhead in processing the Timer2 interrupts which occur each millisecond.

```
// Program TIMER1_1.C
//
// Illustrates the use of Timer 1 with the external 32.768 kHz crystal T1OSC0
// and T1OSC1.
//
// Each second, briefly blips the speaker and displays the elapsed time in
// seconds and in hour:minute:sec format on the LCD.
//
// 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

#define TRUE !0
#define FALSE 0

struct TM
{
    byte hr;
    byte mi;
    byte se;
};

void blip_tone(void);
void increment_time(struct TM *t);

byte timer1_int_occ;    // note that this is global
```

```
main()
{

   byte duty;
   long elapsed_t;
   struct TM t;

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

   lcd_init();

   pspmode = 0;
   portd0 = 0;    // make speaker an ouput 0
   trisd0 = 0;

   // Set up timer2
   PR2 = 250;    // period set to 250 * 4 usecs  = 1 ms

   // Timer 2 post scale set to 1:1
   toutps3 = 0;  toutps2 = 0; toutps1 = 0; toutps0 = 0;

   // Timer 2 prescale set to 1:4
   t2ckps1 = 0; t2ckps0 = 1;

   // Set up timer1
   t1oscen = 1;   // enable external crystal osc circuitry
   tmr1cs = 1;    // select this as the source

   t1ckps1 = 0;   t1ckps0 = 0;   // prescale of 1

   tmr1if = 0;    // kill any junk interrupt
   TMR1L = 0x00;
   TMR1L = 0x80;

   tmr1ie = 1;
   peie = 1;
   gie = 1;

   timer1_int_occ = FALSE;

   elapsed_t = 0;                // start with elapsed time = 0
   t.hr = 0;  t.mi = 0; t.se = 0;

   tmr1on = 1;

   lcd_clr_line(2);
   printf(lcd_char, "Impress the spouse");
   lcd_clr_line(3);
   printf(lcd_char, "with a personal msg!");

   while(1)
   {
       if (timer1_int_occ)
       {
           timer1_int_occ = FALSE;
```

```
            ++elapsed_t;
            increment_time(&t);
            lcd_clr_line(0);
            printf(lcd_char, "%ld    ", elapsed_t);

            lcd_clr_line(1);
            lcd_dec_byte(t.hr, 2);
            lcd_char(':');
            lcd_dec_byte(t.mi, 2);
            lcd_char(':');
            lcd_dec_byte(t.se, 2);
            blip_tone();
        }
        // else do nothing
    }
}

void blip_tone(void)
{
   tmr2ie = 1;    // turn on timer 2 and enable interrupts
   peie = 1;
   tmr2on = 1;
   gie = 1;

   delay_ms(200); // tone for nominally 200 ms

   tmr2ie = 0;
   tmr2on = 0;
}

void increment_time(struct TM *t)
{
    ++t->se;
    if (t->se > 59)
    {
        t->se = 0;
        ++t->mi;
        if (t->mi > 59)
        {
            t->mi = 0;
            ++t->hr;
            if (t->hr > 23)
            {
                t->hr = 0;
            }
        }
    }
}

#int_timer1 timer1_int_handler(void)
{
    timer1_int_occ = TRUE;
    TMR1H = 0x80;
}

#int_timer2 timer2_int_handler(void)
{
```

```
    portd0 = !portd0;
}

#include <lcd_out.c>
```

Program THERM_1.C

This program extends program TIMER1_1.C to also perform an A/D conversion on Channel 1 which is configured with a series 10.0K fixed resistor and a nominal 10K negative temperature coefficient (NTC) thermistor in a voltage divider arrangement. The A/D value is used to determine the resistance of the thermistor (r_therm) and this is used to determine the temperature which is displayed in degrees C and degrees F on the LCD display.

Using voltage division the voltage appearing at the A/D input is;

```
    (1) V_in = r_therm / (r_therm + 10.0K) * V_ref
```

where V_ref is nominally 5.0 VDC.

Using the A/D result, V_in may be calculated;

```
    (2)     V_in = ad_val / 1024.0 * V_ref
```

Equations (1) and (2) may be combined and with a bit of algebra, the value of r_therm may be calculated;

```
    (3)     r_therm = 10.0e3 / ((1024.0 / ad_val) - 1.0)
```

Thus, the value of the NTC resistance may be calculated from the A/D value.

A good model of the NTC thermistor;

```
    (4)     T_K = 1.0 / (a + b * ln(r_therm))
```

where T_K is the temperature in degrees Kelvin and "a" and "b" are constants. For many years I have used values of a = 0.0004132 and b = 0.000320135 for these thermistors. Note that in the C language, the log() function is the natural log.

The values of T_C and T_F may then be calculated;

```
    (5)     T_C = T_K - 273.15
    (6)     T_F = T_C * 1.8 + 32.0
```

```
// Program THERM_1.C
//
// Illustrates the use of Timer 1 with the external 32.768 kHz crystal T1OSC0
// and T1OSC1.
//
// Each second, briefly blips the speaker and displays the elapsed time in
// seconds and in hour:minute:sec format on the LCD.  Also performs A/D conversion
// on A/D Ch 1 and displays the temperature is degrees C and F.
//
```

```
// 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
#include <math.h>

#define TRUE !0
#define FALSE 0

#define THERM_A 0.0004132
#define THERM_B 0.000320135

struct TM
{
    byte hr;
    byte mi;
    byte se;
};

float calc_T_C(long ad_val);
float T_C_to_T_F(float T_C);
long meas_ad1(void);

void blip_tone(void);
void increment_time(struct TM *t);

byte timer1_int_occ;    // note that this is global

main()
{

   long elapsed_t, ad_val;
   struct TM t;
   float T_F, T_C;

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

   lcd_init();

   pspmode = 0;
   portd0 = 0;    // make speaker an ouput 0
   trisd0 = 0;

   // Set up timer2
   PR2 = 250;    // period set to 250 * 4 usecs  = 1 ms

   // Timer 2 post scale set to 1:1
   toutps3 = 0;  toutps2 = 0; toutps1 = 0; toutps0 = 0;

   // Timer 2 prescale set to 1:4
   t2ckps1 = 0; t2ckps0 = 1;
```

```
    // Set up timer1
    t1oscen = 1;   // enable external crystal osc circuitry
    tmr1cs = 1;    // select this as the source

    t1ckps1 = 0;   t1ckps0 = 0;   // prescale of 1

    tmr1if = 0;    // kill any junk interrupt
    TMR1L = 0x00;
    TMR1L = 0x80;

    tmr1ie = 1;
    peie = 1;
    gie = 1;

    timer1_int_occ = FALSE;

    elapsed_t = 0;                 // start with elapsed time = 0
    t.hr = 0;  t.mi = 0; t.se = 0;

    tmr1on = 1;

    while(1)
    {
        if (timer1_int_occ)
        {
            blip_tone();
            timer1_int_occ = FALSE;
            ++elapsed_t;
            increment_time(&t);
            lcd_cursor_pos(0, 0);
            printf(lcd_char, "%ld    ", elapsed_t);

            lcd_cursor_pos(1, 0);
            lcd_dec_byte(t.hr, 2);
            lcd_char(':');
            lcd_dec_byte(t.mi, 2);
            lcd_char(':');
            lcd_dec_byte(t.se, 2);

            ad_val = meas_ad1();

            if (ad_val == 0)
            {
                ad_val = 1;      // avoid a divide by zero error
            }

            T_C = calc_T_C(ad_val);
            T_F = T_C_to_T_F(T_C);

            lcd_cursor_pos(2, 0);
            printf(lcd_char, "T_C = %3.1f  ", T_C);
            lcd_cursor_pos(3, 0);
            printf(lcd_char, "T_F = %3.1f  ", T_F);
        }
        // else do nothing
    }
}
```

```
float calc_T_C(long ad_val)
{
    float ad_val_float, r_therm, T_K, T_C;

    ad_val_float = (float) ad_val;
    r_therm = 10.0e3 / (1024.0/ad_val_float - 1.0);
    T_K = 1.0 / (THERM_A + THERM_B * log(r_therm));
    T_C = T_K - 273.15;
    return(T_C);
}

float T_C_to_T_F(float T_C)
{
    float T_F;
    T_F = T_C * 1.8 + 32.0;
    return(T_F);
}

long meas_ad1(void)
{
    long ad_val;

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

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

    adgo = 1;
    while(adgo)    ;  // poll adgo until zero
    ad_val = ADRESH;
    ad_val = ad_val << 8 | ADRESL;
    adon = 0;
    return(ad_val);
}

void blip_tone(void)
{
   tmr2ie = 1;    // turn on timer 2 and enable interrupts
   peie = 1;
   tmr2on = 1;
   gie = 1;

   delay_ms(200); // tone for nominally 200 ms

   tmr2ie = 0;
   tmr2on = 0;
}

void increment_time(struct TM *t)
{
    ++t->se;
    if (t->se > 59)
    {
        t->se = 0;
```

```
        ++t->mi;
        if (t->mi > 59)
        {
            t->mi = 0;
            ++t->hr;
            if (t->hr > 23)
            {
                t->hr = 0;
            }
        }
    }
}

#int_timer1 timer1_int_handler(void)
{
    timer1_int_occ = TRUE;
    TMR1H = 0x80;
}

#int_timer2 timer2_int_handler(void)
{
    portd0 = !portd0;
}

#include <lcd_out.c>
```

Program THERM_2.C

This routine extends THERM_1.C to also sequentially write ten A/D measurements to program EEPROM memory. This data is then fetched from EEPROM and the results are displayed on the LCD.

The intent of the routine is to illustrate how to write to and read from program EEPROM. The EEPROM is discussed in Section 4 of the PIC16F87X data sheet.

The program memory addresses of the PIC16F877 extend from 0x0000 to 0x1fff (8192 locations). It is critically important that the addresses used for logging data not be used for the actual program.

Note that the program memory is 14-bits wide and thus can accommodate the 10-bit A/D result.

```
// Program THERM_2.C
//
// Illustrates the use of Timer 1 with the external 32.768 kHz crystal T1OSC0
// and T1OSC1.
//
// Each second, briefly blips the speaker and displays the elapsed time in
// seconds and in hour:minute:sec format on the LCD.  Also, every five seconds,
// performs an A/D conversion on A/D Ch 1 and displays the temperature in degrees
// C and F.  The result of the A/D conversion is also written to program EEPROM.
//
// After ten measurements, the data is read from EEPROM and displayed on the LCD.
//
// 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
#include <math.h>

#define TRUE !0
#define FALSE 0

#define THERM_A 0.0004132
#define THERM_B 0.000320135

// #define TEST

struct TM
{
    byte hr;
    byte mi;
    byte se;
};

long get_eeprom(long adr);
void put_eeprom(long adr, long dat);

float calc_T_C(long ad_val);
float T_C_to_T_F(float T_C);
long meas_ad1(void);

void blip_tone(void);
void increment_time(struct TM *t);

byte timer1_int_occ;    // note that this is global

main()
{

   byte num_secs, n, num_samples;
   long elapsed_t, ad_val;
   struct TM t;
   float T_C, T_F;

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

   lcd_init();

   pspmode = 0;
   portd0 = 0;    // make speaker an ouput 0
   trisd0 = 0;

   // Set up timer2
   PR2 = 250;    // period set to 250 * 4 usecs  = 1 ms

   // Timer 2 post scale set to 1:1
   toutps3 = 0;  toutps2 = 0; toutps1 = 0; toutps0 = 0;
```

```
// Timer 2 prescale set to 1:4
t2ckps1 = 0; t2ckps0 = 1;

// Set up timer1
t1oscen = 1;   // enable external crystal osc circuitry
tmr1cs = 1;    // select this as the source

t1ckps1 = 0;   t1ckps0 = 0;   // prescale of 1

tmr1if = 0;    // kill any junk interrupt
TMR1L = 0x00;
TMR1L = 0x80;

tmr1ie = 1;
peie = 1;
gie = 1;

timer1_int_occ = FALSE;

elapsed_t = 0;                // start with elapsed time = 0
t.hr = 0;  t.mi = 0; t.se = 0;

tmr1on = 1;

num_secs = 0;
num_samples = 0;

while(1)
{
    if (timer1_int_occ)
    {

        timer1_int_occ = FALSE;
        ++elapsed_t;
        increment_time(&t);
        lcd_cursor_pos(0, 0);
        printf(lcd_char, "%ld    ", elapsed_t);

        lcd_cursor_pos(1, 0);
        lcd_dec_byte(t.hr, 2);
        lcd_char(':');
        lcd_dec_byte(t.mi, 2);
        lcd_char(':');
        lcd_dec_byte(t.se, 2);

        ++num_secs;

        if (num_secs == 5)
        {
            num_secs = 0;
            blip_tone();

            ad_val = meas_ad1();

            if (ad_val == 0)
            {
```

```
                    ad_val = 1;      // avoid a divide by zero error
                }

                T_C = calc_T_C(ad_val);
                T_F = T_C_to_T_F(T_C);  // convert T_C to T_F

                lcd_cursor_pos(2, 0);
                printf(lcd_char, "T_C = %3.1f  ", T_C);
                lcd_cursor_pos(3, 0);
                printf(lcd_char, "T_F = %3.1f  ", T_F);

                put_eeprom(0x1000 + num_samples, ad_val);

                ++num_samples;
                if (num_samples == 10)
                {
                    break;
                }

            }
        }
        // else do nothing
    }
    // now dump the data

    delay_ms(500);
    lcd_init();

    printf(lcd_char, "Dumping Data");
    for (n=0; n< 10; ++num_samples)
    {
        ad_val = get_eeprom(0x1000 + num_samples);
        if (ad_val == 0)
        {
            ad_val = 1;      // avoid a divide by zero error
        }

        T_C = calc_T_C(ad_val);

        lcd_cursor_pos(0, 0);
        printf(lcd_char, "%2d %3.1f  ", num_samples, T_C);
        delay_ms(1000);
    }

    lcd_clr_line(0);
    printf(lcd_char, "Done");
    delay_ms(1000);
    lcd_init();
    while(1)    // continual loop
    {
#asm
        CLRWDT
#endasm
    }
}

void put_eeprom(long adr, long dat)
```

```
{
    while(gie)       // be sure interrupts are disabled
    {
        gie = 0;
    }
    EEADRH = adr >> 8;
    EEADR = adr & 0xff;

    EEDATH = dat >> 8;
    EEDATA = dat & 0xff;
    eepgd = 1;    // program memory
    wren = 1;
    EECON2 = 0x55;
    EECON2 = 0xaa;
    wr = 1;
#asm
    NOP
    NOP
#endasm
    wren = 0;
    gie = 1;
}

long get_eeprom(long adr)
{
    long eeprom_val;
    EEADRH = adr >> 8;
    EEADR = adr & 0xff;
    eepgd = 1;
    rd = 1;
#asm
    NOP
    NOP
#endasm
    eeprom_val = EEDATH;
    eeprom_val = eeprom_val << 8 | EEDATA;
    return(eeprom_val);
}

float calc_T_C(long ad_val)
{
    float ad_val_float, r_therm, T_K, T_C;

    ad_val_float = (float) ad_val;
    r_therm = 10.0e3 / (1024.0/ad_val_float - 1.0);
    T_K = 1.0 / (THERM_A + THERM_B * log(r_therm));
    T_C = T_K - 273.15;
    return(T_C);
}

float T_C_to_T_F(float T_C)
{
    float T_F;
    T_F = T_C * 1.8 + 32.0;
    return(T_F);
}
```

```
long meas_ad1(void)
{
    long ad_val;

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

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

    adgo = 1;
    while(adgo)    ;           // poll adgo until zero
    ad_val = ADRESH;
    ad_val = ad_val << 8 | ADRESL;
    adon = 0;
    return(ad_val);
}

void blip_tone(void)
{
   tmr2ie = 1;    // turn on timer 2 and enable interrupts
   peie = 1;
   tmr2on = 1;
   gie = 1;

   delay_ms(200); // tone for nominally 200 ms

   tmr2ie = 0;
   tmr2on = 0;
}

void increment_time(struct TM *t)
{
    ++t->se;
    if (t->se > 59)
    {
        t->se = 0;
        ++t->mi;
        if (t->mi > 59)
        {
            t->mi = 0;
            ++t->hr;
            if (t->hr > 23)
            {
                t->hr = 0;
            }
        }
    }
}

#int_timer1 timer1_int_handler(void)
{
    timer1_int_occ = TRUE;
    TMR1H = 0x80;
}
```

```
#int_timer2 timer2_int_handler(void)
{
    portd0 = !portd0;
}

#include <lcd_out.c>
```

Program EEPROM_1.C.

This program illustrates how to write to and read from data EEPROM. EEPROM is treated in Section 4 of the PIC16F87X Data Sheet.

The data EEPROM differs from the flash program memory in that each address consists of eight bits (vs 14 bits), it is high endurance and a write takes considerably more time, typically 10 ms.

I attempted to implement the write_data_eeprom using the EEPROM interrupt, but this effort was not successful. The general idea is to enable the write (wren bit), write the key (0x55, 0xaa), initiate the write (wr bit) and then wait for an EEPROM interrupt. However, the processor was never interrupted and thus I abandoned the interrupt approach in favor of writing the data and then waiting 10 ms. Note that I have left my attempt in the listing and would be very interested if someone finds my error.

In developing the routines for EEPROM, I was surprised and disappointed to find that I was unable to set a break point and examine the content of EEPROM. (There is an EEPROM window, but this is only updated when you initially download the hex file and thus not terribly useful). I was able to do so with earlier ICD firmware and an earlier version of MPLAB and am hopeful this capability returns in later releases.

```
// EEPROM_1.C
//
// Illustrates how to initialize EEPROM, how to read from EEPROM and
// write to EEPROM.  Note that this EEPROM is the data EEPROM on the 16F87X.
//
// EEPROM location 00 is initialized to 100 decimal using the #rom
// directive.  Each time function dec_count is called, the program
// decrements this value and checks to see if it is at zero.  Such an
// arrangement might be used to limit the number of accesses and might
// be used with a debit card.
//
// Program continually flashes LED on PORTD4 at about 250 ms on and 250 msec
// off.  Loops indefinitely.  However, counter in EEPROM is decremented on
// each pass.  Program locks when the EEPROM counter is decremented to zero.
//
// Note that even if power is turned off prior to the completion of 100
// flashes, the latest EEPROM value will be retained for the subsequent
// run of the program.
//
// copyright, Peter H. Anderson, Baltimore, MD, Dec, '00

#case

#device PIC16F877 *=16 ICD=TRUE
#include <defs_877.h>
#include <lcd_out.h>

#define TRUE !0
#define FALSE 0
```

```
// #define INTS

void flash_led(void);
void write_data_eeprom(byte adr, byte d);
byte read_data_eeprom(byte adr);

#ifdef INTS
byte ee_int_occ;
#endif

void main(void)
{
   byte n;

   pspmode = 0;

   portd4 = 0;          // LED
   trisd4 = 0;

   lcd_init();

   while(1)
   {
      n = read_data_eeprom(0x00);   // fetch from EEPROM
      if (n==0)
      {
         break;                              // if at zero, lock up
        }
      lcd_cursor_pos(0, 0);
      printf(lcd_char, "%u   ", n);

      flash_led();                           // flash the LED one time

      --n;
      write_data_eeprom(0x00, n);   // decrement and save
   }

   lcd_clr_line(0);
   printf(lcd_char, "Locked!!!");

#asm        // lock when EEPROM value is decremented to zero
LOCK:
   CLRWDT
   GOTO LOCK
#endasm
}

void flash_led(void)
{
   portd4 = 1;
   delay_ms(250);
   portd4 = 0;
   delay_ms(250);
}

byte read_data_eeprom(byte adr)
```

```
{
   eepgd = 0;           // select data EEPROM
   EEADR=adr;
   rd=1;    // set the read bit
   return(EEDATA);
}

void write_data_eeprom(byte adr, byte d)
{
   eepgd = 0;           // select data EEPROM
#ifdef INTS
   ee_int_occ = FALSE;
   while(gie)           // be sure interrupts are off while executing the key
   {
         gie = 0;
   }

   eeif = 0;
   eeie = 1;
   peie = 1;
#endif
   EEADR = adr;
   EEDATA = d;

   wren = 1;            // write enable
   EECON2 = 0x55; // protection sequence
   EECON2 = 0xaa;

   wr = 1;              // begin programming sequence

#ifdef INTS
   gie = 1;
   while(!ee_int_occ)         ;
   ee_int_occ = FALSE;
#else
   delay_ms(10);
#endif

   wren = 0;            // disable write enable
}

#include <lcd_out.c>

#ifdef INTS

#int_eeprom eeprom_int_handler(void)
{
   ee_int_occ = TRUE;
}

#int_default default_int_handler(void)
{
}
#endif

#rom 0x2100={100} // initialize location 0 to 100
```

Program FIRST_TM.C.

In some applications it is desirable to take an action only when the program is executed the first time and not do so on future executions. In this routine an implementation of a first time function is implemented by initializing four data EEPROM locations (0x00 – 0x03) to distinctive values. Function first_tm() reads these locations and if these locations indicate this is the first time, they are set to zero and TRUE is returned. Otherwise FALSE is returned.

Another useful feature is to marry the PIC with a piece of hardware having a unique identity much like a lock provided with such high end software packages as AutoCad.. This might be used in applications where it is desirable to subsequently provide field updates in hexadecimal format to be downloaded by the customer and you desire to keep your design private, at least from the casual pirate.

Examples of hardware having unique hardware identities include the Dallas DS2401 Silicon Serial Number or a DS1820 Thermometer if you the nature of the design involves measuring temperature. In this routine, function _2401_fetch_ser_num is a "stub". That is, it is simply software that is intended to simulate the fetching of a serial number from an external DS2401.

Thus, the program calls first_time, and if it is the first time, the eight byte serial number is fetched from the DS2401 and programmed to EEPROM locations 0x10 – 0x17.

In order for the task (flashing of an LED) to be run, the serial number of the external DS2401 and the serial number which has been programmed in EEPROM must agree. If they do not match, the task is not performed. (Note that in providing future software updates, you would initialize first time locations to 0x00, or omit is all together, and also initialize locations 0x10 – 0x17 to the DS2401 serial number which you have on file).
As the fetching of the external serial number is actually written in software in this routine, a bit of work was required to introduce an error. Note that in _2401_fetch_ser_num, the number of calls to the function is implemented using EEPROM location 0x08. On the fourth call to the function, an error is introduced in the external serial number.

```
// FIRST_TM.C
//
// Illustrates the use of data EEPROM on the PIC16F87X.
//
// On download, EEPROM locations 0x00 - 0x03 are initialized to a distinctive
// pattern (0x5a, 0xa5, 0x5a, 0xa5).
//
// Each time the program is executed, function is_first_time() is called. If
// the distinctive pattern is detected, the locations are changed to 0x00 and
// TRUE is returned.  Subsequent calls will return FALSE.
//
// If, it is the first time, an 8-byte serial number is read from an external
// device and written to EEPROM locations 0x10 - 0x17.  The external device  might
// be a Dallas DS2401 1-W Silicon Serial Number. (In this routine, a stub is used
// to pass back a serial number).  Thus, the PIC is now married to a unique piece
// of hardware.
//
// Each time the program is executed, the 8-byte serial number is read from the
// external DS2401 and compared with that stored in EEPROM.  If they agree, the
// task (flashing of an LED) is executed.  If not, the program is locked.
//
// As noted, the fetching of the serial number from the DS2401 is actually
```

```
// implemented in software.  Note that on the fifth call to this routine, an error
// is introduced which causes the program to lock.  This is facilitated by using
// EEPROM location 0x08 as a persistent location for variable num_calls.
//
// copyright, Peter H. Anderson, Baltimore, MD, Dec, '00

#case

#device PIC16F877 *=16 ICD=TRUE
#include <defs_877.h>
#include <lcd_out.h>

#define TRUE !0
#define FALSE 0

//#define TEST

byte is_first_time(void);     // returns true if locations 0x00 - 0x03 in EEPROM
are
                              // at specific first time values.  Id so,
sets these
                              // locations to 0x00

byte is_valid_ser_num(void);  // tests if _2401 ser number agrees with
                              // EEPROM locations 0x10 - 0x17

void _2401_fetch_ser_num(byte *ser_num);
                                 // returns a serial number.  This is a stub.
void write_eeprom_ser_num(byte *ser_num); // writes serial number to locations
                              //0x00 - 0x17

void write_data_eeprom(byte adr, byte d);
byte read_data_eeprom(byte adr);

void flash_led(void);

void main(void)
{
   byte ser_num[8];

   pspmode = 0;

   portd4 = 0;          // LED
   trisd4 = 0;

   lcd_init();

   if (is_first_time())       // if the first time, fetch serial number
                              // from DS2401 and save to data EEPROM
   {
         _2401_fetch_ser_num(ser_num);    // fetch ser num from DS2401
         write_eeprom_ser_num(ser_num);   // and save.
         write_data_eeprom(0x08, 0x00);   // zero the number of calls to
                                          // _2401_fetch
   }

   if (is_valid_ser_num())    // if DS2401 and EEPROM ser nums agree
```

```
   {
         lcd_clr_line(0);
         printf(lcd_char, "Valid");
         while(1)
         {
              flash_led();
         }
   }

   else           // there was no match.  Lock the system.
   {
       lcd_clr_line(0);
       printf(lcd_char, "Invalid");
       lcd_clr_line(1);
       printf(lcd_char, "System Locked");
       while(1)
       {
#asm
          CLRWDT
#endasm
       }
   }
}

void write_eeprom_ser_num(byte *ser_num)
{
      byte n;

      for (n=0; n<8; n++)
      {
            write_data_eeprom(n+0x10, ser_num[n]);
      }
}

byte is_valid_ser_num(void)
{
   byte n, ser_num[8];

   _2401_fetch_ser_num(ser_num);    // fetch ser num from DS2401
   for (n = 0; n<8; n++)
   {
         if(read_data_eeprom(n+0x10) != ser_num[n])   // if not the same
         {
               return(FALSE);
         }
   }
   return(TRUE);                          // all eight bytes matched
}

void _2401_fetch_ser_num(byte *ser_num)   // this is a stub
{
   byte n, num_calls;

   const byte _2401_ser_num[8] = {0x77, 0x66, 0x55, 0x44,
                                  0x33, 0x22, 0x11, 0x00};
```

```
   for (n=0; n<8; n++)
   {
      ser_num[n] = _2401_ser_num[n];
   }

   num_calls = read_data_eeprom(0x08);
   ++num_calls;
   write_data_eeprom(0x08, num_calls);

   lcd_clr_line(3);
   printf(lcd_char, "Num Calls = %d", num_calls);

   if (num_calls == 5)        // on the 5th call, introduce an erro
   {
      ser_num[3] = 0x78;      // make serial number incorrect
   }
}

byte is_first_time(void)
{

   byte n;
   const byte x[4] = {0x5a, 0xa5, 0x5a, 0xa5};
   for (n = 0; n<4; n++)
   {
      if (read_data_eeprom(n) != x[n])
      {
           return(FALSE);
      }
   }

   for (n=0; n<4; n++)  // is it is first time, write 0x00s to each location
   {
      write_data_eeprom(n, 0x00);
#ifdef TEST
      read_data_eeprom(n);
#endif
   }

   return(TRUE);
}

void flash_led(void)
{
   portd4 = 1;
   delay_ms(250);
   portd4 = 0;
   delay_ms(250);
}

byte read_data_eeprom(byte adr)
{
   byte retval;
   eepgd = 0;           // select data EEPROM
   EEADR=adr;
   rd=1;    // set the read bit
   retval = EEDATA;
```

```
#ifdef TEST
   lcd_cursor_pos(0, 15);
   printf(lcd_char, "%x %x", adr, retval);
   delay_ms(2000);
#endif
   return(retval);
}

void write_data_eeprom(byte adr, byte d)
{
   eepgd = 0;           // select data EEPROM

   EEADR = adr;
   EEDATA = d;

   wren = 1;            // write enable
   EECON2 = 0x55; // protection sequence
   EECON2 = 0xaa;

   wr = 1;        // begin programming sequence


   delay_ms(10);

   wren = 0;            // disable write enable
}


#include <lcd_out.c>

#rom 0x2100={0x5a, 0xa5, 0x5a, 0xa5}      // initialize EEPROM
```

Program EE_SAVE.C.

This file illustrates how to write a float or a structure to EEPROM and how to read them from EEPROM.

One application might be to save a calibration constant.

Note that in functions save_to_eeprom and read_from_eeprom, a byte pointer which points to the first byte of the quantity is passed.

In some cases, you may wish to ship a product with a calibration constant programmed in EEPROM. Rather than fussing with trying to figure out how CCS stores floats, you might use a simple utility;

```
float a = 0.0004125;
byte n, *p;

p = (byte *) &a;  // address of “a”.  Typecast as pointer to a byte

for(n=0; n<sizeof(float); n++)
{
   printf(“%2x “, *(p+n));
}
```

and then simply initialize four locations in EEPROM to these values.

```
// EE_SAVE.C
//
// Illustrates how to save a quantity to and fetch a quantity from EEPROM.
//
// Saves a float and a struct TM to EEPROM and then fetches them and displays
// on LCD.
//
// Note that a byte pointer which points to the beginning of the quantity is passed
// to each function.
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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

void save_to_eeprom(byte adr, byte *p_dat, byte num_bytes);
void read_from_eeprom(byte adr, byte *p_dat, byte num_bytes);

void write_data_eeprom(byte adr, byte d);
byte read_data_eeprom(byte adr);

struct TM
{
      byte hr;
      byte mi;
      byte se;
};

void main(void)
{
   float float_1 = 1.2e-12, float_2;
   struct TM t1, t2;
   byte *ptr;

   lcd_init();
   t1.hr = 12;    t1.mi = 45;       t1.se = 33;

   ptr = (byte *) &float_1;         // ptr points to first byte of float_1
   save_to_eeprom(0x00, ptr, sizeof(float));    // save flaot_1

   ptr = (byte *) &t1;
   save_to_eeprom(0x10, ptr, sizeof(struct TM));      // save t1

   ptr = (byte *) &float_2;
   read_from_eeprom(0x00, ptr, sizeof(float));

   ptr = (byte *) &t2;
   read_from_eeprom(0x10, ptr, sizeof(struct TM));

   lcd_clr_line(0);                        // print the float
   printf(lcd_char, "float = %1.3e", float_2);
```

```
   lcd_clr_line(1);                        // print the time
   printf(lcd_char, "t2 = ");
   lcd_dec_byte(t2.hr, 2);
   lcd_char(':');
   lcd_dec_byte(t2.mi, 2);
   lcd_char(':');
   lcd_dec_byte(t2.se, 2);

   while(1)
#asm
   CLRWDT
#endasm
}

void save_to_eeprom(byte adr, byte *p_dat, byte num_bytes)
{
   byte n;

   for (n=0; n<num_bytes; n++)
   {
       write_data_eeprom(adr, *p_dat);
       ++adr;
       ++p_dat;
   }
}

void read_from_eeprom(byte adr, byte *p_dat, byte num_bytes)
{
   byte n;

   for (n=0; n<num_bytes; n++)
   {
       *p_dat = read_data_eeprom(adr);
       ++adr;
       ++p_dat;
   }
}

byte read_data_eeprom(byte adr)
{
   byte retval;
   eepgd = 0;           // select data EEPROM
   EEADR=adr;
   rd=1;    // set the read bit
   retval = EEDATA;
#ifdef TEST
   lcd_cursor_pos(0, 15);
   printf(lcd_char, "%x %x", adr, retval);
   delay_ms(2000);
#endif
   return(retval);
}

void write_data_eeprom(byte adr, byte d)
{
   eepgd = 0;           // select data EEPROM
```

```
   EEADR = adr;
   EEDATA = d;

   wren = 1;            // write enable
   EECON2 = 0x55; // protection sequence
   EECON2 = 0xaa;

   wr = 1;        // begin programming sequence

   delay_ms(10);

   wren = 0;            // disable write enable
}

#include <lcd_out.c>
```

Program TMR0_1.C.

Timer 0 is an eight bit counter which may be configured to count using the system clock or using the T0CK1/RA4 external terminal. Although its utility is limited when compared with the 16-bit Timer 1 and 8-bit Timer 2, it is the only timer associated with low end PICs; e.g., 12C67X, 16F84, 558 and thus an understanding of its operation is important if your final application is one of these devices. The Timer 0 Module is discussed in Section 5 of the PIC16F87X Data Sheet.

Program TMR0_1.C uses the system clock (1 usec if using a 4.0 MHz crystal or resonator) to generate a tone and also perform long term timing. A 500 Hz tone is generated on a speaker and an LED is continually turned on for 4.0 secs and off for 4.0 secs while also performing other tasks.

The prescaler is set for 1:8 and thus the counter increments each 8 us. A periodic rollover of 1.0 ms is achieved by setting the counter to the two's complement of 125. Thus, an interrupt occurs every 1.0 ms (125 * 8 us). Note that by the time the program processes the interrupt, some time has elapsed. If this time is known, the value loaded to the timer might be adjusted to provide accurate long term timing, but even then, Timer 0 is not suitable for such applications as a clock / calendar.

In this routine, note the use of a "static" variable in the interrupt service routine to count the number of times the ISR is executed.

```
// Program TMR0_1.C
//
// Illustrates use of TMR0 to time for 1 ms. Generates 500 Hz tone on
// speaker on RD.0 and continually flashes LED on RD.4 on for 4 secs
// and off for 4 seconds.
//
// Note that TMR0 is configured for CLOCK, assigned to OSC, prescale
// by 4.  Thus, 1.00 MHz / 8 = 125 KHz.  Period = 8 usecs.  Thus, TMRO
// is loaded with the twos comp of 125 to achieve interrupt timing of 1 ms
//
// Copyright, Peter H. Anderson, Baltimore, MD, Jan, '01

#case

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

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

#define TRUE !0
#define FALSE 0

#define T_TICK (~125) + 1     // two's complement of 125

void main(void)
{
   lcd_init();
   pspmode = 0;

   portd0 = 0;
   trisd0 = 0;          // speaker

   portd4 = 0;
   trisd4 = 0;          // LED

// configure TMR0
   t0cs = 0;            // use CLK as source
   psa = 0;             // prescaler assigned to TMR0
   ps2 = 0;  ps1 = 1;  ps0 = 0;     // prescale by 8


   t0if = 0;            // clear any existing interrupt
   t0ie = 1;
   gie = 1;

   while(1)
   {
      lcd_clr_line(0);  // do something else as well
      printf(lcd_char, "Hello World");
      delay_ms(500);
   }
}

#int_rtcc timer0_int_handler(void)
{
   static long isr_timer=4000;

   TMR0 = TMR0 + T_TICK;
   portd0 = !portd0;          // invert bit on speaker

   if (--isr_timer==0)
   {
      portd4 = !portd4;       // reverse LED every 4000 ms
      isr_timer=4000;
   }
}

#include <lcd_out.c>
```

Program COUNT_1.C.

This routine counts the number of pulses on input T0CK1 over one second.

Note that the Morgan Logic Probe provides clock sources at 1 and 10 pulses per second. In running this routine, you may wish to connect the 10 PPS output to PIC input T0CK1/RA4.

```
   Logic Probe Term 2 (CLK10) --------------- PIC16F877 Term 6 (T0CK1/RA4)
```

Note that this routine uses the Timer1 interrupt to implement the one second timing and also uses the TMR0 interrupt. Each time the Timer 0 counter rolls over, variable high_byte is incremented. At the end of the one second interval, TMR0 is fetched and this is the low byte. Of course, when using the Logic Probe's 10 PPS output, TMR0 never rolls over.

```
// COUNT_1.C
//
// Illustrates the use of Timer 0 as an event counter and Timer 1 for timing.
//
// Function count_1_sec configures Timer 0 as a counter of events appearing on
// PIC input TOCK1/RA4.
//
// copyright, Peter H. Anderson, Baltimore, Jan, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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

#define TRUE !0
#define FALSE 0

unsigned long count_1_sec(void);

int timer0_int_occurred;
int timer1_int_occurred;

void main(void)
{
   unsigned long count;
   float freq;

   lcd_init();
   while(1)
   {
      count = count_1_sec();

      lcd_clr_line(0);
      printf(lcd_char, "%ld", count);
      delay_ms(1000);
   }
}

unsigned count_1_sec(void)
{
   // uses tmr1 with 32 kHz to time for one second
   // use tmr0 to count the number of events
```

```
byte high_byte = 0, low_byte;
unsigned long count;

timer0_int_occurred = FALSE;
timer1_int_occurred = FALSE;

t1ckps1=0;           // 1:1 prescale
t1ckps0=0;

t1oscen=1;           // enable external osc
t1sync=1;            // don't synch external clock with CPU clock
tmr1cs=1;            // external clock source

delay_ms(5);         // a bit of time to let the oscillator turn on

t0cs =1;             // assign tmr0 counter to RA4
psa = 1;             // prescaler not assigned to counter

TMR0 = 0;            // init counts to zero

TMR1H = 0x80;  // set TMR1 to roll over in 32,768 counts
TMR1L = 0x00;

tmr1on=1;            // turn on timer1
t0se=1;              // enable counting from T0CK1 input

t0if=0;
t0ie=1;

tmr1if = 0;          // clear any interrupt
tmr1ie = 1;

peie = 1;
gie = 1;

while(!timer1_int_occurred)
{
   if(timer0_int_occurred)
   {
      timer0_int_occurred = FALSE;
      ++high_byte;   // there was a rollover of timer 0
   }
}

low_byte = TMR0;     // immediately fetch low 8 bits

while(gie)
{
   gie=0;
}

t0ie = 0;      // clean up
t0se = 0;

tmr1ie = 0;
t1oscen = 0;
tmr1on = 0;
```

```
   count = high_byte;
   count = (count << 8) | low_byte;
   return(count);
}

#int_rtcc timer_0_interrupt_handler()
{
   timer0_int_occurred = TRUE;
}

#int_timer1 timer_1_interrupt_handler()
{
   timer1_int_occurred = TRUE;
}

#int_default default_interrupt_handler()
{
}

#include <lcd_out.c>
```

Program CAPTURE_1.C.

The operation of the Capture/Compare/PWM (CCP) modules is treated in Section 8 of the PIC16F877 Data Sheet.

This routine illustrates the operation of the input capture feature to measure the period of an input appearing at PIC input CCP1/RC2. Here again, the Morgan Logic Probe may be used as a source.

```
   Logic Probe Term 2 (CLK10) ---------------- PIC16F877 Term 17 (CCP1/RC2)
```

The concept of input capture is simply that the value of Timer 1 is written to CCPR1H and CCPR1L when a specified condition appears on input CCP1/RC2. This condition might be a falling edge, rising edge or every fourth or 16th rising edge. In this routine, the specified event is every rising edge. The advantage of the hardware writing to the CCPR1 registers at the moment the event occurs over reading the value of Timer 1 in an interrupt service routine is that a substantial period of time may elapse prior to handling the interrupt. The use of the CCPR registers permits you to write code which is performing other tasks and handle the CCP interrupt as convenient. Of course, the interrupt must be handled prior to another the occurrence of another event.

In this routine, the maximum number of Timer 1 rollovers (nominally 65 ms per rollover) is passed to function measure period(). This prevents the system from hanging if the first rising edge when timing of the period begins or the second rising edge when timing stops never occur.

```
// Capture1.C
//
// Illustrates the use of Timer1 and Input Capture to continually measure a period
// of an input signal on CCP1/RC2.
//
// In function measure_period, Timer 1 is configured for internal clock, 1:1
// prescale. Thus, one usec per click.  The CCP module is configured for interrupt
// on rising edge.
//
// If no CCP1 interrupt occurs within the specified number of rollovers of Timer 1
// success is set to FALSE.  Otherwise, the function breaks from the first while(1)
```

```
// loop and waits up to the specified number of rollovers for a second CCP1
// interrupt. The time difference in usecs is returned.
//
// Copyright, Peter H. Anderson, Baltimore, MD, Jan, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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

#define TRUE !0
#define FALSE 0

float measure_period(byte max_rollovers, byte *p_success);

byte tmr1_int_occ, capture_int_occ;

void main(void)
{
   float t_period;
   byte success;

   lcd_init();

   while(1)
   {
      t_period = measure_period(50, &success);
      if (success)
      {
         lcd_clr_line(0);
         printf(lcd_char, "%3.3e", t_period);
      }
      else
      {
         lcd_clr_line(0);
         printf(lcd_char, "Invalid");
      }
   }
}

float measure_period(byte max_rollovers, byte *p_success)
{
      byte rollovers = 0, is_valid = TRUE;
      unsigned long t1, t2;
      float t_period;

      // fire up timer 1
      tmr1cs = 0;                              // 1 usec at 4.0 MHz
      t1ckps1 = 0;  t1ckps0 = 0;          // prescale 1:1

      TMR1H = 0x00;                            // set Timer 1 to 0
      TMR1L = 0x00;

      CCP1CON = 0x05;                          // interrupt on rising edge
```

```
  tmr1if = 0;
  ccp1if = 0;                                // kill any old interrupts

  tmr1on = 1;       // get it going
  tmr1_int_occ = FALSE;

  tmr1ie = 1;
  ccp1ie = 1;
  peie = 1;
  gie = 1;

  while(1)
  {
     if (tmr1_int_occ)               // if a timer 1 interrupt
     {
        ++rollovers;
        tmr1_int_occ = FALSE;
        if (rollovers == max_rollovers)
        {
           is_valid = FALSE;
           break;
        }
     }

     if (capture_int_occ)       // if an input capture interrupt
     {
        rollovers = 0;
        t1 = CCPR1H;
        t1 = (t1 << 8) | CCPR1L;
        capture_int_occ = FALSE;
        break;
     }
}

while(1)    // now for the second rising edge
{
   if(!is_valid)
   {
      break;
   }

   if (tmr1_int_occ)
   {
      ++rollovers;
      if (rollovers == max_rollovers)
      {
         is_valid = FALSE;
         break;
      }
      tmr1_int_occ = FALSE;
   }

   if (capture_int_occ)
   {
      t2 = CCPR1H;  // the value of Timer 1 is stored in CCP1H & L
      t2 = (t2 << 8) | CCPR1L;
      capture_int_occ = FALSE;
```

```
         break;
      }
   }

   while(gie)
   {
      gie = 0;
   }
   tmr1ie = 0;
   ccp1ie = 0;

   if (is_valid)
   {
      if(t2 > t1)
      {
         t_period = ((float) rollovers) * 65535.0 + (float) (t2 - t1);
      }
      else
      {
         t_period = ((float) rollovers) * 65535.0 - (float) (t1 - t2);
      }
      *p_success = TRUE;
   }

   else
   {
      *p_success = FALSE;
   }

   return(t_period);
}


#int_timer1 timer1_int_handler(void)
{
      tmr1_int_occ = TRUE;
}


#int_ccp1 ccp1_int_handler(void)
{
      capture_int_occ = TRUE;
}

#int_default default_interrupt_handler()
{
}

#include <lcd_out.c>
```

Program CAPTURE_2.C.

This routine is quite similar to CAPTURE_1.C except that it uses the input capture feature to measure either the logic zero or logic one time of the input pulse.

If the “state” to measure is “0”, the CCP module is first configured for falling edge. On interrupt, timing begins and the CCP module is configured for rising edge. In measuring the logic 1 time, the CCP module is configured for rising edge and then falling edge.

```
// Capture2.C
//
// Illustrates use of Timer 1 and CCP1 in the input capture mode to measure the
// logic zero and logic one times of a pulse appearing on CCP1/RC2.
//
// In function measure_pulse(), timer 1 is configured for internal osc, prescale
// of 1:1. Thus, one usec per tick.
//
// If the specified state is 0, the CCP mode is set to 0x04 so as to cause an
// interrupt on a falling edge.  If the specified state is 1, the mode is set to
// 0x05 to cause an interrupt on the rising edge.
//
// After the capture interrupt occurs, the CCP mode is set for either the rising or
// falling edge.
//
// After the second capture interrupt occurs, elapsed time from the first interrupt
// to the second is fetch from CCP1H & L and combined with the value of rollovers.
//
// Note that if, in waiting for either the first or second capture interrupt, the
// number of rollovers of timer 1 equals the specified maximum time to wait,
// success is set to FALSE.
//
// Copyright, Peter H. Anderson, Baltimore, MD, Jan, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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


#define FALSE 0
#define TRUE !0

float measure_pulse(byte state, byte max_rollovers, byte *p_success);

byte tmr1_int_occ, capture_int_occ;

void main(void)
{
   float t_pulse;
   byte success;

   lcd_init();

   while(1)
   {
      t_pulse = measure_pulse(0, 50, &success);       // measure the zero time

      if (success)
      {
```

```
         lcd_clr_line(0);
         printf(lcd_char, "0 %3.3e", t_pulse);
      }
      else
      {
         lcd_clr_line(0);
         printf(lcd_char, "Invalid");
      }

      t_pulse = measure_pulse(1, 50, &success);       // measure the logic one time
      if (success)
      {
         lcd_clr_line(1);
         printf(lcd_char, "1 %3.3e", t_pulse);
      }
      else
      {
         lcd_clr_line(1);
         printf(lcd_char, "Invalid");
      }
   }
}

float measure_pulse(byte state, byte max_rollovers, byte *p_success)
{

      byte rollovers = 0, is_valid = TRUE;
      unsigned long t1, t2;
      float t_pulse;

      // fire up timer 1
      tmr1cs = 0;                                 // 1 usec at 4.0 MHz
      t1ckps1 = 0;  t1ckps0 = 0;                  // prescale 1:1

      TMR1H = 0x00;
      TMR1L = 0x00;

      if (state)
      {
         CCP1CON = 0x05;                          // interrupt on rising edge
      }
      else
      {
         CCP1CON = 0x04;                          // falling edge

      }

      tmr1if = 0;
      ccp1if = 0;                                 // kill any old interrupts

      tmr1on = 1;                                 // get it going
      tmr1_int_occ = FALSE;

      tmr1ie = 1;
      ccp1ie = 1;
      peie = 1;
      gie = 1;
```

```
  while(1)
  {
     if (tmr1_int_occ)
     {
        ++rollovers;
        if (rollovers == max_rollovers)
        {
           is_valid = FALSE;
           break;
        }
        tmr1_int_occ = FALSE;
     }

     if (capture_int_occ)
     {
         rollovers = 0;
         t1 = CCPR1H;                 // fetch time t1 for CCP1H & L
         t1 = (t1 << 8) | CCPR1L;
         capture_int_occ = FALSE;
         break;
     }
}

while (gie)   // turn off interrupts for the moment
{
   gie = 0;
}

ccp1m0 = !ccp1m0;   // change CCP mode for interrupt on opposite edge

ccp1if = 0;
gie = 1;

while(1)                        // now for the second transition
{
   if(!is_valid)
   {
      break;
   }

   if (tmr1_int_occ)
   {
      ++rollovers;
      if (rollovers == max_rollovers)
      {
          is_valid = FALSE;
          break;
      }
      tmr1_int_occ = FALSE;
   }

   if (capture_int_occ)
   {
      t2 = CCPR1H;        // fetch time t2
      t2 = (t2 << 8) | CCPR1L;
      capture_int_occ = FALSE;
```

```
         break;
      }
   }

   while(gie)                          // turn off ints
   {
           gie = 0;
   }
   tmr1ie = 0;
   ccp1ie = 0;

   if (is_valid)
   {
      if(t2 > t1)
      {
         t_pulse = ((float) rollovers) * 65535.0 + (float) (t2 - t1);
      }
      else
      {
         t_pulse = ((float) rollovers) * 65535.0 - (float) (t1 - t2);
      }
      *p_success = TRUE;
   }
   else
   {
      *p_success = FALSE;
     }
     return(t_pulse);
}


#int_timer1 timer1_int_handler(void)
{
   tmr1_int_occ = TRUE;
}


#int_ccp1 ccp1_int_handler(void)
{
   capture_int_occ = TRUE;
}

#int_default default_interrupt_handler()
{
}

#include <lcd_out.c>
```

Program OUT_CMP1.C.

The concept of output compare is to cause an interrupt and take an action when the value of Timer 1 (16-bits) matches the value of the CCPR1H and CCPR1L registers.

In this routine, the Timer is configured to use the external 32.768 kHz crystal with a prescale of 1:8. Thus, the timer rolls over every 16 seconds. In the “set on match” and “clear on match”, the Timer is not reset. Thus, to force an interrupt at a future time, one must advance the CCPR1H & L registers

CCP1 is configured for “set on match” and the Timer is loaded with CCPR1H&L plus 0x1400 ticks (1.25 secs) . On interrupt, the CCP module is configured for “clear on match” and Timer 1 is loaded with CCPR1H&L + 0x4000. Thus, the LED on CCP1/RC2 is on for 4.0 seconds and off for 1.25 seconds.

```
// OUT_CMP1.C
//
// Illustrates Use of CCP1 for Output Compare.
//
// LED on RC2/CCP1 is continually turned off for four seconds and
// then on for 1.25 secs, etc. while processor is continually send dots
// to LCD module.
//
// RC2/CCP1 (term 17) to LED.
//
// Note that a CCP1 interrupt in the Compare, Set on Match and Clr on Match
// does not reset TMR1.  Rather, the periodicity is achieved by adding an
// offset to CCPR1H and L.   This is important when using both CCP modules
// in the compare mode.
//
// Peter H. Anderson, Baltimore, MD, Jan, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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

#define TRUE !0
#define FALSE 0

void main(void)
{

   lcd_init();

   trisc2 = 0;          //CCP1 output to LED
   portc2 = 0;

   t1ckps1 = 1;         // set the prescaler for 1:8.  Thus, full roll is 16 secs
   t1ckps0 = 1;

   t1oscen = 1;         // turn on external 32.768 kHz osc
   tmr1cs = 1;

   t1sync = 0;          // synchronize external clock input.  Uncertain what this
                        // does for me.

   CCP1CON = 0x08;      // configure for output compare, set on match

   TMR1H = 0;           // start timer at 0x0000
   TMR1L = 0;

   CCPR1H = 0x00;
   CCPR1L = 20;         // first interrupt in 20 ticks - the value isn't critical.
```

```
                        // But there is no point in waiting for up to 16 secs.

   tmr1on = 1;          // turn on the timer

   ccp1if = 0;          // clear flag
   ccp1ie = 1;          // enable interrupts

   peie = 1;
   gie = 1;

   while(1)
   {
      lcd_clr_line(0);  // can now be doing other things
      printf(lcd_char, "Hello World");
      delay_ms(1000);
   }
}

#int_ccp1 ccp1_int_handler(void)
{
   unsigned long next_time, current_time;

   if(CCP1CON==0x08)    // if it is currently set on match
   {
       next_time=0x1400;      // for 1.25 seconds
                              // 1/32.768 kHz * 8 * X = 1.25; X=0x1400
   }

   else     // it is currently clear on match
   {
      next_time=0x4000;       // for 4.00 seconds
                              // 1/32.768 kHz * 8 * X = 1.25; X=0x4000

   }
   // set new value of CCPR1H and L
   current_time = CCPR1H;
   current_time = current_time << 8 | CCPR1L;
   next_time = current_time + next_time;
   CCPR1H = next_time >> 8;
   CCPR1L = next_time;

   if(CCP1CON==0x08)    // if it is currently set on match
   {
      CCP1CON = 0x09;   // clear on match
   }
   else     // it is currently clear on match
   {
      CCP1CON=0x08;     // set on match
   }
}

#int_default default_int_handler(void)
{
}

#include <lcd_out.c>
```

Program OUT_CMP2.C.

This routine extends on OUT_CMP1.C.

The second CCP module is configured to simply generate a software interrupt on match. The interrupt service routine advances the CCPR2H & L by 208 so as to force an interrupt nominally every 50 ms. A flag is also set to TRUE. When main() reads this flag as TRUE, PORTB.0 is read and if the pushbutton is depressed, a character is output to the LCD.

```
// OUT_CMP2.C
//
// Illustrates Use of both CCP1 and CCP2 for Output Compare.
//
// LED on RC2/CCP1 is continually turned off for four seconds and
// then on for 1.25 secs, etc. while processsor is continually send dots
// to LCD module.
//
// RC2/CCP1 (term 17) to LED.
//
// Using CCP2 in Compare - Software Interrupt only.  Every 50 ms, scans
// RB0.  If at logic zero zero, keyval is set to 'A' and  displayed.
// Note that this is a cheap excuse for a keyboard scan routine.
//
// Peter H. Anderson, Baltimore, MD, Jan, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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

#define TRUE !0
#define FALSE 0

byte keypressed=FALSE;
byte keyval;

void main(void)
{
   byte pos = 0;
   lcd_init();

   not_rbpu = 0;        // enable weak pull-ups – for pushbutton

   trisc2 = 0;          //CCP1 output to LED
   portc2 = 0;

   t1ckps1 = 1;         // set the prescaler for 1:8.  Thus, full roll is 16 secs
   t1ckps0 = 1;

   t1oscen = 1;         // turn on external 32.768 kHz osc
   tmr1cs = 1;
```

```
   t1sync = 0;          // synchronize external clock input.  Uncertain what this
                        // does for me.


   CCP1CON = 0x08;      // configure for output compare - set on match
   CCP2CON = 0x0a;      // output capture - software interrupt

   TMR1H = 0;
   TMR1L = 0;

   CCPR1H = 0x40;
   CCPR1L = 0x00;       // 4.00 secs

   CCPR2H = 0x00;
   CCPR2L = 208;        // about 50 ms

   tmr1on = 1;          // turn on the timer

   ccp1if = 0;          // clear flags
   ccp2if = 0;
   ccp1ie = 1;          // enable interrupts
   ccp2ie = 1;

   peie = 1;
   gie = 1;

   while(1)
   {
      if (keypressed)
      {
         lcd_cursor_pos(3, pos);
         lcd_char(keyval);
         ++pos;
         if (pos == 20)
         {
            pos = 0;
         }
         keypressed = FALSE;
      }
      else
      {
         lcd_cursor_pos(0, 0);
         printf(lcd_char, "Hello World"); // can now be doing other things
      }
   }
}

#int_ccp1 ccp1_int_handler(void)
{
   unsigned long next_time, current_time;

   if(CCP1CON==0x08)    // if it is currently set on match
   {
       next_time=0x1400;      // for 1.25 seconds
                              // 1/32.768 kHz * 8 * X = 1.25; X=0x1400
   }
```

```
   else     // it is currently clear on match
   {
      next_time=0x4000;       // for 4.00 seconds
                              // 1/32.768 kHz * 8 * X = 1.25; X=0x4000

   }
   // set new value of CCPR1H and L
   current_time = CCPR1H;
   current_time = current_time << 8 | CCPR1L;
   next_time = current_time + next_time;
   CCPR1H = next_time >> 8;
   CCPR1L = next_time;

   if(CCP1CON==0x08)    // if it is currently set on match
   {
      CCP1CON = 0x09;   // clear on match
   }
   else     // it is currently clear on match
   {
      CCP1CON=0x08;     // set on match
   }
}

#int_ccp2 ccp2_int_handler(void)
{
   unsigned long next_time, current_time;
   next_time = 205;     // 1/32.768 kHz * 8 * 205 = 50.04 ms
   current_time = CCPR2H;
   current_time = current_time << 8 | CCPR2L;
   next_time = current_time + next_time;
   CCPR2H = next_time >> 8;
   CCPR2L = next_time;

   if(!rb0) // this is a cheap excuse for a keyboard scan routine
   {
        keypressed = TRUE;
        keyval = 'A';
   }
}

#int_default default_int_handler(void)
{
}

#include <lcd_out.c>
```

Program OUT_CMP3.C.

This routine illustrates the use of the CCP2 module to periodically trigger an A/D conversion. Note that the A/D must be set up and ready to go.

Note that the A/D special event is available only with CCP2. When this mode is used with either CCP module, Timer 1 is reset. I have mixed feeling as to why Microchip opted to do this as it makes this CCP mode on one module incompatible with the “set of match”, “clear on match” and “software interrupt” on the other module.

```
// OUT_CMP3.C
```

```
//
// Illustrates Use of CCP2 for Output Compare - Trigger Special Event.
//
// Performs an A/D conversion on RA0/AN0 (Term 2) every 4.0 seconds.
//
// Peter H. Anderson, Baltimore, MD, Jan, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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

#define TRUE !0
#define FALSE 0

byte new_ad_avail = FALSE;
long ad_val;

void main(void)
{
   lcd_init();

   // configure A/D
   adfm = 1;   // a/d format right justified

   pcfg3 = 1; // configure for AN0 only
   pcfg2 = 1;
   pcfg1 = 1;
   pcfg0 = 0;

   adcs1 = 1;
   adcs0 = 1; // internal RC

   adon = 1;

   chs2 = 0; chs1 = 0; chs0 = 0;    // channel 0

   // set up Timer 1
   t1ckps1 = 1;         // set the prescaler for 1:8.  Thus, full roll is 16 secs
   t1ckps0 = 1;

   t1oscen = 1;         // turn on external 32.768 kHz osc
   tmr1cs = 1;

   t1sync = 0;          // synchronize external clock input.  Uncertain what this
                        // does for me.

   CCP2CON = 0x0b;      // output capture - trigger special event

   TMR1H = 0;
   TMR1L = 0;

   CCPR2H = 0x40;
   CCPR2L = 0x00;       // 4.00 secs
```

```
   tmr1on = 1;          // turn on the timer

   ccp2if = 0;          // kill any pending interrupts
   adif = 0;
                        // enable interrupts
   ccp2ie = 1;
   adie = 1;

   peie = 1;
   gie = 1;

   while(1)
   {
      if (new_ad_avail)
      {
         lcd_clr_line(3);
         printf(lcd_char, "%ld", ad_val);
         new_ad_avail = FALSE;
      }
      else
      {
         lcd_cursor_pos(0, 0);
         printf(lcd_char, "Hello World"); // can now be doing other things
      }
   }
}

#int_ccp2 ccp2_int_handler(void)
{
   unsigned long next_time, current_time;
   next_time = 0x4000;
   current_time = CCPR2H;
   current_time = current_time << 8 | CCPR2L;
   next_time = current_time + next_time;
   CCPR2H = next_time >> 8;
   CCPR2L = next_time;
}

#int_ad ad_int_handler(void)
{
   ad_val = ADRESH;                     // fetch the value
   ad_val = ad_val << 8 | ADRESL;
   new_ad_avail = TRUE;                 // signal that a/d int occurred
}

#int_default default_int_handler(void)
{
}

#include <lcd_out.c>
```