Tutorial by Example – Issue 1D

Introduction.

This is Issue 1D of "Tutorial by Example". The complete package now consists of Issues 1, 1A, 1B, 1C and 1D. Note that all routines have been consolidated into a single file routines.zip. There are now two sub-directories, one for the PIC16F877 and another for the PIC16F628.

This distribution includes RS232 routines to send and receive characters using "bit-bang". This technique is useful in cases where the PIC does not have a hardware UART, or an additional RS232 port is required. Further, using bit-bang, the data may be inverted such that the PIC may "usually" be interfaced with a PC Com Port with no intermediate level shifter such as a DS275 or MAX232.

This distribution also presents routines for the interfacing with an X-Y keypad which uses the interrupt on PortB change feature. This was debugged on a PIC16F877 using the ICD and then mapped over to a PIC16F628.

The distribution also includes a number of weather type applications including measuring barometric pressure using a Motorola MPX4115A pressure sensor, determining wind direction using a special potentiometer which consists of two wipers positioned 180 out of phase with one another and measuring relative humidity using a Honeywell HIH-3610 RH sensor and a Dallas DS2438 1-W Battery Monitor. The DS2438 is an amazing device capable of measuring temperature, its own supply voltage and the voltage appearing at an input which may be as high as 10.0 Volts even when the power is as low as 3.0.

Also included is an extensive discussion of the Dallas DS2435 Battery Monitor which may be used to measure temperature, elapsed time, log temperature in a histogram fashion. The DS2435 also includes 32 EEPROM bytes.

A program to use flash memory is presented. One problem with flash memory is the limited endurance and this program presents a technique to move the actual addresses of the storage so as to achieve an endurance improvement of 32.

Next Distribution.

The next distribution will be in early August.

Advertisement.

Over the summer, I plan to prepare a collection of routines for "Little PICs" including 12-bit devices PIC12C50X, 12CE51X, 16C505, 12HV540 and 14-bit devices 12C67X and 12CE67X. This will probably run about 125 pages, will be distributed via e-mail and will be priced at about $12.00. I will be using the Advanced Transdata RICE-17 Emulator to debug these routines.

In the fall, I will begin a similar tutorial on the 16-bit PIC18C family and for the moment will be using the RICE-17. Right now, I can't imagine just why I would want such power as that provided by the PIC18C, but there was a time when I couldn't see why anyone would ever need a 180K floppy. This will probably be similar to this effort where I develop a bit at a time and the price is adjusted with each new distribution.

So, save those tax rebates!

Bit Bang Serial (9600 Baud).

Program SER_BB.C illustrates a bit bang implementation of sending and receiving RS232 at 9600 baud for a processor running with a 4.0 MHz crystal or resonator. That is, f_osc / 4 = 1.0 MHz.

There are a number of good reasons to use "bit-bang". Not all processors have a hardware UART. For example, I often use the PIC12C672 as it is a 2K device with adequate RAM in a 8-pin package. In some cases it may be desirable to have multiple serial ports and the 16F87X family provides only the single UART. But, one big reason is that one may be able to eliminate a level shifter (MAX232, DS275 or similar) in directly interfacing a PIC with a PC COM Port.

For RS232, recall that on the TTL side, the idle state is a logic one (near +5 VDC). The start bit is a logic zero (near ground) for one bit time (104 us for 9600 baud) followed by the eight data bits and then back to idle (near +5 VDC). Note that this is the output and the expected input of the hardware UART. I refer to these levels as "True" or "Non-inverted".

However, in transmitting to a distant point, a level shifter such as a MAX232 or similar is used to provide a greater swing and also provide hysterisis, which also inverts the logic levels. Thus, on the communications side, a logic one is less than minus 3 VDC and a logic zero is greater than +3 VDC. I refer to this as "inverted" logic levels. (The negative logic has a history going back to the Bell System which used a standard –48 VDC office battery).

On the receive side, the RS232 levels are converted back to TTL.

However, it is possible to perhaps eliminate the intermediate level shifter by sending the TTL inverted. That is, the idle is near ground (close to the less than minus 3.0), the start bit is near +5 VDC which meets the greater than +3 VDC requirement and the data bits are then transmitted inverted, followed by the idle (near ground). My confidence in taking this short cut was considerably bolstered by the introduction of the popular BasicX BX24 which does not provide a level shifter and this seems to work and I have introduced a number of kits that use the same technique and have had no negative feedback. That is, PC Com ports seem to recognize near ground as being an RS232 logic one although it is a tad higher than the specified –3 VDC. Thus, in most cases, you can simplify the circuitry in transmitting to a PC Com port by simply sending the data as inverted. (Again, note that this inversion is not possible with the hardware UART. Indeed, one can invert the data, but the idle condition of the UART is near +5 VDC).

When using a PIC to receive data from a PC, recognize the outputs of the PC are probably good RS232 levels. That is, the idle logic one is less than –3.0 VDC (usually –8.0 VDC and a logic one is greater than +3.0 VDC (typically +8.0 VDC). Here again, the PIC can interpret a level near ground (less than minus 3.0 VDC) as being a logic one and a level near +5 VDC (greater than 3.0 VDC) as a logic zero. However, note that voltages of –8.0 and +8.0 VDC are considerably outside the specification for levels appearing at a PIC input. I usually use a series 22K resistor which in conjunction with the internal protection diodes clips these levels to close to TTL levels.

Thus, bit-bang, permits a direct interface with a PC COM port. Send the data inverted and similarly receive the data with a series 22K resistor.

Note that the disadvantage of "bit bang" is timing. When sending a byte, timing is critical and any interrupts must be turned off. When awaiting the receipt of a byte, the PIC must "camp on" the input to avoid missing the byte.

In function ser_bb_init(), txdata and rxdata are configured as an output and input, respectively. If INV is #defined, the output idle is a TTL logic zero (RS232 logic one).

```
void ser_bb_init(void) // sets TxData in idle state
{
#ifdef INV    // idle is TTL logic zero
#asm
```

```
   BCF STATUS, RP0
   BCF PORTA, TxData
   BSF STATUS, RP0
   BCF TRISA, TxData // TxData is an oputput
   BSF TRISA, RxData // RxData is an input
   BCF STATUS, RP0
#endasm

#else
#asm
   BSF STATUS, RP0
   BCF PORTA, TxData // idle state is TTL logic one
   BSF STATUS, RP0
   BCF TRISA, TxData
   BSF TRISA, RxData
   BCF STATUS, RP0
#endasm
#endif
}
```

In function ser_bb_char() a character is output at 9600 baud. The carry bit is used to implement a nine bit transfer, start bit plus the eight data bits. If INV is #defined, the CY is initially set to a TTL logic one (RS232 logic zero) for the start bit and each of the eight bits are shifted in to the carry bit using the RRF command with the opposite state of the bit being output on TxData. Note that for non-inverted, the logic state output is the same as the state of the bit. The loop time is 4 + 1 + 1 + 31 * 3 + 1 + 1 + 2 or 104 us (1/9600). Clearly, this must be modified when using a clock other than 4.0 MHz.

```
void ser_bb_char(byte ch) // serial output 9600 baud
{
   byte n, dly;
     // start bit + 8 data bits
#ifdef INV    // idle is TTL logic zero
#asm
   BCF STATUS, RP0
   MOVLW 9
   MOVWF n
   BCF STATUS, C

SER_BB_CHAR_1:

   BTFSS STATUS, C            // 4 ~
   BSF PORTA, TxData
   BTFSC STATUS, C
   BCF PORTA, TxData
   MOVLW 31                   // 1 ~
   MOVWF dly                  // 1 ~

SER_BB_CHAR_2:
   DECFSZ dly, F              // 31 * 3 ~
   GOTO SER_BB_CHAR_2         // 1 ~
   RRF ch, F                  // 1 ~
   DECFSZ n, F                // 1 ~
   GOTO SER_BB_CHAR_1   // next bit // 2 ~

   BCF PORTA, TxData    // idle between characters
   CLRWDT
   MOVLW 96
```

```
   MOVWF dly

SER_BB_CHAR_3:
   DECFSZ dly, F
   GOTO SER_BB_CHAR_3
   CLRWDT
#endasm

#else  // idle is TTL logic one

// non-inverted is the same as inverted, except that the logic output is the same as the
// bit state.
```

Function ser_bb_get_ch() provides the capability of fetching a character. If no character is fetched within the specified wait time, 0xff is returned.

When INV is #defined, the program goes into a nominal 10 us loop and exits the loop if a start bit, logic one TTL state, is detected. The program delays for 1.5 bit times such that the first sample occurs near the middle of the receipt of the first bit. The program then fetches each bit in turn, interpreting the bit state as the opposite of the logic actually received on RxData.

If, INV is not #defined, the routine is precisely the same except the wait loop is exited when a logic zero TTL state is detected and each bit state is interpreted as the same as the logic appearing on RxData.

```
byte ser_bb_get_ch(long t_wait)
// returns character.  If no char received within t_wait ms, returns 0xff.
#ifdef INV
{
   byte one_ms_loop,  ser_loop, ser_data, ser_time;
   do
   {
        one_ms_loop = 100; // 100 times 10 usecs
#asm

SCAN_1:
    CLRWDT
    NOP
    NOP
    NOP
    NOP
    BTFSC PORTA, RxData // check serial in - for inverted data
    GOTO  SERIN_1  // if start bit
    DECFSZ  one_ms_loop, F
    GOTO SCAN_1
#endasm
   }while(--t_wait);
   return(0xff);

#asm
SERIN_1:

    MOVLW 8
    MOVWF ser_loop
    CLRF ser_data

    MOVLW 51          // delay for 1.5 bits
```

```
    MOVWF ser_time  // 3 + 51 * 3 = 156 usecs for 9600

SERIN_2:
    DECFSZ ser_time, F
    GOTO SERIN_2

SERIN_3:
    BTFSS PORTA, RxData
    BSF STATUS, C  // reverse these for non inverted
    BTFSC PORTA, RxData
    BCF STATUS, C

    RRF ser_data, F

    MOVLW 23        // one bit delay
    MOVWF ser_time  // 10 + 23 * 4 =  102 usecs

SERIN_4:
    CLRWDT
    DECFSZ ser_time, F
    GOTO SERIN_4

    DECFSZ ser_loop, F // done?
    GOTO SERIN_3 // get next bit

    MOVLW 10
    MOVWF ser_time  // wait for at least 1/2 bit

SERIN_5:
    CLRWDT
    DECFSZ ser_time, F
    GOTO SERIN_5
#endasm
   return(ser_data);
}

#else
// non-inverted
```

Program ser_bb.c also includes functions ser_bb_new_line() which simply sends the CR and LF characters and ser_bb_out_str which outputs a null terminated string. The user may either use printf(ser_bb_char, …) for other output formats or add such functions as ser_hex_byte(), ser_dec_byte() which are implemented in much the same manner as in lcd_out.c except ser_bb_char() vs lcd_char() is used to output the character.

The following two functions are also included in ser_bb.c.

```
byte ser_bb_get_str_1(byte *p_chars, long t_wait_1, long t_wait_2, byte term_char);
byte ser_bb_get_str_2(byte *p_chars, long t_wait_1, long t_wait_2, byte num_chars);
```

Function ser_bb_get_str_1() fetches characters until the defined terminal character is received. The function aborts if the specified t_wait_1 is exceeded with no character being received or if the specified t_wait_2 is exceeded in waiting for any subsequent character.

Function ser_bb_str_2 is similar except that it receives characters until the specified number of characters have been received.

Only the main() is shown below. Note that I used a PC Com Port configured direct, 9600, 8, N, 2 with no flow control for testing. The program simply echoes the typed string when a new line character is sent. The PIC sends a "!" every ten seconds if no character is received.

```
// SER_BB.C
//
// Illustrates RS232 communication using bit-bang.
//
// Program listens for a string terminated with 0x0d from PC Com Port and
// then sends the string to the PC Com Port.
//
//  PC Com Port                         PIC16F877
//
// Tx (term 3) ------- 22K ----------> RA1 (RxData)
// Rx (term 2) <---------------------- RA0 (TxData)
//
// copyright, Peter H Anderson, Baltimore, MD, June, '01

#case

#device PIC16F877 *=16 ICD=TRUE
#include <a:\defs_877.h>
#include <a:\delay.c>

#define FALSE 0
#define TRUE !0

void ser_bb_init(void);
void ser_bb_char(byte ch);
void ser_bb_new_line(void);

void ser_bb_out_str(byte *s);

byte ser_bb_get_str_1(byte *p_chars, long t_wait_1, long t_wait_2, byte term_char);
byte ser_bb_get_str_2(byte *p_chars, long t_wait_1, long t_wait_2, byte num_chars);
byte ser_bb_get_ch(long t_wait);

#define INV  // inverted bit-bang

#define TxData 0 // PortA.0
#define RxData 1 // PortA.1

void main(void)
{
    byte s[20];
    pcfg3 = 0;  pcfg2 = 1;   pcfg1 = 1;  // configure A/D as 0/0
    ser_bb_init();
    while(1)
    {
       if (ser_bb_get_str_1(s, 10000, 1000, 0x0d))
       {
          ser_bb_out_str(s);
          ser_bb_new_line();
       }
       else
```

```
      {
         ser_bb_char('!');   // to show something is going on
      }
   }
  }
}
```

Program ser_bb.c - Final Notes.

I recently used the transmit routines in a temperature monitor and it did not work. I discovered the variables associated with function ser_bb_char(), which was near the end of my very long program were being placed in RAM bank 1. Thus, each time ch, dly and n were being accessed, the program was setting the RP0 bit of STATUS which of course destroyed my 104 us timing. My non elegant corrective action was to declare variables _ch, dly and n as global. In serbbchar(), the passed byte ch was copied to _ch and then variables _ch, _dly and _n were used. Its safe to say that CCS assigns variables as they appear, beginning with globals, and thus I forced the variables to RAM bank 0. There may be more elegant solutions, but this did the job for me.

In functions ser_bb_char() and ser_bb_get_char(), I used the preprocessor directives #ifdef, #else and #endif. Although the only differences between the cases of inverted and non-inverted are a few lines, I opted to duplicate virtually the entire routine in each case. My feeling is that life is complex enough without all kinds of #ifdefs appearing again and again in code. My approach simply chews up paper and disk space, but uses no more PIC resources.

The question often arises as to whether the internal clock associated such devices as the PIC12C50X, 12C67X and 16F628 is accurate enough to assure accurate transmission and receipt of RS232. My reading of the specs is no. I do use it for quick projects and for demos at school where failure doesn't really matter, but I have never sold a kit that uses the internal clock.

Interfacing with a Key Pad.

The following discussion focuses on a PIC interface with a 4 X 4 or similar keypad. When a key is pressed there is a closure between the column and the row. These keypads are readily available from such surplus suppliers as BG Micro.

This is a series of four programs which build on one another to finally implement a Serial Key Pad Decoder with a 20 byte buffer. The first three are implemented using the ICD with a PIC16F877. The final design is implemented on a PIC16F628. One intent in presenting this series is to illustrate how much of a design may be done in the ICD environment leaving very little to chance in finally porting the code to the PIC16F628. Program keypad_1.c simply reads a key and displays it. Program keypad_2.c extends this to include a 20-byte circular buffer where each valid key press is placed in a buffer to be read by the main program at its convenience. Program keypad_3.c extends this such that on receipt of an RS232 serial character, the content of the buffer is sent to an interfacing processor using the hardware UART. Finally, program keypad_4.c is an implementation using a PIC16F628.

Program keypad_1.c.

In this implementation, the rows are connected to the upper nibble of PORTB which is configured as inputs with the internal weak pull-up resistors enabled. The lower nibble of PORTB is configured as output zeroes. Thus, when no key is depressed, the inputs on the high nibble are all at logic one.

When a key is depressed, the corresponding row goes to a logic zero causing an interrupt on change. The program then determines the column and the row by sequentially outputting patterns to the columns with a single zero; 0x0e, 0x0d, 0x0b and 0x07 and reading the row looking at pattern which contains one zero; 0x0e, 0x0d, 0x0b and 0x07. For example if pattern 0x07 is output to the columns and pattern 0x0e is read, the key is col 3, row 0. Note that if no combination of

outputting patterns with a single 0 results in an input pattern with a single zero as would be the case if no key were pressed or if multiple keys were pressed, the global variable key_pressed is left as FALSE in the interrupt service routine. Otherwise, global variable key_pressed is set to TRUE and key to 4 * row + col.

In main, if global variable key_pressed is seen as being TRUE, interrupts are momentarily disabled and and array is used to display the corresponding character on the LCD.

Note that I opted to perform the scan function in the interrupt service routine. There is bound to be some switch bounce when a key is depressed and I implemented debounce as a simple delay function. In this application, I may well have been able to do this by simply using the familiar delay_ms() function. However, I opted to code a separate routine; debounce(). The reason is that the PCM compiler does not support recursion and when using interrupts one might be tricked into recursion. For example, assume the main was executing delay_ms() at the time the interrupt on change on PORTB occurred. If the interrupt service routine then called delay_ms(), we have a case of recursion.

In placing the debounce() is the ISR, its important to note that GIE = 0 and thus no other interrupts can be processed during this wait time.

Note that the ICD uses PORTB.6 and PORTB.7 Thus, in debugging this program, I initially used only rows 0 and 1 (on PORTB4 and 5). The code is slightly different and this may be selected by #define _2_ROW. Once I got this working I undefined _2_ROW and programmed the PIC in the non-debug mode.

This program merely fetches a single key and displays the assigned character.

This routine was "new to me" and as with all such routines, I felt quite elated when it worked. But, then I pause to wonder how bulletproof it really is. How many trouble conditions have I overlooked. Thus, take a good hard look at this code before committing it blindly to an expensive development.

```
// KeyPad_1.C (PIC16F877)
//
// Illustrates an interface with a 4X4 key pad.  Program continually
// loops, checking if key_present and is so displays the key_character
// assigned to the depressed key.
//
// Nothe that the reading of the key is performed in the interrupt service
// routine.
//
// PIC16F877        KeyPad
//
// RB7 <--------- Row 3                       1 2 3 A
// RB6 <--------- Row 2                       4 5 6 B
// RB5 <--------- Row 1                       7 8 9 C
// RB4 <--------- Row 0                       * 0 # D
//                                            Key Pad Layout
// RB3 ---------> Col 3
// RB2 ---------> Col 2
// RB1 ---------> Col 1
// RB0 ---------> Col 0
//
// copyright, Peter H. Anderson, Baltimore, MD, May, '01

#case

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

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

#define TRUE !0
#define FALSE 0

// #define _2_ROW // See Text

byte get_key(byte *p_row, byte *p_col);
void debounce(byte ms);       // separte delay routine to avoid recursion

byte key_present, key;

void main(void)
{
   byte const key_char[16] = { '1', '2', '3', 'A', '4', '5', '6', 'B',
                               '7', '8', '9', 'C', '*', '0', '#', 'D'};
   lcd_init();
   not_rbpu = 0;
   pspmode = 0;
   TRISB = 0xf0;  // high nibble are row inputs, low are col outputs
   PORTB = 0x00;

   key_present = FALSE;

   rbif = 0;
   rbie = 1;
   gie = 1;

   while(1)
   {
      if (key_present)
      {
         while(gie)
         {
            gie = 0;
         }

         lcd_char(key_char[key]);
         key_present = FALSE;
         rbif = 0;      // not really necessary
         rbie = 1;      // ''
         gie = 1;
      }
   } // of while 1
}

void debounce(byte ms)        // note that a separate delay function was used to
{                             // avoid inadvertent recursion.
   byte t;
   do
   {
    t = 100;                        // about 100 * 10 us
#asm
      BCF STATUS, RP0
DELAY_10US_1:
      CLRWDT
```

```
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      DECFSZ t, F
      GOTO DELAY_10US_1
#endasm

   } while(--ms);
}

byte get_key(byte *p_row, byte *p_col)
{
   byte row, col, in_patt;
   byte const patt[4] = {0x0e, 0x0d, 0x0b, 0x07};

   for (col = 0; col < 4; col++)
   {
      TRISB = (TRISB & 0xf0) | patt[col];
#ifdef _2_ROW
      in_patt = (PORTB >> 4) & 0x03;
      if(in_patt != 0x03)     // both high
#else
      in_patt = (PORTB >> 4) & 0x0f;
      if(in_patt != 0x0f)     // all high
#endif
      {
         for (row = 0; row < 4; row++)
         {
#ifdef _2_ROW           // if using only rows 0 and 1
            if (in_patt == (patt[row] & 0x03))
#else
            if (in_patt == (patt[row] & 0x0f))
#endif
            {
               *p_row = row;
               *p_col = col;
               TRISB = TRISB & 0xf0;      // restore ground to all col outputs
               return(TRUE);
            }
         }
      }
   }
   TRISB = TRISB & 0xf0;
   return(FALSE);
}

#int_rb rb_int_handler(void)
{
   byte row, col;
   debounce(50);
   if (get_key(&row, &col) == TRUE)
   {
      key_present = TRUE;
      key = 4 * row + col;
```

```
   }
}

#int_default default_interrupt_handler()
{
}

#include <lcd_out.c>
```

Program keypad_2.c.

Program keypad_2.c is similar in concept except that it includes a 20-byte circular buffer to store depressed keys. This allows the processor to be performing other tasks and take a look at the depressed keys when it has the time. In this example, the "other task" is simply a five second delay. It is important to note that the program spends some 60 ms in the interrupt service routine associated with each depression of the key and each release of the key and this length of time may restrict the nature of these other tasks. However, even if the other task include such time critical routines as the Dallas 1-W, interrupts may be turned off for the 60 us associated with writing a reading a bit.

The circular buffer "keys" is implemented in much the same manner as was done with the receipt of serial characters. Initially, the get_index and put_index are set to zero. In the interrupt on portb change service routine, if a valid key is detected, it is placed in the buffer and the put_index is incremented. If the put_index falls off the bottom of the buffer, it is reset to zero. Thus, the term circular. Note that in implementing this I did not consider the “buffer full” condition. This would be detected by first incrementing the put_index and then testing if it is equal to the get_index.

In main(), if the get_index is not equal to the put_index, there is at least one key in the buffer. Each key is read and displayed and get_index is incremented until get_index is eqaul to the put_index.

```
// KeyPad_2.C (PIC16F877)
//
// Illustrates an interface with a 4X4 key pad with a 20 byte buffer.  Program
// continually loops spending most of the time in a five second delay.  On interrupt
// on change, the interrupt service routine adds a valid key to a 20 byte circular
// buffer.  Periodically, the main checks to see if any keys are in the buffer and
// if so fetches the keys from the 20-byte buffer and displays them on the LCD.
//
// PIC16F877   KeyPad                     C0 C1 C2 C3
//
// RB7 <--------- Row 3        Row 0      1  2  3  A
// RB6 <--------- Row 2        Row 1      4  5  6  B
// RB5 <--------- Row 1        Row 2      7  8  9  C
// RB4 <--------- Row 0        Row 3      *  0  #  D
//                                      Key Pad Layout
// RB3 ---------> Col 3
// RB2 ---------> Col 2
// RB1 ---------> Col 1
// RB0 ---------> Col 0
//
// copyright, Peter H. Anderson, Baltimore, MD, May, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
```

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

#define TRUE !0
#define FALSE 0

#define _2_ROW // See Text

byte get_key(byte *p_row, byte *p_col);
void debounce(byte ms);  // separte delay routine to avoid recursion

byte keys[20], get_index, put_index;

void main(void)
{
   byte const key_char[16] = { '1', '2', '3', 'A', '4', '5', '6', 'B',
                               '7', '8', '9', 'C', '*', '0', '#', 'D'};
   byte key;
   lcd_init();
   not_rbpu = 0;
   // pspmode = 0; // PORTD not used
   TRISB = 0xf0; // high nibble are row inputs, low are col outputs
   PORTB = 0x00;

   put_index = 0;
   get_index = 0;

   rbif = 0;
   rbie = 1;
   gie = 1;

   while(1)
   {
      while (get_index != put_index)
      {
         key = keys[get_index];
         lcd_char(key_char[key]);
         ++get_index;
         if (get_index > 19)
         {
            get_index = 0;
         }
      }
      delay_ms(5000);   // delay for five seconds
   } // of while 1
}

void debounce(byte ms)  // note that a separate delay function was used to
{       // avoid inadvertent recursion.
   byte t;
   do
   {
    t = 100;    // about 100 * 10 us
#asm
      BCF STATUS, RP0
DELAY_10US_1:
      CLRWDT
      NOP
```

```
      NOP
      NOP
      NOP
      NOP
      NOP
      DECFSZ t, F
      GOTO DELAY_10US_1
#endasm

   } while(--ms);
}

byte get_key(byte *p_row, byte *p_col)
{
   byte row, col, in_patt;
   byte const patt[4] = {0x0e, 0x0d, 0x0b, 0x07};

   for (col = 0; col < 4; col++)
   {
      TRISB = (TRISB & 0xf0) | patt[col];
#ifdef _2_ROW
      in_patt = (PORTB >> 4) & 0x03;
      if(in_patt != 0x03)     // both high
#else
      in_patt = (PORTB >> 4) & 0x0f;
      if(in_patt != 0x0f)     // all high
#endif
      {
         for (row = 0; row < 4; row++)
         {
#ifdef _2_ROW
            if (in_patt == (patt[row] & 0x03))
#else
            if (in_patt == (patt[row] & 0x0f))
#endif
            {
               *p_row = row;
               *p_col = col;
               TRISB = TRISB & 0xf0; // restore ground to all col outputs
               return(TRUE);
            }
         }
      }
   }
   TRISB = TRISB & 0xf0;
   return(FALSE);
}

#int_rb rb_int_handler(void)
{
   byte row, col;
   debounce(50);
   if (get_key(&row, &col) == TRUE)
   {
      keys[put_index] = 4 * row + col;
      ++put_index;
      if (put_index > 19)
```

```
      {
        put_index = 0;
      }
   }
}

#int_default default_interrupt_handler()
{
}

#include <lcd_out.c>
```

Program keypad_3.c.

Program keypad_3.c is a modification of keypad_2.c. Keys are added to the buffer in the portb change interrupt service routine. When a serial character is received from a master, the PIC sends the current content of the buffer using the hardware UART. Note that the portb interrupt on change is not disabled while sending the keys in the buffer. This may result in delays of nominally 60 ms if a key is pressed while sending the data. This should not pose a problem when sending serial data using the hardware UART. It would cause a problem if the sending of the serial data were implemented using bit-bang where the timing for each bit is critical.

```
// KeyPad_3.C (PIC16F877)
//
// Illustrates an interface with a 4X4 key pad with a 20 byte buffer.  Program
// continually loops spending most of the time in a five second delay.  On interrupt
// on change, the interrupt service routine adds a valid key to a 20 byte circular
// buffer.  On receipt of a serial character from a master processor the content of the
// key buffer is sent to the master at 9600 baud.  This is terminated by a new line
// <CR><LF>.
//
//                  PIC16F877       KeyPad
//                                                    C0 C1 C2 C3
//                      RB7 <--------- Row 3          R0 1  2  3  A
//                      RB6 <--------- Row 2          R1 4  5  6  B
//                      RB5 <--------- Row 1          R2 7  8  9  C
//                      RB4 <--------- Row 0          R3 *  0  #  D
//                                                       Key Pad Layout
//                      RB3 ---------> Col 3
//                      RB2 ---------> Col 2
//                      RB1 ---------> Col 1
//                      RB0 ---------> Col 0
//
// BX24
//
//   -------->  RX/RC7
//   <--------  TX/RC6
//
// copyright, Peter H. Anderson, Baltimore, MD, May, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <ser_87x.h> // serial routines
```

```
#include <delay.h>

#define TRUE !0
#define FALSE 0

#define _2_ROW // See Text

byte get_key(byte *p_row, byte *p_col);
void debounce(byte ms);  // separate delay routine to avoid recursion

byte keys[20], get_index, put_index, rda_int_occ;

void main(void)
{
   byte const key_char[16] = { '1', '2', '3', 'A', '4', '5', '6', 'B',
                               '7', '8', '9', 'C', '*', '0', '#', 'D'};
   byte key, ch;
   asynch_enable();     // set up UART for 9600 baud
   not_rbpu = 0;
   // pspmode = 0; // PORTD not used
   TRISB = 0xf0; // high nibble are row inputs, low are col outputs
   PORTB = 0x00;

   put_index = 0;
   get_index = 0;

   ch = RCREG;  // get any junk that may be in the buffer
   ch = RCREG;

   rda_int_occ = FALSE;

   rcif = 0;  // receive data interrupt
   rcie = 1;
   peie = 1;

   rbif = 0;
   rbie = 1;

   gie = 1;

   while(1)
   {
      if (rda_int_occ)        // if a serial character was received
      {
         while (get_index != put_index)
         {
            key = keys[get_index];
            ser_char(key_char[key]);
            ++get_index;
            if (get_index > 19)
            {
               get_index = 0;
            }
         }
         ser_char(13);  // terminate the string with new line
         ser_char(10);
         rda_int_occ = FALSE;
```

```
     }
   } // of while 1
}

void debounce(byte ms)  // note that a separate delay function was used to
{       // avoid inadvertent recursion.
   byte t;
   do
   {
    t = 100;    // about 100 * 10 us
#asm
      BCF STATUS, RP0
DELAY_10US_1:
      CLRWDT
      NOP
      NOP
      NOP
      NOP
      NOP
      NOP
      DECFSZ t, F
      GOTO DELAY_10US_1
#endasm

   } while(--ms);
}

byte get_key(byte *p_row, byte *p_col)
{
   byte row, col, in_patt;
   byte const patt[4] = {0x0e, 0x0d, 0x0b, 0x07};

   for (col = 0; col < 4; col++)
   {
      TRISB = (TRISB & 0xf0) | patt[col];
#ifdef _2_ROW
      in_patt = (PORTB >> 4) & 0x03;
      if(in_patt != 0x03)     // both high
#else
      in_patt = (PORTB >> 4) & 0x0f;
      if(in_patt != 0x0f)     // all high
#endif
      {
         for (row = 0; row < 4; row++)
         {
#ifdef _2_ROW
            if (in_patt == (patt[row] & 0x03))
#else
            if (in_patt == (patt[row] & 0x0f))
#endif
            {
               *p_row = row;
               *p_col = col;
               TRISB = TRISB & 0xf0; // restore ground to all col outputs
               return(TRUE);
            }
         }
```

```
      }
   }
   TRISB = TRISB & 0xf0;
   return(FALSE);
}

#int_rb rb_int_handler(void)
{
   byte row, col;
   debounce(50);
   if (get_key(&row, &col) == TRUE)
   {
      keys[put_index] = 4 * row + col;
      ++put_index;
      if (put_index > 19)
      {
        put_index = 0;
      }
   }
}

#int_rda rda_interrupt_handler(void)
{
   byte ch;
   rda_int_occ = TRUE;
   ch = RCREG;
}

#int_default default_interrupt_handler()
{
}

#include <ser_87x.c>
#include <delay.c>
```

Program keypad_4.c.

Finally, this is mapped over to the PIC16F628. Note that the UART RX and TX use RB1 and RB2, respectively. Thus, the column outputs were moved from the lower nibble of PORTB to the lower nibble of PORTA. Note that as these PORTA bits are shared with the comparator and voltage regulator modules, their use as general purpose IOs must be configured.

```
   CMCON = 0x07;  // comparators off
   vroe = 0;   // voltage ref disabled on RA2
```

My point in presenting this relatively complex development is to illustrate a design where the In Circuit Debugger is used for nearly all of the testing leaving only a few minor changes when the code is ported to another processor.

```
// KeyPad_4.C (PIC16F628)
//
// Illustrates an interface with a 4X4 key pad with a 20 byte buffer.  Program
// continually loops. On interrupt on change, the interrupt service routine adds
// a valid key to a 20 byte circular buffer.  On receipt of a serial character from
// a master processor, the keypad decoder PIC sends the content of the key buffer to
// the master at 9600 baud.
//
```

```
//
//                              PIC16F877   KeyPad
//                                                      C0 C1 C2 C3
//                              RB7 <--------- Row 3    R0  1  2  3  A
//                              RB6 <--------- Row 2    R1  4  5  6  B
//                              RB5 <--------- Row 1    R2  7  8  9  C
//                              RB4 <--------- Row 0    R3  *  0  #  D
//                                                        Key Pad Layout
//                              RA3 ---------> Col 3
//                              RA2 ---------> Col 2
//                              RA1 ---------> Col 1
//                              RA0 ---------> Col 0
//
// BX24
//
//       -------->  RX/RB1
//       <--------  TX/RB2
//
// copyright, Peter H. Anderson, Baltimore, MD, May, '01

#case

#device PIC16F628 *=16

#include <defs_628.h>
#include <ser_628.h> // serial routines
#include <delay.h>

#define TRUE !0
#define FALSE 0

byte get_key(byte *p_row, byte *p_col);
void debounce(byte ms);  // separate delay routine to avoid recursion

byte keys[20], get_index, put_index, rda_int_occ;

void main(void)
{
   byte const key_char[16] = { '1', '2', '3', 'A', '4', '5', '6', 'B',
                               '7', '8', '9', 'C', '*', '0', '#', 'D'};
   byte key, ch;

   CMCON = 0x07;  // comparators off
   vroe = 0;   // voltage ref disabled on RA2

   asynch_enable();
   not_rbpu = 0;

   TRISB = 0xff; // high nibble are row inputs, low nibble is TX and RX
   PORTA = 0x00;
   TRISA = 0xf0; // column outputs

   put_index = 0;
   get_index = 0;

   ch = RCREG;  // get any junk that may be in the buffer
   ch = RCREG;
```

```
   rda_int_occ = FALSE;

   rcif = 0;  // receive data interrupt
   rcie = 1;
   peie = 1;

   rbif = 0;
   rbie = 1;

   gie = 1;

   while(1)
   {
      rcie = 1;
      peie = 1;
      rbie = 1;
      gie = 1;
#asm
      CLRWDT
#endasm
      if (rda_int_occ)
      {
            delay_ms(10);          // a bit of a delay for if the interface is a Basic
Stamp
         while (get_index != put_index)
         {
            key = keys[get_index];
            ser_char(key_char[key]);
            ++get_index;
            if (get_index > 19)
            {
               get_index = 0;
            }
         }
         ser_char(13);  // terminate the string with new line
         ser_char(10);
         rda_int_occ = FALSE;
     }
   } // of while 1
}

void debounce(byte ms)  // note that a separate delay function was used to
{                              // avoid inadvertent recursion.
   byte t;
   do
   {
    t = 100;    // about 100 * 10 us
#asm
      BCF STATUS, RP0
DELAY_10US_1:
      CLRWDT
      NOP
      NOP
      NOP
      NOP
      NOP
```

```
      NOP
      DECFSZ t, F
      GOTO DELAY_10US_1
#endasm

   } while(--ms);
}

byte get_key(byte *p_row, byte *p_col)
{
   byte row, col, in_patt;
   byte const patt[4] = {0x0e, 0x0d, 0x0b, 0x07};

   for (col = 0; col < 4; col++)
   {
      TRISA = (TRISA & 0xf0) | patt[col];
#ifdef _2_ROW
      in_patt = (PORTB >> 4) & 0x03;
      if(in_patt != 0x03)     // both high
#else
      in_patt = (PORTB >> 4) & 0x0f;
      if(in_patt != 0x0f)     // all high
#endif
      {
         for (row = 0; row < 4; row++)
         {
#ifdef _2_ROW
            if (in_patt == (patt[row] & 0x03))
#else
            if (in_patt == (patt[row] & 0x0f))
#endif
            {
               *p_row = row;
               *p_col = col;
               TRISA = TRISA & 0xf0; // restore ground to all col outputs
               return(TRUE);
            }
         }
      }
   }
   TRISA = TRISA & 0xf0;
   return(FALSE);
}

#int_rb rb_int_handler(void)
{
   byte row, col;
   debounce(50);
   if (get_key(&row, &col) == TRUE)
   {
      keys[put_index] = 4 * row + col;
      ++put_index;
      if (put_index > 19)
      {
        put_index = 0;
      }
   }
```

```
}

#int_rda rda_interrupt_handler(void)
{
   byte ch;
   rda_int_occ = TRUE;
   ch = RCREG;
}

#int_default default_interrupt_handler()
{
}

#include <ser_628.c>
#include <delay.c>
```

Weather Station Routines.

The following few routines might be used in a weather station application. Typical functions in such a station include measuring one or more temperatures, one or more relative humidities, barometric pressure, wind direction, wind speed and rainfall.

Techniques for measuring temperature using either a NTC thermistor or a DS18S20 have been detailed in previous discussions. The following discusses measuring barometric pressure using a Motorola MPX4115A, measuring relative humidity and temperature using a Dallas DHS XXXX and measuring wind direction using a special dual potentiometer with 360 degrees of travel from Fascinating Electronics.

Functions which are not included in this distribution are wind speed and rainfall accumulation. I would be inclined toward using outboard PICs for these functions and perhaps these will be discussed in future distributions.

Fascinating Electronics provides wind-vane and anemometer assemblies which in my mind are very well done at a reasonable price.

Note that in developing this material, I am presenting one concept at a time and thus used whatever PIC terminals were convenient. Thus, if you take this further and integrate various functions, you will undoubtedly find you have to modify my pin assignments.

Barometric Pressure.

Program barom_1.c illustrates a technique for measuring atmospheric pressure using a Motorola MPX4115A pressure sensor which outputs a voltage proportional to the absolute pressure. This voltage is read using the PIC's A/D converter, the atmospheric pressure is calculated and the barometric pressure is then calculated for the altitude. The result is displayed in millibars and in inches of mercury.

The voltage appearing on ADC input AN0 is calculated as;

```
(1)   V_4115_out = ad_val / 1024 * V_ref
```

where V_ref is the PIC's supply voltage.

The MPX4115A sensor outputs a voltage which is proportional to pressure;

```
(2)   V_4115_out =  0.0009 * P_station - 0.095
```

where P_station is the atmospheric pressure in millibars.

Equating (1) and (2) and solving for P_station;

```
(3) P_station = (ad_val / 1024 + 0.095) / 0.0009
```

or

```
(4) P_station = 1.085 * ad_val + 105.56;
```

Thus, in function meas_pressure(), 100 A/D calculations are performed and averaged and the atmospheric pressure is calculated using equation 4.

Note that atmospheric pressure varies with altitude in accordance with the following;

```
(5) P_sea_level  = K * P_station
```

where;

```
      K =  1/((1-6.8755856*10^-6 * h_feet)^5.2558797)
```

or

```
      K =  1/((1- 2.255e-5 * h_meters)^5.2558797)
```

where h_feet and h_meters are the elevations.

Thus, in function adjust_pressure(). the pressure at sea level is calculated. Note that if you are developing a weather station for yourself, you might consider calculating K for your elevation using a calculator and hard coding it in your program and avoid gobbling up program memory in performing this calculation.

Most of the world uses millibars (or hectoPascals). However, in the United States, we use inches of mercury. The following equation may be used to convert;

```
(6) barom_pressure_Hg_in = 0.02953 * barom_pressure_mb
```

Note that a "fudge" will probably be necessary to adjust the final result to agree with locally reported barometric pressure. When developing this routine, my unit was reporting 30.3 inches of mercury when in fact the locally reported barometric pressure was 30.1. It sure isn't rocket science to simply subtract 0.2.

Hard coding of the elevation factor K and the FUDGE as was done in this routine is viable in one of a kind applications. However, if designing a product for wide use where it is unreasonable for the end user to do the hard coding, I would tend toward providing a calibrate mode such that the user could adjust a potentiometer such that the reading agreed with the local reading and then save this calibration to EEPROM. This concept was illustrated in an earlier routine.

```
// barom_1.c (PIC16F877)
//
// Measures atmospheric pressure using an MPX4115A pressure sensor using an ADC
// on the PIC.  Calculates pressure in millibars and then calculates barometric
// pressure based on the altitude.  Displays in millibars and in inches of mercury.
//
// MPX4115A                        PIC16F877
//
```

```
// Out (term 1) --------------------> AN0 (term 2)
// GRD (term 2)
// +5VDC (term 3)
//
// copyright, Peter H Anderson, Baltimore, MD, May, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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

#define ELEVATION_METERS 133.0
#define FUDGE -0.2

float meas_pressure(void);
float adjust_pressure(float pressure, float alt_meters);
float power(float a, float b);

main()
{
   float atmos_pressure_mb, barom_pressure_mb, barom_pressure_Hg_in;

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

   lcd_init();

   while(1)
   {
      atmos_pressure_mb = meas_pressure();
      barom_pressure_mb = adjust_pressure(atmos_pressure_mb, ELEVATION_METERS);
      lcd_clr_line(0);
      printf(lcd_char, "Pressure = %3.1f", barom_pressure_mb);
      barom_pressure_Hg_in = 0.02953 * barom_pressure_mb + FUDGE;
      lcd_clr_line(1);
      printf(lcd_char, "Pressure = %3.2f", barom_pressure_Hg_in);

      delay_ms(5000);   // five second delay
   }
}

float meas_pressure(void)
{
   byte n;
   long ad_val;
   float sum = 0.0, ad_val_avg, atmos_pressure;

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

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

```
   for (n=0; n<100; n++)      // perform 100 A/D conversions and average
   {
      adgo = 1;
      while(adgo)    ;  // poll adgo until zero
      ad_val = ADRESH;
      ad_val = ad_val << 8 | ADRESL;
      sum = sum + (float) ad_val;
   }
   ad_val_avg = sum / 100.0;
   atmos_pressure = 1.085 * ad_val_avg + 105.56;
   return(atmos_pressure);
}

float adjust_pressure(float pressure, float alt_meters)
{
   float x, y, k;

   x = 2.255e-5 * alt_meters;
   x = 1.0 - x;
   y = power(x, 5.2558797);
   k = 1.0/y;
   return(k * pressure);
}

float power(float a, float b)
{
   float c;

   c = exp(b*log(a));
   return(c);
}

#include <lcd_out.c>
```

Wind Direction.

Fascinating Electronics offers a wind vane which turns a special 5K potentiometer which is capable of freely turning. The potentiometer provides 270 degrees of wiper surface, but there are two wipers which are positioned 180 degrees relative to one another. Thus, when one wiper is in the 90 degree dead zone, the other wiper is in contact with the conductive surface.

Thus, the angle measured using wiper #1 is adval * 270 / 1024. The angle measured using wiper #2 is adval * 270 / 1024 + 180.

I used 100K pull-down resistors to ground on the two wipers such that when a wiper is in the dead zone (open), the A/D reading will be close to 0.

Thus, 100 measurements are performed on A/D Ch 1 and they are averaged. If the adval result is greater than 100.0, it is assumed wiper #1 is not in the dead zone and the angle is calculated as 0.2636 * adval_float. However, if wiper #1 appears to be in the dead zone (adval <= 100), 100 measurements are performed on A/D Ch 3 (wiper #2) and the angle is calculated as 0.2636 * adval_float + 180.0.

Of course, getting a wind vane up on the roof with your trusty Boy Scout compass and compensating for the declination and trying to twist a mast such that when the vane points north the reading is zero will probably lead to problems with the

spouse. This can be avoided by putting up the wind vane with no attention to the direction and then adding a FUDGE such that the value displayed agrees with reality.

Note that in adding the FUDGE, the result may well be 360 or more and thus the angle is adjusted to the range 0 - 359;

```
      angle = angle % 360;
```

Note that I have been able to buy only the potentiometer from Fascinating for about $15.00.

```
// Program WIND_DIR.C
//
// Uses Fascinating Electronics dual wiper potentiometer to determine wind
// direction (0 - 359).  Displays readings on LCD.
//
//  Dual Wiper Pot                  PIC
//    +5 -- term 1
//    Wiper 1 (term 2) ----------> RA1/AN1
//    Wiper 2 (term 3) ----------> RA3/AN3
//    GRD -- term 4                Note 100K pull-down resistors to GRD on RA1 and RA3
//
// copyright, Peter H. Anderson, Baltimore, MD, May, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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

#define TRUE !0
#define FALSE 0

#define FUDGE 36
long meas_wind_dir(void);
float ad_meas(byte channel, byte num_samps);

void main(void)
{
 long angle;

 lcd_init();

 while(1)
 {
    angle = meas_wind_dir(); // measure the angle
    lcd_clr_line(0);   // and display with leading zero suppression
    if (angle > 99)
    {
      lcd_dec_byte(angle / 100, 1);
      lcd_dec_byte(angle % 100, 2);
    }
    else if (angle > 9)
    {
      lcd_dec_byte((byte) angle, 2);
    }
    else
```

```
    {
      lcd_dec_byte((byte) angle, 1);
    }
    delay_ms(200);   // with a brief delay
  }
}

long meas_wind_dir(void)
{
   float adval_float, angle_float;
   long angle;

   adval_float = ad_meas(1, 100); // avg of 100
   if (adval_float > 100.0)
   {
     angle_float = 0.2636 * adval_float;
   }
   else
   {
     adval_float = ad_meas(3, 100);
     angle_float = 0.2636 * adval_float + 180.0;
   }
   angle = (long) angle_float;
   angle = angle + FUDGE;     // see text
   angle = angle % 360;
   return(angle);
}

float ad_meas(byte channel, byte num_samps)
{
   byte n;
   long adval;
   float sum = 0.0;

   pcfg3 = 0;  pcfg2 = 1;  pcfg1 = 0; pcfg0 = 0; // 3/0 configuration
   adfm = 1;

   adcs1 = 1;  adcs0 = 1;
   adon = 1;
   chs2 = 0;

   switch (channel)
   {
      case 1:  chs1 = 0; chs0 = 1;
           break;
      case 3:  chs1 = 1; chs0 = 1;
           break;
      default:
           return(-99.9);
   }

   for (n = 0; n<num_samps; n++)
   {
      adgo = 1;
      while(adgo)               ;
      adval = ADRESH;
      adval = (adval << 8) + ADRESL;
```

```
      sum = sum + (float) adval;
   }
   return(sum/(float) num_samps);
}

#include <lcd_out.c>
```

Relative Humidity.

The Honeywell HIH-3610 is a relative humidity to voltage converter sensor in a three terminal package. The accuracy is specified as two percent and the operational temperature range is specified as –40 to 85 degrees C. The output is;

```
(1)   V_out = V_supply (0.0062 * RH + 0.16)
```

Solving for RH;

```
(2)   RH = ((V_out / V_supply) – 0.16) / 0.0062
```

This is corrected for temperature;

```
(3)   RH_corrected = RH / (1.0546 – 0.00216 * T_C)
```

Note that measuring RH involves measuring V_out, the supply voltage and temperature. This might be easily done by interfacing the HIH-3610 with a PIC A/D. Equation (2) is then;

```
(4)       RH = ((ad_val / 1024) – 0.16) / 0.0062
```

A thermistor might then be used to measure temperature.

However, the Dallas DS2438 1-W Battery Management IC is an interesting device in that it is capable of measuring its supply voltage, an analog input voltage and its temperature. This device in itself is powerful in that it permits the remote measurement of voltages and one temperature at a remote location up to 300 feet with only a single twisted pair. In fact, the DS2438 may be powered off the signal line and although this supply voltage may be less than 3.0 VDC, the DS2438 is capable of measuring voltages as high as ten volts.

Dallas offers a DSHS01K kit at their i-Button web site which consists of a DS2438 and Honeywell HIH-3610 mounted on a small printed circuit board about 0.6 inches square. The cost is nominally $20.00. Power is derived from the signal line using a diode and capacitor and thus the only interface with the processor and this relative humidity PCB is a single twisted pair (signal DQ and ground).

The DS2438 is a relatively powerful device which provides all manner of capabilities including the sensing of current and a primitive elapsed time counter. In this discussion, only the temperature and voltage measurement capabilities are discussed.

The DS2438 provides eight pages of memory, each consisting of eight bytes. Page zero is used for temperature and voltage measurements. Byte 0 of page zero is a Status / Configuration register. The only bit of interest in this discussion is bit 3. When set to one (0x08), the voltage measured by the A/D is V_dd. When set to zero, the voltage measured is that appearing on the AD input. Bytes 1 and 2 of page zero contain results of the most recent temperature conversion and bytes 3 and 4 contain the results of the most recent A/D conversion.

Thus, a sequence is to configure byte 0 of page 0, perform the required operation and then read page 0 and use the appropriate bytes. This is shown in the following example;

```
0xcc  // skip ROM
0x4e  // write scratch pad
      0x00  // beginning at byte 0
0x08  // value to write to byte 0 for V_dd), 0x00 for V_ad

0xcc  // skip ROM
0x48  // copy scratchpad
0x00  // scratch pad is to be copied to page 0

0xcc  // skip ROM
0xb4  // perform an A/D conversion.  Use 0x44 for temperature

0xcc  // skip ROM
0xb8  // recall memory to scratchpad
0x00  // memory page to be transferred

0xcc  // skip ROM
0xbe  // read scratchpad
// now read the eight bytes plus CRC
```

Note that the only differences in measuring V_DD and V_AD is whether the value 0x0x08 or 0x00 is written to the status / configuration register. The difference in measuring temperature is simply issuing the command 0x44 rather than 0xb4.

The result of a temperature measurement is contained in bytes 1 and 2, least significant byte first. These are put together in a signed long which is of the form;

```
S XXXXXXX  YYYYY000
```

Where S is the sign bit (1 is minus) and XXXXXX is the whole number and YYYYY is the decimal portion.

Note that by dividing by 8 results in;

```
S 000 XXXXX XXXYYYY
```

It is important to note that division by eight is different than shifting to the right three places (>>3) as division by eight retains the sign bit in the most significant bit position.

The result may be typecast to a float and multiplied by 0.03125.

The voltage result is contained in bytes 3 and 4. These are put together in a long which is of the form;

```
000000XX YYYYYYYY
```

Dallas is clever. This result may be simply typecast as a float and multiplied by 0.01.

In routine humid_2, measurements of temperature, V_AD and V_DD are performed and the RH is calculated and corrected for temperature as described above. The values of T_C, V_AD, V_DD and the corrected RH are displayed on the LCD.

```
// HUMID_2.C (PIC16F877)
//
// Illustrates an interface with a DS2438 Battery Monitor and
// a Honeywell HIH-3610 Humidity Sensor.
```

```
//
// Continually measures V_dd (Voltage source), V_ad (output of humidity
// sensor) and temperature T_C and calculates RH and RH corrected for
// temperature.
//
//  PIC16F877                    DS2438
//
//  PORTD0 (term 19) ------------ DQ (term 8)
//
//               HIH-3610
//                   V_out ------ VAD (term 4)
//
// 4.7K pullup to +5 VDC on DQ.
//
// Note that a module consisting of the DS2438 and an HIH3610 on a small
// PCB is available from Dallas Semi as "DSHS01K Humidity Sensor Experimenter's
// Kit"
//
// copyright, Peter H. Anderson, Baltimore, MD, June, '01

#case
#device PIC16F877 *=16 ICD=TRUE

#include <a:\defs_877.h>
#include <a:\lcd_out.h>
#include <a:\_1_wire.h>

#define FALSE 0
#define TRUE !0

#define V_AD_SOURCE 1
#define V_DD_SOURCE 0

float meas_2438_T_C(void);
float meas_2438_V(byte source);

float calc_RH(float V_DD, float V_AD);
float calc_RH_temp_corrected(float RH, float T_C);

void main(void)
{
   float V_AD, V_DD, T_C, RH, RH_corrected;

   pspmode = 0;

   lcd_init();
   printf(lcd_char, ".............");  // to show that something is going on

   while(1)
   {
      T_C = meas_2438_T_C( );

      V_AD = meas_2438_V(V_AD_SOURCE);
      V_DD = meas_2438_V(V_DD_SOURCE);

      RH = calc_RH(V_DD, V_AD);
      RH_corrected = calc_RH_temp_corrected(RH, T_C);
```

```
      lcd_clr_line(0);
      printf(lcd_char, "V_dd = %3.2f", V_DD);

      lcd_clr_line(1);
      printf(lcd_char, "V_ad = %3.2f", V_AD);

      lcd_clr_line(2);
      printf(lcd_char, "T_C = %3.2f", T_C);

      lcd_clr_line(3);
      printf(lcd_char, "RH = %3.1f", RH_corrected);

      delay_ms(5000);
  }
}

void meas_2438_T_C(void)
{
   byte a[8], n;
   signed long T_C_long;
   float T_C_float;

   _1w_init(0); // first set config byte
   _1w_out_byte(0, 0xcc);
   _1w_out_byte(0, 0x4e);
   _1w_out_byte(0, 0x00); // page 0
   _1w_out_byte(0, 0x00);


   _1w_init(0);
   _1w_out_byte(0, 0xcc);
   _1w_out_byte(0, 0x44); // temperature conversion
   delay_ms(1000);

   _1w_init(0);
   _1w_out_byte(0, 0xcc); // recall memory
   _1w_out_byte(0, 0xb8);
   _1w_out_byte(0, 0x00);

   _1w_init(0);   // send data
   _1w_out_byte(0, 0xcc);
   _1w_out_byte(0, 0xbe);
   _1w_out_byte(0, 0x00); // page 0

    for (n=0; n<9; n++)
    {
         a[n] = _1w_in_byte(0);
    }

    T_C_long = a[2];
    T_C_long = (T_C_long << 8) + a[1]; // put the two bytes together
    T_C_long = T_C_long / 8;   // see text
    T_C_float = (float) T_C_long * 0.03125;
    return(T_C_float);
}
```

```
float meas_2438_V(byte source)
{
   byte a[8], n;
   long ad_val;

   _1w_init(0); // first set config byte
   _1w_out_byte(0, 0xcc);
   _1w_out_byte(0, 0x4e);
   _1w_out_byte(0, 0x00); // page 0
   if (source == V_AD_SOURCE)
   {
      _1w_out_byte(0, 0x00);
   }
   else
   {
      _1w_out_byte(0, 0x08);
   }

   _1w_init(0);
   _1w_out_byte(0, 0xcc); // perform ADC
   _1w_out_byte(0, 0xb4);
   delay_ms(1);

   _1w_init(0);
   _1w_out_byte(0, 0xcc); // recall memory
   _1w_out_byte(0, 0xb8);
   _1w_out_byte(0, 0x00);

   _1w_init(0);   // send data
   _1w_out_byte(0, 0xcc);
   _1w_out_byte(0, 0xbe);
   _1w_out_byte(0, 0x00); // page 0

    for (n=0; n<9; n++)
    {
         a[n] = _1w_in_byte(0);
    }

    ad_val = a[4]; // high byte
    ad_val = (ad_val << 8) | a[3];
    return(0.01 * (float) ad_val);
}

float calc_RH(float V_DD, float V_AD)
{
   float RH;
   RH = ((V_AD / V_DD) - 0.16) / 0.0062;
   return(RH);
}

float calc_RH_temp_corrected(float RH, float T_C)
{
   float RH_corrected;
   RH_corrected = RH * (1.0546 - 0.00216 * T_C);
   return(RH_corrected);
}
```

```
#include <a:\lcd_out.c>
#include <a:\_1_wire.c>
```

Wind Direction.

Fascinating Electronics offers a wind vane which turns a special 5K potentiometer which is capable of freely turning. The potentiometer provides 270 degrees of wiper surface, but there are two wipers which are positioned 180 degrees relative to one another. Thus, when one wiper is in the 90 degree dead zone, the other wiper is in contact with the conductive surface.

Thus, the angle measured using wiper #1 is adval * 270 / 1024. The angle measured using wiper #2 is adval * 270 / 1024 + 180.

I used 100K pull-down resistors to ground on the two wipers such that when a wiper is in the dead zone (open), the A/D reading will be close to 0.

Thus, 100 measurements are performed on A/D Ch 1 and they are averaged. If the adval result is greater than 100.0, it is assumed wiper #1 is not in the dead zone and the angle is calculated as 0.2636 * adval_float. However, if wiper #1 appears to be in the dead zone (adval <= 100), 100 measurements are performed on A/D Ch 3 (wiper #2) and the angle is calculated as 0.2636 * adval_float + 180.0.

Of course, getting a wind-vane up on the roof with your trusty Boy Scout compass and compensating for the declination and trying to twist a mast such that when the vane points north the reading is zero will probably lead to problems with the spouse. This can be avoided by putting up the wind vane with no attention to the direction and then adding a FUDGE such that the value displayed agrees with reality.

Note that in adding the FUDGE, the result may well be 360 or more and thus the angle is adjusted to the range 0 - 359;

```
      angle = angle % 360;
```

Note that I have been able to buy only the potentiometer from Fascinating for about $15.00.

```
// Program WIND_DIR.C
//
// Uses Fascinating Electronics dual wiper potentiometer to determine wind
// direction (0 - 359).  Displays reading on LCD
//
//  Dual Wiper Pot                          PIC
//    +5 -- term 1
//    Wiper 1 (term 2) ----------> RA1/AN1
//    Wiper 2 (term 3) ----------> RA3/AN3
//    GRD -- term 4           Note 100K pull-down resistors to GRD on RA1 and RA3
//
// copyright, Peter H. Anderson, Baltimore, MD, May, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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

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

```
#define FUDGE 36
long meas_wind_dir(void);
float ad_meas(byte channel, byte num_samps);

void main(void)
{
 long angle;

 lcd_init();

 while(1)
 {
    angle = meas_wind_dir(); // measure the angle
    lcd_clr_line(0);   // and display with leading zero suppression
    if (angle > 99)
    {
      lcd_dec_byte(angle / 100, 1);
      lcd_dec_byte(angle % 100, 2);
    }
    else if (angle > 9)
    {
      lcd_dec_byte((byte) angle, 2);
    }
    else
    {
      lcd_dec_byte((byte) angle, 1);
    }
    delay_ms(200);   // with a brief delay
  }
}

long meas_wind_dir(void)
{
   float adval_float, angle_float;
   long angle;

   adval_float = ad_meas(1, 100); // avg of 100 meas on Ch 1
   if (adval_float > 100.0)   // if wiper #1 not in dead zone
   {
     angle_float = 0.2636 * adval_float;
   }
   else           // otherwise, use wiper #2
   {
     adval_float = ad_meas(3, 100); // avg of 100 meas on Ch 3
     angle_float = 0.2636 * adval_float + 180.0;
   }
   angle = (long) angle_float;
   angle = angle + FUDGE;     // see text
   angle = angle % 360;
   return(angle);
}

float ad_meas(byte channel, byte num_samps)
{
   byte n;
   long adval;
```

```
   float sum = 0.0;

   pcfg3 = 0;  pcfg2 = 1;  pcfg1 = 0; pcfg0 = 0; // 3/0 configuration
   adfm = 1;

   adcs1 = 1;  adcs0 = 1;
   adon = 1;
   chs2 = 0;

   switch (channel)
   {
      case 1:  chs1 = 0; chs0 = 1;
           break;
      case 3:  chs1 = 1; chs0 = 1;
           break;
      default:
           return(-99.9);
   }

   for (n = 0; n<num_samps; n++)
   {
      adgo = 1;
      while(adgo)               ;
      adval = ADRESH;
      adval = (adval << 8) + ADRESL;
      sum = sum + (float) adval;
   }
   return(sum/(float) num_samps);
}

#include <lcd_out.c>
```

DS2435 Battery Monitor IC – Temperature Histogram.

I was attracted to the Dallas DS2435 for its ability to "log" temperature in histogram form on a standalone basis. Thus a "logger" consisting of nothing more than the DS2435, a battery and a pull-up resistor might be shipped with a crate of fruit or other temperature sensitive cargo. On receipt of the shipment, the user might then read the histogram to assure the quality of the shipment. The histogram is considerably better than a min-max logger as the value of fruit declines with heat over time and is useful in these types of applications where the exact times that various temperatures occurred is simply not necessary. This is an amazing capability for about $3.00.

However, in tinkering with the device, I can see many other possibilities as well. In addition to logging temperature, the 2435 provides a good deal of nonvolatile EEPROM, a primitive onboard timer and a counter.

Dallas provided the counter feature for monitoring the number of times a battery was placed in a smart charger. Note that any of a number of such chargers might be used and each charger simply increments the counter. However, this capability might be used in any application where the DS2435 is a part of a product which is operated on by a number of devices. For example, as part of a software protection scheme, the DS2435 might be a part of a "key" which may be moved from one PC to another.

Unlike most devices in the Dallas 1-W family, the DS2435 is not addressable. That is, there is no 64 bit serial number and the DS2435 cannot share an IO with another DS2435 or other 1-W device. The device does have a two byte ID number, but this is not unique. Rather, a battery manufacturer might request a different ID for each type of battery such that the charger could identify the battery type and adjust the charging algorithm.

The nature of the DS2435 command set suggests a non electrical engineering / computer science influence as the RAM memory map consists of pages 1 - 5, rather than 0 - 4. It also suggests at least two divergent development groups as there are two different approaches to writing to memory. However, as with most Dallas devices, the data sheet is well done.

Program DS2435_1.C

This program illustrates how to read the two byte ID and temperature, how to reset, increment and read the counter and how to clear and later read the elapsed time counter.

Note that reads are initiated with the command 0xb2 followed by the byte address of the first byte to be read. Thus, reading the ID is a matter of sending the command 0xb2, followed by the address 0x80 followed by two reads. Reading the temperature is a matter of sending the command byte 0xb2 followed by the address 0x60 followed by two reads. Reading the counter is similarly a matter of sending the 0xb2 command, followed by 0x82 followed by two reads and reading the elapsed time is sending 0xb2 followed by address 0x74, followed by three reads.

Specific commands are provided to initiate a temperature conversion (0xd2), reset the counter (0xb8) and increment the counter (0xb5).

The sample rate is configured by issuing the command 0xef, followed by the address of the register (0x8b) followed by the value.

However, writing to the elapsed time counters is implemented with a specific command (0xe6) followed by the three values to be written.

It is my understanding that when the DS2435 is in the communication mode, the elapsed time counter stops. The device is released to perform its off line function by issuing a reset (_1w_init) with no subsequent command.

```
// DS2435_1.C (PIC16F877)
//
// Interface with Battery Monitor DS2435
//
// Reads and displays two byte ID.  Reads and diplays temperature.  Resets
// cycle counter and then increments counter five times, reads counter and
// displays.
//
// Sets sample rate to 0.5 minutes, clears elapsed time counter and then goes
// into loop, continually reading and displaying the elapsed time every ten
// seconds.
//
// PIC16F877                                          DS2435
//                                                          +5 VDC --
// PORTD0 (term 19) ----------------------------- DQ
//                                                          GRD -----
//
// Note that a 4.7K pullup resistor to +5 VDC is on the DQ lead.
//
// This was developed by Ernest N. Wells, Jr. as a part of his Senior Project
// at Morgan State University.
//
// copyright, Peter H. Anderson, Baltimore, MD, May, '01

#case
```

```
#device PIC16F877 *=16 ICD=TRUE
#include <a:\defs_877.h>
#include <a:\lcd_out.h>
#include <a:\_1_wire.h>

#define FALSE 0
#define TRUE !0

void read_2435_ID(byte *a);
void read_temperature(byte *a);

void reset_2435_cycle_counter(void);
void increment_2435_cycle_counter(void);
void read_2435_cycle_counter(byte *a);

void set_2435_sample_rate(byte v);
void set_2435_elapsed_time(byte *a);
void read_elapsed_time(byte *a);


void main(void)
{
   byte a[6], n;

   lcd_init();

   pspmode = 0;            // configure PORT as GPIO

   read_2435_ID(a);     // two byte result in array;
   printf(lcd_char, "ID = %2x%2x", a[1], a[0]);  // display ID

   read_temperature(a); // perform a temperature measurement
   lcd_clr_line(1);
   printf(lcd_char, "T_C = %3.1f  %d", (float) a[0]/2.0, a[1]);
     // display in the two formats

   reset_2435_cycle_counter();

   for (n=0; n<5; n++)
   {
     increment_2435_cycle_counter();
   }

   read_2435_cycle_counter(a); // fetch two byte value of cycle counter
   lcd_clr_line(2);
   printf(lcd_char, "Count = %2x%2x", a[1], a[0]);  // display cycles, high byte first

   set_2435_sample_rate(0x00); // set sample rate for 1/2 minute

   a[0] = 0x00; a[1] = 0x00;  a[2] = 0x00; // set clock to zero
   set_2435_elapsed_time(a);  // pass setting in first three bytes of array a

   while(1)    // continually read the elapsed time and display every 10 secs
   {
      read_elapsed_time(a);
      lcd_clr_line(3);
      printf(lcd_char, "ET = %u", a[0]); // display only the least sig byte
```

```
      _1w_init(0);  // release DS2435 to perform timing
      delay_ms(10000);
   }
}

void read_2435_ID(byte *a)
{
   _1w_init(0);       /* reset */
   _1w_out_byte(0,0xb2);   /* read registers on page 5 (ID bytes) */
   _1w_out_byte(0,0x80);   /* ID address 80h and 81h */
   a[0] = _1w_in_byte(0);
   a[1] = _1w_in_byte(0);
}

void read_temperature(byte *a)
{
   _1w_init(0);
   _1w_out_byte(0,0xd2);     // initiate temperature conversion cycle
    delay_ms(1000);

   _1w_init(0);
   _1w_out_byte(0,0xb2);      // read registers
   _1w_out_byte(0,0x60);

   a[0] = _1w_in_byte(0);     // 1/2 celceus temperature 0 - 127.5
   a[1] = _1w_in_byte(0);
}

void reset_2435_cycle_counter(void)
{
   _1w_init(0);
   _1w_out_byte(0,0xb8);
}
void increment_2435_cycle_counter(void)
{
   _1w_init(0);
   _1w_out_byte(0,0xb5);
}
void read_2435_cycle_counter(byte *a)
{
   _1w_init(0);
   _1w_out_byte(0,0xb2);      // read registers
   _1w_out_byte(0,0x82);

   a[0] = _1w_in_byte(0);     // low byte
   a[1] = _1w_in_byte(0);    // high byte
}

void set_2435_sample_rate(byte v)
{
   _1w_init(0);
   _1w_out_byte(0,0xef);      // write registers
   _1w_out_byte(0,0x8b);
   _1w_out_byte(0,v);
}
```

```
void set_2435_elapsed_time(byte *a)
{
   _1w_init(0);
   _1w_out_byte(0,0xe6);      // set clock
   _1w_out_byte(0,a[0]);
   _1w_out_byte(0,a[1]);
   _1w_out_byte(0,a[2]);
}

void read_elapsed_time(byte *a)
{
   _1w_init(0);
   _1w_out_byte(0,0xb2);      // read registers
   _1w_out_byte(0,0x74);

   a[0] = _1w_in_byte(0);     // low byte
   a[1] = _1w_in_byte(0);    // mid byte
   a[2] = _1w_in_byte(0);    // high byte
}

#include <a:\lcd_out.c>
#include <a:\_1_wire.c>
```

Program DS2435_2.c.

This routine illustrates how the DS2435 may be configured as a primitive data logger which periodically records data in a histogram fashion..

The elapsed time is set to 0 and the sample time is to 0.5 minutes.

The seven temperature boundaries are defined by issuing the 0xef command followed by the address of the first byte (0x84) followed by the seven temperature boundaries. The current histogram is cleared by issuing the command 0xe1. The DS2435 is then released to perform its task (_1w_init). Note that the DS2435 (powered) may now be removed from the processor and packaged with a shipment of strawberries or other temperature sensitive cargo.

The histogram is read by issuing the command 0xb2 followed by the first address which is to be read (0x64) and then reading the value of each of the eight bins. Note that each bin consists of two bytes permitting bin counts of up to 65535.

```
// DS2435_2.C (PIC16F877)
//
// Illustrates histogram feature of the DS2435.
//
// Clears the elapsed time counter and sets the sample rate for 0.5 minutes.
// Sets seven temperature boundaries, TA - TG, to define eight temperature bins.
// Clears the current histogram.
//
// The program then enters a continual loop.  The DS2435 is released (_1w_init)
// to perform its off line operation and after a 30 second delay, the histogram
// is read and displayed.
//
// PIC16F877                              DS2435
//                                        +5 VDC --
// PORTD0 (term 19) ----------------------------- DQ
//                                        GRD -----
//
```

```
// Note that a 4.7K pullup resistor to +5 VDC is on the DQ lead.
//
// This was developed by Ernest N. Wells, Jr. as a part of his Senior Project
// at Morgan State University.
//
// copyright, Peter H. Anderson, Baltimore, MD, May, '01

#case
#device PIC16F877 *=16 ICD=TRUE

#include <a:\defs_877.h>
#include <a:\lcd_out.h>
#include <a:\_1_wire.h>

#define FALSE 0
#define TRUE !0

void set_2435_temperature_bounds(byte *a);

void reset_2435_histogram(void);
void read_2435_histogram(byte *a);

void set_2435_sample_rate(byte v);
void set_2435_elapsed_time(byte *a);

void main(void)
{
    byte const temp_bounds[7] = {20, 21, 22, 23, 24, 25, 26};
    byte a[16], line, n;

    lcd_init();

    printf(lcd_char, ".............");

    a[0] = 0x00;  a[1] = 0x00;  a[2] = 0x00;
    set_2435_elapsed_time(a);

    set_2435_sample_rate(0x00);
    reset_2435_histogram();

    for (n=0; n<7; n++)    // seven boundaries
    {
       a[n] = temp_bounds[n];
    }
    set_2435_temperature_bounds(a);

    while(1)
    {
       _1w_init(0);
       delay_ms(30000); // 30 second delay

       read_2435_histogram(a);
       lcd_init();
       for (n=0, line = 0; n<16; n+=2)
       {
         if (((n%4) == 0) && (n!=0))
         {
```

```
            ++line;
            lcd_clr_line(line);
          }
          printf(lcd_char, "%2x%2x  ", a[n+1], a[n]);
       }
    }
}

void set_2435_temperature_bounds(byte *a)
{
    byte n;
    _1w_init(0);
    _1w_out_byte(0,0xef);  /* write register */
    _1w_out_byte(0,0x84);

    for (n=0; n<7; n++)
    {
       _1w_out_byte(0, a[n]);
    }
}

void reset_2435_histogram(void)
{
   _1w_init(0);
   _1w_out_byte(0,0xe1);
}

void read_2435_histogram(byte *a)
{
   byte n;
   _1w_init(0);   /* reset */
   _1w_out_byte(0,0xb2);     /* read registers */
   _1w_out_byte(0,0x64); /* starting address */

   for (n = 0; n < 16; n++)
   {
     a[n] = _1w_in_byte(0);
   }
}

void set_2435_sample_rate(byte v)
{
   _1w_init(0);
   _1w_out_byte(0,0xef);      // write registers
   _1w_out_byte(0,0x8b);
   _1w_out_byte(0,v);
}

void set_2435_elapsed_time(byte *a)
{
   _1w_init(0);
   _1w_out_byte(0,0xe6);
   _1w_out_byte(0,a[0]);
   _1w_out_byte(0,a[1]);
   _1w_out_byte(0,a[2]);
}
```

```
#include <a:\lcd_out.c>
#include <a:\_1_wire.c>
```

Program DS2435_3.c.

The DS2435 also includes 24 bytes plus 8 bytes of EEPROM and 32 bytes of static RAM. This might be used in applications where the application demands measuring temperature and this EEPROM may then permit a very inexpensive PIC to be used. In fact, I feel this is becoming less and less of an issue, but the PIC12C672 is an example of an inexpensive PIC with no EEPROM.

However, the EEPROM might be used to store data which is peculiar to the product associated with the DS2435. In my example of shipping a DS2435 histogram logger with strawberries, this EEPROM might be used to identify the nature of the shipment, the shipper and the date and time the product was shipped at the departure point. At the receiving point, this data and the histogram might be read and a printout send to a central location to determine the quality of the shipment and whether the shipper is to be paid or penalized.

As previously noted, the memory associated with the DS2435 is defined in a manner a bit at odds with convention.

Page 1	0x00 – 0x1f
Page 2	0x20 – 0x3f
Page 3	0x40 – 0x5f
Page 4	0x60 – 0x7f
Page 5	0x80 – 0x9f

All functions which have been previously discussed; ID, counter, elapsed time, temperature, histogram boundaries and the histogram bins are associated with pages 4 and 5.

24 bytes of EEPROM are associated with Page 1, 8 bytes of EEROM with page 2 and 32 bytes of SRAM are associated with Page 3.

Writing to memory is a matter of issuing the “write scratchpad” (0x17) command, followed by the start address, followed by the data bytes. I confined the number of bytes to eight, but in scanning the data sheet, I am unsure if this was not a self imposed restriction.

The scratchpad is copied to the EEPROM associated with Page 1, or to the EEPROM associated with Page 2 or to the SRAM associated with Page 3 by issuing commands 0x22, 0x25 or 0x28, respectively. The association of the page and the command is implemented using a constant array.

As added security, the DS2435 provides a lock on page 1. The EEPROM is unlocked with command 0x44, the data is transferred from scratchpad to EEPROM and the EEPROM may then again be locked with command 0x43.

Reading data from EEPROM or SRAM is a matter of first transferring it from EEPROM or SRAM to the scratchpad by issuing commands 0x71, 0x77 or 0x7a for pages 1, 2 and 3, respectively. Here again, a constant array was used to map the page number into the command.

The data is then read from the scratchpad by issuing command 0x11, followed by the address to begin reading from and then sequentially reading each byte.

In the following routine, the string “Morgan State University” is written to the 24 EEPROM bytes associated with Page 1, two floats are written to the eight EEPROM bytes associated with Page 2 and various numbers are written to the 32 SRAM bytes associated with Page 3. This is then read and displayed on the LCD.

```
// DS2435_3.C (PIC16F877)
//
// Illustrates how to use EEPROM and static RAM associated with the DS2435.
// Write the string "Morgan State University" to the 24 bytes of EEPROM on page
// 1, writes two floats to the eight EEPROM bytes on page 2 and 32 bytes to SRAM
// on page 3.
//
// Then reads this data back and displays on LCD.
//
// PIC16F877                              DS2435
//                                        +5 VDC --
// PORTD0 (term 19) ----------------------------- DQ
//                                        GRD -----
//
// Note that a 4.7K pullup resistor to +5 VDC is on the DQ lead.
//
// This was developed by Ernest N. Wells, Jr. as a part of his Senior Project
// at Morgan State University.
//
// copyright, Peter H. Anderson, Baltimore, MD, May, '01

#case

#device PIC16F877 *=16 ICD=TRUE
#include <a:\defs_877.h>
#include <a:\lcd_out.h>
#include <a:\_1_wire.h>

#define FALSE 0
#define TRUE !0

void clear_2435_scratchpad_all(void);
void write_2435_scratchpad(byte adr, byte *a, byte num_vals);
void read_2435_scratchpad(byte adr, byte *a, byte num_vals);

void transfer_2435_scratchpad_to_mem(byte page);
void transfer_2435_mem_to_scratchpad(byte page);

void main(void)
{
   byte const a1[24] = {"Morgan State University"};
   byte a[8], n, x;
   float f1 = 1.23e-6, f2 = 17.3e-6;

   lcd_init();
   pspmode = 0;

   for (n=0; n<8; n++)  // write to page 1, eight bytes at a time
   {
     a[n] = a1[n];
   }
   write_2435_scratchpad(0x00, a, 8);
```

```
for (n=0; n<8; n++)
{
  a[n] = a1[n+8];
}
write_2435_scratchpad(0x08, a, 8);

for (n=0; n<8; n++)
{
  a[n] = a1[n+16];
}
write_2435_scratchpad(0x10, a, 8);

transfer_2435_scratchpad_to_mem(1);

write_2435_scratchpad(0x20, &f1, 4); // write to page 2
write_2435_scratchpad(0x24, &f2, 4);
transfer_2435_scratchpad_to_mem(2);

for (n=0; n<32; n++)     // write to page 3
{
   x = 255 - n;
   write_2435_scratchpad(0x40+n, &x, 1);
}
transfer_2435_scratchpad_to_mem(3);

clear_2435_scratchpad_all();

// Now read the data back and display
transfer_2435_mem_to_scratchpad(1);
transfer_2435_mem_to_scratchpad(2);
transfer_2435_mem_to_scratchpad(3);

printf(lcd_char, "Page 1");

lcd_clr_line(1);
read_2435_scratchpad(0x00, a, 8);
for (n=0; n<8; n++)
{
   lcd_char(a[n]);
}

lcd_clr_line(2);
read_2435_scratchpad(0x08, a, 8);
for (n=0; n<8; n++)
{
   lcd_char(a[n]);
}

lcd_clr_line(3);
read_2435_scratchpad(0x10, a, 8);
for (n=0; n<8; n++)
{
   lcd_char(a[n]);
}

delay_ms(1000);
```

```
   lcd_init();
   printf(lcd_char, "Page 2 - floats");
   read_2435_scratchpad(0x20, &f1, 4);
   read_2435_scratchpad(0x23, &f2, 4);

   lcd_clr_line(1);
   printf(lcd_char, "%e", f1);

   lcd_clr_line(2);
   printf(lcd_char, "%e", f2);

   delay_ms(1000);

   lcd_init();
   printf(lcd_char, "Page 3");

   for (n=0x40; n<0x60; n++)
   {
     lcd_clr_line(1);
     read_2435_scratchpad(n, &x, 1);
     lcd_dec_byte(x, 3);
     delay_ms(100);
   }
   while(1)          ;
}

void clear_2435_scratchpad_all(void)
{
   byte n;
   _1w_init(0);
   _1w_out_byte(0, 0x17);
   _1w_out_byte(0, 0x00);
   for (n=0; n<0x5f; n++)
   {
      _1w_out_byte(0, 0x00);
   }
}

void write_2435_scratchpad(byte adr, byte *a, byte num_vals)
{
    byte n;

   _1w_init(0);
   _1w_out_byte(0, 0x17);
   _1w_out_byte(0, adr);
   for (n=0; n<num_vals; n++)
   {
     _1w_out_byte(0, a[n]);
   }
}

void read_2435_scratchpad(byte adr, byte *a, byte num_vals)
{
   byte n;

   _1w_init(0);
```

```
   _1w_out_byte(0, 0x11);
   _1w_out_byte(0, adr);
   for (n=0; n<num_vals; n++)
   {
     a[n] = _1w_in_byte(0);
   }
}

void transfer_2435_scratchpad_to_mem(byte page)
{
    byte const command[4] = {0x00, 0x22, 0x25, 0x28}; // zeroth element not used

    if (page == 1)
    {
       _1w_init(0);
       _1w_out_byte(0, 0x44);  // unlock
    }

    _1w_init(0);
    _1w_out_byte(0, command[page]);
    delay_ms(50);

    if (page == 1)
    {
       _1w_init(0);
       _1w_out_byte(0, 0x43);  // lock
    }
}

void transfer_2435_mem_to_scratchpad(byte page)
{
    byte const command[4] = {0x00, 0x71, 0x77, 0x7a}; // zeroth ele not used

    _1w_init(0);
    _1w_out_byte(0, command[page]);
    delay_ms(10);
}

#include <a:\lcd_out.c>
#include <a:\_1_wire.c>
```

Flash EEPROM.

Saving to flash memory attractive as it requires considerably less time than saving to EEPROM. On the down side, the endurance of flash memory is severely limited when compared with the data EEPROM.

Program FLSH_EE.C illustrates a technique which moves the assignment of eight “persistent” bytes from one block to another after 256 writes. In this scheme, 32 blocks, each consisting of eight bytes of flash memory are used. In addition, 32 counters are implemented in flash memory, one counter for each of the 32 blocks. In addition, a single byte is used to identify the current block which is being used. Thus, using this technique, the endurance is improved by a factor of 32 at the expense of 256 plus 32 plus 1 or 289 bytes. This may not be a serious penalty in applications where the PIC has more than enough flash memory to spare and the nature of the product demands more endurance than afforded by using the same flash memory addresses again and again.

All of the intricacies of moving the 8-byte block from one location to another is relatively hidden. For example, a "persistent" float and three "persistent" bytes might be saved;

```
float fl = 1.23e4
byte a = 0, b = 90, c = 100, *p;
byte dat[8];

p = (byte *) &fl;
dat[0] = *p;         // fill up the array
dat[1] = *(p+1);
dat[2] = *(p+2);
dat[3] = *(p+3);

dat[4] = a;
dat[5] = b;
dat[6] = c;

write_flash_block(dat);    // write it to flash memory

//....

read_flash_block(dat)      // later, read it back

p = (byte *) &fl;

*p = dat[0];
*(p+1) = dat[1];
*(p+2) = dat[2];
*(p+3) = dat[3];

a = dat[4];
b = dat[5];
c = dat[6];
```

In the following, on boot, the program checks to see if this is the first time the program has been run. This is implemented using data EEPROM and was discussed previously. If it is the first time, each of the 32 counters in flash memory are set to 0xff and the current block number is set to 0. The data is written to the eight memory locations specified by the current block number and the counter is incremented.

On each subsequent call to write a block, the block address is determined by reading the current block number, the data is written and the counter is incremented. If the counter rolls over to 0xff, the current block number is incremented such that subsequent writes are made to the next 8-byte block.

If the increment of the current block number is greater than 31, it is reset to zero and the process is repeated.

I am skeptical that anyone will actually use this routine in its entirety. Note that the same algorithm might also be adapted to an Atmel high density flash memory. In addition, FLSH_EE does illustrate another application of "first time", how to perform such operations as saving and reading floats to and from flash memory and incrementing a persistent byte.

```
// Program FLSH_EE.C
//
// Illustrates how to write and read a block of eight bytes to flash
// EEPROM.  After 256 writes to the same eight locations, the address is
// changed to eight different locations, and this is repeated 32 times and
// the process is repeated.  This permits variables to be saved 32 times
```

```
// as often as if the same eight locations were used.
//
// copyright, Peter H. Anderson, Baltimore, MD, May, '01

#case

#device PIC16F877 *=16 ICD=TRUE

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

#define TRUE !0
#define FALSE 0

#define BLOCK_NUM_ADR 0x1cff
#define BASE_COUNTER_ADR 0x1c00
#define BASE_STORE_ADR 0x1d00

#define TEST

byte read_block_number(void);
write_block_number(byte block_num);
byte read_counter(byte block_num);
write_counter(byte block_num, counter);
read_flash_block(byte *d);
write_flash_block(byte *d);

long get_flash_eeprom(long adr);
void put_flash_eeprom(long adr, long dat);

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


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

void main(void)
{
   byte n, dat[8], block_num, counter;
   long num = 0;

   lcd_init();

   if (is_first_time())
   {
      write_block_number(0x00);
      for (n=0; n< 32; n++)
      {
         write_counter(n, 0xff); // zero all of the counters
      }
      dat[0] = 1;
      // leave others as garbage
      write_flash_block(dat);
   }
```

```
   while(1)
   {
      read_flash_block(dat);
      if ((num%100) == 0)
      {
         lcd_clr_line(0);
         block_num = read_block_number();
         counter = read_counter(block_num);
         lcd_clr_line(0);
         printf(lcd_char, "Blk = %u", block_num);
         lcd_clr_line(1);
         printf(lcd_char, "Counter = %u", counter);
         lcd_clr_line(2);
         printf(lcd_char, "Data = %u", dat[0]);
         delay_ms(1000);
      }
      ++dat[0];
      write_flash_block(dat);
      ++num;
   }
}

byte read_block_number(void)
{
    byte block_number;
    block_number = (byte) get_flash_eeprom(BLOCK_NUM_ADR);
    return(block_number);
}

write_block_number(byte block_num)
{
    put_flash_eeprom(BLOCK_NUM_ADR, (long) block_num);
}

byte read_counter(byte block_num)
{
    byte counter;
    long adr;

    adr = BASE_COUNTER_ADR + block_num;
    counter = get_flash_eeprom(adr);
    return(counter);
}

write_counter(byte block_num, counter)
{
    long adr;
    adr = BASE_COUNTER_ADR + block_num;
    put_flash_eeprom(adr, (long) counter);
}

write_flash_block(byte *d)
{
    byte block_num, n, counter;
    long adr;

    block_num = read_block_number();
```

```
    counter = read_counter(block_num);
    ++counter;
    if (counter == 0xff)
    {
      ++block_num;
      if (block_num > 31)
      {
          block_num = 0;
      }
      write_block_number(block_num);
   }
   write_counter(block_num, counter);
   adr = (long) block_num * 8 + BASE_STORE_ADR;
   for (n = 0; n<8; n++)
   {
       put_flash_eeprom(adr, (long) d[n]);
       ++adr;
   }
}

read_flash_block(byte *d)
{
    byte block_num, n;
    long adr;

    block_num = read_block_number();
    adr = (long) block_num * 8 + BASE_STORE_ADR;
    for (n = 0; n<8; n++)
    {
       d[n] = (byte)get_flash_eeprom(adr);
       ++adr;
    }
}

void put_flash_eeprom(long adr, long dat)
{
    while(gie)       // be sure interrupts are disabled
    {
        gie = 0;
    }
    EEADRH = adr >> 8;
    EEADR = adr & 0xff;

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

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

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

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

   return(TRUE);
}

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

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

```
   eepgd = 0;  // select data EEPROM

   EEADR = adr;
   EEDATA = d;

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

   wr = 1;  // begin programming sequence

   delay_ms(10);

   wren = 0;  // disable write enable
}

#include <lcd_out.c>

#rom 0x2100={0x5a, 0xa5, 0x5a, 0xa5}      // used for first time
```