Tutorial by Example – Issue 1B

Introduction.

This is the fourth distribution to those who have purchased the PIC16F87X Development Package. The total package consists of Tutorial by Example – Issue 1 and Issue 1A and finally this discussion. All routines for all distributions are contained in file routines.zip.

This distribution discusses two more SPI devices, the Dallas DS1305 Real Time Clock and the Atmel AT45 high density Flash EEPROMs. It also deals with implementing the I2C protocol using both "bit bang" and using the SSP module as an I2C Master and illustrates interfaces with the Microchip 24LC256 EEPROM, Philips PCF8574 8-bit I/O expander, Dallas DS1803 Dual Potentiometer, Maxim MAX518 Dual D/A, Dallas DS1307 Real Time Clock, Dallas DS1624 Thermometer and EEPROM and Philips PCF8583 Real Time Clock and Event Counter.

Note that none of these parts nor schematics were included in the Development package. However, a brief ASCII diagram is included with each routine and using this along with the manufacturer's data sheet should be adequate. I stock all of the above devices except the Atmel AT45 EEPROM. All routines assume the LCD arrangement and possibly an LED provided with the Development Package.

This installment is a bit late and not as complete as I would have liked. Once school begins, there isn't all that much "free" time when my mind is also alert and it doesn't take much to bog me down for some time. For example, I apparently overheated an AT45 EEPROM when mounting it on an adaptor and managed to waste a weekend fooling with a dead device.

The next installment will be distributed well before April 15, but it may be somewhat abbreviated. My goal is the Dallas 1-W, I2C Slave, SPI Slave and RS232 routines using both bit bang and using the UART. However, don't depend on getting all of this.

Note that other enthusiasts may now buy a standalone subscription to this tutorial which includes all of the prior material as well anything developed in the future for $18.00. The price goes up as more material is developed. Any exposure you can give in promoting this effort and feel comfortable in doing so, would be greatly appreciated.

I enjoy doing this, which is good as the arithmetic is not. Currently there are 42 subscribers and allocating $13 each, one arrives at the staggering sum of nominally $550. Spread over 170 pages, this is about $3.00 per page which translates into about $1.00 per hour. But, as I say, I do enjoy it, but would like to see it become a bit larger revenue generator. Of course, I haven't taken into account the movie royalties!

SPI Master (continued).

Program DS1305_1.c

This program illustrates how an interface with the Dallas DS1305 Real Time Clock.

The DS1305 is one of the most versatile real time clocks I have used with such features as;

1. Powering the device with a single supply, powering with a single supply while trickle charging a second backup supply and a single supply with battery backup. The DS1305 provides a register and the associated circuitry to control the trickle charge. A power fail output (/PF) is also provided.

2. The SDO output logic “one” level is defined by the voltage appearing at input V_CCIF.

3. The device provides a number of time alarm features including alarm each day, hour, minute and second or alarm when the time matches the alarm registers and this is tied to two alarm outputs which may be used as interrupts.

However, this routine is limited to writing a date and time to the device and then periodically reading and displaying the date and time and reading the time, calculating the elapsed time in the current day and writing to and reading from the RAM at locations 0x20 – 0x7f.

The DS1305 is unique in my experience in that the device is selected by bringing the CS (or CE) lead high. All other devices I have used use an active low. Note that failure to observe this and erroneously output logic ones to all devices on an SPI bus will in fact cause the DS1305 to be active and will cause problems when communicating with another SPI device. However, aside from being different, this peculiarity of the DS1305 causes no complications.

Writing to the DS1305 is implemented by bringing the CE high, sending the register start address plus 0x80 to indicate a write and then the data, byte by byte. The sequence is terminated by bringing CE low.

Reading from the device is implemented by bringing CE high, sending the address of the first register to be read and then reading the data byte by byte.

One common problem I have noted in various discussions on news groups is a failure to clear the WP bit in the control register to zero and to enable the oscillator by clearing bit /EOSC.

Thus, in this routine, the SPI bus is configured for interfacing with the DS1305 and 0x00 is written to the control register.

```
   _1305_setup_SPI();
   _1305_write_config(0x00); // clear WP and /EOSC
```

Note that the date and time are stored in BCD format which is nice when displaying the date and time, but not for performing calculations to compute a future date and time as one may wish to do when using the device in an alarm mode.

In this routine, an array of seven bytes where each element has been #defined as SEC, MINI, etc, is used to pass the date and time to functions to write, read and display the date and time. A structure is used in another routine for the Dallas DS1307 RTC which uses the Philips I2C bus, but in fact, I see no advantage in one technique over the other.

A base date and time is written to the DS1305 and then the date and time is periodically read and displayed on the LCD.

Using the RAM at locations 0x20 – 0x7f is implemented in much the same manner as the date and time. In this routine, ten values are written to RAM and then read back and displayed.

A “timer” function which returns the number of seconds in the current day is implemented. Note that this poses a problem as there are 86,400 seconds in a day which cannot be stored in a 16 bit CCS long and working with a structure to implement a 17 or 24 bit variable is cumbersome. However, the CCS float uses a 23 bit mantissa and thus, I think one can use a float and still maintain a resolution of one second. I say, “I think” as I wouldn’t want anyone designing life support equipment on the assumption that I am always correct.

The utility of such a “timer” function is in performing periodic tasks. For example;

```
      zero the time
      t_old = 0.0;
      while(1)
      {
          t_new = timer();
          if (t_new >= t_old + 180.0)     // three minutes
          {
              t_old = t_old + 180.0;
              portd0 = portd0 ^ 0x01;     // do the task
          }
          // do other stuff
      }
```

Note that the code becomes a bit more complex when the time rolls over to a new day. I believe there is a sample routine (in Visual Basic) for the BasicX BX24 on my web site.

```
// DS1305_1.C
//
// Illustrates interface with DS1305 real time clock and RAM.
//
// Configures control register to zero WP bit and enable oscillator.
// Writes a time and date to timer registers (addresses 0x00 - 0x06)
// and then reads and displays time and date at nominally 2 sec
// intervals.
//
// Illustrates how to write to and read from RAM at locations 0x20 –
// 0x7f.
//
// Illustrates how to fetch the time and calculate the number of
// elapsed seconds since the start of the day.
//
//    PIC16F877                           DS1305                  To Other SPI Dev
//
// RC5/SDO (term 24) -------------------> SI (term 12) ------>
// RC4/SDI (term 23) <------------------- SO (term 13) <------
// RC3/SCK (term 18) -------------------> SCK (term 11) ----->
// RB0/CS (term 33) --------------------> CE (term 10)
//
// In this example;
//
//  VCC2 (term 1)  +5VDC
//  VCC1 (term 16) GRD
//  VBAT (term 2) GRD
//  VCCIF (term 14) +5 VDC (Determines level of logic one)
//  SERMODE (term 9) +5VDC (SPI Mode)
//  X1, X2 (terms 3, 4) 32.768 kHz Crystal
//  INT0, /INT1, /PF - Open (Not used)
//
// Copyright, Peter H. Anderson, Baltimore, MD, Feb, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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

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

```
#define SEC 0
#define MINI 1
#define HOUR 2
#define DAY 3
#define DATE 4
#define MONTH 5
#define YEAR 6

void _1305_setup_SPI(void);
void _1305_write_config(byte control_byte);
void _1305_write_date_time(byte *dt);
void _1305_read_date_time(byte *dt);
void _1305_write_ram(byte adr, byte *d, byte num_bytes);
void _1305_read_ram(byte adr, byte *d, byte num_bytes);

void display_date_time(byte *dt);
void display_RAM_data(byte *d, byte num_ele);

float timer(void);

byte nat_to_BCD(byte x);
byte BCD_to_nat(byte x);

void main(void)
{
    byte dt[7] = {0x00, 0x59, 0x11, 0x00, 0x28, 0x02, 0x01};
                         // Feb 28, '01  11:59:00
    byte RAM_dat[10];
    byte n;

    float elapsed_time;

    lcd_init();
    _1305_setup_SPI();
    _1305_write_config(0x00);
    _1305_write_date_time(dt);

    for (n = 0; n < 10; n++)        // continually read and display
                                    // date and time
                                    // this section requires nominal 20
                                    // secs to execute
    {
        _1305_read_date_time(dt);
        display_date_time(dt);
        delay_ms(2000);
    }

    lcd_init();                     // illustrate writing to and
                                    // reading from RAM
    printf(lcd_char, "RAM Test");
    delay_ms(1000);

    for (n = 0; n<10; n++)
    {
        RAM_dat[n+0x20] = 0xff - n; // fill the array with some data
    }
    _1305_write_ram(0x00 + 0x20, RAM_dat, 10);
                                // write data to DS1305 RAM

    _1305_read_ram(0x00, RAM_dat, 10);    // read the data back
    display_RAM_data(RAM_dat, 10);

    delay_ms(1000);

    lcd_init();
    while(1)                    // continually display elapsed time
                                // since midnight
    {
         elapsed_time = timer();
         lcd_clr_line(0);
         printf(lcd_char, "ET = %5.0f", elapsed_time);
         delay_ms(1500);
    }
}

void _1305_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 = 0;
    stat_smp = 0;

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

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

    rb0 = 0;
    trisb0 = 0;
}

void _1305_write_config(byte control_byte)
{
      byte dummy;

      rb0 = 1;

      SSPBUF = 0x8f;
      while(!stat_bf)   /* loop */         ;
      dummy = SSPBUF;

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

      rb0 = 0;
}

void _1305_write_date_time(byte *dt)
```

```
{
      byte dummy, n;

      rb0 = 1;

      SSPBUF = 0x80;
      while(!stat_bf)   /* loop */         ;
      dummy = SSPBUF;

      for (n=0; n<7; n++)
      {
         SSPBUF = dt[n];
         while(!stat_bf)            ;
         dummy = SSPBUF;
    }

    rb0 = 0;
}

void _1305_read_date_time(byte *dt)
{
      byte dummy, n;

      rb0 = 1;

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

      for (n=0; n<7; n++)
      {
         SSPBUF = dummy;
         while(!stat_bf)            ;
         dt[n] = SSPBUF;
    }

    rb0 = 0;
}

void _1305_write_ram(byte adr, byte *d, byte num_bytes)
{
      byte dummy, n;

      rb0 = 1;

      SSPBUF = adr + 0x80;
      while(!stat_bf)   /* loop */         ;
      dummy = SSPBUF;

      for (n=0; n<num_bytes; n++)
      {
         SSPBUF = d[n];
         while(!stat_bf)            ;
         dummy = SSPBUF;
      }

      rb0 = 0;
```

```
}

void _1305_read_ram(byte adr, byte *d, byte num_bytes)
{
      byte dummy, n;

      rb0 = 1;

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

      for (n=0; n<num_bytes; n++)
      {
         SSPBUF = dummy;
         while(!stat_bf)            ;
         d[n] = SSPBUF;
      }

      rb0 = 0;
}

void display_date_time(byte *dt)
{
   static byte line = 0;

   lcd_clr_line(line);

   lcd_hex_byte(dt[MONTH]);
   lcd_char('/');
   lcd_hex_byte(dt[DATE]);
   lcd_char('/');
   lcd_hex_byte(dt[YEAR]);

   lcd_char(' ');

   lcd_hex_byte(dt[HOUR]);
   lcd_char(':');
   lcd_hex_byte(dt[MINI]);
   lcd_char(':');
   lcd_hex_byte(dt[SEC]);

   ++line;
   if (line == 4)
   {
      line = 0;
   }
}

void display_RAM_data(byte *d, byte num_ele)
{
      byte line = 0, m, n;
      lcd_clr_line(line);
      for (n = 0, m = 0; n<num_ele; n++, m++)
      {
          if (m==4)
          {
              m=0;
              ++line;
              if (line == 4)
              {
                  line = 0;
              }
              lcd_clr_line(line);
          }

          lcd_hex_byte(d[n]);
          lcd_char(' ');
      }
}

float timer(void)
{
      float elapsed_time;
      byte dt[8];
      _1305_read_date_time(dt);
      elapsed_time = 3600.0 * (float) BCD_to_nat(dt[HOUR])
               + 60.0 * (float) BCD_to_nat(dt[MINI])
               + (float) BCD_to_nat(dt[SEC]);

      return(elapsed_time);
}

byte nat_to_BCD(byte x)
{
      byte h_nib, l_nib;
      h_nib = x/10;
      l_nib = x % 10;
      return ((h_nib << 4) | l_nib);
}

byte BCD_to_nat(byte x)
{
      byte h_nib, l_nib;
      h_nib = x >> 4;
      l_nib = x & 0x0f;
      return(10 * h_nib + l_nib);
}

#include <lcd_out.c>
```

Program AT45_1.c.

Atmel manufactures a number of high density SPI flash memories, up to 8M-bits with two RAM buffers. Of course, as with all relatively new parts, what one sees on a manufacturer's site and what you can actually buy are two different things. I was able to purchase some low end devices (128k bytes with a single RAM buffer) from Arrow for about $4.50 and did not pay an inordinate shipping and handling charge.

As with many new devices, these are not available in DIP packages. Mouser has Aries SOIC to DIP adapters for nominally $5.25. I found the trick to mounting these is to carefully anchor one terminal on each side of the device and then gob solder all over one side at a time and mop it up with soldering wick. I show the results to my students and they are pretty impressed.

Thus, the final device used in prototyping is about $10.00, but for $10.00, it provides many hours of fun. These devices might be used for ordinary data logging or for storing speech. My specific interest, for a Civil Engineering instructor, was to save 256 A/D samples during an impact.

The device I used was an AT45D011 1-megabit devices which is organized as 264 bytes X 512 pages with a single 264 RAM buffer.

The idea is that one can write quickly to the RAM buffer and then command the device to program this to a flash memory page which requires some 7 ms. During this time, a user might be writing to the RAM buffer of another device or use a AT45 device which provides two RAM buffers.

Note that the RAM buffer and each of the flash pages are 264 bytes. The intent in providing eight bytes beyond the usual 256 bytes is to permit the user to attach information relative to the 256 byte data chunk. This might be a time stamp or it might be the location of where the next data chunk is stored.

However, it does lead to a rather interesting addressing scheme as nine bytes are required to specify the address in the RAM buffer or Flash page, and for this device, nine bytes are required to specify identify a Flash page.

Thus, a full address;

```
0000 00PP / PPPP PPPA / AAAA AAAA
```

where P is the page address and A is the address within that page.

Note that not all combinations are used as the maximum value of the address within a page is 263.

Thus, in writing to or reading from the RAM buffer, a command byte is sent, followed by a byte consisting of only the highest bit of the address (0000 000A) and then the low byte of the address followed by the either writing or reading the data.

But, in transferring the RAM buffer to a flash page or the reverse, the page is specified as

```
0000 00PP PPPP PPPX
```

Thus, in writing the RAM buffer to a specific page, note that the page (16 bits) is shifted left and then parsed into two bytes; h and l as shown.

```
page = page << 1;
portd0 = 0;

SSPBUF = BUFF_TO_MEM_PAGE_WITH_ERASE;
while(!stat_bf)         ;
dummy = SSPBUF;

h = (page >> 8) & 0x03;       // highest two bits
SSPBUF = h;
while(!stat_bf)         ;
dummy = SSPBUF;

l = page & 0xff;        // low 7 address bits
SSPBUF = l;
while(!stat_bf)         ;
dummy = SSPBUF;
```

```
    SSPBUF = dummy;
    while(!stat_bf)         ;
    dummy = SSPBUF;

    portd0 = 1;
    delay_ms(20);                      // allow time for programming
```

In reading directly from a specified address on a specified page;

```
    SSPBUF = MEM_PAGE_READ;
    while(!stat_bf)         ;
    dummy = SSPBUF;

    page = page << 1;
    h = (page >> 8) & 0x03;
    l = (page & 0xff) + (adr >> 8);

    SSPBUF = h;
    while(!stat_bf)         ;
    dummy = SSPBUF;

    SSPBUF = l;
    while(!stat_bf)         ;
    dummy = SSPBUF;

    SSPBUF = adr & 0xff;
    while(!stat_bf)         ;
    dummy = SSPBUF;
```

As above, the page is shifted right and parsed into two bytes. However, the least significant bit of the second byte is the high byte of the address and the final byte is the low byte of the address.

The following routine illustrates how to write to the RAM buffer and read the buffer, transfer the buffer to a flash page and a flash page to the RAM buffer and to directly read from flash.

There are a number of additional capabilities which I didn't explore. The Atmel data sheet is quite good and there is also an excellent Application Note titled "Using Atmel's Serial Data Flash"

Note that in configuring the SSP;

```
  ckp = 1;                  // idle state for clock is one
  stat_cke = 1;
  stat_smp = 1;
```

I just kept fooling until it worked and can't say I thoroughly understand this. College professors have this luxury, but my experience in industry was that anything I didn't thoroughly understand had a tendency to blow up in my face, long about the time it would cost $5 million to fix.

```
// Program AT45_1.C
//
// Illustrates an interface with an Ateml AT45D011 512 page X 264 byte
// Flash EEPROM using MSSP in SPI Master Mode.
//
```

```
// Illustrates how to write (and read) a single byte and sequential
// bytes to (and from) the 264 byte RAM buffer.  Transfer of the RAM
// buffer to Flash EEPROM and Flash EEPROM to RAM buffer and direct
// sequential read from Flash EEPROM.
//
//    PIC16F877                          AT45D011
//
// RC5/SDO (term 24) -------------------> SI (term 1) To other SPI
// RC4/SDI (term 23) <------------------- SO (term 8) Devices
// RC3/SCK (term 18) -------------------> SCK (term 2)
// RD0/CS (term 19) --------------------> CE (term 4)
//
// In this example;
//
//    /WP (term 5) and /RESET (term 4) are open (logic one).  The AT45
// device provides internal pull-up resistors on these inputs.
//
// Copyright, Peter H. Anderson, Baltimore, MD, Feb, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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

#define TRUE !0
#define FALSE 0

//ATMEL SerialDataFlash Commands

#define MEM_PAGE_READ 0x52
#define BUFF_READ 0x54
#define BUFF_WRITE 0x84
#define MEM_PAGE_TO_BUFF_TRANSFER 0x53
#define MEM_PAGE_TO_BUFF_COMPARE 0x60
#define BUFF_TO_MEM_PAGE_WITH_ERASE 0x83
#define BUFF_TO_MEM_PAGE_NO_ERASE 0x88
#define MEM_PAGE_ERASE 0x81
#define MEM_BLOCK_ERASE 0x50
#define STATUS_REG 0x57

void AT45_setup_SPI(void);

byte AT45_read_data_flash_status(void);

void AT45_write_buffer_byte(unsigned long adr, byte dat);
byte AT45_read_buffer_byte(unsigned long  adr);

void AT45_write_buffer_sequential(unsigned long adr, byte *d,
                        byte num_bytes);
void AT45_read_buffer_sequential(unsigned long adr, byte *d,
                        byte num_bytes);

void AT45_buffer_to_flash_copy_with_erase(unsigned long page);
void AT45_flash_to_buffer(unsigned long page);
void AT45_read_flash_sequential(unsigned long page, unsigned long adr,
                                    byte *d, byte num_bytes);

void main(void)
{
    byte dat, n;
    byte d[4] = {0xaa, 0xbb, 0xcc, 0xdd};

    lcd_init();
    AT45_setup_SPI();
    pspmode = 0;

    dat = AT45_read_data_flash_status();  // read STATUS
    printf(lcd_char, "Status = %2x", dat);
    delay_ms(2000);

    lcd_init();               // write and read bytes, byte by byte
    printf(lcd_char, "RAM Byte Test");
    delay_ms(1000);

    for (n = 0; n<4; n++)
    {
        dat = 0xaa + n;
        AT45_write_buffer_byte(0x0000+n, dat);
    }

    lcd_clr_line(1);
    for (n = 0; n<4; n++)
    {
        dat = AT45_read_buffer_byte(0x0000+n);
        lcd_hex_byte(dat);
        lcd_char(' ');
    }

    delay_ms(2000);
    lcd_init();

    printf(lcd_char, "RAM Seq Test");     // sequential write and read
    delay_ms(1000);

    AT45_write_buffer_sequential(0x0000, d, 4);

    for (n=0; n<4; n++)
    {
        d[n] = 0;
    }

    AT45_read_buffer_sequential(0x0000, d, 4);
    lcd_clr_line(1);
    for (n = 0; n<4; n++)
    {
        lcd_hex_byte(d[n]);
        lcd_char(' ');
    }

    delay_ms(2000);
```

```
lcd_init();

printf(lcd_char, "Transfer Test");
                    // illustrates transfer of RAM to flash
                    // and flash page to RAM buffer
for (n=0; n<4; n++)
{
     d[n] = 0xb0 + n;
}
AT45_write_buffer_sequential(0x0000, d, 4);
AT45_buffer_to_flash_copy_with_erase(0);    // transfer to page 0

for (n=0; n<4; n++)
{
     d[n] = 0xc0 + n;
}

AT45_write_buffer_sequential(0x0000, d, 4);
AT45_buffer_to_flash_copy_with_erase(511);  // transfer to page 511

AT45_flash_to_buffer(0);                    // page 0 to buffer
AT45_read_buffer_sequential(0x0000, d, 4);  // read and display
lcd_clr_line(1);
for (n = 0; n<4; n++)
{
    lcd_hex_byte(d[n]);
    lcd_char(' ');
}

AT45_flash_to_buffer(511);                  // page 511 to buffer
AT45_read_buffer_sequential(0x0000, d, 4);  // read and display
lcd_clr_line(2);
for (n = 0; n<4; n++)
{
     lcd_hex_byte(d[n]);
     lcd_char(' ');
}

delay_ms(2000);

lcd_init();
printf(lcd_char, "Read from Flash");

AT45_read_flash_sequential(0, 0x0000, d, 4);
lcd_clr_line(1);
for (n = 0; n<4; n++)
{
     lcd_hex_byte(d[n]);
     lcd_char(' ');
}

AT45_read_flash_sequential(511, 0x0000, d, 4);
lcd_clr_line(2);
for (n = 0; n<4; n++)
{
     lcd_hex_byte(d[n]);
     lcd_char(' ');
    }

    delay_ms(2000);

    lcd_init();
    printf(lcd_char, "Done");

    while(1)            ;     // continual loop
}

void AT45_setup_SPI(void)
{
    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 one
    stat_cke = 1;
    stat_smp = 1;

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

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

    portd0 = 0;
    trisd0 = 0;
}

byte AT45_read_data_flash_status(void)
{
    byte d, dummy;

    portd0 = 0;

    SSPBUF = STATUS_REG;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = dummy;
    while(!stat_bf)           ;
    d = SSPBUF;

    portd0 = 1;

    return(d);
}

void AT45_write_buffer_byte(unsigned long adr, byte dat)
{
    byte dummy;

    portd0 = 0;
    SSPBUF = BUFF_WRITE;
    while(!stat_bf)           ;
    dummy = SSPBUF;
    SSPBUF = dummy;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = adr >> 8;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = adr & 0xff;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = dat;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    portd0 = 1;
}

byte AT45_read_buffer_byte(unsigned long adr)
{
    byte dummy, dat;

    portd0 = 0;
    SSPBUF = BUFF_READ;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = dummy;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = adr >> 8;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = adr & 0xff;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = dummy;                 // dummy byte
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = dummy;
    while(!stat_bf)           ;
    dat = SSPBUF;

    portd0 = 1;

    return(dat);
}

void AT45_write_buffer_sequential(unsigned long adr, byte *d, byte num_bytes)
{
    byte dummy, n;
    portd0 = 0;
    SSPBUF = BUFF_WRITE;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = dummy;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = adr >> 8;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = adr & 0xff;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    for (n=0; n<num_bytes; n++)
    {
         SSPBUF = d[n];
         while(!stat_bf)            ;
         dummy = SSPBUF;
    }

    portd0 = 1;

}

void AT45_read_buffer_sequential(unsigned long  adr, byte *d,
                                    byte num_bytes)
{
    byte dummy, n;

    portd0 = 0;
    SSPBUF = BUFF_READ;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = dummy;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = adr >> 8;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = adr & 0xff;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = dummy;                 // dummy byte
    while(!stat_bf)           ;
    dummy = SSPBUF;

    for (n=0; n<num_bytes; n++)
    {
        SSPBUF = dummy;
        while(!stat_bf)        ;
        d[n] = SSPBUF;
    }
    portd0 = 1;
}

void AT45_buffer_to_flash_copy_with_erase(unsigned long page)
{
    byte dummy, h, l;

    page = page << 1;
    portd0 = 0;

    SSPBUF = BUFF_TO_MEM_PAGE_WITH_ERASE;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    h = (page >> 8) & 0x03;   // highest two bits
    SSPBUF = h;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    l = page & 0xff;          // low 7 address bits
    SSPBUF = l;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = dummy;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    portd0 = 1;
    delay_ms(20);                   // allow time for programming
}

void AT45_flash_to_buffer(unsigned long page)
{
    byte dummy, h, l;

    portd0 = 0;

    SSPBUF = MEM_PAGE_TO_BUFF_TRANSFER;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    page = page << 1;

    h = (page >> 8) & 0x03;   // highest two bits
    SSPBUF = h;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    l = page & 0xff;          // low 7 address bits
    SSPBUF = l;
    while(!stat_bf)           ;
    dummy = SSPBUF;
    SSPBUF = dummy;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    portd0 = 1;
    delay_ms(1);
}

void AT45_read_flash_sequential(unsigned long page, unsigned long adr,
                              byte *d, byte num_bytes)
{
    byte dummy, h, l, n;

    portd0 = 0;

    SSPBUF = MEM_PAGE_READ;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    page = page << 1;
    h = (page >> 8) & 0x03;
    l = (page & 0xff) + (adr >> 8);

    SSPBUF = h;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = l;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    SSPBUF = adr & 0xff;
    while(!stat_bf)           ;
    dummy = SSPBUF;

    for (n=0; n<4; n++)       // 32 don't care bits
    {
        SSPBUF = dummy;
        while(!stat_bf)       ;
        dummy = SSPBUF;
    }

    for (n=0; n<num_bytes; n++)
    {
        SSPBUF = dummy;
        while(!stat_bf)       ;
        d[n] = SSPBUF;
    }

    portd0 = 1;
}

#include <lcd_out.c>
```

Philips Inter IC Protocol (I2C) – Master Mode.

This section presents a brief overview of the Philips I2C Master protocol in the context of context of manipulating any two general purpose I/O pins on a PIC. I have termed this “bit bang”.

Use of the PICs SSP module in the I2C Master Mode is also presented. Routines for interfacing with a Microchip 24LC256 EEPROM, Maxim MAX518 2-channel D/A, Philips PCF8574 8-bit I/O expander, Dallas DS1803 dual potentiometer, Dallas DS1624 digital thermometer and Dallas DS1307 RTC and Philips PCF8583 are presented.

In my mind, there is precious little advantage to using the SSP module to implement an I2C master or for that matter an SSP master. If the SSP is available, I am inclined to use it, but if my final platform doesn’t have an SSP or the SSP is being used for another application, “bit banging” is okay. I note this as I see people overlooking inexpensive PICs feeling they absolutely need an SSP module or if they have both SPI and I2C devices, they are looking for a PIC with two SSP modules. My comment is not applicable when working with a PIC as a slave I2C or SPI device. The SSP becomes mighty important when operating in the slave mode.

The I2C protocol uses two leads; data (SDA) and clock (SCL). The clock is controlled by the master. The SDA lead is bi-directional; commands and data are sent to the slave and data is received from the slave.

The logic states of each of these leads are near ground (logic zero) and high impedance (logic one). External pull-up resistors, nominally 4.7K, are used such that when the output is in a high Z state (logic one), the slave sees +5VDC through the 4.7K pull-up resistor.

Implementations for bringing these leads to high and low logic states;

```
void i2c_high_sda(void)
{
   // bring SDA to high impedance
   SDA_DIR = 1;
   // delay_10us(5);
}

void i2c_low_sda(void)
{
   SDA_PIN = 0;
   SDA_DIR = 0;  // output a hard logic zero
   // delay_10us(5);
}

void i2c_high_scl(void)
{
   SCL_DIR = 1;   // high impedance
   // delay_10us(5);
}

void i2c_low_scl(void)
{
   SCL_PIN = 0;
   SCL_DIR = 0;
   // delay_10us(5);
 }
```

When idle, the master maintains both the SDA and SCL leads in a high impedance state (logic one).

```
   void i2c_setup_bb(void)
   {
      i2c_high_sda();
      i2c_high_scl();
   }
```

The master calls all devices to “listen up” by send a “start” sequence which is bringing SDA low while SCL is high.

```
   void i2c_start_bb(void)
   {
      i2c_low_scl();    // See note
      i2c_high_sda();   // See note
      i2c_high_scl();   // bring SDA low while SCL is high
      i2c_low_sda();
      i2c_low_scl();
   }
```

Note that the first three instructions are not really required if both SDA and SCL are already high.

After, sending the “start” sequence, all slave devices are listening and the master sends an address byte of the form;

```
      AAAA AAAB
```

Where AAA AAA is a unique seven bit address and bit B indicates whether this is a “write” (logic 0) or “read” (logic 1) sequence.

The highest four bits are assigned by the manufacturer under the authority of Philips. Thus, to my knowledge, all EEPROM devices are use the address 1010. The lower three address bits may either be assigned by the manufacturer or the user may configure at least some of the bits by strapping terminals on the device.

For example, the 24LC256 EEPROM provides three terminals, A2, A1 and A0. If these are strapped as 101, the full 7-bit address for this EEPROM is 1010101 which is then followed by a 0 or 1 depending on whether the sequence is a write or read sequence. Thus, one can configure up to eight 24LC256 (or 1010) devices on the same bus, each strapped for a different 3-bit address.

However, it is nonsensical to have as many as eight real time clocks on the same bus and in the interest of limiting the terminal count, the manufacturer may implement one or more of these lower three address bits in the IC and leave the user with only one or two bits which are assigned by strapping. For example, with the MAX518 Dual D/A, the family code is 0101. However, Maxim opted to also assign one of the lower three bits as “1” and leave the user with two bits which are defined by strapping terminals AD1 and AD0. Thus, the full 7-bit address for a MAX 518 is 0101 1 AD1 AD0.

(Note that some EEPROM devices violate this concept of the lower three address bits identifying a specific device on the I2C bus. With the Microchip 24LC16 EEPROM these bits are used to identify the page in the 24LC16. This is a bit confusing as Microchip has actually brought out leads identified as A2, A1 and A0, but these are not used. Thus, only one 24LC16 may be used on a bus and no other devices having a 1010 family code can be used on the same bus).

All bytes, including this address byte, are sent to the slave starting with the most significant bit. For each bit, SDA is brought to the appropriate state and SCL is then brought high and then low. After sending the byte, the master then outputs a single clock pulse with SDA in a high impedance state to provide the slave the opportunity to acknowledge receipt of the byte.

```
   void i2c_out_byte_bb(byte o_byte)
   {
      byte n;
      for(n=0; n<8; n++)
      {
         if(o_byte&0x80)      // test most significant bit
         {
            i2c_high_sda();   // set up SDA
         }
         else
         {
            i2c_low_sda();
         }
         i2c_high_scl();      // and clock out the data
         i2c_low_scl();
         o_byte = o_byte << 1;
      }
      i2c_high_sda();

      i2c_high_scl();         // nack
      i2c_low_scl();
   }
```

Important Note.

Note that this implementation of outputting a byte differs from that used in my book "PIC C Routines" and routines which are presented on my web site in that the above incorporates the nack pulse into the out_byte routine. The reason for this modification is to agree with the implementation when using the SSP module.

That is, in my "PIC C" book;

```
      i2c_out_byte(x);
      i2c_nack();
```

In this discussion, this is simply;

```
      i2c_out_byte_bb(x);
```

A data byte is fetched by the Master, by bringing SCL high and reading the state of SDA and then bringing SCL low. This is repeated for each of the eight bits, beginning with the most significant bit.

On receipt of a byte from a slave, the Master provides an additional clock pulse with SDA at a logic zero to acknowledge the receipt of the byte. However, if this is the last byte prior to terminating the sequence with a "stop", this additional clock pulse is sent with SDA at a high impedance.

```
byte i2c_in_byte_bb(byte ack)
{
   byte i_byte, n;
   i2c_high_sda();
   for (n=0; n<8; n++)
   {
      i2c_high_scl();

      if (SDA_PIN)
      {
         i_byte = (i_byte << 1) | 0x01; // msbit first
      }
      else
      {
         i_byte = i_byte << 1;
      }
      i2c_low_scl();
   }

   if (ack) // if ack is desired, bring SDA low for a clock pulse
   {
         i2c_low_sda();
   }
   else
   {
         i2c_high_sda();// not really necessary as it is already high
   }

   i2c_high_scl();            // clock for ack or nack
   i2c_low_scl();

   i2c_high_sda();            // be sure to leave routine with SDA high

   return(i_byte);
}
```

Note that in this implementation, variable "ack" is passed to the function to indicate whether the additional clock pulse is to be a zero (ack = TRUE) or high impedance (ack = FALSE).

Important Note.

Here again, this differs from the approach used in my "PIC C" book in that in the above, the sending of the ACK (zero) or NACK (high Z) has been incorporated into the i2c_in_byte_bb() function.

Data interchange with a specific device is terminated with a "stop" sequence which is implemented by bringing SDA high while clock is high. Note that this returns the bus to the idle state (both SCL and SDA in a high impedance state).

```
void i2c_stop_bb(void)
{
   i2c_low_scl();
   i2c_low_sda();
   i2c_high_scl();
   i2c_high_sda();  // bring SDA high while SCL is high
   // idle is SDA high and SCL high
}
```

I2C_BB.C. (Bit Bang Routines).

All of the above routines are implemented in file I2C_BB.C and thus the header file (I2C_BB.H) and the implementations (I2C_BB.C) may be included in the main file in the same manner as the LCD routines. As all of these routines have been discussed above, the file I2C_BB.C is not included in this narrative.

However, the header file might be useful as a memory jogger;

```
// I2C_BB.H
//
// Header file for I2C bit bang routines.
//
// copyright, Peter H. Anderson, Baltimore, MD, Feb, '01

void i2c_setup_bb(void);
byte i2c_in_byte_bb(byte ack);
void i2c_out_byte_bb(byte o_byte);
void i2c_start_bb(void);
void i2c_stop_bb(void);

void i2c_high_sda(void);
void i2c_low_sda(void);
void i2c_high_scl(void);
void i2c_low_scl(void);
```

Note that when using i2c_bb.c, SDA_PIN, SCL_PIN, SDA_DIR and SCL_DIR must be defined in the main routine.

Program 24_256_1.c.

This routine illustrates how to interface with the Microchip 24LC256 32K X 8 EEPROM. This program may be used with other EEPROMs as well. The only difference is the range of valid addresses.

Device	Size (Bytes)	Address Range
24X32	4096	0x0000 – 0x0fff
24X64	8192	0x0000 – 0x1fff
24X128	16384	0x0000 – 0x3fff
24X256	32768	0x0000 – 0x7fff

Note that in this routine, files i2c_bb.h and i2c_bb.c are included.

In function random_write, the sequence begins with the “start” followed by the I2C address byte with the W/R bit at zero (write), followed by the high and low memory address bytes followed by the data to be written, followed by a “stop”. A delay is provided to permit the data to be burned to EEPROM.

```
void random_write(byte dev_adr, unsigned long mem_adr, byte dat)
{
   i2c_start_bb();
   i2c_out_byte_bb(0xa0 | (dev_adr << 1));
   i2c_out_byte_bb((mem_adr >> 8) & 0xff);
   i2c_out_byte_bb(mem_adr & 0xff);
   i2c_out_byte_bb(dat);
   i2c_stop_bb();
   delay_ms(25); // allow for the programming of the eeprom
}
```

In function random_read, the “start”, followed by the I2C address byte with the W/R bit at zero (write), followed by the high and low memory address bytes. The idea of a “write”, when the idea is to read, bothered me when I first used this device, but note that the write indicates that the next byte or bytes are being written, while a “read” is an invitation for the slave to transmit beginning with the next byte.

This is followed by another “start” with no intermediate “stop”, commonly termed a “repeated start” and the I2C address byte with the W/R bit at 1 (read). The data is read followed by the “stop”. Note that the master does not provide an ACK clock pulse after reading the byte to signal the slave that this is the last byte to be read prior to the “stop”.

```
byte random_read(byte dev_adr, unsigned long mem_adr)
{
   byte y;
   i2c_start_bb();
   i2c_out_byte_bb(0xa0 | (dev_adr << 1));
   i2c_out_byte_bb((mem_adr >> 8) & 0xff);
   i2c_out_byte_bb(mem_adr & 0xff);

   i2c_start_bb();
   i2c_out_byte_bb(0xa1 | (dev_adr << 1));
   y=i2c_in_byte_bb(FALSE);   // no ack prior to stop
   i2c_stop_bb();
   return(y);
}
```

The program writes 16 values to EEPROM beginning at memory location 0x700 and then reads these back and displays them on the LCD.

The program also illustrates sequential writes and reads which are much the same. In writing, each data byte is output one after the other and terminated with the “stop” and a delay for programming to EEPROM. In reading, each byte is read from the slave and is acknowledged by the master, except for the last byte as shown;

```
   for(n=0; n<num_vals; n++)
   {
      if(n!=(num_vals-1))
      {
         d[n] = i2c_in_byte_bb(TRUE);     // ack after each byte
      }
      else
      {
         d[n] = i2c_in_byte_bb(FALSE);    // except the last byte
      }

   }
   i2c_stop_bb();
```

In this routine, a “dummy” routine to simulate the results of a measurement sequence is used to generate data values. Note the use of a static, which in this case is initialized to 0 only on the first call to the function. The variable is updated (by adding 2) prior to leaving the routine and this is the value of the variable when the function is again called.

```
void make_meas_seq(byte *d)   // generates four values on each call
{
   static byte n=0;
   d[0]=0xf0+n;
   d[1]=0xa0+n;
   d[2]=0x80+n;
   d[3]=0x40+n;
   n+=2;
}
```

```
// Program 24_256_1.C
//
// Illustrates how to write a byte to an address and read a byte from
// an address.  The I2C interface is implemented using "bit bang"
// routines.
//
// Program writes the 16 values 0xff, 0xfe, etc to locations beginning
// at memory adr 0x0700.  Reads them back and displays on LCD.
//
// Also illustrates sequential write and read.
//
//    PIC16F877                    24LC256
//
//    RB1 (term 33)---------------- SCL (term 6) ----- To Other
//    RB2 (term 34) --------------- SDA (term 5) ----- I2C Devices
//
// Note that the slave address is determined by A2 (term 3), A1
// (term 2) and A0 (term 1) on the 24LC256.  The above SCL and SDA
// leads may be multipled to eight group "1010" devices, each strapped
// for a unique A2 A1 A0 setting.
//
// 4.7K pullup resistors to +5VDC are required on both signal leads.
//
// copyright, Peter H. Anderson, Baltimore, MD, Feb, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <i2c_bb.h>

#define TRUE !0
#define FALSE 0

#define SDA_DIR trisb2
#define SCL_DIR trisb1

#define SDA_PIN rb2
#define SCL_PIN rb1

// routines used for 24LC256 byte write and read
void random_write(byte dev_adr, unsigned long mem_adr, byte dat);
byte random_read(byte dev_adr, unsigned long mem_adr);

// routines used for 24LC256 seq byte write and read
void make_meas_seq(byte *d);
void seq_write(byte dev_adr, unsigned long mem_adr, byte *d, byte num_vals);
void seq_read(byte dev_adr, unsigned long mem_adr, byte *d, byte num_vals);

void main(void)
{
   unsigned long mem_adr;
   byte dat, m, n, line;
   byte d[4];
```

```
lcd_init();
i2c_setup_bb();

// illustrates byte write and byte read
printf(lcd_char, "Byte Write Demo");
delay_ms(2000);

lcd_init();

mem_adr=0x0700;
for(n=0; n<16; n++)
{
   lcd_char('!');          // to indicate something is going on
   dat = 0xff-n;
   random_write(0x00, mem_adr, dat);
   ++mem_adr;
}

line = 0;
lcd_clr_line(line);
mem_adr=0x0700;

for(n=0, m=0; n<16; n++, m++)
{
   if (m==4)
   {
      m = 0;
      ++line;
      lcd_clr_line(line);
   }
   dat = random_read(0x00, mem_adr);
   lcd_hex_byte(dat);
   lcd_char(' ');
   ++mem_adr;
}
delay_ms(2000);

// illustrates sequential write and read
lcd_init();
printf(lcd_char, "Seq Byte Demo");
delay_ms(2000);

mem_adr = 0x0700;          // write the data
for(n=0; n<3; n++)         // three chuncks of 4 data bytes
{
    make_meas_seq(d);
    seq_write(0x00, mem_adr, d, 4);
    mem_adr +=4;
    lcd_char('!');         // to show something is happening
}

delay_ms(1000);

lcd_init();

mem_adr=0x0700;                  // now read it back
for(n=0; n<3; n++)
```

```
   {
       seq_read(0x00, mem_adr, d, 4);

       lcd_clr_line(n);       // and display it on the LCD
       for(m=0; m<4; m++)
       {
           lcd_hex_byte(d[m]);
           lcd_char(' ');
       }
       mem_adr +=4;
    }

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

void random_write(byte dev_adr, unsigned long mem_adr, byte dat)
{
   i2c_start_bb();
   i2c_out_byte_bb(0xa0 | (dev_adr << 1));
   i2c_out_byte_bb((mem_adr >> 8) & 0xff);
   i2c_out_byte_bb(mem_adr & 0xff);
   i2c_out_byte_bb(dat);
   i2c_stop_bb();
   delay_ms(25); // allow for the programming of the eeprom
}

byte random_read(byte dev_adr, unsigned long mem_adr)
{
   byte y;
   i2c_start_bb();
   i2c_out_byte_bb(0xa0 | (dev_adr << 1));
   i2c_out_byte_bb((mem_adr >> 8) & 0xff);
   i2c_out_byte_bb(mem_adr & 0xff);

   i2c_start_bb();
   i2c_out_byte_bb(0xa1 | (dev_adr << 1));
   y=i2c_in_byte_bb(FALSE);   // no ack prior to stop
   i2c_stop_bb();
   return(y);
}

void make_meas_seq(byte *d)   // generates four values on each call
{
   static byte n=0;
   d[0]=0xf0+n;
   d[1]=0xa0+n;
   d[2]=0x80+n;
   d[3]=0x40+n;
   n+=2;
}

void seq_write(byte dev_adr, unsigned long mem_adr, byte *d, byte num_vals)
{
   byte n;
   i2c_start_bb();
   i2c_out_byte_bb(0xa0 | (dev_adr << 1));
   i2c_out_byte_bb((mem_adr >> 8) & 0xff);
   i2c_out_byte_bb(mem_adr & 0xff);

   for (n=0; n<num_vals; n++)
   {
      i2c_out_byte_bb(d[n]);
   }
   i2c_stop_bb();
   delay_ms(25);
}

void seq_read(byte dev_adr, unsigned long mem_adr, byte *d, byte num_vals)
{
   byte n;

   i2c_start_bb();
   i2c_out_byte_bb(0xa0 | (dev_adr << 1));
   i2c_out_byte_bb((mem_adr >> 8) & 0xff);
   i2c_out_byte_bb(mem_adr & 0xff);

   i2c_start_bb();                  // repeated start
   i2c_out_byte_bb(0xa1 | (dev_adr << 1));

   for(n=0; n<num_vals; n++)
   {
      if(n!=(num_vals-1))
      {
             d[n] = i2c_in_byte_bb(TRUE); // ack after each byte
      }
      else
      {
             d[n] = i2c_in_byte_bb(FALSE);      // its the last byte
        }

   }
   i2c_stop_bb();
}

#include <lcd_out.c>
#include <i2c_bb.c>
```

Use of the SSP Module as an I2C Master.

I2C_MSTR.C includes routines which are functionally similar to those in I2C_BB.C except they use the PIC's SSP module in the I2C Master mode. The operation of the SSP module is discussed in Section 9 of the PIC16F87X Data Sheet.

Note that when using the SSP, the SCL and SDA terminals correspond to PORTC3 and PORTC4.

In function i2c_master_setup(), the SCL and SDA terminals are configured in a high impedance state. The "cke" bit in register SSPSTAT is set to a zero to such that the levels conform to I2C specifications. (I am out of my league here!).

The clock speed in controlled by the setting of register SSPADD. I opted for 250 kHz, but as I recall, even the slowest I2C devices are rated at 400 kHz. Note that the value of SSPADD will vary depending on the PIC's clock. For example, if the PIC is running at 20 MHz, f_ocs/4 = 5.0 MHz, and an SSPADD value of 12 will provide an I2C clock of 384 kHz.

Bits sspm3::sspm0 are set to 1 0 0 0 to configure the SSP module in the I2C Master mode and the module is enabled by setting sspen.

I found that a review of the various status bits in register SSPCON2 was helpful in understanding the operation of the I2C master.

In initiating a “start” sequence, bit “sen” is set and the program loops until this bit goes to zero. Similarly, in initiating a “stop”, bit “pen” is enabled and the program loops until this bit goes to zero.

A byte is output by loading the SSPBUFF and then waiting for bit “stat_bf” to go to zero. The program then loops until the slave acknowledges (bit ackstat at zero).

In receiving a byte, bit, bit “rcen” is set and the program loops until this bit goes to a zero. Bit “ackdt” is set to 0 or 1 to either acknowledge or not acknowledge upon receipt of the byte and bit “acken” is set and the program loops until this bit goes to zero. The received byte is fetched from SSPBUF.

Note that in developing these routines I sought to avoid error conditions from bringing the PIC to a grinding halt by setting a counter to 50 and decrementing each time the status bit was sampled.

```
void i2c_master_out_byte(byte o_byte)
{
   byte n=50;
   SSPBUF = o_byte;
   while(stat_bf && --n)      ;

   n=50;
   while(ackstat && --n)      ;
}
```

In the above, if there were no device attached to the PIC, the slave device would never acknowledge and without the counter, the program would loop indefinitely.

However, I decided not to complicate this discussion by identifying the error to the calling process. This might be done by returning an error code.

```
byte i2c_master_in_byte(byte ack, byte *p_ibyte)
{
   byte n=50;
   rcen = 1;
   delay_ms(1);
   while(rcen & --n)  ;
   if (n==0)
   {
      return(ERROR_RCEN);
   }

   n= 50;
   ackdt = ack ? 0 : 1;       // nack or ack
   acken = 1;
   while(acken & --n)   ;
   if (n == 0)
   {
      return(ERROR_ACKEN)
   }
   *p_ibyte = SSPBUF
   return(SUCCESS);
}
```

Note that the received byte is passed by reference in the above example.

As I say, I abandoned the idea of dealing with errors in the interest of clarity and limited the implementation to one of avoiding "hanging".

In function i2c_master_in_byte, I used the dreaded "? :" construct;

```
   ackdt = ack ? 0 : 1;       // nack or ack
```

This is simply a shorthand for;

```
if (ack)
{
   ackdt = 0;
}
else
{
   ackdt = 1;
}
```

Note that I2C_MSTR.C includes a special routine for a "repeated start".   Use of this is illustrated in routine 24_256_2.C below.

Routines I2C_MSTR.H and I2C_MSTR.C are to be #included in the main routine.

```
// I2C_MSTR.C
//
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '01

void i2c_master_setup(void)
{
   trisc3 = 1;    // scl - term 18
   trisc4 = 1;    // sda - term 23
   stat_smp = 1;
   stat_cke = 0;  // input levels conform to i2c

   SSPADD = 3;
        //  1.0 MHz / (SSPADD + 1)  - 250 kHz

   sspm3 = 1;     // i2c master mode
   sspm2 = 0;
   sspm1 = 0;
   sspm0 = 0;

   sspen = 1;     // enable ssp
}

void i2c_master_start(void)
```

```
{
   byte n=50;
   sen = 1;
   while(sen && --n)          ;     // n used to avoid infinite loop
}

void i2c_master_repeated_start(void)
{
   byte n=50;
   rsen = 1;
   while(rsen && --n)         ;
}

void i2c_master_stop(void)
{
   byte n=50;
   pen = 1;
   while(pen && --n)          ;
}

void i2c_master_out_byte(byte o_byte)
{
   byte n=50;
   SSPBUF = o_byte;
   while(stat_bf && --n)      ;

   n=50;
   while(ackstat && --n)      ;
}

byte i2c_master_in_byte(byte ack)
{
   byte n=50;
   rcen = 1;
   delay_ms(1);
   while(rcen & --n)  ;

   n= 50;
   ackdt = ack ? 0 : 1;       // nack or ack
   acken = 1;
   while(acken & --n)   ;

   return(SSPBUF);
}
```

Program 24_256_2.C.

This program is functionally the same as 24_256_1.c except that it uses the PIC’s SSP module.

When reading data from the EEPROM, note that the sequence begins with the I2C address byte with the R/W bit set to zero (write), followed by the high and low memory address bits.

However, unlike the “bit bang” implementation, a “repeated start’ is then sent rather than another “start”. I am uncertain I clearly understand just why the PIC needs to know the difference between a “start” and a “repeated start” , but when things work as specified in the manual, why argue the point.

```
byte random_read(byte dev_adr, int mem_adr)
{
   byte y;
   i2c_master_start();
   i2c_master_out_byte(0xa0 | (dev_adr << 1));
   i2c_master_out_byte((mem_adr >> 8) & 0xff);
   i2c_master_out_byte(mem_adr & 0xff);

   i2c_master_repeated_start();     // no intermediate stop
   i2c_master_out_byte(0xa1 | (dev_adr << 1));  // read operation
   y=i2c_master_in_byte(1);
   i2c_master_stop();
   return(y);
}


// 24_256_2.C
//
// Interface with 24LC256 using MSSP Module in I2C Master Mode
//
// This routine is functionally identical to 24_256_1.c.
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <i2c_mstr.h>

#define TRUE !0
#define FALSE 0

void random_write(byte dev_adr, int mem_adr, byte dat);
byte random_read(byte dev_adr, int mem_adr);

// routines used for 24LC256
void make_meas_seq(byte *d);
void seq_write(byte dev_adr, unsigned long mem_adr, byte *d, byte num_vals);
void seq_read(byte dev_adr, unsigned long mem_adr, byte *d, byte num_vals);

void main(void)
{
   long mem_adr;
   byte dat, m, n, line;
   byte d[4];

   i2c_master_setup();

   lcd_init();
   printf(lcd_char, "Byte Demo");
   delay_ms(2000);

   lcd_clr_line(0);
   mem_adr=0x0700;
   for(n=0; n<16; n++)  // write some data to the EEPROM
   {
      dat = 0xff - n;
      random_write(0x00, mem_adr, dat);
      ++mem_adr;
      lcd_char('!');           // to show something is going on
   }

   // now, read the data back and display
   lcd_init();

   mem_adr=0x0700;
   for(n=0, m=0, line = 0; n<16; n++, m++)
   {
      if (m==4)
      {
          m = 0;
          ++line;
          lcd_clr_line(line);
      }
      dat = random_read(0x00, mem_adr);
      lcd_hex_byte(dat);
      lcd_char(' ');
      ++mem_adr;
   }
   delay_ms(2000);

   lcd_init();
   printf(lcd_char, "Seq Byte Demo");
   delay_ms(2000);

   lcd_init();

   mem_adr = 0x0700;           // write the data
   for(n=0; n<3; n++)          // three chunks of 4 data bytes
   {
       make_meas_seq(d);
       seq_write(0x00, mem_adr, d, 4);
       mem_adr +=4;
   }

   mem_adr=0x0700;                   // now read it back

   for(n=0; n<3; n++)
   {
      seq_read(0x00, mem_adr, d, 4);
      lcd_clr_line(n);
      for(m=0; m<4; m++)
      {
         lcd_hex_byte(d[m]);
         lcd_char(' ');
      }
      mem_adr +=4;
   }

   while(1)             ;
}
```

```
void random_write(byte dev_adr, int mem_adr, byte dat)
{
   i2c_master_start();
   i2c_master_out_byte(0xa0 | (dev_adr << 1));
   i2c_master_out_byte((mem_adr >> 8) & 0xff);
                  // high byte of memory address
   i2c_master_out_byte(mem_adr & 0xff);   // low byte of mem address
   i2c_master_out_byte(dat);              // and finally the data
   i2c_master_stop();
   delay_ms(25); // allow for the programming of the eeprom
}

byte random_read(byte dev_adr, int mem_adr)
{
   byte y;
   i2c_master_start();
   i2c_master_out_byte(0xa0 | (dev_adr << 1));
   i2c_master_out_byte((mem_adr >> 8) & 0xff);
   i2c_master_out_byte(mem_adr & 0xff);

   i2c_master_repeated_start();     // no intermediate stop
   i2c_master_out_byte(0xa1 | (dev_adr << 1));  // read operation
   y=i2c_master_in_byte(1);
   i2c_master_stop();
   return(y);
}

void make_meas_seq(byte *d)
{
   static byte n = 0;
   d[0]=0xf0+n;
   d[1]=0xa0+n;
   d[2]=0x80+n;
   d[3]=0x40+n;
   n+=2;
}

void seq_write(byte dev_adr, unsigned long mem_adr, byte *d, byte num_vals)
{
   byte n;
   i2c_master_start();
   i2c_master_out_byte(0xa0 | (dev_adr << 1));
   i2c_master_out_byte((mem_adr >> 8) & 0xff);
   i2c_master_out_byte(mem_adr & 0xff);
   for (n=0; n<num_vals; n++)
   {
      i2c_master_out_byte(d[n]);
   }
   i2c_master_stop();
   delay_ms(10);
}

void seq_read(byte dev_adr, unsigned long mem_adr, byte *d, byte num_vals)
{
   byte n;
   i2c_master_start();
   i2c_master_out_byte(0xa0 | (dev_adr << 1));
   i2c_master_out_byte((mem_adr >> 8) & 0xff);
   i2c_master_out_byte(mem_adr & 0xff);

   i2c_master_repeated_start();
   i2c_master_out_byte(0xa1 | (dev_adr << 1));

   for(n=0; n<num_vals; n++)
   {
      if(n!=(num_vals-1))
      {
          d[n] = i2c_master_in_byte(1);
      }
      else
      {
          d[n] = i2c_master_in_byte(0);
        }
   }
   i2c_master_stop();
}

#include <lcd_out.c>
#include <i2c_mstr.c>
```

Program MAX518_1.C.

The Maxim MAX518 is a dual 8-bit D/A. I have used this in applications using a 200 mV panel meter with external scaling resistors to display quantities. Another application is in conjunction with a Texas Instruments DRV101T 20KHz PWM Driver where the duty cycle is controlled by a voltage or resistance.   The DRV101T is available from Digikey.

A D/A sequence consists of a “start”, followed by the I2C address byte with the R/W bit set to zero (write) , followed by a command byte, followed by the D/A value, followed by the “stop”. The D/A to be controlled is specified in the command byte as either 0 or 1.

In addition, both D/A’s may be reset by simply sending the I2C address byte (write mode) followed by a command byte with a one in the bit 4 position (0x10). The device may be powered down by sending a command byte with a one in the bit 3 position (0x08).

In this routine, one output channel is set to a constant and a crude triangular wave is output on the other channel. After about 15 seconds, the device is powered down.

```
// Program MAX518_1.C,
//
// Illustrates an interface with a MAX518 Dual D/A.  This routine
// uses the MSSP Module as an I2C Master.
//
// Program outputs a constant 0x80 on D/A 0. The result is nominally
// 2.5V on OUT0.  In addition, outputs a triangular wave on D/A1.
//
//    PIC16F877                     MAX518
//
// SCL/RC3 (term 18)----- SCL (term 3) ----- To Other
// SDA/RC4 (term 23)----- SDA (term 4) ----- I2C Devices
//
```

```
// Note that the slave address is 0101 1 AD1 AD0 where AD1 and AD0
// correspond to terminals 5 and 6, respectively.  In this program
// they are strapped to ground and thus the slave address is 0x58.
//
// External pull-up resistors to +5VDC are not required on the SDA
// and SCL signal leads.  However, they may be present to maintain
// compatibility with other I2C devices.
//
// In command byte;
//           RST (bit 4) set to 1 resets both D/As
//           PD (bit 3) set to one for power down
//           A0 (bit 0) identifies whether data is for D/A0 or D/A1
//
// copyright, Peter H. Anderson, Baltimore, MD, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <i2c_mstr.h>

#define TRUE !0
#define FALSE 0

void max518_d_a(byte dev_adr, byte d_a, byte dat);
// dev_adr is  AD1, AD0 strapping, d_a is either 0 or 1, dat is data
// to be output
void max518_power_down(byte dev_adr);


void main(void)
{
   byte i=0;
   byte const triangular[16]=
          {0x00, 0x20, 0x40, 0x60, 0x80, 0xa0, 0xc0, 0xe0,
           0xe0, 0xc0, 0xa0, 0x80, 0x60, 0x40, 0x20, 0x00};

   unsigned long j=5000;

   lcd_init();          // used for possible debugging
   i2c_master_setup();
   max518_d_a(0x00, 0, 0x80); // constant on A/D0
   while(j) // loop 5000 times
   {

      max518_d_a(0x00, 1, triangular[i]);

      ++i;
      if (i>15)
      {
         i=0;
      }
      --j;
      delay_ms(3);      // 3 ms * 15,000 = 15 seconds
   }
   max518_power_down(0);   // and then power down
   while(1)  /* endless loop */     ;
}

void max518_d_a(byte dev_adr, byte d_a, byte dat)
// dev_adr is  AD1, AD0 strapping, d_a is either 0 or 1, dat is data
// to be output
{
   i2c_master_start();
   i2c_master_out_byte(0x58 | (dev_adr<<1));    // address the device
   i2c_master_out_byte(d_a);  // selects D/A
   i2c_master_out_byte(dat);  // d/a data
   i2c_master_stop();
}

void max518_power_down(byte dev_adr)
{
   i2c_master_start();
   i2c_master_out_byte(0x58 | (dev_adr<<1));    // address the device
   i2c_master_out_byte(0x08);       // sets PD bit of command register
   i2c_master_stop();
}

#include <lcd_out.c>
#include <i2c_mstr.c>
```

Program DS1803_1.C.

The Dallas DS1803 is a dual 256 position potentiometer. It is available in 10K, 50K and 100K versions.

The DS1803 does not include non-volatile EEPROM, but fortunately, on power up, both pots initialize at zero. Actually, with the virtually free EEPROM available on a PIC, I am uncertain the absence of EEPROM on the DS1803 is all that serious.

The setting of a potentiometer is implemented with a sequence consisting of the "start", followed by the I2C address with the R/W bit at 0 (write), followed by either the command 0xa9 or 0xaa to identify which of the two potentiometers are to be set, followed by the setting. In reading the data sheet, I note that the command 0xaa may be followed by a setting to set both pots to the same value.

The value of the pots may be read by sending the I2C address byte with the R/W bit at 1 (read) and the setting of each of the pots are then read.

The following program simply sets the pots to 0x40 (1/4 of full setting) and 0x80 (1/2 setting) and then reads these values and displays them on the LCD.

```
// DS1803_1.BS2
//
// Illustrates how to control DS1803 Addressable Dual Potentiometer
//
// 16F877                           DS1803
//
//  SCL/RC3 (term 18 ) --------- SCL (term 9) ----- To Other
//  SDA/RC4 (term 23) ---------- SDA (term 10) ----- I2C Devices
//
// Note that the slave address is determined by A2 (term 5), A1 (term
//  6) and A0 (term 7) on the 1803.  The above SCL and SDA leads may be
// multipled to eight devices, each strapped for a unique A2 A1 A0
// setting.  In this example A2, A1 and A0 are strapped to ground.
//
// Pot 0 is set to a value of 0x40 (1/4) and Pot 1 to 0x80 (1/2).  The
// settings of the two pots are then read from the 1803 and displayed
// on the LCD.
//
// copyright Peter H. Anderson, Baltimore, MD, Mar, 01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <i2c_mstr.h>

#define TRUE !0
#define FALSE 0

void ds1803_write_pot(byte device, byte pot, byte setting);
void ds1803_read_pots(byte device, byte *p_setting_0, byte *p_setting_1);

void main(void)
{
   byte pot_setting_0, pot_setting_1;

   lcd_init();
   i2c_master_setup();

   ds1803_write_pot(0, 0, 0x40);
                 // dev 0, pot 0, setting 0x40 (1/4 of full)
   ds1803_write_pot(0, 1, 0x80);  // pot 1 setting 0x80 (1/2 of full)

   ds1803_read_pots(0, &pot_setting_0, &pot_setting_1);

   lcd_clr_line(0);
   printf(lcd_char, "POT 0 = ");
   lcd_hex_byte(pot_setting_0);
   lcd_clr_line(1);
   printf(lcd_char, "POT 1 = ");
   lcd_hex_byte(pot_setting_1);

   while(1)       ;
}

void ds1803_write_pot(byte device, byte pot, byte setting)
//writes specified setting to specified potentiometer on specified device
{
   i2c_master_start();
   i2c_master_out_byte(0x50 | (device << 1));
   i2c_master_out_byte(0xa9 + pot);  // 0xa9 for pot 0, 0xaa for pot 1
   i2c_master_out_byte(setting);
   i2c_master_stop();
}
```

```
void ds1803_read_pots(byte device, byte *p_setting_0, byte *p_setting_1)
//reads data from both potentiometers
{
   i2c_master_start();
   i2c_master_out_byte(0x51 | (device << 1));
   *p_setting_0 = i2c_master_in_byte(TRUE);
   *p_setting_1 = i2c_master_in_byte(FALSE);
   i2c_master_stop();
}

#include <lcd_out.c>
#include <i2c_mstr.c>
```

Program 8574_1.C.

The Philips PCF8574 8-bit I/O expander is convenient device to expand the number of I/O. I considered using it in implementing the LCD that is shipped with the PIC16F87X Dev Package, but favored the 74HC595 for a few dollars less at the expense of only one extra PIC terminal. I have often entertained using the 8574 with a keypad, but have never quite gotten sufficiently excited.

The 8574 provides for three user address straps, permitting up to eight PCF8574s on the same bus.

In addition, Philips has coded two different devices. A PCF8574 has a manufacturer's family address of 0100 and a PCF8574A has a family address of 0111. Thus, if there were no other devices having these family addresses, up to eight of each could be accommodated on the same bus. And, of course, if the "bit bang" approach is used, you could well have multiple I2C busses.

Each of the eight I/O's may be configured as an output zero or as an input (high impedance). Note that an output logic one is actually the high Z associated with an input. Thus, when driving an external device, the device can only sink current (logic zero). In the high-Z logic one state, the amount of source current is limited by the value of pull-up resistor which is used.

Although defining an I/O as an input and outputting a logic one are precisely the same thing, the following routine treats them as variables "dirs" and "patt". Thus, a dir value of 0x80 and a patt of 0x7e is, in fact, the same as outputting the value 0xfe.

To write a value to the 8574, the sequence is "start" followed by the I2C address byte with the R/W bit set to zero (write) followed by the desired state of the IOs. In my case, this is simply dirs | patts.

To read a value, the sequence is "start", followed by the I2C address byte with the R/W bit set to 1 (read). The value is then read. Note that if a bit has been defined as an output logic zero, it will be read as a zero.

In the following routine, an LED on 8574 output P0 is flashed if P7 on the 8574 is at ground.

Note that the 8574 also includes an open drain output which is latched when any input changes. This is an open drain and thus several devices may be wire wire ored together to a single pull-up resistor to the PICs external interrupt on RB0. Thus, on interrupt, the PIC would read each of the 8574's on the bus to determine which bit or bits had changed. The read of the 8574 clears the latch and the output returns to its normal high-Z state. The following routine does not treat this feature.

```
// 8574_1.C
//
// Illustrates control of 8574.  Flashes LED on P0 of 8574 if switch at
// P7 of 8574 is at zero.  Uses SSP Module in the I2C Master Mode.
//
```

```
//    PIC16F877                     PCF8574
//
//  SCL/RC3 (term 18)---- SCL (term 14) ----- To Other
//  SDA/RC4 (term 23)---- SDA (term 15) ----- I2C Devices
//
// Note that the slave address is determined by A2 (term 3), A1
// (term 2) and A0 (term 1) on the 8574.  The above SCL and SDA leads
// may be multipled to eight devices, each strapped for a unique A2
// A1 A0 setting.  In this example, A2, A1 and A0 are strapped to
// ground.
//
// Pullup resistors to +5VDC are required on both SDA and SCL
// signal leads.
//
// copyright, Peter H. Anderson, Baltimore, MD, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <i2c_mstr.h>

#define TRUE !0
#define FALSE 0

// routines used for 8574
byte in_patt(byte dev_adr);
void out_patt(byte dev_adr, byte dirs, byte patt);

void main(void)
{
   byte inputs;

   i2c_master_setup();

   out_patt(0x00, 0x80, 0x7f);
   // address = 0, dirs = 0x80, patt = 0x7f

   while(1)
   {
      inputs = in_patt(0x00); // read inputs

      if (inputs&0x80)  // if switch at in7==1 then turn off LED
      {
         out_patt(0x00, 0x80, 0x7f);
      }
      else // flash the LED one time
      {
         out_patt(0x00, 0x80, 0x7e);  // turn the LED on
         delay_ms(500);
         out_patt(0x00, 0x80, 0x7f);  // turn it on
         delay_ms(500);
      }
   }
}
```

```
byte in_patt(byte dev_adr)
{
   byte y;
   i2c_master_start();
   i2c_master_out_byte(0x40 | (dev_adr<<1) | 0x01);
   y=i2c_master_in_byte(FALSE);
   i2c_master_stop();
   return(y);
}

void out_patt(byte dev_adr, byte dirs, byte patt)
{
   i2c_master_start();
   i2c_master_out_byte(0x40 | (dev_adr << 1));
   i2c_master_out_byte(dirs | patt);
   i2c_master_stop();
}

#include <lcd_out.c>
#include <i2c_mstr.c>
```

Program DS1307_1.C.

The Dallas DS1307 RTC might be thought of as a RAM, with seconds at location 0x00, followed by minutes, hours, weekday, date, month and year. There is also a control register at location 0x07.

Note that bit 7 in the seconds register (0x00) is used to disable or enable the clock's oscillator. Setting this bit to a logic 1 disables the clocking. Bit 6 in the hours register (0x02) controls whether the hours are in 12 or 24 hour format. When this bit is set to one (12 hour format), bit 5 of the hours register is interpreted as AM or PM.

As with the DS1307 and virtually every RTC I have tinkered with, the time and date is stored in BCD format.

There is also a control register at location 0x07 which is used to configure an output to provide an output square wave having frequencies of 1 Hz, 4.096, 8.192 or 32.767 kHz.

Address locations 0x08 – 0x3f (56 locations) are simply RAM.

Writing to the device is implemented by the "start" followed by the I2C address byte with the R/W bit set to write (0), followed by the address to start writing data. This is then followed by one or more data bytes. Note that the DS1307 auto increments to the next address after the receipt of each byte.

Reading from the device is quite similar to the 24LC256 EEPROM. The address in the DS1307 is first written to the device; "start", I2C address with R/W at write and then the DS1307 address to begin reading. This is followed by a "repeated start" (no intermediate "stop") followed by the I2C address byte with the R/W bit set to read. Data is then read byte by byte.

```
// DS1307.C
//
// Writes a base time and date to DS1307.  About every second reads
// time and date and displays to serial LCD on RA.0.
//
// Offers a good example of working with structures.  Note that
// structures may only be passed using pointers.
//
```

```
//    DS1307                         DS1307
//
// SCL (term 18)------------------- SCL (term 6) ----- To Other
// SDA (term 23) ------------------- SDA (term 5) ----- I2C Devices
//
// Note that there is no provision for the user defining the
// secondary I2C address using straps.
//
// copyright, Peter H. Anderson, Baltimore, MD, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <i2c_mstr.h>

#define TRUE !0
#define FALSE 0

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

struct Date
{
   byte yr;
   byte mo;
   byte da;
   byte weekday;
};

// routines used for DS1307
void set_up_clock(int control_reg);
void write_clock(struct Time *p_t, struct Date *p_d);
void read_clock(struct Time *p_t, struct Date *p_d);

void main(void)
{
   byte mem_adr;
   byte n;

   struct Time t_base={0x23, 0x59, 0x00}, t;
   struct Date d_base={0x00, 0x02, 0x28, 0x07}, d;

   lcd_init();
   i2c_master_setup();

   set_up_clock(0x10);  // sqwe enabled, 1 Hz output
   write_clock(&t_base, &d_base);

   while(1)
   {
      read_clock(&t, &d);
      lcd_init();
      printf(lcd_char, "%x/%x/%x %x:%x:%x",
                              d.yr, d.mo, d.da, t.hr, t.mi, t.se);
      lcd_clr_line(1);
      printf(lcd_char, "%x", d.weekday);
      delay_ms(1000);
   }
}

void set_up_clock(int control_reg)
{
   i2c_master_start();
   i2c_master_out_byte(0xd0);
   i2c_master_out_byte(0x07);  // control register address
   i2c_master_out_byte(control_reg);
   i2c_master_stop();
}

void write_clock(struct Time *p_t, struct Date *p_d)
{
   i2c_master_start();
   i2c_master_out_byte(0xd0);   // address
   i2c_master_out_byte(0x00);     // first address
   i2c_master_out_byte(p_t->se);
   i2c_master_out_byte(p_t->mi);
   i2c_master_out_byte(p_t->hr);
                  // 24 hour format - 0x40 for 12 hour time
   i2c_master_out_byte(p_d->weekday);
   i2c_master_out_byte(p_d->da);
   i2c_master_out_byte(p_d->mo);
   i2c_master_out_byte(p_d->yr);
   i2c_master_stop();
}

void read_clock(struct Time *p_t, struct Date *p_d)
{
   i2c_master_start();
   i2c_master_out_byte(0xd0);   // 1101 000 0
   i2c_master_out_byte(0x00);  // first address
   i2c_master_stop();

   i2c_master_repeated_start();
   i2c_master_out_byte(0xd1);
   p_t->se = i2c_master_in_byte(TRUE) & 0x7f;
   p_t->mi = i2c_master_in_byte(TRUE);
   p_t->hr = i2c_master_in_byte(TRUE) & 0x3f;

   p_d->weekday = i2c_master_in_byte(TRUE) & 0x07;
   p_d->da = i2c_master_in_byte(TRUE) & 0x3f;
   p_d->mo = i2c_master_in_byte(TRUE) & 0x3f;
   p_d->yr = i2c_master_in_byte(FALSE);
   // no ack prior to stop
   i2c_master_stop();
}

#include <lcd_out.c>
```

```
#include <i2c_mstr.c>
```

Program DS1624_1.C.

The Dallas DS1624 combines a thermometer with 256 bytes of EEPROM. The EEPROM might be used for a limited data logger or in creating a temperature histogram, in defining high and low trip points or the times to perform a measurement, or it may be used for any application with a PIC not having on-board EEPROM.

As with many Dallas temperature sensors, Dallas has defined specific command codes;

Access Config Register	0xac
Access Memory (EEPROM)	0x17
Start Temp Conversion	0xee
Read Temperature	0xaa
Stop Temp Conversion	0x22

Thus, to write to the non-volatile configuration register, the sequence is “start” followed by the I2C address byte with the R/W bit at zero (write), followed by command 0xac, followed by the desired setting of the register. A delay is required for this to burn into EEPROM. For the DS1624, there is only one bit in this register which is of interest; 1SHOT.

In the 1SHOT mode, each temperature measurement must be initiated by sending command code 0xee which consists of “start”, followed by the I2C address byte with the R/W bit set to zero (write) followed by the start temperature conversion command, 0xee. After a delay to allow time for the measurement, the temperature result is read with the sequence “start” followed by the I2C address byte (write), followed by the read temperature command 0xaa. A “repeated start” is then followed by the I2C address byte with the R/W bit in the read mode (one) and the two byte result is read.

The advantage of the 1SHOT mode is one of saving power. The disadvantage is having to initiate the measurement each time and then wait up to one second for the measurement result.

If the configuration register is programmed for continuous operation (1SHOT bit at zero), the start temperature conversion command need only be sent one time and the temperature may then be read at any time by issuing the read temperature command. The penalty is that the DS1624 is continually performing temperature conversions and thus consuming power. The continual conversion may be turned off by issuing the stop temperature conversion command (0x22).

The full 13-bit temperature result consists of two bytes and is of the form;

```
SWWW WWWW  FFFF FXXX
```

Where S is the sign bit (0 for plus) and WWW WWW is the whole part. The fractional part is in the high five bits of the second byte.

Dallas indicates these two bytes are in two’s compliment format.

Writing to EEPROM consists of “start”, followed by the I2C address byte with the R/W bit in the write mode (zero), followed by the access EEPROM command (0x17), followed by the one byte address, followed by one or more data bytes. A delay is required for the data to be programmed into memory.

Reading from EEPROM is implemented by “start”, followed by the I2C address byte with the R/W byte in the write mode, followed by the access EEPROM command, followed by the EEPROM address. This is followed by a “repeated start”, followed by the I2C address byte with the R/W bit in the read mode (one). One or more data bytes are then read.

Note that in developing this routine, I wrote each byte to EEPROM one byte at a time and did the same in reading them. I now note the data sheet indicates the DS1624 supports a sequential data write and read of up to eight bytes, much the same as presented for the Microchip 24LC256. However, I am reluctant to fool too much with a tested routine and have left this routine as byte by byte.

In the following routine, the configuration register is placed in the 1SHOT mode. Ten measurements are performed and each is displayed on the LCD and also saved to EEPROM. The data is then read from EEPROM and the logged temperature results are again displayed on the LCD.

Note that in converting the raw data for display on the LCD, the high bit is tested as to the sign and if it is a one (minus), a minus is output on the LCD and a two's compliment operation is performed on the two byte reading.

It is important to note that taking the two's compliment of a two byte quantity is not a matter of taking the two's compliment of each byte. Rather, each byte is complemented and one is added to the low byte. One is added to the high byte only if there is an overflow in the low byte (low byte at zero). I say, that one is added to the low byte, but, in fact, there are only five significant in the low byte and thus adding one is a matter of adding 0x08. There are a number of points here that eluded me for a good deal of time.

In developing this routine I did not run leads over to the refrigerator or out the window to actually test the measurements below zero degrees C. If you should do this, I would appreciate any feedback.

In computing the fractional part, I used an array. Note that quantities after the "binary point" are of the form;

```
      2^-1        2^-2        2^-3        2^-3        2^-4
```

or;

```
      50/100      25/100      12.5/100    6.25/100    3.125/100
```

Thus, I used a constant array consisting of 3, 6, 12, 25 and 50 and added each element to variable sum if the corresponding bit is a logic one. This technique avoids the use of floats.

```
byte ds1624_compute_fraction(byte t_fract)
// converts high five bits in t_fract to a number in the range of
// 0 - 100.
{
    byte sum = 0;
    byte y, n;
    const byte dec_vals[5] = {3, 6, 12, 25, 50};
                        // 3 / 100, 6/100, 12/100, etc
      y = t_fract >> 3;
                        // fractional part is now in lowest five bits
      for (n=0; n<5; n++)
      {
         if (y&0x01)
         {
               sum = sum + dec_vals[n];
         }
         y = y >> 1;
      }
      return(sum);
}
```

Note that a bit more resolution might have been achieved by using a const array of longs having values of 31, 62, 125, 250 and 500.

As an aside, Dallas uses the same format in the 1-W DS18B20 and the lower cost DS1822 Thermometer. Note that I have not actually ever seen a DS1822 and am uncertain if it is a real product.

```
// 1624_1.C
//
//    PIC16F877                  DS1624
//
//    SCL (term 18) ----------- SCL (term 2) ----- To Other
//    SDA (term 23) ----------- SDA (term 1) ----- I2C Devices
//
//
// Performs 10 temperature measurements and displays each result on the
// LCD and also save the data to the DS1624's EEPROM.  The data is then
// read from EEPROM and displayed on the LCD.
//
// copyright, Peter H. Anderson, Baltimore, MD, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <i2c_mstr.h>

#define TRUE !0
#define FALSE 0

// functions in this program
void ds1624_config(byte dev_adr, byte mode);
void ds1624_start_conv(byte dev_adr);
void ds1624_meas_temp(byte dev_adr, byte *p_whole, byte *p_fract);
byte ds1624_compute_fraction(byte t_fract);

void ds1624_ee_write(byte dev_adr, byte ee_adr,  byte dat);
byte ds1624_ee_read(byte dev_adr, byte ee_adr);

#define NUM_SAMPS 10

void main(void)
{
   byte m, n, line, ee_adr, dat_h, dat_l, temp_h, temp_l,
                  t_whole, t_fract;

   lcd_init();
   i2c_master_setup();

   ds1624_config(0x02, 0x01);
                  // dev adr is 0x02, mode is 0x01 - 1 shot

   for(n = 0, ee_adr=0; n<NUM_SAMPS; n++)
   {
```

```
  ds1624_start_conv(0x02);
  delay_ms(1000);
  ds1624_meas_temp(0x02, &dat_h, &dat_l);
         // note that pointers are passed
  if(dat_h & 0x80)         // result is negative
  {
      temp_l = ((~dat_l) & 0xf8) + 0x08;
      temp_h = ~dat_h;
      if (temp_l == 0)
      {
         ++temp_h;
      }
      lcd_char('-');
  }
  else
  {
       temp_h = dat_h;
       temp_l = dat_l;
  }
  t_whole = temp_h;
  t_fract = ds1624_compute_fraction(temp_l);
                             // convert to decimal format
  lcd_clr_line(0);
  lcd_dec_byte(t_whole, 2);
  lcd_char('.');     // decimal point
  lcd_dec_byte(t_fract, 2);

  ds1624_ee_write(0x02, ee_adr, dat_h);
  ds1624_ee_write(0x02, ee_adr+1, dat_l);
  ee_adr += 2;
  delay_ms(1000);
}

lcd_init();

for (ee_adr = 0, n = 0, m = 0, line = 0; n<NUM_SAMPS; n++)
         // now fetch each from EEPROM and display
{
   if (m==2)         // two readings per line
   {
        m=0;
        ++line;
        if (line == 4)
        {
           line = 0;
           lcd_init();
        }
        lcd_clr_line(line);
   }
   ++m;

   dat_h = ds1624_ee_read(0x02, ee_adr);
   dat_l = ds1624_ee_read(0x02, ee_adr+1);
   ee_adr += 2;

   if(dat_h & 0x80)        // result is negative
   {
         temp_l = (~dat_l & 0xf8) + 0x08;
         temp_h = ~dat_h;
         if (temp_l == 0)
         {
            ++temp_h;
         }
         lcd_char('-');
      }

      else
      {
         temp_h = dat_h;
         temp_l = dat_l;
      }

      t_whole = temp_h;
      t_fract = ds1624_compute_fraction(temp_l);
                                 // convert to decimal format
      lcd_dec_byte(t_whole, 2);
      lcd_char('.');    // decimal point
      lcd_dec_byte(t_fract, 2);

      lcd_char(' ');
      delay_ms(500);
   }

   while(1) /* loop */        ;
}

void ds1624_config(byte dev_adr, byte mode)
// configures DS1624 in 1SHOT temperature conversion mode
{
   i2c_master_start();
   i2c_master_out_byte(0x90 | (dev_adr << 1));
   i2c_master_out_byte(0xac); // access configuration
   i2c_master_out_byte(mode);
   i2c_master_stop();
   delay_ms(25);        // wait for EEPROM to program
}

void ds1624_start_conv(byte dev_adr)
{
   i2c_master_start();
   i2c_master_out_byte(0x90 | (dev_adr << 1));
   i2c_master_out_byte(0xee); // start conversion
   i2c_master_stop();
}

void ds1624_meas_temp(byte dev_adr, byte *p_whole, byte *p_fract)
// fetches temperature result.  Values t_whole and t_fract returned
// using pointers
{
   i2c_master_start();
   i2c_master_out_byte(0x90 | (dev_adr << 1));
   i2c_master_out_byte(0xaa); // fetch temperature

   i2c_master_repeated_start();     // no intermediate stop
   i2c_master_out_byte(0x90 | (dev_adr << 1) | 0x01);
   *p_whole=i2c_master_in_byte(TRUE);
                  // value pointed to by p_whole
   *p_fract=i2c_master_in_byte(FALSE);
   i2c_master_stop();
}

byte ds1624_compute_fraction(byte t_fract)
// converts high five bits in t_fract to a number in the range of
// 0 - 100.
{
    byte sum = 0;
    byte y, n;
      const byte dec_vals[5] = {3, 6, 12, 25, 50};
                        // 3 / 100, 6/100, 12/100, etc
      y = t_fract >> 3;
                        // fractional part is now in lowest five bits
      for (n=0; n<5; n++)
      {
         if (y&0x01)
         {
               sum = sum + dec_vals[n];
         }
         y = y >> 1;
      }
      return(sum);
}

byte ds1624_ee_read(byte dev_adr, byte ee_adr)
// returns content location of location ee_adr in DS1624 EEPROM
{
   byte y;

   i2c_master_start();
   i2c_master_out_byte(0x90 | (dev_adr << 1));
   i2c_master_out_byte(0x17);       // access memory

   i2c_master_out_byte(ee_adr);     // the eeprom address

   i2c_master_repeated_start();     // no intermediate stop
   i2c_master_out_byte(0x90 | (dev_adr << 1) | 0x01);

   y = i2c_master_in_byte(FALSE);
   i2c_master_stop();

   return(y);
}

void ds1624_ee_write(byte dev_adr, byte ee_adr, byte dat)
// writes content of dat to specified address ee_adr
{
   i2c_master_start();
   i2c_master_out_byte(0x90 | (dev_adr << 1));
   i2c_master_out_byte(0x17);       // access memory

   i2c_master_out_byte(ee_adr);     // the eeprom address
   i2c_master_out_byte(dat);        // the eeprom data
   i2c_master_stop();
   delay_ms(25);              // wait for eeprom to program
}

#include <lcd_out.c>
#include <i2c_mstr.c>
```

Philips PCF8583 RTC and Counter.

The Philips PCF8583 is the most robust real time clock I have worked with and even after implementing a number of routines, I am uncertain I have fully utilized all of it's capabilities. It's weakness is that there is no provision for a back-up battery.

One interesting capability is that it may be used either for time or for counting events which opens up possibilities for an outboard rain gage using a tipping bucket, tachometer or totalizer. One might opt for this approach when using a PIC having only the single TMR0 which is being used for timing or for any PIC when the requirement is to count a number of sources.

It is important to note that the assigned I2C family code for the 8583 is 1010 which is the same code used by the Microchip 24LCXX EEPROM family. Thus, when using the 8583 on a bus with EEPROMs it is important to be certain that the user straps are set to avoid duplications of the I2C address. I learned this the hard way and it was not an easy problem to find.

Philips uses the term "count" whether the counting is the timing crystal or the counting of events. Thus, in the following, I try to use the words "time" or "event" to distinguish between the two modes.

In the following, routine 8583_1.c illustrates setting and reading time. 8583_2.c illustrates the use of the dated alarm and 8583_3.c illustrates the use of the periodic alarm function.

Routines 8583_4.c and 8583_5.c illustrate the counting of events and are similar in concept to 8583_2.c and 8583_3.c.

Perhaps I spent too much time on this device, but the routines provide good illustrations of various structures including calculating a new date and time and working with multi-byte quantities in BCD format.

Program 8583_1.c.

As with other RTCs, the 8583 might be thought of as RAM. Writing data is implemented with a "start" followed by the I2C address byte in the R/W bit in the write mode, followed by the address, followed by one or more data bytes. Reading data is implemented by the sequence "start" followed by the I2C address byte with the R/W bit in the write mode followed by the address to begin reading. This is followed with a "repeated start", the I2C address byte in the read mode and one or more data bytes are read.

Note that locations 0x10 – 0xff (240 locations) are simply RAM.

The timer (counter) registers are at locations 0x00 – 0x07 and the alarm registers, if used, are at locations 0x08 – 0x0f.

Location 0x00 is a control register which among other things, controls the counting source; either 32.767 kHz timer mode or event counter mode, a stop count bit, a bit to control whether the last count is held, an alarm enable bit and a few others that I didn't fool with.

In the timing mode, locations 0x01 – 0x06 are hundredths of seconds, seconds, minutes, year and date packed into one byte and weekday and month packed into one byte. Note that only two bits are used for the year and thus only values 0 – 3 are permissible. Thus, it is for the interfacing circuitry to distinguish whether the year is 2000, 2004 or 2008. When the year is 0, a leap day is inserted which will work until 2100, but I'm not sure there will be all that many circuits designed today still in service by then. Location 0x07 is a "timer". More on this later in routine 8583_3.c.

In the following routine, the control register is configured for the timer mode, counting on, no alarm.

A base date and time is written to the device and the program continually reads the date and time about every second and displays this on the LCD.

```
// 8583_1.C
//
// Writes a base time and date to clock.  Reads clock about every
// second and displays on LCD.
//
// PIC16F877                                         PCF8583
//
//  SCL (term 18) ------------------- SCL (term 6) ----- To Other
//  SDA (term 23) ------------------- SDA (term 5) ----- I2C Devices
//
// I2C address is 0xa0 or 0xa2 depending on strapping of A0 (terminal
// 3). In this example, A0 is at ground.
//
// copyright, Peter H. Anderson, Baltimore, MD, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <i2c_mstr.h>

#define TRUE !0
#define FALSE 0

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

struct Date
{
   byte yr;
   byte mo;
   byte da;
   byte weekday;
};

// routines used for PCF8583
void _8583_configure_control_register(byte control_reg);
void _8583_write_clock(struct Time *p_t, struct Date *p_d);
void _8583_read_clock(struct Time *p_t, struct Date *p_d);

void _8583_display_date_time(struct Time *p_t, struct Date *p_d);
```

```
byte to_BCD(byte natural_binary);
byte to_natural_binary(byte BCD);

void main(void)
{
   struct Time t_base, t;
   struct Date d_base, d;
   t_base.hr=23;  t_base.mi=59;  t_base.se=00;
   d_base.yr=3; d_base.mo=2; d_base.da=28; d_base.weekday=1;
   // Note that year is 0 through 3.  Thus, if the base year is 2000
   // this is Feb 28, 2003, Monday

   lcd_init();
   i2c_master_setup();

   _8583_configure_control_register(0x00);  // 32.768 kHz, no alarm
   _8583_write_clock(&t_base, &d_base);

   while(1)
   {
      _8583_read_clock(&t, &d);
      _8583_display_date_time(&t, &d);
      delay_ms(1000);
   }
}

void _8583_write_clock(struct Time *p_t, struct Date *p_d)
{
// Note that most sig bit of hours is 12/24 hour format
// 4 year is in bits 7 and 6 of 0x05.  Lower six bits are day in BCD
// Location 0x06.  Weeks day is in bits 7, 6, 5 and month in lower
// five bits.

   byte v;

   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x01); // address of first write
   i2c_master_out_byte(0x00); // hundreths of a second
   v = to_BCD(p_t->se);
   i2c_master_out_byte(v);
   v = to_BCD(p_t->mi);
   i2c_master_out_byte(v);    //  location 0x03
   v = to_BCD(p_t->hr);
   i2c_master_out_byte(v);
       // 24 hour format - 0x80 for 12 hour time
   v = to_BCD(p_d->da);
   i2c_master_out_byte((p_d->yr << 6) | v);
                    // YY TT UUUU
   v = to_BCD(p_d->mo);
   i2c_master_out_byte((p_d->weekday << 5) | v );
                   // DDD T UUUU
   i2c_master_stop();
}

void _8583_read_clock(struct Time *p_t, struct Date *p_d)
{
   byte v;
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x02); // begin with seconds

   i2c_master_repeated_start();
   i2c_master_out_byte(0xa1);

   v = i2c_master_in_byte(TRUE) & 0x7f;
   p_t->se = to_natural_binary(v);
   v = i2c_master_in_byte(TRUE);
   p_t->mi = to_natural_binary(v);
   v = i2c_master_in_byte(TRUE) & 0x3f;
   p_t->hr = to_natural_binary(v);

   v = i2c_master_in_byte(TRUE);
   p_d->yr = v >> 6;          // year is in two most sig bits
   v = v & 0x3f;  // day in lower six bits
   p_d->da = to_natural_binary(v);

   v = i2c_master_in_byte(FALSE);
   p_d->weekday = v >> 5;
   v = v & 0x1f;
   p_d->mo = to_natural_binary(v);
   // no ack prior to stop
   i2c_master_stop();
}

void _8583_configure_control_register(byte control_reg)
{
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x00);  // control register address
   i2c_master_out_byte(control_reg);
   i2c_master_stop();
}

void _8583_display_date_time(struct Time *p_t, struct Date *p_d)
{
  lcd_init();
  lcd_dec_byte(p_d->yr + 00, 2); // assumes base year of 2000
  lcd_char('/');
  lcd_dec_byte(p_d->mo, 2);
  lcd_char('/');
  lcd_dec_byte(p_d->da, 2);
  lcd_char(' ');
  lcd_dec_byte(p_t->hr, 2);
  lcd_char(':');
  lcd_dec_byte(p_t->mi, 2);
  lcd_char(':');
  lcd_dec_byte(p_t->se, 2);

  lcd_clr_line(1);
  lcd_dec_byte(p_d->weekday, 1);
}

byte to_BCD(byte natural_binary)
```

```
{
   return ( ((natural_binary/10) << 4) + natural_binary%10 );
}

byte to_natural_binary(byte BCD)
{
   return(  ((BCD >> 4) * 10) + (BCD & 0x0f)  );
}

#include <lcd_out.c>
#include <i2c_mstr.c>
```

Program 8583_2.c.

The intent of this routine was to illustrate the dated alarm feature. Alarm locations, 0x09 – 0x0e define the alarm date and time and are in the same format as 0x01 – 0x06; hundredths of a second, seconds, etc. The idea is that when there is a match between the current time and the alarm time, either an alarm flag is set or an output goes to zero which I used to interrupt the PIC.

The control register at 0x00 is configured for the timer mode, count on and alarm enabled. With the alarm bit enabled, the 8583 then uses locations 0x08 – 0x0f.

Location 0x08 is the alarm control register and here again, there are many options including a bit to control whether the alarm output is to go low on alarm and the alarm mode, either timer" or "dated". In this routine, the dated alarm feature was used.

For this tutorial, I probably should have simply written the current date and time and an alarm date and time to locations 0x01 – 0x06 and 0x09 – 0x0e and let it go at that.

However, as I am prone to do, I got carried away and opted to calculate the alarm date and time which is "m" minutes from the current date and time. Imagine the utility of setting the alarm for every 10,232 minutes!

The calculation of the new alarm date and time might be implemented by simply incrementing the minutes, :m times which would necessitate providing for incrementing the hour and of course the date. However, the number of possible iterations may be reduced by calculating the number of hours "h" and the remaining minutes "m" and then incrementing the hours "h" times which involves provision for incrementing the date and then incrementing the minutes "m" times which involves a provision for incrementing for incrementing the hours.

I'm sure this is about a clear as mud. And, I really wouldn't bet the ranch on my code being bullet proof. The important thing is to write a time and date to 0x01 – 0x06 and then to simply write the alarm date and time to 0x09 – 0x0d and understand that the interrupt occurs when the current time and date matches the alarm time and date. If you care to dig into my calc_new_time(), it's there and if you do more fully test it than I did, please let me know. One thing this does drive home is the advantage of C over assembly. I just couldn't imagine wasting time trying to do this in assembly.

```
// 8583_2.C
//
// Illustrates how to force a periodic interrupt using the PCF8583s
// alarm function.
//
// Sets clock to a base date and time.  Sets alarm to two minutes
// later.  Configures for dated alarm.  Reads and displays the date and // time.
//
// On interrupt, sets alarm for 2 minutes later, displays the date
// and time and momentarily flashes an LED on PORTD7
```

```
//
// PIC16F877                        PCF8583
//
//  SCL (term 18) ---------- SCL (term 6) ----- To Other
//  SDA (term 23) ---------- SDA (term 5) ----- I2C Devices
//
// I2C address is 0xa0 or 0xa2 depending on strapping of A0 (terminal 3)
// In this example, A0 is at ground.
//
//  PCF8583                         PIC16F877
//
//  /INT (term 7) ------------------- RB0
//
// copyright, Peter H. Anderson, Baltimore, MD, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <i2c_mstr.h>

#define TRUE !0
#define FALSE 0

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

struct Date
{
   byte yr;
   byte mo;
   byte da;
   byte weekday;
};

// routines used for PCF8583
void _8583_configure_control_register(byte control_reg);
void _8583_configure_alarm_register(byte alarm_control_reg);

void _8583_write_clock(struct Time *p_t, struct Date *p_d);
void _8583_write_alarm(struct Time *p_t, struct Date *p_d);
void _8583_read_clock(struct Time *p_t, struct Date *p_d);
void _8583_read_alarm(struct Time *p_t, struct Date *p_d);

void _8583_display_date_time(struct Time *p_t, struct Date *p_d);

void calc_new_time(struct Time *p_t, struct Date *p_d,
                  unsigned long minutes);
void increment_time_minutes(struct Time *p_t, struct Date *p_d);
void increment_time_hours(struct Time *p_t, struct Date *p_d);
void increment_date(struct Date *p_date);
```

```
byte to_BCD(byte natural_binary);
byte to_natural_binary(byte BCD);

#define LED_DIR trisd7
#define LED_PIN portd7

byte const days_in_month[13] = {0, 31, 28, 31, 30, 31, 30,
                                  31, 31, 30, 31, 30, 31};
// Note that Jan is month 1.  Element 0 is not used

byte ext_int_occurred;        // global variable to indicate an interrupt occurred

void main(void)
{
   struct Time t;
   // note these numbers are in natural binary
   struct Date d;
   t.hr=23;  t.mi=59;  t.se=00;
   d.yr=3; d.mo=2; d.da=28; d.weekday=1;

   lcd_init();
   i2c_master_setup();

   not_rbpu = 0;  // enable internal pull-ups on PORTB

   pspmode = 0;         // use PORTD as general purpose IO
   LED_DIR = 0;
   LED_PIN = 0;   // turn off LED

   ext_int_occurred = FALSE;
   _8583_write_clock(&t, &d);
   _8583_display_date_time(&t, &d);
   delay_ms(1000);

   calc_new_time(&t, &d, 2);              // two minutes
   _8583_display_date_time(&t, &d);
               // to verify calculation is working
   _8583_write_alarm(&t, &d);
   _8583_configure_alarm_register(0x80 | 0x20 | 0x10); // dated alarm
   _8583_configure_control_register(0x04); // 32.768 kHz, alarm
   delay_ms(1000);

   _8583_read_clock(&t, &d);        // read and display the time
   _8583_display_date_time(&t, &d);
   delay_ms(1000);

   _8583_read_alarm(&t, &d);  // read an display the alarm value
   _8583_display_date_time(&t, &d);
   delay_ms(1000);

   intf = 0;      // kill any pending interrupts
   intedg = 0;    // negative edge
   inte = 1;
   gie = 1;       // enable interrupts

   while(1)
   {
      if(ext_int_occurred)
      {
         while (gie)    // for the moment disable interrupts
         {
            gie = 0;
         }
         ext_int_occurred = FALSE;
         _8583_read_alarm(&t, &d);
         calc_new_time(&t, &d, 2);  // add two minutes
         _8583_write_alarm(&t, &d);
         _8583_configure_alarm_register(0x80 | 0x20 | 0x10);
                        // dated alarm
         _8583_configure_control_register(0x04); // 32.768 kHz, alarm

         _8583_read_alarm(&t, &d);  // read and display the new alarm
         _8583_display_date_time(&t, &d);
         LED_PIN = 1;   // momentarily wink the LED
         delay_ms(1000);
         LED_PIN = 0;
         gie = 1; // enable interrupts again
      }
   }
}

void _8583_display_date_time(struct Time *p_t, struct Date *p_d)
{
  lcd_init();
  lcd_dec_byte(p_d->yr + 00, 2); // assumes base year of 2000
  lcd_char('/');
  lcd_dec_byte(p_d->mo, 2);
  lcd_char('/');
  lcd_dec_byte(p_d->da, 2);
  lcd_char(' ');
  lcd_dec_byte(p_t->hr, 2);
  lcd_char(':');
  lcd_dec_byte(p_t->mi, 2);
  lcd_char(':');
  lcd_dec_byte(p_t->se, 2);

  lcd_clr_line(1);
  lcd_dec_byte(p_d->weekday, 1);
}

void calc_new_time(struct Time *p_t, struct Date *p_d,
                                          unsigned long minutes)
{
   byte n, hours, mins;
   hours = (byte)(minutes / 60);
                  // split into number of hours and remaining minutes
   mins = (byte)(minutes % 60);

   for (n=0; n<hours; n++)
   {
         increment_time_hours(p_t, p_d);
   }
   for (n=0; n<mins; n++)
   {
         increment_time_minutes(p_t, p_d);
   }
}

void increment_time_minutes(struct Time *p_t, struct Date *p_d)
{
      ++p_t->mi;
      if (p_t->mi > 59)
      {
            p_t->mi = 0;
            increment_time_hours(p_t, p_d);
      }
}

void increment_time_hours(struct Time *p_t, struct Date *p_d)
{
      ++p_t->hr;
      if (p_t->hr > 23)
      {
            p_t->hr = 0;
            increment_date(p_d);
      }
}

void increment_date(struct Date *p_date)
{
   if ((p_date->da) == days_in_month[p_date->mo])
      // currently last day of the month
   {
      if(  (p_date->yr==0) && (p_date->mo == 2)  ) // leap year
      {
         ++p_date->da;  // Feb 29
      }
      else if (p_date->mo == 12)    // Dec 31
      {
          ++p_date->yr;
          if (p_date->yr == 4)
          {
             p_date->yr=0;    // roll over the century
          }
          p_date->mo=1;
          p_date->da=1;       // Jan 1
      }
      else
      {
        ++p_date->mo;         // set to first day of new month
        p_date->da=1;
      }
   }
   else // not the last day of the month
   {
      ++p_date->da;  // simply increment the date
   }
}
```

```
void _8583_write_clock(struct Time *p_t, struct Date *p_d)
{
// Note that most sig bit of hours is 12/24 hour format
// 4 year is in bits 7 and 6 of 0x05.  Lower six bits are day in BCD
// Location 0x06.  Weeks day is in bits 7, 6, 5 and month in lower
// five bits.

   byte v;

   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x01); // address of first write
   i2c_master_out_byte(0x00); // hundreths of a second

   v = to_BCD(p_t->se);
   i2c_master_out_byte(v);

   v = to_BCD(p_t->mi);
   i2c_master_out_byte(v);    // 0x03

   v = to_BCD(p_t->hr);
   i2c_master_out_byte(v);
       // 24 hour format - 0x80 for 12 hour time

   v = to_BCD(p_d->da);
   i2c_master_out_byte((p_d->yr << 6) | v);
                    // YY TT UUUU

   v = to_BCD(p_d->mo);
   i2c_master_out_byte((p_d->weekday << 5) | v );
                   // DDD T UUUU
   i2c_master_stop();
}

void _8583_write_alarm(struct Time *p_t, struct Date *p_d)
{
// Note that most sig bit of hours is 12/24 hour format
// 4 year is in bits 7 and 6 of 0x05.  Lower six bits are day in BCD
// Location 0x06.  Weeks day is in bits 7, 6, 5 and month in lower
// five bits.
//
// Note that this function might have been combined with
// _8583_write_clock by passing an indication as to whether clock
// or alarm.

   byte v;

   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x01+8);// address of first write
   i2c_master_out_byte(0x00); // hundreths of a second

   v = to_BCD(p_t->se);
   i2c_master_out_byte(v);

   v = to_BCD(p_t->mi);
   i2c_master_out_byte(v);    // 0x03 + 8
   v = to_BCD(p_t->hr);
   i2c_master_out_byte(v);
       // 24 hour format - 0x80 for 12 hour time

   v = to_BCD(p_d->da);
   i2c_master_out_byte((p_d->yr << 6) | v);
                    // YY TT UUUU

   v = to_BCD(p_d->mo);
   i2c_master_out_byte((p_d->weekday << 5) | v );
                   // DDD T UUUU
   i2c_master_stop();
}

void _8583_read_clock(struct Time *p_t, struct Date *p_d)
{
   byte v;
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x02); // begin with seconds

   i2c_master_repeated_start();
   i2c_master_out_byte(0xa1);

   v = i2c_master_in_byte(TRUE) & 0x7f;
   p_t->se = to_natural_binary(v);
   v = i2c_master_in_byte(TRUE);
   p_t->mi = to_natural_binary(v);
   v = i2c_master_in_byte(TRUE) & 0x3f;
   p_t->hr = to_natural_binary(v);

   v = i2c_master_in_byte(TRUE);
   p_d->yr = v >> 6;          // year is in two most sig bits
   v = v & 0x3f;  // day in lower six bits
   p_d->da = to_natural_binary(v);

   v = i2c_master_in_byte(FALSE);
   p_d->weekday = v >> 5;
   v = v & 0x1f;
   p_d->mo = to_natural_binary(v);
   // no ack
   i2c_master_stop();
}

void _8583_read_alarm(struct Time *p_t, struct Date *p_d)
{
   byte v;
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x02 + 8);   // begin with alarm seconds

   i2c_master_repeated_start();
   i2c_master_out_byte(0xa1);

   v = i2c_master_in_byte(TRUE) & 0x7f;
   p_t->se = to_natural_binary(v);
   v = i2c_master_in_byte(TRUE);
   p_t->mi = to_natural_binary(v);
   v = i2c_master_in_byte(TRUE) & 0x3f;
   p_t->hr = to_natural_binary(v);

   v = i2c_master_in_byte(TRUE);
   p_d->yr = v >> 6;          // year is in two most sig bits
   v = v & 0x3f;  // day in lower six bits
   p_d->da = to_natural_binary(v);

   v = i2c_master_in_byte(FALSE);
   p_d->weekday = v >> 5;
   v = v & 0x1f;
   p_d->mo = to_natural_binary(v);
   // no ack
   i2c_master_stop();
}

void _8583_configure_control_register(byte control_reg)
{
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x00);  // control register address
   i2c_master_out_byte(control_reg);
   i2c_master_stop();
}

void _8583_configure_alarm_register(byte alarm_control_reg)
{
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x08);  // control register address
   i2c_master_out_byte(alarm_control_reg);
   i2c_master_stop();
}

byte to_BCD(byte natural_binary)
{
   return ( ((natural_binary/10) << 4) + natural_binary%10 );
}

byte to_natural_binary(byte BCD)
{
   return(  ((BCD >> 4) * 10) + (BCD & 0x0f)  );
}

#int_ext ext_int_handler(void)
{
    ext_int_occurred = TRUE;   // flag that there has been an interrupt
}

#int_default default_int_handler(void)
{
}

#include <lcd_out.c>
#include <i2c_mstr.c>
```

Program 8583_3.c.

On a more practical note, the PCF8583 may be configured to generate an alarm, every 01 - 99 intervals of time where the interval may be 1/00 of a second, seconds, minutes, hour or even days.

The control register at 0x00 is set as above, 32,767 kHz timer mode, counter on, alarm on.

However, the alarm register at 0x08 is configured for “timer” vs “dated” alarm and the timer function is set for seconds. However, this function might be set for minutes, hours or days.

This causes location 0x07 in the timer section to increment the specified quantity with each passing second, minute, hour or day. An alarm occurs when location 0x07 matches the alarm value set in location of 0x0f.

Thus a periodic interrupt may be generated by setting location 0x07 to 0x00 and setting location 0x0f to the timeout value.

In this example, an LED is turned off and the alarm timer (0x0f) is set to 0x20. On interrupt, location 0x07 is cleared and a new value of the timeout (10 seconds) is written to the timer alarm and the LED is turned on. On the next interrupt, location 0x07 is again cleared and a timeout of 20 seconds is written to the timer alarm and the LED is turned off.

As I write this, I realize that I don’t really know if seconds is selected as the timer mode, if the permissible timeout values are limited to 00 – 59 or include the full range of 00 – 99 and for hours, if the range is limited to 00 – 23 or if a timeout of 72 hours is permitted. The documentation is very clear in noting it will accommodate up to 99 days and seems to imply this is true for the other time quantities as well.

As an aside, the alarm register may also be configured to provide a “daily alarm”. As noted, the PCF8583 is quite a robust design.

```
// 8583_3.C
//
// Illustrates use of the timer in location 0x07.
//
// Sets timout for 20 seconds.  On timeout, turns on LED and sets
// timeout for 10 secs.  On timeout turns LED off.  The LED is on
// PORTD7.
//
// PIC16F877                        PCF8583
//
//  SCL (term 18) ----------- SCL (term 6) ----- To Other
//  SDA (term 23)------------ SDA (term 5) ----- I2C Devices
//
// I2C address is 0xa0 or 0xa2 depending on strapping of A0 (terminal
// 3). In this example, A0 is at logic zero.
//
// PCF8583                  PIC16F877
//
// /INT (term 7) ------------- RB0 (term 33)
//
// copyright, Peter H. Anderson, Baltimore, MD, Mar, '01

#case

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

```
#include <defs_877.h>
#include <lcd_out.h>
#include <i2c_mstr.h>

#define TRUE !0
#define FALSE 0

// routines used for PCF8583
void _8583_configure_control_register(byte control_reg);
  // 1 in bit 2 to enable alarm
void _8583_configure_alarm_register(byte alarm_control_reg);
  // 1 1 00  0  010
void _8583_zero_clock(void);
void _8583_set_timeout(byte seconds);
  // zero location 0x07 and set location 0x0f to timeout val

#define LED_DIR trisd7
#define LED_PIN portd7

int ext_int_occurred;   // global

void main(void)
{
   lcd_init();
   i2c_master_setup();

   pspmode = 0;
   not_rbpu = 0;

   ext_int_occurred = FALSE;  // defined globally

   LED_PIN = 0;
   LED_DIR = 0;

   _8583_zero_clock();
   // configure to timeout in 20 seconds
   _8583_set_timeout(0x20);   // note that this is BCD
   _8583_configure_alarm_register(0xc2);
   // alarm flag interrupt, timer alarm, seconds

   _8583_configure_control_register(0x04);      // enable alarm

   intf = 0;      // clear any interrupt
   intedg = 0;
   inte = 1;
   gie = 1;

   while(1)
   {

      if (ext_int_occurred)
      {
          while(gie)    // momentarily turn off interrupts
          {
             gie = 0;
          }
          ext_int_occurred = FALSE;

          if(!LED_PIN)  // if LED currently at zero
                  // make it a one for 10 seconds
          {
             _8583_set_timeout(0x10);     // note that this is BCD
             LED_PIN = 1;
          }
          else
          {
             _8583_set_timeout(0x20);
             LED_PIN = 0;
          }
          _8583_configure_alarm_register(0xc2);
          // alarm flag interrupt, timer alarm, seconds
          // 0xc3 for minutes, 0xc4 for hours, 0x05 for days
          _8583_configure_control_register(0x04);     // enable alarm
               // note that this also clears alarm flag
          gie = 1;
      } // end of if

   }
}

void _8583_configure_control_register(byte control_reg)
{
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x00);  // control register address
   i2c_master_out_byte(control_reg);
   i2c_master_stop();
}

void _8583_configure_alarm_register(byte alarm_control_reg)
{
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x08);  // control register address
   i2c_master_out_byte(alarm_control_reg);
   i2c_master_stop();
}

void _8583_set_timeout(byte seconds)
  // zero location 0x07 and set location 0x0f to timeout val
{
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x07);  // timer
   i2c_master_out_byte(0x00);
   i2c_master_stop();

   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x0f);  // timer alarm location
   i2c_master_out_byte(seconds);
   i2c_master_stop();
}
```

```
void _8583_zero_clock(void)
{
// set hours, mintues and secs to zero

   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x01); // address of first write
   i2c_master_out_byte(0x00); // hundreths of a second
   i2c_master_out_byte(0x00);
   i2c_master_out_byte(0x00); // 0x03
   i2c_master_out_byte(0x00);
      // 24 hour format - 0x80 for 12 hour time
   i2c_master_stop();
}

#int_ext ext_int_handler(void)
{
    ext_int_occurred = TRUE;   // flag that there has been an interrupt
}

#int_default default_int_handler(void)
{
}

#include <lcd_out.c>
#include <i2c_mstr.c>
```

Program 8583_4.c.

As noted above, the 8583 may be configured to count the number of events appearing at input OSC1 (term 1). In developing the following routines, I used the Morgan Logic Probe clock outputs at terminals 2 and 3 as a simple clock source.

In the event count mode, locations 0x01, 0x02 and 0x03 are the units, hundreds and ten thousands of events, respectively, all in BCD. Thus, the event counter is capable of counting up to 99 99 99 and thus has a modulus of one million

The control register at location 0x00 is configured for event count mode, clock on. If there is no need for an alarm, the alarm enable bit may be cleared to zero.

This in itself is quite valuable. In a weather station, a tipping bucket rain gauge might be interfaced with the PCF8583, with suitable de-bouncing, and the PIC might periodically read the count. Or a PIC might be deployed in an application where many counting processes are to be monitored and each PCF8583 might be periodically queried.

In the “dated” alarm, better termed, “alarm on match”, the corresponding alarm event count registers are at 0x09, 0x0a and 0x0b.

Here again, I may have gone a bit overboard in providing the ability to generate an alarm when the number of events is say, 34,424 more than it is currently. Again, perhaps, not all that practical, but the exercise was useful in understanding how to add an offset to a three byte BCD quantity short of converting the quantity to natural binary, adding the offset and the converting back to BCD. This is no small task without having 32 bit integers.

The count is stored in a structure consisting of three bytes; ten-thousands (tt), hundreds (hu) and units (un). Rather than increment this 34,424 times, the number of ten-thousands is isolated and the base quantity.tt is incremented;

```
   for (n = 0; n<tt; n++)
   {
         p->tt = increment_BCD(p->tt);
   }
```

The remaining number of hundreds is calculated and the quantity.hu is incremented that number of times. Note that is there is a rollover; the quantity.tt is incremented.

```
   for (n=0; n<hu; n++)
   {
         p->hu = increment_BCD(p->hu);
         if (p->hu == 0)
         {
               p->tt = increment_BCD(p->tt);
         }
   }
```

Finally, the quantity.hu is incremented by the remaining number of units;

```
   for (n=0; n<un; n++)
   {
         p->un = increment_BCD(p->un);
         if (p->un == 0)
         {
               p->hu = increment_BCD(p->hu);
               if (p->hu == 0)
               {
                    p->tt = increment_BCD(p->tt);
               }
         }
   }
```

Note that these bytes are incremented in a BCD fashion. That is, the low nibble is incremented and if the result is greater than 9, it is set to zero and the high nibble is incremented. If this result is greater than 9, it is set to zero. Thus, the return of a value of 00 indicates there was a rollover.

The important thing is that the 8583 has the ability to "interrupt on match" and this is a simple matter of writing the new alarm value to locations 0x09, 0x0a and 0x0b. I took a side trip of calculating this new alarm value by using rather simple concepts in working with a three byte quantity expressed in BCD. This actually turned out to simpler than I had thought.

In this routine, the alarm count is advanced by 50 and on interrupt, an LED on PORTD7 is toggled, the new alarm count is calculated and written to the alarm registers.

Note that when using the 8583 in the count mode, it is important to avoid an open condition on the OSC1 input. Thus, when using a momentary closure to ground, it is important to include a pull-up resistor to avoid the OSC1 input from being open. An "open" causes the 8583 to count the noise.

```
// 8583_4.C
//
// Illustrates use of the PCF8583 in a counting mode.  Use of the
// counting alarm (interrupt on match) is also illustrated.
//
// Counter is set to 00 00 00 and alarm to 00 00 50.  On interrupt
// alarm is advanced by 50 and an LED on PORTD is winked, and this
// process continues.  The 10 pps output of the Morgan Logic Probe
// (term 2) may be used as a counter source.
//
// Note that the counter and alarm values are each stored as three
// two digit BCD values in a structure.  Calculation of a new alarm
// value is implemented by manipulating this structure.
//
// This program closely follows program 8583_2.C except this program
// counts events rather than performing a timing function.
//
// PIC16F877                        PCF8583
//
//  SCL (term 18) ----------- SCL (term 6) ----- To Other
//  SDA (term 23)------------ SDA (term 5) ----- I2C Devices
//
// I2C address is 0xa0 or 0xa2 depending on strapping of A0 (terminal
// 3) In this example, A0 is at logic zero.
//
// PCF8583                             PIC16F877
//
// /INT (term 7) ------------- RB0 (term 33)
//
// copyright, Peter H. Anderson, Baltimore, MD, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <i2c_mstr.h>

#define TRUE !0
#define FALSE 0

struct BCD3
{
   byte tt;
   byte hu;
   byte un;
};

// routines used for PCF8583
void _8583_configure_control_register(byte control_reg);
void _8583_configure_alarm_register(byte alarm_control_reg);

void _8583_write_event_count(struct BCD3 *p);
void _8583_write_event_alarm(struct BCD3 *p);
void _8583_read_event_count(struct BCD3 *p);
void _8583_read_event_alarm(struct BCD3 *p);

void _8583_display_event_count(struct BCD3 *p);

void calc_new_BCD3(struct BCD3 *p, long num_events);
byte increment_BCD(byte x);
byte to_BCD(byte natural_binary);
byte to_natural_binary(byte BCD);

#define LED_DIR trisd7
#define LED_PIN portd7

byte ext_int_occurred;

void main(void)
{
   struct BCD3 count;
   byte line = 0;
   // note these numbers are in BCD
   count.tt=0x00;  count.hu=0x00;  count.un=0x00;

   lcd_init();
   i2c_master_setup();

   not_rbpu = 0;
   pspmode = 0;

   LED_DIR = 0;
   LED_PIN = 0;   // turn off LED

   ext_int_occurred = FALSE;
   _8583_write_event_count(&count);             // zero the counter
   calc_new_BCD3(&count, 50);
   lcd_clr_line(0);
   _8583_display_event_count(&count);
               // a check that the calc_new_BCD3
               // is working

   _8583_write_event_alarm(&count);
   _8583_configure_alarm_register(0x80 | 0x10); // event alarm
   _8583_configure_control_register(0x24); // event counter, alarm

   _8583_read_event_count(&count);
               // read and display the count

   lcd_clr_line(1);
   _8583_display_event_count(&count);
               // to verify circuit is working

   delay_ms(1000);

   intf = 0;      // kill any pending interrupts
   intedg = 0;    // negative edge
   inte = 1;
   gie = 1;

   while(1)
   {
      _8583_read_event_count(&count);  // continually read and
                                         //display the count
      lcd_clr_line(0);
      printf(lcd_char, "Count  ");
      _8583_display_event_count(&count);
      delay_ms(50);
      if(ext_int_occurred)                      // if an interrupt
      {
         while (gie)    // for the moment disable interrupts
         {
            gie = 0;
         }
         ext_int_occurred = FALSE;
         _8583_read_event_alarm(&count);  // read the alarm
         calc_new_BCD3(&count, 50); // 50 more counts
         _8583_write_event_alarm(&count); // new alarm value
         lcd_clr_line(1);
         printf(lcd_char, "Alarm ");
         _8583_display_event_count(&count);
         _8583_configure_alarm_register(0x80 | 0x10); // event alarm
         _8583_configure_control_register(0x24); // event, alarm
         LED_PIN = 1;   // momentarily wink the LED
         delay_ms(1000);
         LED_PIN = 0;
         gie = 1; // enable interrupts again
      }
   }
}

void _8583_display_event_count(struct BCD3 *p)
{
  lcd_hex_byte(p->tt);  // note that these values are stored in BCD
  lcd_hex_byte(p->hu);
  lcd_hex_byte(p->un);
}

void _8583_write_event_count(struct BCD3 *p)
{

   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x01); // address of first write
   i2c_master_out_byte(p->un);
   i2c_master_out_byte(p->hu);
   i2c_master_out_byte(p->tt);
   i2c_master_stop();
}

void _8583_write_event_alarm(struct BCD3 *p)
{

   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x09); // address of first write
   i2c_master_out_byte(p->un);
   i2c_master_out_byte(p->hu);
   i2c_master_out_byte(p->tt);
   i2c_master_stop();
}

void _8583_read_event_count(struct BCD3 *p)
{
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x01);

   i2c_master_repeated_start();
   i2c_master_out_byte(0xa1);
   p->un = i2c_master_in_byte(TRUE);      // units
   p->hu = i2c_master_in_byte(TRUE);      // hundreds
   p->tt = i2c_master_in_byte(FALSE);     // ten thousands

   i2c_master_stop();
}

void _8583_read_event_alarm(struct BCD3 *p)
{
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x09);

   i2c_master_repeated_start();
   i2c_master_out_byte(0xa1);
   p->un = i2c_master_in_byte(TRUE);
   p->hu = i2c_master_in_byte(TRUE);
   p->tt = i2c_master_in_byte(FALSE);
   i2c_master_stop();
}

void _8583_configure_control_register(byte control_reg)
{
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x00);  // control register address
   i2c_master_out_byte(control_reg);
   i2c_master_stop();
}

void _8583_configure_alarm_register(byte alarm_control_reg)
{
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x08);  // control register address
   i2c_master_out_byte(alarm_control_reg);
   i2c_master_stop();
}

void calc_new_BCD3(struct BCD3 *p, unsigned long num_events)
{
   byte tt, hu, un, n;

   tt = (byte) (num_events / 10000);
   num_events = num_events % 10000;
   hu = (byte) (num_events / 100);
   un = (byte) (num_events % 100);

   for (n = 0; n<tt; n++)
   {
         p->tt = increment_BCD(p->tt);
   }

   for (n=0; n<hu; n++)
   {
         p->hu = increment_BCD(p->hu);
         if (p->hu == 0)
         {
               p->tt = increment_BCD(p->tt);
         }
   }

   for (n=0; n<un; n++)
   {
         p->un = increment_BCD(p->un);
         if (p->un == 0)
         {
               p->hu = increment_BCD(p->hu);
               if (p->hu == 0)
               {
                    p->tt = increment_BCD(p->tt);
               }
         }
   }
}

byte increment_BCD(byte x)
{
      byte h, l;

      h = x / 16;
      l = x % 16;

      ++l;
      if (l > 9)
      {
         l = 0;
         ++h;
         if (h>9)
         {
             h = 0;
         }
      }
      return ((h * 16) + l);
}

byte to_BCD(byte natural_binary)
{
   return ( ((natural_binary/10) << 4) + natural_binary%10 );
}

byte to_natural_binary(byte BCD)
{
   return(  ((BCD >> 4) * 10) + (BCD & 0x0f)  );
}

#int_ext ext_int_handler(void)
{
    ext_int_occurred = TRUE;   // flag that there has been an interrupt
}

#int_default default_int_handler(void)
{
}

#include <lcd_out.c>
#include <i2c_mstr.c>
```

Program 8583_5.c.

As with the timing function, an interrupt may also be generated when the quantity in location 0x07 matches that in location 0x0e.

As with the previous routine, the control register at 0x00 is configured for event counting, counter on with the alarm bit enabled.

However, the alarm control register at 0x08 is configured for interrupt enabled, “timer alarm”. Use of the word “timer” is a bit confusing in this context, but the “timer” mode may be configured for units, hundreds, ten thousands or millions. Thus, if the mode is set for millions, location 0x07 will increment with each million event counts and when this matches the value in location 0x0e, interrupt will occur. Thus, an interrupt may be generated over the range of;

	01-99 events
	0100 – 9900 events
	010000 – 990000 events
or	01 million to 99 million events.

Note that if the value at locations 0x07 and 0x0e are left unchanged, an interrupt will occur every 100, 10,000, 1 million or 100 million events. Handy, eh!

In this routine, the alarm control register is configured for “timer” mode, units and location 0x07 is set to zero and 0x20 is written to location 0x0e. Thus, after 20 events, interrupt occurs, an LED is turned on, location 0x07 is cleared and the “timeout” value at location 0x0e is set to 0x10. Thus, interrupt occurs ten events later, the LED is turned off, etc.

```
// 8583_5.C
//
// Similar to routine 8583_3.C except PCF8583 is configured to count
// events.
//
// Illustrates use of the "timer" in location 0x07.
//
// Sets "timout" for 20 events.  On interrupt, turns on LED on PORTD7
// and sets next interrupt for 10 events.  On interrupt, turns LED off
// and sets  the next interrupt for 20 events.  The Morgan Logic Probe
// provides nominal one and ten pps outputs which may be used as a
// counter source.
//
// PIC16F877                         PCF8583
//
//  SCL (term 18) ----------- SCL (term 6) ----- To Other
//  SDA (term 23)------------ SDA (term 5) ----- I2C Devices
//
```

```
// I2C address is 0xa0 or 0xa2 depending on strapping of A0 (terminal
// 3) In this example, A0 is at logic zero.
//
// PCF8583                            PIC16F877
//
// /INT (term 7) ------------- RB0 (term 33)
//
// copyright, Peter H. Anderson, Baltimore, MD, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <i2c_mstr.h>

#define TRUE !0
#define FALSE 0

// routines used for PCF8583
void _8583_configure_control_register(byte control_reg);
  // 1 in bit 2 to enable alarm
void _8583_configure_alarm_register(byte alarm_control_reg);
  // 1 1 00  0  010
void _8583_zero_event_counter(void);
void _8583_set_event_trip(byte events);
  // zero location 0x07 and set location 0x0f to event trip

#define LED_DIR trisd7
#define LED_PIN portd7

int ext_int_occurred;   // global

void main(void)
{

   lcd_init();          // for possible debugging
   i2c_master_setup();

   pspmode = 0;
   not_rbpu = 0;

   LED_PIN = 0;
   LED_DIR = 0;

   ext_int_occurred = FALSE;  // defined globally

   _8583_zero_event_counter();
   // configure to timeout in 20 events
   _8583_set_event_trip(0x20);      // note that this is BCD
   _8583_configure_alarm_register(0xc1);
   // alarm flag interrupt, "timer" alarm, units

   _8583_configure_control_register(0x24);
                           // enable alarm, event counter
   intf = 0;      // clear any interrupt
   intedg = 0;    // 1 -> 0 causes interrupt
   inte = 1;
   gie = 1;

   while(1)
   {

      if (ext_int_occurred)
      {
          while(gie)    // momentarily turn off interrupts
          {
             gie = 0;
          }
          ext_int_occurred = FALSE;

          if(!LED_PIN)  // if RB5 currently at zero
                  // make it a one for 10 seconds
          {
             _8583_set_event_trip(0x10);  // note that this is BCD
             LED_PIN = 1;
          }
          else
          {
             _8583_set_event_trip(0x20);
             LED_PIN = 0;
          }
          _8583_configure_alarm_register(0xc1);
          // alarm flag interrupt, event mode, units
          // 0xc2 for hundreds, 0xc3 for 10,000, 0xc4 for millions
          _8583_configure_control_register(0x24);     // enable alarm
               // note that this also clears alarm flag
          gie = 1;
      } // end of if

   }
}

void _8583_configure_control_register(byte control_reg)
{
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x00);  // control register address
   i2c_master_out_byte(control_reg);
   i2c_master_stop();
}

void _8583_configure_alarm_register(byte alarm_control_reg)
{
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x08);  // control register address
   i2c_master_out_byte(alarm_control_reg);
   i2c_master_stop();
}

void _8583_set_event_trip(byte events)
```

```
  // zero location 0x07 and set location 0x0f to events
{
   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x07);  // timer
   i2c_master_out_byte(0x00);
   i2c_master_stop();

   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x0f);  // timer alarm location
   i2c_master_out_byte(events);
   i2c_master_stop();
}

void _8583_zero_event_counter(void)
{
// set units, hundreds and 10,000 to zero

   i2c_master_start();
   i2c_master_out_byte(0xa0);
   i2c_master_out_byte(0x01); // address of first write
   i2c_master_out_byte(0x00); // units
   i2c_master_out_byte(0x00); // hundreds
   i2c_master_out_byte(0x00); // 10,000
   i2c_master_stop();
}

#int_ext ext_int_handler(void)
{
    ext_int_occurred = TRUE;   // flag that there has been an interrupt
}

#int_default default_int_handler(void)
{
}

#include <lcd_out.c>
#include <i2c_mstr.c>
```