Tutorial by Example – Issue 1A

Introduction.

This is the third distribution of sample C routines to those people who have purchased the PIC16F87X Dev Package. However, I did not merge this into the previous material for a number of reasons; my laziness, and my feeling that many people may have printed out the previous and may be inconvenienced in again printing out a new version and then trying to find the additions.

Thus, the previous discussion will be termed Issue 1. Additions will be 1A, 1B, etc. If I ever get the energy to merge all of the additions, this will become Issue 2.

The zip file includes all files, both those previously distributed along with the new ones.

This distribution deals briefly with bit fields, unions and a technique for using a potentiometer in conjunction with an A/D converter and EEPROM to adjust and set a value. Interfacing with devices using the Motorola SPI protocol is treated in considerable detail. As always, I was not able to do as much as I had hoped.

And, try as I might, I am sure there are more than a few errors in this discussion. Note that all routines have been tested, but files can get mixed up.

In developing prior material, only one irresolvable problem was encountered. I was unable to implement a write to the data EEPROM using interrupts

In this distribution, two confidence breakers were encountered, which is in part why I didn't get more done. These are noted in the discussion, but, briefly;

1. Program UNION_1.C compiles under PCW. However, when compiling in MPLAB, MPLAB hangs. I saw this problem at a recent workshop and also a posting on the PICLIST describing the same problem.

2. In Program TLC2543_2.C, I found that arrays of longs were not being properly handled in a function.

Both of these problems are disconcerting. I have been teaching for close to 20 years and scarcely a day goes by that I don't see some amazingly weird thing. However, I have found that over time I find the answer. But, often this is later rather than sooner.

The next distribution will be on or about Feb 20. School is again in session and I have less flexibility. I try to assign problems that permit me to refine some routines in teaching my students and I have many students and workers on a number of different projects. Thus, right now, I am uncertain just what will be in the next distribution.

Bit Fields.

Assume you are using PORTD with bits 0 – 3 as outputs for one task and bit 4 as an output for another task and bits 5 – 7 for yet another. Dealing with the single output on bit 4 is relatively simple;

```
portd4 = 1; // turn on LED on bit 4
```

However, dealing with the three upper bits can result in code that looks a bit confusing. For example, assume the variable n in the range of 0 - 5 is to be displayed on output bits 5 – 7;

```
PORTD = (PORTD & 0x1f) | (n << 5);
```

Note that the upper three bits of PORTD are zeroed and this is then ored with variable n shifted left by five places. Variable n must be shifted left 5 places so as to align with bits 5 - 7 of PORTD.

Modifying the low four bits;

```
PORTD = (PORTD & 0xf0) | m;
```

An alternative which can greatly simplify things is to use bit fields;

```
struct PORTD_BITS
{
   byte FOUR_LEDS : 4;        // bits 0 - 3
   byte ONE_LED : 1;          // bit 4
   byte THREE_LEDS: 3;        // bits 5, 6, 7
};

struct PORTD_BITS portd_bits;  // global as opposed to passing to functions
```

This permits bit 0 – 3 to be referred to as portd_bits.FOUR_LEDS and bits 5 – 7 as simply portd_bits.THREE_LEDS.

The above examples are implemented as;

```
portd_bits.THREE_LEDS = n;
PORTD = portd_bits;
```

Note that the order of declaring bits in the structure is critically important. Low bits to high bits.

It is also the users responsibility to assure the variable is limited to the size declared in the structure. For example;

```
m = 17;
portd_bits.FOUR_LEDS = m;
```

Note that this will cause an undesired change in bit 4. as five bits are required to store the number 17.

I haven't really investigated the matter, but I assume that although the use of bit fields make the C code appear more compact, the amount of code required is marginally more than manipulating bits with ands and ors. However, this is not an issue until one runs out of memory.

```
// BITS_1.C
//
// Illustrates the use of bit fields.
//
// Configuration - LEDS (8) on PORTD0 - PORTD7
//
// Each 250 ms, inverts the state of the LED on PORTD4 and displays a three bit
// count (0 – 5) on PORTD5::7.  At the end of the sequence, the number of times the
// for loop has been executed is displayed on LEDs on bits 0 – 3.
//
```

```
// 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

struct PORTD_BITS
{
   byte FOUR_LEDS : 4;        // bits 0 - 3
   byte ONE_LED : 1;          // bit 4
   byte THREE_LEDS: 3;        // bits 5, 6, 7
};

struct PORTD_BITS portd_bits;  // global as opposed to passing to functions

void main(void)
{
   int m = 0, n;

   pspmode = 0;

   portd_bits = 0x00;
   PORTD = portd_bits;

   TRISD = 0x00;        // all outputs

   while(1)
   {
      for (n=0; n<6; n++)
      {
           portd_bits.ONE_LED = !portd_bits.ONE_LED;  // toggle bit 4
           portd_bits.THREE_LEDS = n;
           PORTD = portd_bits;
           delay_ms(250);
      }
      portd_bits.FOUR_LEDS = m;  // display the number of times the loop is executed
      PORTD = portd_bits;

      ++m;
      if (m == 16)      // limit the variable to 0 - 15
      {
         m = 0;
      }
   }
}

#include <lcd_out.c>
```

Unions.

Caution. At a recent workshop, one participant had a frustrating problem where the program compiled under PCW. However, compiling in MPLAB, caused MPLAB to hang. I have had the same problem with the file UNION_1.C. I have scanned the file again and again to assure it is free of a virus and tried to run the program again and again, without success. I noted a posting on the PICLIST precisely describing a similar problem the very same problem which left me feeling a bit better in a perverse sort of way.

As I say, I have scanned the file for viruses, but you might wish to use caution. If you can figure out the problem, I would like to say, you win $10,000. But, I can't afford to do that. But, it sure is frustrating.

The idea of unions is to use the same RAM addresses to share variables. One application is to use the address space in one task and then reuse the address space in another task where the variables in one task are not used in the other. This could be a big plus with a PIC, particularly when using an array in a task.

The amount of memory allocated to the union is the size of the largest element. Thus one might declare a union as consisting of an array of 30 bytes and a structure consisting of five byte variables. Thirty bytes will be allocated for the union.

Another application of unions is in dealing with longs (16-bits) and the high and low bytes. An example is combining ADRESH and ADRESL into a 16 bit ad_val.

In the following, note that union LONG consists of a long and a struct TWO_BYTES. Thus, ad_val.w refers to the entire 16 bit quantity and ad_val.b.h and ad_val.b.l refer to the high and low bytes.

Thus, such code as;

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

May be simplified as;

```
ad_val.b.h = ADRESH;
ad_val.b.l = ADRESL;
```

The entire quantity may then be referred to as ad_val.w. However, note that this works only if the order of definitions in struct TWO_BYTES is declared in the order shown; low byte first..

Note that I opted to use a structure to define the high and low bytes, but I often see this done with a two byte array.

In fact, I don't tend to use unions in my programming. Usually, I am too eager to dig into the problem to fool with this, but the idea that you can do this is interesting.

```
// Program UNION_1.C
//
// Illustrates the use of a union to such that a long and two bytes share
// the same memory space.
//
// 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

struct TWO_BYTES
{
      byte l;
      byte h;
};

union LONG
{
      unsigned long w;
      struct TWO_BYTES b;
};

unsigned long make_ad_meas(void);

void main(void)
{

   union LONG adval;
   lcd_init();
   adval.b.h = 0x01;
   adval.b.l = 0x80;
   printf(lcd_char, "%lx", adval.w);

   adval.w = make_ad_meas();
   lcd_clr_line(1);
   printf(lcd_char, "%lx", adval.w);

   while(1)       /* loop continually */  ;
}

unsigned long make_ad_meas(void)
{
   union LONG x;
   x.b.h = 0x02;        // high byte
   x.b.l = 0x80;        // low byte
   return(x.w);         // the whole thing
}

#include <lcd_out.c>
```

Program CALIB.C.

In many applications, it is desirable to permit the user to set a quantity which is then saved to EEPROM.

For example, I recently developed a temperature measurement system for agricultural grain bins by measuring the forward voltage across a diode. Cables consisting of eight diodes spaced some six feet apart had already been installed in the bins. I was concerned with that diodes may well vary from one to another and thus decided to use a calibration potentiometer to permit the end user to "tweak" the measurement by -10.0 to 10.0 degrees.

The concept is that on boot, the A/D associated with the potentiometer is read and if it is near ground (adval < 23), the value previously stored in EEPROM is used in all subsequent calculations. However, if, on boot, the A/D value is above ground (adval >= 23), the adval is written to EEPROM, but the A/D is read and this value is used in all subsequent calculations. Thus, in my case, the user may adjust the potentiometer until the temperature values measured by my system agreed with some reference they might have available. Once satisfied, they would leave the potentiometer at that setting and reboot the processor. The processor would then read the A/D and on finding it is above ground, would write this value to EEPROM. The user would then turn power off and either set the pot to its low value or replace it with a ground. On all subsequent boots, the processor reads the pot A/D and on finding it near ground, it uses the value previously stored in EEPROM.

Note that A/D values in the range of 0 - 22 are not valid calibration values, leaving 1001 values in the range of 23 to 1023. The actual value of the parameter you are adjusting may be calculated;

```
q = ((float)(adval - 23)) /1000.0 * (highest - lowest) + lowest;
```

For example, in the example below, where the thermostat settings may be set between -40 and 150;

```
q = ((float)(adval - 23))/1000.0 * (150.0 - (-40.0)) + (-40.0)
```

This general idea might be adapted to any similar measurements or it might be used to permit the user to set thermostat trip points or to adjust the duty cycle of a PWM output or to set the period of a PIC output.

Using a potentiometer in conjunction with the PICs A/D converters is inexpensive and is far easier for the end user than asking that they interface the PIC circuit with a laptop to set data which is peculiar to their installation.

```
// CALIB.C
//
// On boot, reads the value of a potentiometer on A/D CH 0.  If near ground,
// uses a value previously stored in EEPROM and ignores the potentiometer.
// Otherwise, writes the potentiometer value to EEPROM, but uses potentiometer
// value.
//
// In this example. the potentiometer is used to set a thermostat trip point
// in the range  of -40.0 to 150.0 degrees.
//
// copyright, Peter H. Anderson, Jan, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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

#define TRUE !0
#define FALSE 0

unsigned long meas_pot(void);
float calc_therm_set(unsigned long adval);

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

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

void main(void)
{
   byte use_eeprom_flag;
   long adval;
   float T_F;

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

   lcd_init();

   adval = meas_pot();     // read calibration potentiometer

   if (adval < 23)         // if near ground, use the value stored in eeprom
   {
       use_eeprom_flag = TRUE;
   }

   else
   {
       // save pot val to EEPROM, but use potentiometer for the value
       save_to_eeprom(0x30, (byte *) &adval, 2);
       use_eeprom_flag = FALSE;
   }

   while(1)    // continually
   {
       if (use_eeprom_flag)
       {
              read_from_eeprom(0x30, (byte *) &adval, 2);
         }

         else
         {
              adval = meas_pot();
         }

         T_F = calc_therm_set(adval);
         lcd_clr_line(0);
       printf(lcd_char, "T_F = %3.2f", T_F);

       delay_ms(500);
   }
}

unsigned long meas_pot(void)
{
      unsigned long adval;

      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
```

```
      adgo = 1;
      while(adgo)    ;  // poll adgo until zero
      adval = ADRESH;
    adval = adval << 8 | ADRESL;
    return(adval);
}

float calc_therm_set(unsigned long adval)
{
      float T_F;
      unsigned long x;  // intermediate variable
    if (adval < 23)
    {
            T_F = -40.0;
      }
      else
      {
          T_F = ((float) (adval - 23)) * 190.0/1000.0 - 40.0;     // -40 to 150 degrees
      }
      return(T_F);
}

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;
   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>
```

Interfacing with SPI Devices, (Master).

This section illustrates using the PIC as a master in controlling such SPI devices as the Microchip 25LC640 EEPROM, Maxim MAX7219 7-Segment LED Display Controller, Texas Instruments TLC2543 11-channel 12-bit A/D and the new Microchip MCP3208 8-channel 12-bit A/D.

Note that none of these devices were supplied with the PIC16F87X Development Package. All are available from a number of sources including me. Note that the 7-segment LEDs used with the MAX7219 7-seg LED Driver are common cathode. I have both MAN74 (0.3 inch) and Ligitek LDS8151-10 (0.8 inch) 7-segment LEDs and will provide a number of schematics illustrating the connections of the MAX7219 to the LEDs.

The MSSP Module is discussed in Section 9 of the PIC16F87X Data Sheet.

The SPI interface consists of three common leads which are multipled to all devices on the bus and a unique select lead for each device as illustrated below.

```
PIC16F877             24LC640      TLC2543      MCP3208     MAX7219

RC5/SDO (term 24) -----SI (5)------DIN (17)-----DIN (11)---->DIN (term 1)
RC4/SDI (term 23) <----SO (2)------DOUT (16)----DOUT (12)
RC3/SCK (term 18) -----SCK (6)-----CLK (18)-----CLK (13)---->CLK (term 13)

RB5 (term 38) ---------/CS (1)
RB4 (term 37) ----------------------/CS (15)
RB3 (term 36) ----------------------------------/CS (10)
RB2 (term 35) ------------------------------------------> /CS (term 12)
```

This configuration was used in debugging all of the following routines. Having two different A/D converters on the same bus probably isn't all that practical, but one might, otherwise, use such a configuration for logging data and displaying data using the MAX7219.

Note that all outputs from the devices, identified as SO or DOUT are wire ored together. When a device is not selected it's ouput is in a high impedance state. When the device is selected, the output is normal logic; hard logic one or zero. Thus, only one device may be selected at a time.

For all of the above devices, a device is selected by bringing it's /CS lead low. (I know only of one device, a Dallas DS13XX Real Time Clock, where the device is enabled with a logic one). I have frequently gotten myself into trouble by leaving the /CS for devices I am not using in a high impedance input mode only to discover that one of the devices on the bus other than the one I was addressing was active. That is, a high impedance may not be adequate to maintain a device in the inactive mode. Thus, either use pullup resistors to +5VDC on the CS leads or be sure to bring the CS leads associated with all inactive devices high.

Each communications sequence begins by first selecting the device and then transferring eight bits out on SDO while receiving also receiving bits on SDI. The sequence may be simply one byte or it may be multiple bytes, but each sequence is terminated by "deselecting" the device.

A "bit-bang implementation of the I/O sequence is shown below;

```
byte spi_io(byte spi_byte)
{
    byte n;

    for(n=0; n<8; n++)
    {
       if (spi_byte & 0x80)  /* most sign bit first */
       {
               SDO_PIN = 1;
       }
       else
       {
           SDO_PIN = 0;
       }
       SCK_PIN = 1;

       spi_byte = (spi_byte << 1) | SDI_PIN;
       SCK_PIN = 0;
   }
   return(spi_byte);
}
```

Note that spi_byte is passed to the function. SDO_PIN is brought to the state of the most significant bit and this is read by the slave when SCK_PIN is brought high. spi_byte is then shifted to the left and the state of SDI is inserted in the least significant bit position. The next bit is then clocked out on SDO and the state of SDI is placed in the least significant bit position. Thus, at the end of the eight bit sequence, a byte has been output and the byte which was read is now in spi_byte which is returned to the calling function. Thus, the nature of the SPI bus is one of 8-bit transfers.

Although this function implies that a useful byte is being sent and a useful byte is being received, this usually is not the case. For example, a byte may be output to configure the 24LC640 EEPROM, but no useful data is received. Or, in reading a byte from the EEPROM, the byte which is sent is useless to the 24LC640. In the case of the MAX7219 Display Driver, no data is returned to the PIC and thus the byte which is read by the PIC is junk.

However, in the case of the TLC2543 and MCP3208 A/Ds, useful data is both sent to and received from the A/D in the same byte. (I assume the speed advantage of the SPI over the Philips I2C is related to the fact that with the SPI there is no "start", "address" and "stop" and data may be both sent and received at the same time.)

The user must configure the directions of the SDO, SDI and SCK terminals and the set the idle state of the clock.

```
void _25_640_setup_SPI(void)
{
    SDI_DIR = 1;        // configure SDI as input, SDO and SCK as outputs
    SDO_DIR = 0;
    SCK_DIR = 0;

    SCK_PIN = 0;        // be sure clock is at zero
}
```

In this case, the idle state of the SCK is zero, the data is read by the slave on the rising edge of the clock and the data is read by the master when the clock is high. All of these may vary from one device to another. For example, with the Microchip MCP3208, the clock idles high, data is read by the 3208 on the rising edge of the clock and output to the master on the falling edge. With the bit-bang approach, all of this is simply a matter of slightly modifying the above two routines. When using the PIC's SSP Module, these parameters are configured using the "ckp", stat_cke and stat_smp bits. But, whether using the bit-bang approach or the SSP module, recognize that if there are multiple devices on the bus, the formats may differ.

In summary, the SPI bus uses three leads, SDO, SDI and SCK which are multipled from one device to another on the bus. A unique chip select is associated with each SPI slave device and only one device may be selected. A sequence begins by selecting the device, sending and receiving one or more bytes and then "deselecting" the device.

The byte transfer may be bi-directional. The idle state of the clock (SCK), when the slave reads the master's output (SDO) and when the slave buts data on the master's input (SDI) may vary from one device to another.

Program 25_640BB.C.

This routine illustrates an interface with the Microchip 25LC640 EEPROM using a "bit-bang" implementation of the SPI protocol.

To guard against accidental writes to the EEPROM, the device is addressed (rb4 = 0) and a single byte write enable code (WREN) and the device is then "deselected". The EEPROM is again addressed and a write command code is sent, followed by the high byte of the EEPROM address, the low byte of the address and then, the actual data. The sequence is then ended by bringing the /CS back to logic one. A 5 ms delay is required to assure the data is burned into EEPROM.

Data is read by selecting the device, sending a read command code, followed by the high and low bytes of the EEPROM address. At this point, any byte may be clocked out and the data value is read.

```
// 25_640BB.C
//
// Illustrates the interface with Microchip 25LC640 EEPROM (SPI) using a bit bang
// implementation. This is useful for PICs not having an SSP interface or in
// applications where the SSP module is used for other purposes.
//
// Uses single byte write and read.  Writes 10 values beginning at location
// 0x0700 and then reads back the values and displays them on the LCD.
//
// Note that although the PIC terminals which are used are the same as the
// SSP module, any terminals may be used when using this bit bang approach.
//
//    PIC16F877                                              25LC640
```

```
//
// RC5/SDO (term 24) ------------------> SI (term 5)
// RC4/SDI (term 23) <------------------ SO (term 2)
// RC3/SCK (term 18) ------------------> SCK (term 6)
// RB4/CS (term 37) -------------------> /CS (term 1)
//
// 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 SPI_WREN 0x06         // various 24LC640 command codes defined
#define SPI_WRITE 0x02
#define SPI_READ 0x03

#define SCK_DIR trisc3        // SPI terminals defined
#define SDO_DIR trisc5
#define SDI_DIR trisc4
#define CS_DIR_25640 trisb4

#define SCK_PIN portc3
#define SDO_PIN portc5
#define SDI_PIN portc4
#define CS_PIN_25640 rb4

void _25_640_setup_SPI(void);
void _25_640_write_byte(unsigned long adr, byte dat);
byte _25_640_read_byte(unsigned long adr);
byte spi_io(byte spi_byte);

void main(void)
{
   byte n, dat;
   unsigned long  adr;

   lcd_init();

   CS_PIN_25640 = 1;          // Chip Select at logic one
   CS_DIR_25640 = 0;          // output

   _25_640_setup_SPI();

   lcd_cursor_pos(0, 0);
   for (n=0, adr = 0x0700; n<10; n++, adr++)
   {
     _25_640_write_byte(adr, n+10);       // write 10, 11, 12, etc
     lcd_char('!');                       // to see that something is happening
   }

   for (n=0, adr = 0x0700; n<10; n++, adr++)
```

```
   {
       dat = _25_640_read_byte(adr);            // now read back the data
       lcd_clr_line(1);
       lcd_dec_byte(dat, 2);                    // and display
       delay_ms(250);
   }

   while(1)       ;                                  // continual loop
}

void _25_640_setup_SPI(void)
{
    SDI_DIR = 1;        // configure SDI as input, SDO and SCK as outputs
    SDO_DIR = 0;
    SCK_DIR = 0;

    SCK_PIN = 0;        // be sure clock is at zero
}

void _25_640_write_byte(unsigned long adr, byte dat)
{
    byte dummy;

    CS_PIN_25640 = 0;                           // CS low
    dummy = spi_io(SPI_WREN);
    CS_PIN_25640 = 1;                           // CS high - end of WREN sequence

    CS_PIN_25640 = 0;                           // begin another session - 4 bytes
    dummy = spi_io(SPI_WRITE);
    dummy = spi_io((byte) (adr >> 8));          // high byte of adr
    dummy = spi_io((byte) adr);                 // low byte of address
    dummy = spi_io(dat);                        // data
    CS_PIN_25640 = 1;

    delay_ms(5);  // allow time for programming EEPROM
}

byte _25_640_read_byte(unsigned long adr)
{
    byte dummy, dat;

    CS_PIN_25640 = 0;               // begin 40byte sequence
    dummy = spi_io(SPI_READ);
    dummy = spi_io((byte) (adr >> 8));          // high byte of adr
    dummy = spi_io((byte) adr);                 // low byte of address
    dat = spi_io(dummy);                        // data
    CS_PIN_25640 = 1;

    return(dat);
}

byte spi_io(byte spi_byte)
{
    byte n;

    for(n=0; n<8; n++)
    {
```

```
        if (spi_byte & 0x80)  /* most sign bit first */
        {
            SDO_PIN = 1;
        }
        else
        {
            SDO_PIN = 0;
        }
        SCK_PIN = 1;

        spi_byte = (spi_byte << 1) | SDI_PIN;
        SCK_PIN = 0;
    }
    return(spi_byte);
}

#include <lcd_out.c>
```

Program 25_640_1.C.

This program is functionally the same as 25_640BB.C except that this program does use the PIC's SSP module.

In function *_25*_640_setup(), SSP module is configured as an SPI master with SCK running at f_osc / 64 (sspm3::sspm0 bits set to 0010). The idle state of the clock is defined as being zero (ckp = 0), the SDO data is defined as appearing on a positive clock transition (stat_cke = 1) and the SDI is to be read on the negative transition of the clock (stat_smp = 1). As with the bit-bang implementation, the directions of the SCK, SDO and SDI terminals must be configured and the state of SCK must be initialized to a zero when communicating with the 25LC640 EEPROM. In addition, the idle state of the clock is set to zero.

The equivalent function of the spi_io() function used in the bit-bang implementation is illustrated in the following snippet;

```
    SSPBUF = spi_byte;
    while(!stat_bf)   /* loop */         ;
    spi_byte = SSPBUF;
```

Note that the byte to be sent is loaded into SSPBUFF and it is automatically clocked out using the stat_cke parameter. And, just as with spi_io(), the byte received on SDI is clocked into SSPBUFF using the stat_smp bit as a definition of when to sample. Note that the program loops until the buffer full (stat_bf) bit is at one indicating the receive is complete.

```
// 25_640_1.C
//
// Illustrates the use of the SSP module in interfacing with a 25LC640
// EEPROM using the SPI protocol.  Illustrates single byte mode.
//
// Uses single byte write and read.  Writes 10 values beginning at location
// 0x0700 and then reads back the values and displays them on the LCD.
//
//    PIC16F877                                              25LC640
//
// RC5/SDO (term 24) -------------------> SI (term 5)
// RC4/SDI (term 23) <------------------- SO (term 2)
// RC3/SCK (term 18) -------------------> SCK (term 6)
// RB4/CS (term 37) --------------------> /CS (term 1)
//
//
```

```
// 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 SPI_WREN 0x06
#define SPI_WRITE 0x02
#define SPI_READ 0x03

void _25_640_setup_SPI(void);
void _25_640_write_byte(unsigned long adr, byte dat);
byte _25_640_read_byte(unsigned long adr);

void main(void)
{
   byte n, dat;
   unsigned long  adr;

   lcd_init();

   rb4 = 1;       // CS for 25LC640
   rb5 = 1;       // CS for TLC2543
   trisb4 = 0;
   trisb5 = 0;

   _25_640_setup_SPI();

   lcd_cursor_pos(0, 0);
   for (n=0, adr = 0x0700; n<10; n++, adr++)
   {
     _25_640_write_byte(adr, n+10);       // write 10, 11, 12, etc
     lcd_char('!');                       // to see that something is happening
   }

  for (n=0, adr = 0x0700; n<10; n++, adr++)
  {
     dat = _25_640_read_byte(adr);        // now read back the data
     lcd_clr_line(1);
     lcd_dec_byte(dat, 2);                // and display
     delay_ms(250);
  }

  while(1)        ;                       // continual loop
}

void _25_640_setup_SPI(void)
{
    sspen = 0;
    sspen = 1;
    sspm3 = 0; sspm2 = 0; sspm1 = 1; sspm0 = 0; // Configure as SPI Master, fosc / 64
```

```
    ckp = 0;                                    // idle state for clock is zero
    stat_cke = 1;                               // data transmitted on rising edge
    stat_smp = 1;                           // input data sampled at end of clock pulse

    portc3 = 0;
    trisc3 = 0;   // SCK as output 0

    trisc4 = 1;   // SDI as input
    trisc5 = 0;   // SDO as output
}

void _25_640_write_byte(unsigned long adr, byte dat)
{
    byte dummy;
    rb4 = 0;                                // CS low

    SSPBUF = SPI_WREN;
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    rb4 = 1;                                // CS high - end of WREN sequence

    rb4 = 0;                                // begin another session - 4 bytes

    SSPBUF = SPI_WRITE;
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    SSPBUF = (byte) (adr >> 8);     // high byte of adr
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    SSPBUF = (byte) (adr);    // low byte of adr
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    SSPBUF = dat; // data
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    rb4 = 1;
    delay_ms(5);  // allow time for programming EEPROM
}

byte _25_640_read_byte(unsigned long adr)
{
    byte high_adr, low_adr, dummy, dat;

    rb4 = 0;            // CS low

    SSPBUF = SPI_READ;
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    high_adr = adr >> 8;

    SSPBUF = high_adr;  // high byte of adr
```

```
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    low_adr = adr;
    SSPBUF =  low_adr;  // low byte of adr
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    SSPBUF = dummy;
    while(!stat_bf)   /* loop */         ;
    dat = SSPBUF;
    rb4 = 1;

    return(dat);
}

#include <lcd_out.c>
```

Program 25_640_2.C.

This program extends on the previous to illustrate a sequential write and read..

In writing data, the EEPROM is write enabled by sending the WREN command. This sequence is followed with sending the WRITE command, the high and low bytes of the EEPROM address and then each of the eight data bytes. In reading data, the READ command is sent, followed by the high and low bytes of the address and the eight bits are then read.

Up to 32 bytes may be written or read. However, all bytes must reside on the same 32 byte page. Thus, 0x07e0 would be a valid start address for a 32 byte sequential write. Addresses 0x07e1 or 0x7f0 would not be valid.

Note that in main(), I used two const arrays. One limitation of the CCS compiler is that pointers to const arrays may not be passed to a function. Thus, I simply copied the const array to a conventional RAM array and then called the function to write the data to the EEPROM.

There is one detail in function display() that is worthy of note; the use of the mod (%) operator to advance the cursor to the next line. Although this looks nice and compact in C, the mod operator, particularly with long ints, uses a good deal of program memory as the implementation is one of dividing and then taking the result and multiplying and then subtracting this from the quantity;

```
      q = (n+1) / 4;
      m = q * 4;
      mod = (n+1) – m;
```

Program memory is not an issue until you run out of it. However, a more efficient implementation of the advancing of the cursor which really doesn't sacrifice clarity;

```
void display(byte *read_dat, byte num_bytes)
{
    byte m, n, line = 0;
    lcd_clr_line(0);
    for (n=0, m = 0; n<num_bytes; n++, m++)
    {
       if ((m == 4)           // four values per line
       {
```

```
         m = 0;
         ++line;
         lcd_clr_line(line);
       }
       lcd_hex_byte(read_dat[n]);
       lcd_char(' ');
   }
}

// 25_640_2.C
//
// Illustrates the use of the SSP module in interfacing with a 25LC640
// EEPROM using the SPI protocol.  Illustrates block write and read mode.
//
// Writes one block of eight bytes beginning to locations beginning at
// 0x0700 and another block to 0x0708.  Then reads back the values and
// displays them on the LCD.
//
//    PIC16F877                                              25LC640
//
// RC5/SDO (term 24) -------------------> SI (term 5)
// RC4/SDI (term 23) <------------------- SO (term 2)
// RC3/SCK (term 18) -------------------> SCK (term 6)
// RB4/CS (term 37) --------------------> /CS (term 1)
//
// 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 SPI_WREN 0x06
#define SPI_WRITE 0x02
#define SPI_READ 0x03

void _25_640_setup_SPI(void);
void _25_640_write_seq_bytes(unsigned long adr, byte *write_dat, byte num_bytes);
void _25_640_read_seq_bytes(unsigned long adr, byte *read_dat, byte num_bytes);

void display(byte *read_dat, byte num_bytes);

void main(void)
{
    byte dat_array[8], n;
    byte const a[8] = {0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00};
    byte const b[8] = {0x17, 0x16, 0x15, 0x14, 0x13, 0x12, 0x11, 0x10};

    lcd_init();

    rb4 = 1;            // CS for 25LC640
```

```
    rb5 = 1;            // CS for TLC2543
    trisb4 = 0;
    trisb5 = 0;

    _25_640_setup_SPI();

    for(n=0; n<8; n++)              // write a block of eight bytes
    {
       dat_array[n] = a[n];
    }

    _25_640_write_seq_bytes(0x700, dat_array, 8);

    for(n=0; n<8; n++)              // write another block
    {
       dat_array[n] = a[n];
    }

    _25_640_write_seq_bytes(0x708, dat_array, 8);  // write another


    _25_640_read_seq_bytes(0x700, dat_array, 8);  // read each block and display
    display(dat_array, 8);

    _25_640_read_seq_bytes(0x708, dat_array, 8);
    display(dat_array, 8);

    while(1)      ;
}

void _25_640_setup_SPI(void)
{

    sspen = 0;
    sspen = 1;
    sspm3 = 0; sspm2 = 0; sspm1 = 1; sspm0 = 0; // Configure as SPI Master, fosc / 64
    ckp = 0;                                    // idle state for clock is zero
    stat_cke = 1;                               // data transmitted on rising edge
    stat_smp = 1;                          // input data sampled at end of clock pulse

    portc3 = 0;
    trisc3 = 0;   // SCK as output 0

    trisc4 = 1;   // SDI as input
    trisc5 = 0;   // SDO as output
}

void _25_640_write_seq_bytes(unsigned long adr, byte *write_dat, byte num_bytes)
{
    byte dummy, n;

    rb4 = 0;            // CS low - begin WREN sequence

    SSPBUF = SPI_WREN;
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;
```

```
    rb4 = 1;

    rb4 = 0;

    SSPBUF = SPI_WRITE;
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    SSPBUF = (byte) (adr >> 8);     // high byte of adr
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    SSPBUF = (byte) (adr);    // low byte of adr
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    for (n=0; n<num_bytes; n++)
    {

       SSPBUF = write_dat[n]; // data
       while(!stat_bf)   /* loop */         ;
       dummy = SSPBUF;
    }

    rb4 = 1;                  // end of sequence
    delay_ms(5);  // allow time for programming
}

void _25_640_read_seq_bytes(int adr, byte *read_dat, byte num_bytes)
{
    byte high_adr, low_adr, dummy, n;

    rb4 = 0;            // CS low - begin sequence

    SSPBUF = SPI_READ;
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    high_adr = adr >> 8;

    SSPBUF = high_adr;  // high byte of adr
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    low_adr = adr;

    SSPBUF =  low_adr;  // low byte of adr
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    for (n=0; n<num_bytes; n++)
    {
       SSPBUF = dummy;
       while(!stat_bf)   /* loop */         ;
       read_dat[n] = SSPBUF;
    }
    rb4 = 1;            // end of sequence
```

```
}

void display(byte *read_dat, byte num_bytes)
{
    byte n, line = 0;
    lcd_clr_line(0);
    for (n=0; n<num_bytes; n++)
    {
      lcd_hex_byte(read_dat[n]);
      lcd_char(' ');

      if (((n+1)%4) == 0)           // four values per line
      {
         ++line;
         lcd_clr_line(line);
      }
   }
}

#include <lcd_out.c>
```

Program 2543_1.C.

This routine illustrates an interface with a Texas Instruments TLC2543, 11-channel, 12-bit A/D.

A command byte of the form;

```
WWWW XX Y Z
```

where WWW is the channel (0-11), XX is the data length (11 for 16 bits) , Y is either most (0) or least (1) significant bit first and Z indicates whether the data is to be returned in unipolar or bipolar format.

Thus, for a specific channel;

```
command = (channel << 4) | 0x0c | bipolar;
```

In function ad_meas(), the command byte is sent, followed by a dummy byte so as to make the sequence 16 bytes long. Note that at the same time this command is being sent to the TLC2543, the result of the previous command is being returned. However, in this context, we have no way of knowing what the previous command was and the data is ignored.

The data is then fetched in the next two byte sequence. The 12 data bits are the first 12 bits returned and thus the result could be calculated as;

```
ad_val = high_byte;
ad_val = (ad_val << 8) | low_byte;        // put the two bytes together
ad_val = ad_val >> 4                      // move to low 12-bits
```

I opted for a more efficient, but less clear implementation;

```
ad_val = (((unsigned long) high_byte) << 4) | (low_byte >> 4);
```

With the bipolar command bit set to zero the result is in the range of 0x000 – 0xfff with 0x800 being the midpoint. With the bipolar bit set to a one, the midpoint is 0x000 and extends up to 0x7ff, and down from the midpoint from 0x000 to 0xfff and down to 0x800. Thus, its is a two's complement representation relative to the midpoint.

```
// 2543_1.C
//
// Illustrates interfacing with a TI TLC2543 11-channel 12-bit A/D using the PIC's
// SSP module in the SPI mode.
//
// Performs an A/D unipolar measurement on Channel 0 and a bipolar measurement on Ch 1
// and displays the result on the LCD.
//
//    PIC16F877                                        TLC2543
//
// RC5/SDO (term 24) ---------------- DIN (term 17)
// RC4/SDI (term 23) <--------------- DOUT (term 16)
// RC3/SCK (term 18) ---------------- CLK (term 18)
// RB5 (term 38) ------------------> /CS (term 15
//
// 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

unsigned long ad_meas(byte channel, byte bipolar);
void display(byte channel, unsigned long ad_val);

void main(void)
{
  byte channel;
  unsigned long ad_val;

  lcd_init();

  while(1)
  {
      channel = 0;
      ad_val = ad_meas(channel, 0); // unipolar measurement on Ch 0
      display(channel, ad_val);

      channel = 1;
      ad_val = ad_meas(channel, 1); // bipolar measurement on Ch 1
      display(channel, ad_val);

      delay_ms(1000);
   }
}

unsigned long ad_meas(byte channel, byte bipolar)
{
    byte high_byte, low_byte, dummy;
    unsigned long ad_val;
```

```
    sspen = 0;
    sspen = 1;
    sspm3 = 0; sspm2 = 0; sspm1 = 1; sspm0 = 0; // Configure as SPI Master, fosc / 64
    ckp = 0;                                // idle state for clock is zero
    stat_cke = 1;                           // data transmitted on rising edge
    stat_smp = 1;                           // input data sampled at end of clock pulse

    portc3 = 0;
    trisc3 = 0;   // SCK as output 0

    trisc4 = 1;   // SDI as input
    trisc5 = 0;   // SDO as output

    rb5 = 1;
    trisb5 = 0;   // be sure CS is high

                        // write the command
    rb5 = 0;            // bring CS low
    delay_10us(10);

    SSPBUF = (channel << 4) | (0x0c + bipolar);
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    SSPBUF = 0x00;      // send a dummy byte
    while(!stat_bf)   /* loop */           ;
    dummy = SSPBUF;

    rb5 = 1;            // CS high

                        // now read the result
    rb5 = 0;            // bring CS low
    delay_10us(10);

    SSPBUF = 0x00;      // send a dummy byte
    while(!stat_bf)   /* loop */         ;
    high_byte = SSPBUF;

    SSPBUF = 0x00;      // send a dummy byte
    while(!stat_bf)   /* loop */           ;
    low_byte = SSPBUF;

    rb5 = 1;            // CS high

    ad_val = (((unsigned long) high_byte) << 4) | (low_byte >> 4);
    return(ad_val);
}

void display(byte channel, unsigned long ad_val)
{
   lcd_clr_line(channel);
   printf(lcd_char, "%d  %4lx", channel, ad_val);
}

#include <lcd_out.c>
```

Program 2543_2.C.

This program illustrates how multiple measurements may be rapidly performed with a command for a new measurement being sent to the TLC2543 while at the same time fetching the result of the previous command.

Note that in performing four measurements, five two-byte sequences are required. On the first sequence, the command is sent, but the result is ignored. On the fifth sequence, the command has no meaning but is executed to fetch the fourth measurement.

In developing this routine I noted a very troubling deficiency with the CCS complier; the inability to handle arrays of longs in a function.

The original code was;

```
m = n – 1;
ad_vals[m] = ad_val;
```

where n is in the range of 1 to 4 and thus m is in the range of 0 to 3.

The problem was that the address location was not advancing by two bytes to accommodate each long, but rather by a single byte. This had me going for many hours and I am surprised that I have never seen this behavior prior to this time.

I did find a work around in incrementing the pointer;

```
++ad_vals
*ad_vals = ad_val;
```

This did work. In incrementing the pointer, it does advance by the correct two bytes to accommodate each long.

The version (2.686) of PCM I am currently using is more than a year old. Hopefully, this has been corrected as bugs in such fundamentals are troubling.

```
// 2543_2.C
//
// Illustrates interfacing with a TI TLC2543 11-channel 12-bit A/D using the PIC's
// SSP module in the SPI mode.
//
// Performs a sequence of A/D measurements on specified channels (array channels[]) with
// specified polarities (array polarities[]).  The results are displayed on the LCD.
//
// This is an example of how the TLC2543 may be configured for the next A/D measurement
// while at the same time receiving the result of the previous command.
//
//
//    PIC16F877                                        TLC2543
//
// RC5/SDO (term 24) ---------------- DIN (term 17)
// RC4/SDI (term 23) <--------------- DOUT (term 16)
// RC3/SCK (term 18) ---------------- CLK (term 18)
// RB5 (term 38) ------------------> /CS (term 15
//
// 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


void mult_ad_meas(byte *channels, byte *polarities, long *ad_vals, byte num_channels);
void display(byte *channels, long *ad_vals, byte num_channels);

#define NUM_CHANNELS 4

void main(void)
{
  byte channels[NUM_CHANNELS] = {0, 1, 3, 4};
  byte polarities[NUM_CHANNELS] = {0, 1, 0, 0};
  long ad_vals[NUM_CHANNELS];

  lcd_init();
  while(1)
  {
     mult_ad_meas(channels, polarities, ad_vals, NUM_CHANNELS);
     display(channels, ad_vals, NUM_CHANNELS);
     delay_ms(1000);
  }
}

void mult_ad_meas(byte *channels, byte *polarities, long *ad_vals, byte num_channels)
{
   byte high_byte, low_byte, m, n;
   unsigned long ad_val;
   sspen = 0;
   sspen = 1;
   sspm3 = 0; sspm2 = 0; sspm1 = 1; sspm0 = 0;  // Configure as SPI Master, fosc / 64
   ckp = 0;                                     // idle state for clock is zero
   stat_cke = 1;                                // data transmitted on rising edge
   stat_smp = 1;                                // input data sampled at end of clock pulse

   portc3 = 0;
   trisc3 = 0;    // SCK as output 0

   trisc4 = 1;    // SDI as input
   trisc5 = 0;    // SDO as output

   rb5 =1;
   trisb5 = 0;    // be sure CS is high

   for (n=0; n<num_channels+1; n++)
   {
      rb5 = 0;          // CS low
      delay_10us(10);

      if (n==num_channels)    // if it is the last meas
      {
         SSPBUF = 0x00; // dummy
```

```
      }
      else
      {
         SSPBUF = (channels[n] << 4) | (0x0c + polarities[n]);
      }

      while(!stat_bf)   /* loop */         ;
      high_byte = SSPBUF;

      SSPBUF = 0x00;    // send a dummy byte
      while(!stat_bf)   /* loop */           ;
      low_byte = SSPBUF;

      rb5 = 1;          // CS high

      if (n!=0)   // if it is not the first
      {
          ad_val = (((unsigned long) high_byte) << 4) | (low_byte >> 4);
          *ad_vals = ad_val;
          ++ad_vals;                // ******** See text *******
      }
   }
}

void display(byte *channels, long *ad_vals, byte num_channels)
{
   byte n;

   for (n=0; n<num_channels; n++)
   {
      lcd_clr_line(n);
      printf(lcd_char, "%d %4lx", channels[n], *ad_vals);
      ++ad_vals;                    // ******** See text ********
   }
}

#include <lcd_out.c>
```

Program 3208_1.C.

The MCP320X family of A/D converters are relatively new offerings by Microchip. The MCP3208 8-channel 12-bit A/D is somewhat less expensive than the TLC2543, about $5.00.

The device may be configured for either single ended measurements or for differential measurements between channels.

For single ended measurements, the five command bits consist of

```
1 1 CCC
```

where CCC is the specific channel.

For differential measurements the five command bits are;

```
1 0 CC P
```

Where CC is the channel pair and P is the polarity. For example Channel 6 relative to Channel 7 is channel pair 3 with a polarity of 0.

Life becomes interesting, as these five bits are split between two bytes. The three most significant command bits are the lowest three bits of the first byte and the lowest two bits are the highest two bits or the second byte.

Thus for single ended measurements;

```
      command = 0x04 + 0x02 + ((channel >> 2) & 0x01);      // first byte
      command = (channel & 0x03) << 6;    // next byte - low two bits of channel are
                                          // highest two bits
```

For differential measurements;

```
      command = 0x04 + 0x00 + (channel >> 1) & 0x01;  // 10 and high bit of channel
      command = (((channel & 0x01) << 1) | polarity) << 6;  // next byte
```

Unlike the TLC2543, the MCP returns the result in the same two byte sequence.

```
// 3208_1.C
//
// Illustrates interfacing with a Microchip MCP3208 8-channel 12-bit A/D.  Performs
// A/D measurements on Ch 0 and Ch 1 and differential measurements Ch0+ relative to Ch 1
// and Ch1+ relative to Ch 0.  Displays results to LCD.
//
// RC5/SDO (term 24) -------------> DIN (11)
// RC4/SDI (term 23) <------------- DOUT (12)
// RC3/SCK (term 18) -------------> CLK (13)
// RB3 (term 36) -----------------> /CS (10)
//
// 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

unsigned long ad_meas_single_end(byte channel);
unsigned long ad_meas_diff(byte channel, byte polarity);

void display(byte line, unsigned long ad_val);

void main(void)
{
  byte channel;
  unsigned long ad_val;

  lcd_init();

  while(1)
  {
```

```
      ad_val = ad_meas_single_end(0);           // single end measurement on Ch 0
      display(0, ad_val);

      ad_val = ad_meas_single_end(1);           // and Ch 1
      display(1, ad_val);

      ad_val = ad_meas_diff(0, 0);
          // differential between Ch0 and Ch1 - Ch 0 more positive
      display(2, ad_val);

      ad_val = ad_meas_diff(0, 1);        // differential - Ch 1 more postive
      display(3, ad_val);

      delay_ms(3000);
   }
}

unsigned long ad_meas_single_end(byte channel)
{
    byte command, dummy, high_byte, low_byte;
    unsigned long ad_val;

    sspen = 0;
    sspen = 1;
    sspm3 = 0; sspm2 = 0; sspm1 = 1; sspm0 = 0; // Configure as SPI Master, fosc / 64
    ckp = 1;                                // idle state for clock is zero
    stat_cke = 0;                           // data transmitted on rising edge
    stat_smp = 1;                           // input data sampled at end of clock pulse

    portc3 = 1;
    trisc3 = 0;   // SCK as output 0

    trisc4 = 1;   // SDI as input
    trisc5 = 0;   // SDO as output

    rb3 =1;
    trisb1 = 0;   // be sure CS is high

    rb3 = 0;
    delay_10us(10);

    command = 0x04 + 0x02 + ((channel >> 2) & 0x01);

    SSPBUF = command;
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    command = (channel & 0x03) << 6;      // low two bits of channel

    SSPBUF = command;
    while(!stat_bf)   /* loop */         ;
    high_byte = SSPBUF;

    SSPBUF = 0x00;
    while(!stat_bf)   /* loop */         ;
    low_byte = SSPBUF;
```

```
    rb3 = 1;

    ad_val = high_byte & 0x0f;
    ad_val = (ad_val << 8) | low_byte;
    return(ad_val);
}

unsigned long ad_meas_diff(byte channel, byte polarity)
{
    byte command, dummy, high_byte, low_byte;
    unsigned long ad_val;

    sspen = 0;
    sspen = 1;
    sspm3 = 0; sspm2 = 0; sspm1 = 1; sspm0 = 0; // Configure as SPI Master, fosc / 64
    ckp = 1;                                    // idle state for clock is zero
    stat_cke = 0;                           // data transmitted on rising edge
    stat_smp = 1;                           // input data sampled at end of clock pulse

    portc3 = 1;
    trisc3 = 0;   // SCK as output 0

    trisc4 = 1;   // SDI as input
    trisc5 = 0;   // SDO as output

    rb3 = 1;
    trisb3 = 0;   // be sure CS is high

    rb3 = 0;
    delay_10us(10);

    command = 0x04 + 0x00 + (channel >> 1) & 0x01;

    SSPBUF = command;
    while(!stat_bf)   /* loop */         ;
    dummy = SSPBUF;

    command = (((channel & 0x01) << 1) | polarity) << 6;
                                          // low bit of channel and polarity

    SSPBUF = command;
    while(!stat_bf)   /* loop */         ;
    high_byte = SSPBUF;

    SSPBUF = 0x00;
    while(!stat_bf)   /* loop */         ;
    low_byte = SSPBUF;

    rb3 = 1;

    ad_val = high_byte & 0x0f;
    ad_val = (ad_val << 8) | low_byte;
    return(ad_val);
}

void display(byte line, unsigned long ad_val)
{
```

```
   lcd_clr_line(line);
   printf(lcd_char, "%4lx", ad_val);
}

#include <lcd_out.c>
```

Program 7219_1.C.

The MAX7219 is capable of interfacing with up to eight 7-segment plus decimal point common anode LEDs. The theory of the device is that the state of the eight anodes to display the desired digit is output to a common bus to all 7-segment LEDs and the cathode on the that display is brought low. The MAX7219 then moves to the next digit, etc. Thus, only a single display is on at one time with the MAX7219 rapidly moving from one display to the next.

The MAX7219 consists of registers "0 – f". Registers "1 – 8" are associated with the eight display devices, "9" with the decoding of the data, "a" with the intensity, "b" with the scan limit (number of displays), "c" with shutdown and "f" with LED test.

Note that no external series limiting resistors are required. The current is controlled with a single resistor and the intensity register.

Thus, each transfer to the MAX7219 consists of two bytes; the register to be written followed by the data. Note that no data is returned by the MAX7219.

This routine uses "Code B" decoding. Thus to display the character "8", the data is simply 0x08. Data values 0x0a – 0x0f display the characters "-", "E", "H", "L", "P" and "blank", respectively.

The alternative to Code B decoding is to write the actual segments as "0111 1111" (0x7f) where all segments except the decimal point are operated in displaying the character "8". This permits a user to define any character, at least within the limits of what one can do with seven segments.

Note that you might use a cadmium sulfide resistor in a voltage divider arrangement and use an A/D converter to control the intensity so as to compensate for ambient light.

As an aside, one might use a MAX7219 to control up to 64 discrete LEDs and use non-code B decoding to generate all manner of fancy patterns. (There are Christmas Star packages on the market that use a MAX7219 and a handful of conventional LEDs costing pennies apiece). If your spouse wasn't all that impressed with you finally getting an LED to wink after several hundred dollars and many days, they may feel you are making progress if you offer up 64 LEDs or extend to two MAX7219s to 128 that write their name.

```
// 7219_1.C,
//
// Illustrates the use of a MAX7219 8-digit LED driver to control four
// common cathode 7-segment LEDs.
//
// Sets up MAX7219 for code B decoding (4-bit), blanks all digits and displays 0 - 15
// on least significant LED display.
//
// Displays 995 through 1005 with leading zero suppression.
//
//    PIC16F877                          MAX7219
//
// RC5/SDO (term 24) ------------------->DIN (term 1)
// RC4/SDI (term 23) <------
```

```
// RC3/SCK (term 18) ------------------->CLK (term 13)
// RB2/CS (term 35) -------------------->/CS (term 12)
//
// 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

void _7219_dec_display_unsigned(unsigned long v, byte zero_suppression);
void _7219_blank_all(void);
void _7219_setup(void);
void _7219_shutdown(void);

void _7219_display_4(byte *patts);
void _7219_digit_display(byte digit_num, byte d);

void _7219_out(unsigned long d);

#define NO_OP 0x0000
#define DECODE_MODE 0x0900
#define INTENSITY 0x0a00
#define SCAN_LIMIT 0x0b00
#define SHUT_DOWN 0x0c00
#define DISP_TEST 0x0f00

#define BLANK 0x0f
#define MINUS 0x0a      // Code B for a minus sign

void main(void)
{
   int n;
   unsigned long v;

   while(1)
   {
      _7219_setup();      // turn on, code B decoding, medium intensity, 4 digits
      _7219_blank_all();
      for (n=0; n<16; n++)
      {
         _7219_digit_display(0, n);
         // display each of the characters on least signif display
         delay_ms(1000);
      }

      for (n=0; n<10; n++)    // display the numbers 995 - 1005
      {
         v = 995 + n;
         _7219_dec_display_unsigned(v, 1);   // leading zero suppression on
         delay_ms(1000);
      }
```

```
   }
}


void _7219_dec_display_unsigned(unsigned long v, byte zero_suppression)
// leading zero suppression
{
   byte d, digits[4];

   d = v/1000;    // display the number of thousands
   if( (d) || (!zero_suppression) )
   // if the digit is not zero OR if there is no zero suppression
   {
      digits[3] = d;
      zero_suppression = 0;         // no zero suppression
   }
   else  // d is zero and zero suppression
   {
      digits[3] = BLANK;
   }
   v = v % 1000;

   d = v / 100;   // display the number of hundreds
   if( (d) || (!zero_suppression))
   {
      digits[2] = d;
      zero_suppression = 0;
   }
   else
   {
      digits[2] = BLANK;
   }

   v = v % 100;
   d = v / 10;    // tens
   if( (d) || (!zero_suppression))
   {
      digits[1] = d;
   }
   else
   {
      digits[1] = BLANK;
   }

   v = v % 10;
   d = v;      // units
   digits[0] = d;
   _7219_display_4(digits);
}

void _7219_blank_all(void)
{
   byte n, digits[4];
   for(n=0; n<4; n++)
   {
      digits[n]=BLANK;
   }
```

```
   _7219_display_4(digits);
}

void _7219_display_4(int *patts)   // display content of patts
{
   byte n;
   for(n=0; n<4; n++)
   {
      _7219_digit_display(n, patts[n]);
   }
}

void _7219_digit_display(byte digit_num, byte d)
      // display d on display digit_num
{
   long v;
   v = ((long) (digit_num + 1) << 8) | d; // display number is in high byte
   _7219_out(v);
}


void _7219_setup(void)
{
   sspen = 0;
   sspen = 1;
   sspm3 = 0; sspm2 = 0; sspm1 = 1; sspm0 = 0;  // Configure as SPI Master, fosc / 64
   ckp = 0;                          // idle state for clock is zero
   stat_cke = 1;                     // data transmitted on rising edge
   stat_smp = 1;                     // not really necessary in this application

   portc3 = 0;
   trisc3 = 0;    // SCK as output 0

   trisc4 = 1;    // SDI as input
   trisc5 = 0;    // SDO as output

   rb2 = 1;
   trisb2 = 0;    // be sure CS is high

   _7219_out(DISP_TEST | 0x00);     // normal
   _7219_out(SHUT_DOWN | 0x01);     // take it out of low power mode
   _7219_out(SCAN_LIMIT | 0x04);    // 4 digits
   _7219_out(DECODE_MODE | 0xff);   //code B decode
   _7219_out(INTENSITY | 0x08);     // medium intensity
}

void _7219_shutdown(void)
{
   _7219_out(SHUT_DOWN | 0x00);  // turn it off
}

void _7219_out(unsigned long d)
{
   byte dummy;

   rb2 = 1;
   trisb2 = 0;
```

```
   rb2 = 0;       // bring CS low
   delay_10us(10);

   SSPBUF = (byte) (d >> 8);
   while(!stat_bf)   /* loop */           ;
   dummy = SSPBUF;

   SSPBUF = (byte) (d);
   while(!stat_bf)   /* loop */           ;
   dummy = SSPBUF;

   rb2 = 1;       // CS high
}

#include <lcd_out.c>
```