Tutorial by Example – Issue 1C

Introduction.

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

This distribution presents routines for the Dallas 1-wire family. Only the DS1820 temperature sensor is presented.

The distribution also includes use of the UART for sending and receiving data and use of the PIC16F877 as an I2C slave and an SPI slave. These are a bit more challenging to debug as additional hardware was required to send RS232 serial to the PIC and perform the I2C and SPI master functions.

I used a BasicX BX24 for these functions. The BX24 is a powerful "stamp-like" processor which uses Visual Basic. I opted for the BX24 over the Basic Stamp as the language is far more structured than Parallax PBASIC and my feeling was that anyone who can program in C can read and understand Visual Basic. Not to force your hand, but I have found the BX24 a nice companion in developing PIC applications. Get it working on a BX24 and then map it over to a PIC.

Some routines required an RS232 serial device to display data and I opted to use a BasicX Serial LCD+. Any LCD introduces additional complexity with specific commands to clear the LCD, clear a specific line, etc., and my feeling was that I am writing for a pretty advanced audience that is quite capable of mapping the material over to whatever serial LCD (or PC COM Port) you happen to have.

This distribution also includes routines for the PIC16F628. This is available in an 18-pin DIP and shares many of the features associated with the PIC16F87X family. There currently is no emulator nor debugger support for this new entry. However, I found that one could debug using the PIC16F877 and the Serial ICD and then modify a few lines of code. The one time I boldly decided to skip the debugging on the 877 cost me more hours than I care to note.

Next Distribution.

The next distribution will be in early June.

Hanging of CCS Compiler in MPLAB.

At a workshop in December we suffered with one participant having MPLAB hang when compiling his file in MPLAB. The file was renamed and extensively modified, all to no avail. MPLAB hung when attempting to import the .hex file. Mighty strange and very very frustrating. In a previous distribution, I noted the same problem with a routine related to unions.

I encountered the same annoying problem yet again and of course, these problems always seem to occur when you have the least time and the program is "oh, so simple". I tried MPLAB versions 5.15, 5,20 and 5.30 and several different PCs running various versions of Windows, with no success.

I then discovered I was using an old version of the CCS compiler, from Nov, ’99. Geeze. After installing a version dated May, ’00, and rebooting the PC, the problem disappeared and I have not any problems since that time.

Use Parenthesis.

I preach to my students that parenthesis are one of those few things in life that are free. We all know that aside from the Parallax BS2, that multiplication and division take precedence over addition and subtraction. After that, use parenthesis.

The problem I had when I boldly wrote code directly for the 16F628, “saving” time by sidestepping the debugging step was;

```
if (ad_lo = ser_get_ch(100) == 0xff)
{
   return(0x7fff);               // if no response
}
```

Function ser_get_ch() returns a character. However, if there is a timeout, the value is 0xff. Thus, in the above my intent was to fetch the character, copy it to variable ad_lo and then test if it is 0xff. Actually, it was always 0x00 as the above expression was evaluated as;

```
if (ad_lo = (ser_get_ch(100) == 0xff))
{
   return(0x7fff);               // if no response
}
```

The comparison was false and ad_lo was thus set to 0x00.

This was corrected to;

```
if ((ad_lo = ser_get_ch(100)) == 0xff)
{
   return(0x7fff);               // if no response
}
```

What a difference the parenthesis make.

This was also the point when I got the hanging problem as well. Points;

1. Use parenthesis.
2. Don’t try to save time by skipping the use of the In Circuit Debugger.
3. Be sure you are not running flaky software.

Dallas 1-Wire Interface.

Logic Levels.

Two logic levels are used in interfacing with Dallas 1-W devices; a hard logic zero and a high impedance. The high impedance is implemented by configuring the PIC terminal as an input.

Thus, a logic zero level might be implemented as;

```
      portd0 = 0;
      trisd0 = 0;
```

and a logic one as;

```
      trisd0 = 1;
```

Note that an external pull-up resistor to +5 VDC provides a TTL logic one to either the external 1-wire device or to the PIC, depending on the direction of communication.

However, in the following routines I desired to provide the user with the ability to use any of the PORTD bits 0-7 without disturbing the other bits in either the TRISD or PORTD registers.

Two constant arrays are globally declared;

```
   byte const mask_one[8] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};
   byte const mask_zero[8] = {0xfe, 0xfd, 0xfb, 0xf7, 0xef, 0xdf, 0xbf, 0x7f};
```

Note that in array mask_one, the nth element is simply a single one in the nth bit position;

```
      00000001
      00000010
      etc
```

and in array mask_zero, the nth element is a single zero in the nth bit position;

```
      11111110
      11111101
      etc;
```

Thus, if variable "bit_pos" is used to specify which bit is to be used for communicating with a 1-W device, a logic zero level is implemented as;

```
      PORTD = PORTD & mask_zero[bit_pos];
      TRISD = TRISD & mask_zero[bit_pos];
```

and a logic one level as;

```
      TRISD = TRISD | mask_one[bit_pos];
```

Note that in taking this approach only the specified bit in TRISD and PORTD is modified.

Protocol.

Note that file _1_wire.c consists of routines to call the external device's attention, to output a byte to the external device, to input a byte from the external device and to provide a source of +5 VDC as might be required for performing a temperature conversion.

File _1_wire.h consists of the prototypes and also declares the two global constant arrays discussed above.

```
// _1_wire.h
```

```
byte const mask_one[8] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};
byte const mask_zero[8] = {0xfe, 0xfd, 0xfb, 0xf7, 0xef, 0xdf, 0xbf, 0x7f};

byte _1w_init(byte bit_pos);
byte _1w_in_byte(byte bit_pos);
void _1w_out_byte(byte bit_pos, byte d);
void _1w_strong_pull_up(byte bit_pos);
```

Thus, the user may simply #include this .h file and .c file much as was done with the lcd_out files. Note that routines in _1_wire.c do use delay_ms() and delay_10us() and thus it is assumed that they exist in another included file such as lcd_out.c.

The following discusses each of the routines in _1_wire.c.

When idle, the IO terminal associated with the 1-wire device is at a high impedance. The external device's attention is attracted by bringing the IO terminal to a logic zero state for nominally 500 us and then back to a high impedance input. The PIC then looks for a presence pulse. That is, a momentary logic zero, from the external device.

This is implemented in function _1_w_init() as illustrated below. Note that the function returns TRUE or FALSE depending on whether or not the presence pulse was detected.

```
byte _1w_init(byte bit_pos)
{
   byte n=250, dir_in, dir_out;

   dir_in = TRISD | mask_one[bit_pos];
   dir_out = TRISD & mask_zero[bit_pos];

   TRISD = dir_in;            // be sure DQ is high
   PORTD = PORTD & mask_zero[bit_pos];
   TRISD = dir_out;

   delay_10us(50);      // low for 500 us

   TRISD = dir_in;

   while((PORTD & mask_one[bit_pos]) && (--n))  /* loop */  ;

   delay_10us(50);

   if (n)
   {
      return(TRUE);
   }
   else
   {
      return(FALSE);
   }
}
```

Commands and data are sent byte by byte to the external 1-wire device using function _1_w_out_byte() which sends the byte d, bit by bit beginning with the least significant. In sending each bit, the PIC brings the IO terminal low and the external device reads the logic state nominally 15 usecs after this high to low transition. Thus, in sending a one, the IO terminal is immediately returned to a logic one state such that the slave 1-wire device reads a logic one level. In sending a logic zero, the IO terminal is left low for nominally 60 usecs such that the slave sees a logic zero level.

```
void _1w_out_byte(byte bit_pos, byte d)
{
   byte n, dir_in, dir_out;

   dir_in = TRISD | mask_one[bit_pos];
   dir_out = TRISD & mask_zero[bit_pos];

   PORTD = PORTD & mask_zero[bit_pos];

   for(n=0; n<8; n++)
   {
      if (d&0x01)
      {
         TRISD = dir_out;           // momentary low
         TRISD = dir_in;
         delay_10us(6);
      }

      else
      {
          TRISD = dir_out;
          delay_10us(6);
          TRISD = dir_in;
      }
      d=d>>1;                       // next bit in lsbit position
   }
}
```

Timing is important as the external device reads the state of the line some 15 usecs after the one to zero transition. Thus, the input and output values of the TRISD register are calculated and saved in variables dir_in and dir_out prior to communicating with the external device. The result is that the IO terminal may be manipulated quickly with predictable timing as;

```
// send a logic one
TRISD = dir_out;        // momentary low
TRISD = dir_in;         // and then quickly back to high impedance
delay_10us(6);
```

or

```
// send a logic zero
TRISD = dir_out;        // low for 60 usecs
delay_10us(6);
TRISD = dir_in;
```

Note that if the application involves other processes using interrupts, interrupts should be turned off during this critical timing. For example;

```
// send a logic one
while(gie)
{
    gie = 0;                  // turn off interrupts
}
TRISD = dir_out;        // momentary low
TRISD = dir_in;         // and then quickly back to high impedance
delay_10us(6);
gie = 1;
```

This should not be a serious impediment as the interrupts are disabled for only 60 - 70 usecs and it is important to note that the interrupt are not lost, but the processing is simply delayed.

Data is received using function _1_w_in_byte().

```
byte _1w_in_byte(byte bit_pos)
{
   byte n, i_byte, temp, dir_in, dir_out;

   dir_in = TRISD | mask_one[bit_pos];
   dir_out = TRISD & mask_zero[bit_pos];

   PORTD = PORTD & mask_zero[bit_pos];

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

      // disable interrupts if required
      TRISD = dir_out;  // bring low pin low
      TRISD = dir_in;   // and back to high Z
#asm
      NOP
      NOP
      NOP
      NOP
#endasm
      temp = PORTD;           // read port
      if (temp & mask_one[bit_pos])
      {
        i_byte=(i_byte>>1) | 0x80;  // least sig bit first
      }
      else
      {
        i_byte=i_byte >> 1;
      }
      delay_10us(6);
      // enable interrupts if necessary
   }
   return(i_byte);
}
```

Note that data is read, bit by bit, beginning with the least significant bit, by bringing the IO terminal low and then quickly configuring back to a high impedance input and reading the state of the lead.

Here again, timing is critical and this is reflected in the routine in several ways.

1. As with the outputting of a byte, the two values of the TRIS register are first calculated thus leaving the winking of the IO terminal as simply;

```
       TRISD = dir_out;  // bring low pin low
       TRISD = dir_in;   // and back to high Z
```

2. In reading the state of the IO, PORTD is copied into variable "temp" and "temp" is then operated on to determine the state of the IO lead.

3. If interrupt processes are running, interrupts should be disabled and may again be enabled as noted above.

Note that as each bit is read, variable i_byte is shifted to the right and the state of the bit is inserted in the most significant bit position such that after reading eight bits, the first bit read is in the least significant bit position of i_byte.

When the external 1-wire device is a thermometer such as the DS1820 and it is operated in the parasite power mode, the +5 VDC though 4.7K is not sufficient to provide the current required to perform the temperature conversion. Thus, routine _1w_strong_pull_up outputs a hard logic one (+5 VDC) for nominally 750 ms.

```
void _1w_strong_pull_up(byte bit_pos)
{
   byte dir_in, dir_out;

   dir_in = TRISD | mask_one[bit_pos];
   dir_out = TRISD & mask_zero[bit_pos];


   PORTD = PORTD | mask_one[bit_pos];     // hard logic one
   TRISD = dir_out;
   delay_ms(750);
   TRISD = dir_in;                  // and back to high impedance

   PORTD = PORTD & mask_zero[bit_pos];
}
```

Notes.

Many Dallas application notes show an outboard FET to supply the current necessary when performing a temperature measurement. I assume their reason for doing so is that TTL devices and some CMOS devices can sink currents of typically 10 mA, but are unable to source any appreciable current without the output voltage drooping. However, this limitation does not apply to the Microchip PIC family and thus this outboard FET or similar is not required.

The specification for the Dallas DS1820 to perform a temperature conversion was a minimum of 500 ms. However, in late '00, the DS1820 was discontinued and replaced with the DS18S20 and I am told the "S" indicated software compatibility with the DS1820. However, a close examination of the data sheet for the DS18S20 indicates this minimum has been increased to 750 ms. Thus, the "S" might be interpretted as "Sort" of "Software" compatible. To add to the confusion, the DS18S20 devices are actually labeled "DS1820". However, the two devices may be distinguished by their packaging. The old DS1820 was packaged in a PR35 package (an elongated TO92). The new DS18S20 (marked DS1820) is in a TO92 package.

In developing the various routines in file _1_wire.c, I attempted to be robust in permitting any bit on PORTD to be used without disturbing other bits that might be used for other purposes. I happened to use PORTD. However, any port might be used by simply modifying the references to TRIS_DS1820 and PORT_DS1820 and then using #define statements. For example;

```
#define TRIS_DS1820 TRISB
#define PORT_DS1820 PORTB
```

Program 1820_1.C.

This routine performs temperature measurements on each of up to four DS1820s and displays the results of each measurement on the LCD. Note that only a single DS1820 is connected to each PIC IO.

Each measurement sequence begins with a call to _1_w_init() which returns TRUE if a presence pulse is detected. If the presence pulse is not detected, the function returns FALSE and the program displays "Not Detected" on the LCD. Note that this requires that each IO be pulled to +5 VDC through a 4.7K resistor. Otherwise, an open input may be read as a zero which is interpreted as the sensor being present.

If the sensor is present, the "Skip ROM" command (0xcc) is output. This indicates to the DS1820 device that the 64-bit serial number will not be sent. That is, there is no addressing. This is followed by the command to perform a temperature measurement (0x44), followed by nominally 750 ms of strong pull-up to provide sufficient current for the DS1820 to perform the measurement.

The result is then read by a call to _1_w_init(), followed by the "Skip ROM" command and the command to read the results (0xbe). The nine bytes are then read and displayed on the LCD.

The program then proceeds to the next sensor.

```
// 1820_1.C
//
// Illustrates implementation of Dallas 1-wire interface.
//
// Continually loops, reading nine bytes of data from DS1820 thermometer
// devices on PORTD0 - PORTD3 and displays the results for each sensor on
// the LCD.
//
// PIC16F877                             DS1820
//
// PORTD3 (term 22) --------------------- DQ
// PORTD2 (term 21) --------------------- DQ
// PORTD1 (term 20) --------------------- DQ
// PORTD0 (term 19) --------------------- DQ
//
// Note that a 4.7K pullup resistor to +5 VDC is on each DQ lead.
//
// copyright, Peter H. Anderson, Georgetown, SC,, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <_1_wire.h>

#define FALSE 0
```

```
#define TRUE !0

#define MAX_SENSORS 4

void set_next_line(byte *p);

void main(void)
{
   byte buff[9], sensor, n, line = 0;

   pspmode = 0;   // configure parallel slave port as general purpose port

   while(1)
   {
      lcd_init();

      for (sensor=0; sensor<MAX_SENSORS; sensor++)
      {
         line = 0;
         if(!_1w_init(sensor)) // if a DS1820 is not detected
         {
            lcd_clr_line(line);
            lcd_hex_byte(sensor);
            set_next_line(&line);

            lcd_clr_line(line);
            printf(lcd_char, "Not Detected");
            set_next_line(&line);
            delay_ms(500);
        }

        else      // otherwise, perform a temperature meas
        {
            _1w_out_byte(sensor, 0xcc);  // skip ROM

            _1w_out_byte(sensor, 0x44);  // perform temperature conversion
            _1w_strong_pull_up(sensor);

            _1w_init(sensor);
            _1w_out_byte(sensor, 0xcc);  // skip ROM

            _1w_out_byte(sensor, 0xbe);

            for (n=0; n<9; n++)                 // fetch the nine bytes
            {
               buff[n]=_1w_in_byte(sensor);
            }

            lcd_clr_line(line);
            lcd_hex_byte(sensor);         // display the sensor number
            set_next_line(&line);

            lcd_clr_line(line);
            for (n=0; n<4; n++)                 // and the results
            {
               lcd_hex_byte(buff[n]);
               lcd_char(' ');
            }

            set_next_line(&line);
            lcd_clr_line(line);

            for (n=4; n<9; n++)
            {
               lcd_hex_byte(buff[n]);
               lcd_char(' ');
            }


            delay_ms(500);
         }  // end of else
      }  // of for
  } // of while
}


void set_next_line(byte *p)
{
   ++(*p);
   if (*p == 4)
   {
      *p = 0;
   }
}

#include <lcd_out.c>
#include <_1_wire.c>
```

Program 1820_2.C.

Note that this program is a nonsense program that reads the serial number of a DS1820, stores the serial number to the PICs flash EEPROM and then continually reads the serial number from flash EEPROM and uses this to address the DS1820. I say "nonsense" as the "Read ROM" will only work if there is but a single DS1820 on the PIC terminal and if there is but a single DS1820, there really is no need to use the "Match ROM" feature.

However, the intent of this program is to illustrate how to read the 8-byte DS1820 serial number and then use the "Match ROM" feature to address the DS1820 and perform a temperature measurement. One might adapt this to read the serial numbers of eight different devices, each on dedicated IO terminals and save these to EEPROM and then reconfigure with all eight devices on the same PIC IO terminal.

The serial number is read by _1_w_init, followed by the "Read ROM" command (0x33). The eight byte serial number is then read, displayed on the LCD and written to the PIC's flash EEPROM, beginning at EEPROM address 0x1000.

The serial number is then read from EEPROM and the "Match ROM" feature of the DS1820 is used. This begins with _1_w_init, followed by the "Match ROM" command (0x55), followed by the 8-byte serial number, followed by the command to perform a temperature conversion (0x44) followed by 750 ms of strong pull-up.

The result is then fetched with the sequence; _1_w_init, followed by the "Match ROM" command, followed by the 8-byte serial number, followed by the "Read Temperature" command. The nine byte result is then read and displayed on the LCD.

```
// Program 1820_2.C
//
// Reads 64-bit address from DS1820, saves to the 16F877's flash EEPROM
// and displays it on LCD.
//
// Uses 64-bit address to perform temperature measurement.  Data is
// is displayed on LCD
//
// 16F877                              DS1820
//
// PORTD0  ---------------------------- DQ (term 2)
//
// copyright, Peter H. Anderson, Georgetown, SC, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <_1_wire.h>

#define FALSE 0
#define TRUE !0

void display_data(byte *d, byte num_vals);

void put_flash_eeprom_byte(long adr, byte d);   // write a single byte to flash
void put_flash_eeprom_bytes(long adr, byte *d, byte num_vals); // write multiple
bytes

byte get_flash_eeprom_byte(long adr);     // read single byte
void get_flash_eeprom_bytes(long adr, byte *d, byte num_vals);    // read multiple
bytes

void ds1820_read_rom(byte sensor, byte *ser_num);     // fetch 8 byte serial number
void ds1820_make_temperature_meas(byte sensor, byte *ser_num, byte *result);
                           // using match ROM

void main(void)
{
   byte ser_num[8], result[9];

   lcd_init();

   pspmode = 0;   // configure parallel slave port as general purpose port


   ds1820_read_rom(0, ser_num);     // read serial number from DS1820

   printf(lcd_char, "Serial Number");
   delay_ms(1000);

   lcd_init();
   display_data(ser_num, 8);        // display the result on LCD
   delay_ms(2000);
   put_flash_eeprom_bytes(0x1000, ser_num, 8);
            // save to flash eeprom, beginning at adr 0x1000, 8 bytes

   // now fetch the serial number, address and continually perform temperature
   // measurments

   lcd_init();
   printf(lcd_char, "Now Measuring");
   delay_ms(1000);

   while(1)
   {
       lcd_init();
       get_flash_eeprom_bytes(0x1000, ser_num, 8);
                // fetch from flash ROM, 8 bytes and return in array ser_num

       ds1820_make_temperature_meas(0, ser_num, result);
       display_data(result, 9);     // display the 9 byte temperature result
       delay_ms(2000);
   }
}

void display_data(byte *d, byte num_vals)
{
  byte n, v, line = 0;

  lcd_clr_line(line);

  for (n=0; n<num_vals; n++)
  {
     v = d[n];          // intermediate variable used for debugging

     if ((n%4 == 0) && (n!=0))
     {
      ++line;
      if (line == 4)
      {
          line = 0;
      }
      lcd_clr_line(line);
     }

    lcd_hex_byte(v);
    lcd_char(' ');
  }
}

void put_flash_eeprom_bytes(long adr, byte *d, byte num_vals)
{
   byte n;
   for(n=0; n<num_vals; n++, adr++)
   {
      put_flash_eeprom_byte(adr, d[n]);
   }
}
```

```
void get_flash_eeprom_bytes(long adr, byte *d, byte num_vals)
{
   byte n;
   for(n=0; n<num_vals; n++, adr++)
   {
      d[n]=get_flash_eeprom_byte(adr);
   }
}

void put_flash_eeprom_byte(long adr, byte d)
{
   EEADRH = adr >> 8;
   EEADR = adr & 0xff;

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

byte get_flash_eeprom_byte(int adr)
{

   EEADR = adr;
   EEADRH = (adr>>8);
   eepgd = 1;
   rd = 1;
#asm
   NOP
   NOP
#endasm
   return(EEDATA);
}

void ds1820_read_rom(byte sensor, byte *ser_num)
{
   byte n, v;

   _1w_init(sensor);
   _1w_out_byte(sensor, 0x33);      // "Read ROM" command

   for(n=0; n<8; n++)
   {
      v =_1w_in_byte(sensor);
      ser_num[n] = v;         // intermediate variable used for debugging
   }
}
```

```
void ds1820_make_temperature_meas(byte sensor, byte *ser_num, byte *result)
{
   byte n;

   _1w_init(sensor);
   _1w_out_byte(sensor, 0x55);      // match ROM
   for(n=0; n<8; n++)   // followed by the 8-byte ROM address
   {
      _1w_out_byte(sensor, ser_num[n]);
   }

   _1w_out_byte(sensor, 0x44);      // start temperature conversion
   _1w_strong_pull_up(sensor);

   _1w_init(sensor);
   _1w_out_byte(sensor, 0x55);      // match ROM
   for(n=0; n<8; n++)   // followed by the 8-byte ROM address
   {
      _1w_out_byte(sensor, ser_num[n]);
   }
   _1w_out_byte(sensor, 0xbe);      // fetch temperature data (nine bytes)

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

#include <lcd_out.c>
#include <_1_wire.c>
```

Program 1820_3.C.

The intent of this program is to illustrate how to calculate an 8-bit cyclic redundancy check (CRC) which is used by Dallas to assure the integrity of the data received from a 1-wire device.

The model which is used is a shift register with feedback. This is nicely illustrated in the Dallas data sheets.

The value of the shift register is initialized to zero.

With each data bit, a feedback bit which is calculated as the data bit XORed with the least bit of the SR. The SR is then shifted right. If the feedback bit is a one, bits 7, 3 and 2 of the SR are inverted.

```
   sr_lsb = shift_reg & 0x01;
   fb_bit = (data_bit ^ sr_lsb) & 0x01;
   shift_reg = shift_reg >> 1;
   if (fb_bit)
   {
       shift_reg = shift_reg ^ 0x8c;
   }
```

This process is repeated for each bit, beginning with the least significant bit, of each byte. The final result is the final value of the SR.

This model has the interesting property that if the SR is operated on by the value in the SR, the result is zero. For example, in sending the nine byte temperature measurement, the first eight bytes are data and the last is the CRC calculated by the DS1820. Thus, if we were to use the SR model and operate on it with the first eight bytes, the result should agree with the CRC calculated by the DS1820. But, if the model is further operated on by this CRC calculated by the DS1820, a zero result should result.

There is another interesting, but troubling and undocumented property of this model. Recall, the initial value of the SR is 0x00. If this is operated on by 0x00 (same as the SR), the result will be 0x00 and you can readily see that if operated on by any number of bytes having a value of 0x00, the result will be 0x00, which indicates success.

My point is, that if you read data and have a hard ground (or possibly simply a floating input that is read consistently as a logic zero), the erroneous data may well all be 0x00 values and the CRC model will calculate this as success.

Thus, two tests are really required in ascertaining the integrity of data. If, all bytes are 0x00, it is a failure. Else, use the CRC algorithm. If the result is non-zero, it is a failure. Else, success.

In all honesty, I am uncertain I fully appreciate why the CRC algorithm is better than a checksum, but I do know there are far wiser folks out there than me and assume they know their stuff.

In this routine, a temperature measurement is performed using the "Skip ROM" mode and then operates on the SR with the nine byte result to calculate a CRC value. A value of 0x00 indicates success.

Note that I did not follow my own advice in first testing for an "all zero" condition.

```
// 1820_3.C
//
// Cyclic redundancy check (CRC).
//
// Performs temperature measurement and displays the nine values on serial LCD.
//
// Then calculates and displays CRC.  Note that the CRC of the 9 bytes should be
// zero.
//
// copyright, Peter H. Anderson, Georgetown, SC, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <_1_wire.h>

#define FALSE 0
#define TRUE !0

byte calc_crc(byte *buff, byte num_vals);

void main(void)
{
   byte buff[9], sensor=0, crc, n;

   while(1)
   {
      _1w_init(sensor);
      _1w_out_byte(sensor, 0xcc);  // skip ROM

      _1w_out_byte(sensor, 0x44);  // perform temperature conversion
      _1w_strong_pull_up(sensor);

      _1w_init(sensor);
      _1w_out_byte(sensor, 0xcc);  // skip ROM

      _1w_out_byte(sensor, 0xbe);

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

      lcd_init();

      for (n=0; n<4; n++)
      {
         lcd_hex_byte(buff[n]);
         lcd_char(' ');
      }

      lcd_clr_line(1);

      for (n=4; n<9; n++)
      {
         lcd_hex_byte(buff[n]);
         lcd_char(' ');
      }

      lcd_clr_line(2);

      crc = calc_crc(buff, 9);
      lcd_hex_byte(crc);

      delay_ms(2000);
   }
}

byte calc_crc(byte *buff, byte num_vals)
{
   byte shift_reg=0, data_bit, sr_lsb, fb_bit, i, j;

   for (i=0; i<num_vals; i++) // for each byte
   {
      for(j=0; j<8; j++)      // for each bit
      {
         data_bit = (buff[i]>>j)&0x01;
         sr_lsb = shift_reg & 0x01;
         fb_bit = (data_bit ^ sr_lsb) & 0x01;
         shift_reg = shift_reg >> 1;
         if (fb_bit)
         {
            shift_reg = shift_reg ^ 0x8c;
         }
      }
   }
   return(shift_reg);
}

#include <lcd_out.c>
#include <_1_wire.c>
```

RS232 Communication using the USART.

Nearly 30 years ago, a collegue and myself developed an Intel 8008 based processor to control an automated telephone test system. We were stars and on reflection, it was a pretty amazing design which predated C and in fact, predated an assembler. But, Lou and I frequently noted that documentation related to the 1200 baud serial communication interface always seemed to occupy 3/4 of our storage space. After six months, three drawers in a file cabinet, after a year, six drawers spread over two cabinets.

If you want a modestly priced geek door stop, pick up a book on programming a serial port. Some 1500 pages. Admittedly, much is fluff.

It isn't that serial communication is all that difficult. Just that it takes a few more words than I am prepared to write! But, let’s try a bit.

When idle, an idle is a logic one. Transmission of a byte begins with a start bit which is a logic zero, followed by each of the eight bits, beginning with the least significant, followed by a return to the idle state. Note that a "stop bit" is the same as idle and when you specify 1, 1.5 or 2.0 stop bits, you are really specifying the minimum idle time between the sending the next character.

Each bit time is 1/baud rate. Thus, for 9600 baud, each bit is close to 104 us.

On the TTL side, a logic one is near +5 VDC and a logic zero is near ground. Thus, when idle, the level is near +5 VDC, the start bit is one bit time near ground, followed by the data bits and finally, a return to the idle state, near +5 VDC. This has become known as "non-inverted" levels.

A point of confusion is that these levels are shifted to RS232 levels when going from one point to another. An RS232 logic one is defined as being less than -3.0 Volts and a logic zero as being greater than +3.0 VDC. This level shifting is performed using a MAX232, DS275, MC1488 or 1489. These voltage levels are often referred to as "inverted"

A PC Comm Port uses these RS232 levels as do all external modems, or for that matter, anything you attach to a PC COM Port. Idle is logic one and a logic one is less than -3 VDC.

However, the USART associated with the PIC uses TTL levels. Idle is a logic one and a logic one is near +5 VDC.

Thus, in interfacing a PIC's USART with a PC or a modem or similar requires a level shifter; MAX232 or similar.

However, many LCD units are capable of operating on either the TTL side (non-inverted) or the RS232 side (inverted). Thus, if a PIC is talking to an LCD unit capable of being operated in a non-inverted mode and the LCD is co-located with the PIC, there is no need for any intermediate level shifters.

In the following routines I used a BasicX Serial LCD+ to receive data from the PIC and used a BasicX BX-24 to send data to the PIC. However, this may easily be modified to interface with a PC COM port. However, in using a

PC, note that it is necessary to use a MAX232 or equivalent level shifter and of course, an extra serial port is required beyond that used to control the In Circuit Debugger.

File SER_87X.C.

File SER_87X.C contains common utility routines for serial transmission and reception. In developing this file, I attempted to maintain some consistency with module LCD_OUT.C. Thus ser_char() or ser_hex_byte() perform the same functions as lcd_char and lcd_hex_byte. Routines which are unique to the BasicX Serial LCD+ are identified with the preface; ser_lcd. For example, ser_lcd_init() and ser_lcd_backlight_on().

In addition, I opted to develop a new file, DELAY.C to implement the two delay routines; delay_ms() and delay_10us. Note that file LCD_OUT.C may also be used as a local display. However, this will require removing duplicate functions; num_to_char(), delay_ms() and delay_10us().

The USART is enabled in function asynch_enable().

```
void asynch_enable(void)
{
   trisc6 = 1; // make tx (term 25 an input)
   trisc7 = 1; // rx (term 26) input

   sync = 0;   // asynchronous
   brgh = 1 ;  // baud rate generator high speed
   SPBRG = 25;  // 9600 with 4.0 MHz clock
            // SPGRG = 129 for 9600 baud with 20.0 MHz clock

   spen = 1;    // serial port enabled

   txen = 1;  // as appropriate
   cren = 1;
}
```

Note that both the Tx and Rx IO terminals are configured as inputs (high impedance).   This state on the Tx output might be interpeted by an interfacing device as either a logic zero or a one or worse yet, wandering between the two states and thus it may be advantageous to provide a pull-up resistor to +5 VDC.

The baud rate is controlled using bit "brgh" and register SPBRG.

A character is sent using function ser_char().

```
void ser_char(char ch) // no interrupts
{
   byte n = 250;

   delay_ms(5);         //  required for Serial LCD+
   while(!txif && --n)  /* txif goes to one when buffer is empty */
   {
         delay_10us(1);
   }
   TXREG = ch;          //  This clears txif
}
```

Note that the program loops until txif is at a logic one indicating the transmit buffer is empty. As in the past, I provided a mechanism to assure the program did not loop forever but did not provide a means to deal with this error.

The 5 msec pacing delay is provided for the Serial LCD+ as some command functions require more time than one "stop" bit. This could be eliminated or at least reduced if interfacing with a terminal such as a PC COM port.

Functions ser_hex_byte() and ser_dec_byte() are implemented in the same manner as in LCD_OUT.C except that the functions call ser_char.

Function ser_out_str() transmits each character in a string until the NULL terminator is encountered.

Other output functions relate to controlling the Serial LCD+;

```
   void ser_lcd_init(void);
      // sets contrast, backlight intensity, cursor style, clears LCD
      // and locates cursor at upper left
   void ser_lcd_backlight_off(void);
   void ser_lcd_backlight_on(void);
   void ser_lcd_set_backlight(byte v);    // range is 0x00 - 0xff
   void ser_lcd_set_contrast(byte v);     // range is 0x00 - 0xff
   void ser_lcd_set_cursor_style(byte v); // underline, block, no cursor
   void ser_lcd_clr_all(void);
   void ser_lcd_clr_line(byte line);
      // clears specified line and postions cursor at left
      // of the specified line
   void ser_lcd_cursor_pos(byte col, byte line);
      // col is 0 - 19, line is 0 - 3
   void ser_lcd_set_beep_freq(byte v);    // range is 0x00 - 0xff
   void ser_lcd_beep(void);
```

On the receive side, a character is fetched using ser_get_ch(). Note that an argument indicating the maximum time in millisecs to wait for a character is passed. On detecting bit rcif as being high, the character is read from RCREG and returned to the calling routine. If timeout occurs, 0xff is returned.

My intent in implementing the wait routine was a 10 usec loop which is executed 100 times and variable t_wait is decremented. In fact, I have never spent any time poring over this innermost loop to assure it is 10 usecs.

```
char ser_get_ch(long t_wait)
// returns 0xff if no char received within t_wait ms
{
   byte loop_one_ms, n;
   char ch;

   do
   {
      loop_one_ms = 100;
      do
      {                 // assumed to be 10 instruction cycles
#asm              // check and adjust with NOPs as neccessary
      CLRWDT
#endasm
         if(rcif)
         {
            return(RCREG);
         }
      } while(loop_one_ms--);
   } while(t_wait--);
   return(0xff);  // if timeout
}
```

Two string fetch routines are provided, one to fetch characters until a specified terminal character is received and the other to fetch a specified number of characters.

```
   byte ser_get_str_1(char *p_chars, long t_wait_1,
                              long t_wait_2, char term_char);
   byte ser_get_str_2(char *p_chars, long t_wait_1,
                              long t_wait_2, byte num_chars);
```

Note that two wait times are passed, one, the maximum time to wait for the first character and the other the maximum time to wait for subsequent characters. If a timeout occurs, the number of characters received is returned.

Program TST_SER2.C.

This program illustrates the various output routines in ser_87x.c.

Note that characters may be output using either printf(ser_char, " ……") or by directly calling ser_char.

```
// TST_SER2.C
//
// Illustrates the use of ser_87x.c utility routines.
//
// Initializes UART and initializes Serial LCD (BasicX Serial LCD+)
// Continually displays "Hello World" and a byte in both decimal and
// hex formats, a long and a float.
//
// PIC16F877                  Serial LCD+
//
// RC6/TX (term 25) -------> (term 2)
//
// copyright, Peter H. Anderson, Baltimore, MD, Apr, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <ser_87x.h>
#include <delay.h>

void main(void)
{
   byte bb = 196, n;
   long ll = 1234;
   float ff = 3.2;

   ser_init();
   ser_lcd_init();
   ser_lcd_set_beep_freq(100);

   for (n = 0; n<5; n++)      // attract some attention by flashing and beeping
                              // LCD
   {
      ser_lcd_backlight_off();
      ser_lcd_beep();
      delay_ms(200);
      ser_lcd_backlight_on();
      ser_lcd_beep();
      delay_ms(200);
   }

   while(1)
   {
      ser_lcd_beep();
      ser_lcd_clr_all();
      ser_lcd_cursor_pos(0, 0);
      printf(ser_char, "Hello World");
      ser_lcd_cursor_pos(0, 1);
      printf(ser_char, "%3.2f ", ff);     // display a float

      ser_dec_byte(ll/100, 2);            // display a long
      ser_dec_byte(ll%100, 2);

      ser_lcd_cursor_pos(0, 2);
      printf(ser_char, "%u %x", bb, bb);

      ++ll;                   // modify the values
      ++bb;
      delay_ms(500);
   }
}

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

Program TST_SER3.C.

Note that in testing the receive routines, I used a BasicX BX24 to continually output a string every three seconds. The string is terminated with character 13 (0x0d). Note that file HELLO_1.BAS must be built with files SERCOM1.BAS and SERCOM3.BAS which are included in the routines. An alternative is to use a PC COM port with a MAX232 or equivalent on the PIC side.

```
' Program HELLO_1.Bas
'
' Continually sends the string "Hello World" to PIC16F877 with a delay
' of 3.0 seconds.
'
' 9600 baud, noninverted.
'
'  BX24                        PIC16F877
'
'   Term 12 -------------------> RC7/RX (term 26)
'
' Compile with SerCom3.Bas and SerialCom1.Bas
```

```
'
' copyright, Peter H. Anderson, Georgetown, SC, Mar, '01

Sub Main()

   Dim Str as String *15

   Call OpenSerialPort(1, 19200) ' for debugging

   Call DefineCom3(0, 12, &H08) ' noninverted, no parity, 8 data bits
                  ' input, output

   Call OpenSerialPort_3(9600)

   Str = "Hello World"
   Do
      Call PutStr_3(Str)
      Call PutByte_3(13)
      Call Sleep(3.0)
   Loop

End Sub
```

Program TST_SER3.C.

Program TST_SER3.C illustrates the fetching of character strings. An attempt is made to fetch a string until either timeout or until the terminating character 0x0d (13 decimal) is received and the string is then displayed on the serial LCD. An attempt is then made to fetch a second string either timeout or until five characters are received. This is then displayed on the serial LCD.

Note that this routine does not use interrupts and thus, the PIC must “camp” on-line awaiting the receipt of the characters. However, I have used this type of routine in many applications where the PIC is simply waiting for a command string from a PC or similar and then performing a task.

```
// Program TST_SER3.C
//
// Illustrates input from BX24 and output to serial LCD.
//
// BX24                 PIC16F877               Serial LCD+
// (term 12) ---------> RC7/RX (term 26)
//                      RC6/TX (term 25) ------> (term 2)
//
// Fetches character string until the character 13 (0x0d)
// is received and then outputs the string to the serial LCD.
//
// Fetches character string until five characters are received.
// Displays on the serial LCD.
//
// copyright, Peter H. Anderson, Baltimore, MD, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <ser_87x.h>
```

```
#include <delay.h>
#include <string.h>

void set_next_line(byte *p_line);

void main(void)
{
   char s[20];
   byte n, line = 0, num_chars;

   ser_init();
   ser_lcd_init();

   while(1)
   {

      if((num_chars = ser_get_str_1(s, 10000, 1000, 13)) !=0)
                        // up to 10 secs on first
                        // 1 sec on subsequent
                        // until new line char
      {
           ser_lcd_clr_line(line);
           printf(ser_char, "%d ", num_chars);
           ser_out_str(s);
      }
      else
      {
           ser_lcd_clr_line(line);
           printf(ser_char, "NULL");
      }
      set_next_line(&line);

      delay_ms(500);

      if((num_chars = ser_get_str_2(s, 10000, 1000, 5)) !=0)
                        // up to 10 secs on first
                        // 1 sec on subsequent
                        // first five characters
      {
           ser_lcd_clr_line(line);
           printf(ser_char, "%d ", num_chars);
           ser_out_str(s);
      }
      else
      {
           ser_lcd_clr_line(line);
           printf(ser_char, "NULL");
      }
      set_next_line(&line);
      delay_ms(500);
   }
}

void set_next_line(byte *p_line)
{
   ++(*p_line);
   if (*p_line == 4)
   {
      *p_line = 0;
   }
}

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

Program GETCHAR2.C.

This program uses interrupts to receive characters and implements a circular receive buffer. This permits the PIC to be performing other tasks while receiving characters which are read at a later time. Although this has been tested and appears to work well, it is new to me. I have never fielded a product using this and thus suggest that you use caution in using this. I have been known to make an error.

The buffer is implemented as an array, in this case 20 bytes with two “pointers”; rx_buffer_put_index and rx_buffer_get_index.   Both are initialized to 0.

When a character is received, the character is put into the array at index rx_buffer_put_index and the put pointer is incremented for the next character. If the put_index falls off the bottom of the array, it is wrapped back to 0. Thus, the term “circular buffer”. If the put_index is the same value as the get_index, the buffer is full and byte rx_buffer_full is set to TRUE.

```
#int_rda rda_interrupt_handler(void)
{
   x = x;   // used for debugging
   if (rx_buff_full == FALSE)
   {
      rx_buff[rx_put_index] = RCREG;      // fetch the character
      ++rx_put_index;
      if (rx_put_index >= RX_BUFF_MAX)
      {
         rx_put_index = 0;   // wrap around
      }
      if (rx_put_index == rx_get_index)
      {
         rx_buff_full = TRUE;
      }
   }
}
```

In fetching a character, if the rx_buffer_full is TRUE, the pointers are initialized to 0 and an error code of 0xff is returned to the calling routine to indicate a possible error condition. Otherwise, if the get_index is the same as the put_index, there is no new character and an error code of 0x00 is returned. Otherwise, the rx_buffer_get_index is incremented and if it exceeds the size of the array, wrapped back to 0 and the character is passed by reference to the calling routine and the error code 0x01 is returned. The significance of the 0x01 is that a character was fetched from the buffer.

```
byte get_rx_buff(byte *ch_ptr)
{
   byte error_code;
   if (rx_buff_full == TRUE)
   {
      rx_buff_full = FALSE;
      error_code = 0xff;      // overflow of rx_buff
      rx_put_index = 0;
      rx_get_index = 0;
      *ch_ptr = 0;
                  // buff was full.  returned character has no meaning
   }
   else if (rx_get_index == rx_put_index) // there is no character
   {
       error_code = 0;        // no character
      *ch_ptr = 0;
   }
   else
   {
       *ch_ptr = rx_buff[rx_get_index];
       ++rx_get_index;
       if (rx_get_index >= RX_BUFF_MAX)
       {
          rx_get_index = 0;
       }
       error_code = 1;  // success
   }
   return(error_code);
}
```

Thus, when the processor has the time, it may check to see if there is a character;

```
      success = get_rx_buff(&ch));
```

If success is 0x01, a character was available and it was fetched in variable ch. If success is 0x00, no character is available and the value of ch has no meaning. A value of 0xff indicates a buffer overflow condition.

In this routine, each character is fetched to a string s, until the character 13 (0x0d) is read. The string is null terminated and displayed on the serial LCD.

Note that in using this routine, care must be used to assure that interrupts are not turned off (gie = 0) for more than one ms for receipt of 9600 baud. That is, this routine may be used in conjunction with such time critical tasks as interfacing with the Dallas 1-wire family where the general interrupt enable is turned off for nominally 60 usecs.

```
// GETCHAR2.C
//
// Illustrates the use of a circular buffer to receive characters
// using interrupts.
//
/// BX24                 PIC16F877               Serial LCD+
// (term 12) ---------> RC7/RX (term 26)
//                      RC6/TX (term 25) ------> (term 2)
//
// copyright, Peter H. Anderson, Georgetown, SC, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <ser_87x.h>
```

```
#include <delay.h>
#include <string.h>

#define TRUE !0
#define FALSE 0

#define RX_BUFF_MAX 20

void open_rx_com(void);
int get_rx_buff(byte *ch_ptr);

void set_next_line(byte *p_line);


byte rx_buff[RX_BUFF_MAX], rx_put_index, rx_get_index, rx_buff_full;
byte x;

void main(void)
{
   char ch, s[20];
   byte success, n=0, line = 0;

   ser_init();
   ser_lcd_init();

   open_rx_com();

   pspmode = 0;
   trisd7 = 0;

   while(1)
   {
      portd7 = 1;       // blink the LED
      delay_ms(200);
      portd7 = 0;
      delay_ms(200);
      // ser_char('.');

      while((success = get_rx_buff(&ch))==1)
      {
          if (ch == 13)
          {
              s[n] = 0;
              ser_lcd_clr_line(line);
              ser_out_str(s);
              set_next_line(&line);
              n = 0;
          }

          else
          {
              s[n] = ch;
              ++n;
              if (n>=RX_BUFF_MAX)
              {
                  ser_lcd_clr_line(line);
                  printf(ser_char, "Error");
                  set_next_line(&line);
                  n = 0;
              }
           }
        }  // of inner while

        if(success == 0xff)
        {
             ser_lcd_clr_line(line);
             printf(ser_char, "Overflow");
             set_next_line(&line);
        }

   } // of outter while
}

byte get_rx_buff(byte *ch_ptr)
{
   byte error_code;
   if (rx_buff_full == TRUE)
   {
      rx_buff_full = FALSE;
      error_code = 0xff;      // overflow of rx_buff
      rx_put_index = 0;
      rx_get_index = 0;
      *ch_ptr = 0;
                  // buff was full.  returned character has no meaning
   }
   else if (rx_get_index == rx_put_index) // there is no character
   {
       error_code = 0;        // no character
      *ch_ptr = 0;
   }
   else
   {
       *ch_ptr = rx_buff[rx_get_index];
       ++rx_get_index;
       if (rx_get_index >= RX_BUFF_MAX)
       {
          rx_get_index = 0;
       }
       error_code = 1;  // success
   }
   return(error_code);
}

void open_rx_com(void)
{
    char ch;
    asynch_enable();
    rx_put_index = 0;
    rx_get_index = 0;
    rx_buff_full = FALSE;
    ch = RCREG;         // get any junk that may be in the buffer
    ch = RCREG;

    rcif = 0;
    rcie = 1;
    peie = 1;
    gie = 1;
}

void set_next_line(byte *p_line)
{
    ++(*p_line);
    if (*p_line == 4)
    {
        *p_line = 0;
    }
}

#int_rda rda_interrupt_handler(void)
{
   x = x;   // used for debugging
   if (rx_buff_full == FALSE)
   {
      rx_buff[rx_put_index] = RCREG;      // fetch the character
      ++rx_put_index;
      if (rx_put_index >= RX_BUFF_MAX)
      {
         rx_put_index = 0;   // wrap around
      }
      if (rx_put_index == rx_get_index)
      {
         rx_buff_full = TRUE;
      }
   }
}

#int_default default_interrupt_handler(void)
{
}

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

I2C Slave.

In developing the following I2C slave routines, a BX24 was configured as an I2C Master and a PIC16F877 as an I2C slave having an I2C address of 0x40.

In the following .Bas routine, the BX24 addresses the slave by sending the “start” followed by the I2C address byte with the R/W bit set to 0 (write) followed by a command. The BX24 provides a two second delay and then addresses the slave with the I2C address byte with the R/W bit at one (read). The BX24 then reads each of nine bytes, acknowledging each except the last one prior to sending a “stop”.

The BX24 I2C master routines are implemented in I2C_BX24.Bas. They are not presented in this narrative but do appear in the “routines” file. These implementations are much the same as the bit bang implementations for the PIC. Note that they do differ from the BX24 routines presented on my web page in that the sending of the NACK and ACK has been incorporated in I2C_out_byte() and I2C_in_byte().

Routines to use the BX24's Com3 are implemented in SerCom3.Bas and routines to interface with the BasicX Serial LCD+ are in LCDCntrl.Bas. These are included in the "routines".

```
' I2C_1.Bas (BX24)
'
' Compile with I2C_BX24.Bas and SerCom3.Bas and LCDCntrl.Bas
'
' Used to test I2C_SLV1.C (PIC16F877).
'
' Address I2C device with address 0x40 and commands it to perform a temperature
' measurement on Ch 0 followed by a one second delay.  The nine bytes are then
' read and displayed on a serial LCD.
'
' BX24                  Serial LCD+       PIC16F877
'
' Term 13 -------------> (term 2)
'
' Term 15 <-----------------------------------> RC4/SDA (term 23)
' Term 14 ------------------------------------> RC3/SCL (term 18)
'
' 4.7K pull-up resistors to +5 VDC on both SDA and SCL
'
' copyright, Peter H. Anderson, Georgetown, SC, Mar, '01

Public Const SDA_PIN as Byte = 15
Public Const SCL_PIN as Byte = 16

Sub Main()

   Dim Buff(1 to 9) as Byte
   Dim N as Integer

   Call DefineCom3(0, 13, &H08) ' noninverted, no parity, 8 data bits
                              ' input, output

   Call OpenSerialPort_3(9600)
   Call LCDInit()

   Do
     ' perform a temperature measurement
      Call I2C_start()
      Call I2C_out_byte(&H40)

      Call Sleep(0.005)
      Call I2C_out_byte(&H80) ' temperature measurement on Ch 0
      Call I2C_stop()

      Call Sleep(2.0)

      Call I2C_start()
      Call I2C_out_byte(&H41) '  Read
      Call Sleep(0.005)
      For N = 1 to 8

         Buff(N) = I2C_in_byte(TRUE)      ' ack after each byte
      Next

      Buff(9) = I2C_in_byte(FALSE)  ' no ack
      Call I2C_stop()

      Call DisplayResult(Buff)

   Loop
End Sub

Sub DisplayResult(ByRef Vals() as Byte)
   Dim N as Integer
   Call LCDClearAll()

   For N = 1 to 4
      Call PutHexB_3(Vals(N))
      Call PutByte_3(Asc(" "))
   Next

   Call LCDSetCursorPosition(20)    ' beginning of second line

   For N = 5 to 8
      Call PutHexB_3(Vals(N))
      Call PutByte_3(Asc(" "))
   Next

   Call LCDSetCursorPosition(40)    ' beginning of third line
   Call PutHexB_3(Vals(9))
End Sub

Sub PutHexB_3(ByVal X as Byte)
   Dim Y as Byte

   Y = X\16
   Y = ToHexChar(Y)
   Call PutByte_3(Y)
   Y = X MOD 16
   Y = ToHexChar(Y)
   Call PutByte_3(Y)
End Sub

Function ToHexChar(ByVal X as Byte) as Byte
   Dim ReturnVal as Byte
   If (X < 10) Then
      ReturnVal = X + Asc("0")
   Else
      ReturnVal = X - 10 + Asc("A")
   End If
   ToHexChar = ReturnVal
End Function
```

Program I2CSLV_1.C.

This is an implementation of an I2C slave which is driven by the BX24 as discussed above.

In I2C_slave_setup(), the SSP module is configured in the I2C slave mode and the SDA and SCL leads are configured as inputs.

In this routine, the seven bit address is hard coded;

```
      SSPADD = 0x40;
```

Note that some flexibility might be given to the user in configuring external strapping options. For example, if the lowest three bits of PORTB were used;

```
      not_rbpu = 0;           // enable weak pull-ups
      trisb2 = 1;   trisb1 = 1;   trisb0 = 1;   // inputs
      user_adr = PORTB & 0x07;
      SSPADD = 0x40 | (user_adr << 1);
```

Note that the ssp interrupt is enabled.

In main(), interrupts are enabled and the program loops until an ssp interrupt occurs. My understanding is that this occurs when a valid “start” is received, followed by the receipt of an I2C address byte which matches the assigned address.

On determining the I2C address byte was a “write” (bit stat_rw), the program then loops until the next interrupt occurs. My understanding is that this now occurs on the receipt of a byte. This is copied to variable “command”. Note that I did not use this variable in this routine.

In this routine, the task was simply one of incrementing each of nine bytes.

The program then loops, waiting for an I2C address byte. On receipt, if the W/R bit was a “read”, the nine bytes are sent to the master. After sending each byte, the program loops until interrupt. My understanding is that this interrupt occurs after receipt of the ACK from the slave.

```
// I2C_SLV1.C
//
// Illustrates use of a PIC16F87X as an I2C Slave device.  The slave address
// is fixed at 0x40 (SSPADD = 0x40).
//
// The program loops waiting for an interrupt which occurs when the assigned
// slave address is received.  If the I2C address byte is a read, nine bytes
// are sent to the master.  Otherwise, the PIC loops, waiting for a command and
// on receipt of the command, performs a task.
//
// BX24                 Serial LCD+                  PIC16F877
//
// Term 13 -------------> (term 2)
//
// Term 15 <-------------------------------------> RC4/SDA (term 23)
// Term 14 --------------------------------------> RC3/SCL (term 18)
//
// copyright, Peter H. Anderson, Georgetown, SC, Mar, '01

#case

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

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

#define TRUE !0
#define FALSE 0

void I2C_slave_setup(void);
void I2C_slave_send_bytes(byte *buff, byte num_bytes);
void do_task(byte *buff);

byte ssp_int_occurred;

void main(void)
{
    byte dummy, command;
    byte buff[9] = {0xf0, 1, 2, 3, 4, 5, 6, 7, 8};

    I2C_slave_setup();
    ssp_int_occurred = FALSE;
    gie = 1;
    while(1)
    {
       while(!ssp_int_occurred)                 /* loop */   ;

       ssp_int_occurred = 0;
       dummy = SSPBUF;

       if(stat_rw)                        // it was a read command
       {
          I2C_slave_send_bytes(buff, 9);
       }

       else
       {
          while(!ssp_int_occurred)  /* loop waiting for command*/ ;

          ssp_int_occurred = 0;
          command = SSPBUF;
          do_task(buff);
       }
   }
}

void do_task(byte *buff)
{
   byte n;
   for (n=0; n<9; n++)
   {
      ++buff[n];
   }
}

void I2C_slave_send_bytes(byte *buff, byte num_bytes)
{
   byte n;
   for (n=0; n<num_bytes; n++)
   {
      SSPBUF = buff[n];
      ckp = 1;
      n=n;  // debugging
      while(!ssp_int_occurred)      /* loop waiting for ack */    ;
      ssp_int_occurred = FALSE;
   }
}

void I2C_slave_setup(void)
{
// set that GEI is not set in this routine

   sspen = 0;

   sspm3 = 0;  sspm2 = 1;  sspm1 = 1; sspm0 = 0;      // I2C Slave Mode - 7-bit

   trisc3 = 1;    // SCL an input
   trisc4 = 1;  // SDA an input

   gcen = 0;      // no general call
   stat_smp = 0;  // slew rate controlled
   stat_cke = 0;  // for I2C
   ckp = 1;       // no stretch

   SSPADD = 0x40;

   sspif = 0;
   sspie = 1;
   peie = 1;

   sspen = 1;     // enable the module
}

#int_ssp ssp_int_handler(void)
{
      ssp_int_occurred = TRUE;
}

#int_default default_interrupt_handler(void)
{
}

#include <lcd_out.c>
```

I2C Slave - Continued.

Buoyed by my success, I extended this such that the master sent two different commands to the addressed slave;

> 0x70 + num_times – Flash an LED num_times.
> 0x80 + channel – Perform a temperature measurement on the specified channel

I2C_2.BAS (BX24).

In the following BX24 master I2C routine, the slave is addressed followed by the command 0x80 to perform a temperature measurement on channel 0. This data is then fetched and displayed on the serial LCD+. The master then addresses the slave and sends the command 0x78 to flash the LED eight times.

```
' I2C_2.Bas (BX24)
'
' Compile with I2C_BX24.Bas LCDCntrl.Bas and SerCom3.Bas
'
' Used to test I2C_SLV2.C (PIC16F877).
'
' Addresses I2C device with address 0x40 and commands it to perform a
' temperature measurement on Ch 0 (0x80 + channel)followed by a two second
' delay.  The nine bytes are then read and displayed on a serial LCD.
'
' Then addresses the same device and commands it to flash an LED 8 times (0x70 +
' num_flashes).
'
' BX24                  Serial LCD+       PIC16F877
'
' Term 13 -------------> (term 2)
'
' Term 15 <-------------------------------------> RC4/SDA (term 23)
' Term 14 -------------------------------------> RC3/SCL (term 18)
'
' 4.7K pull-up resistors to +5 VDC on both SDA and SCL
'
' copyright, Peter H. Anderson, Georgetown, SC, Mar, '01

Public Const SDA_PIN as Byte = 15
Public Const SCL_PIN as Byte = 16

Sub Main()

   Dim Buff(1 to 9) as Byte
   Dim N as Integer

   Call DefineCom3(0, 13, &H08) ' noninverted, no parity, 8 data bits
                              ' input, output

   Call OpenSerialPort_3(9600)
   Call LCDInit()

   Do
      'perform a temperature measurement
       Call I2C_start()
       Call I2C_out_byte(&H40)      ' address, write

       Call Sleep(0.005)
       Call I2C_out_byte(&H80)      ' temperature measurement on Ch 0
       Call I2C_stop()

       Call Sleep(2.0)

       Call I2C_start()
       Call I2C_out_byte(&H41)      ' address, read
       For N = 1 to 8
          Buff(N) = I2C_in_byte(TRUE)
       Next

       Buff(9) = I2C_in_byte(FALSE)
       Call I2C_stop()

       Call DisplayResult(Buff)

       ' flash LED
       Call I2C_start()
       Call I2C_out_byte(&H40)      ' address write
       Call Sleep(0.005)
       Call I2C_out_byte(&H78)      ' flash LED 8 times
       Call I2C_stop()
       Call Sleep(1.0)  ' be sure task has time to complete
   Loop
End Sub

Sub DisplayResult(ByRef Vals() as Byte)
' same as I2C_1.Bas
End Sub

Sub PutHexB_3(ByVal X as Byte)
' same as I2C_1.Bas
End Sub

Function ToHexChar(ByVal X as Byte) as Byte
' same as I2C_1.Bas
End Function
```

Program I2C_SLV2.C.

This is an extension of I2C_SLV1.C.

The program loops waiting for the I2C address match.

If the address byte indicates a “write”, the command is read. The PIC then either performs a temperature measurement on the specified DS1820 or winks the LED the specified number of times.

If the I2C address byte is a “read”, the slave sends the nine byte temperature result.

```
// I2C_SLV2.C
//
// Illustrates use of a PIC16F87X as an I2C Slave device.  The slave address
// is fixed at 0x40 (SSPADD = 0x40).
//
// The program loops waiting for an interrupt which occurs when the assigned
// slave address is received.  If the I2C address byte is a read, nine bytes are
// sent to the master.  Otherwise, the PIC loops, waiting for a command and on
// receipt of the command, performs a task.
//
// If the command is 0x8X, the slave performs a temperature measurement using a
// Dallas DS1820 on the specified channel.  If the command is 0x7X, the slave
// flashes an LED X times.
```

```
//
// BX24                 Serial LCD+                   PIC16F877
//
// Term 13 -------------> (term 2)
//
// Term 15 <------------------------------------> RC4/SDA (term 23)
// Term 14 -------------------------------------> RC3/SCL (term 18)
//
// copyright, Peter H. Anderson, Georgetown, SC, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <_1_wire.h>

#define TRUE !0
#define FALSE 0

void I2C_slave_setup(void);
void I2C_slave_send_bytes(byte *buff, byte num_bytes);

void flash_led(byte num_flashes);
void DS1820_make_temperature_meas(byte sensor, byte *buff);

byte ssp_int_occurred;

void main(void)
{
    byte dummy, command, buff[9];

    pspmode = 0;

    I2C_slave_setup();
    ssp_int_occurred = FALSE;
    gie = 1;

    while(1)
    {
       while(!ssp_int_occurred)                 /* loop */   ;

       ssp_int_occurred = FALSE;
       dummy = SSPBUF;

       if(stat_rw)                        // it was a read command
       {
           I2C_slave_send_bytes(buff, 9);
       }

       else
       {
           while(!ssp_int_occurred) /* loop waiting for command*/ ;

           ssp_int_occurred = FALSE;
           command = SSPBUF;
           if((command >> 4) == 7)  // if high nibble is a 7
           {
              flash_led(command & 0x0f);
               // flash number of times in low nibble of command
           }
           else if ((command >> 4) == 8)  // high nibble is an 8
           {
                     // perform measurement on channel in lower nibble
                     DS1820_make_temperature_meas(command&0x0f, buff);
           }
        }
   }
}

void flash_led(byte num_flashes)
{
   byte n;
   portd7 = 0;
   trisd7 = 0;    // make LED pin an output

   for (n=0; n<num_flashes; n++)
   {
       portd7 = 1;
       delay_ms(40);
       portd7 = 0;
       delay_ms(40);
   }
}

void DS1820_make_temperature_meas(byte sensor, byte *buff)
{
   byte n;

   _1w_init(sensor);
   _1w_out_byte(sensor, 0xcc);  // skip ROM

   _1w_out_byte(sensor, 0x44);  // perform temperature conversion
   _1w_strong_pull_up(sensor);

   _1w_init(sensor);
   _1w_out_byte(sensor, 0xcc);  // skip ROM

   _1w_out_byte(sensor, 0xbe);

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

void I2C_slave_send_bytes(byte *buff, byte num_bytes)
{
   byte n;
   for (n=0; n<num_bytes; n++)
   {
       SSPBUF = buff[n];
       ckp = 1;
       n=n;  // debugging
       while(!ssp_int_occurred)     /* loop waiting for ack */    ;
       ssp_int_occurred = FALSE;
   }
}

void I2C_slave_setup(void)
{
// note that GEI is not set in this routine

   sspen = 0;

   sspm3 = 0;  sspm2 = 1;  sspm1 = 1; sspm0 = 0;      // I2C Slave Mode - 7-bit

   trisc3 = 1;    // SCL an input
   trisc4 = 1;  // SDA an input

   gcen = 0;      // no general call
   stat_smp = 0;  // slew rate controlled
   stat_cke = 0;  // for I2C
   ckp = 1;       // no stretch

   SSPADD = 0x40;

   sspif = 0;
   sspie = 1;
   peie = 1;

   sspen = 1;     // enable the module
}

#int_ssp ssp_int_handler(void)
{
    ssp_int_occurred = TRUE;
}

#int_default default_interrupt_handler(void)
{
}

#include <lcd_out.c>
#include <_1_wire.c>
```

I2C Slave – Observations.

These routines were new for me. I was happy to see them work.

However, I have some concerns with the handling of trouble conditions. In the above routines, the slave is a sequential state machine moving from one state to another and the code is written with the assumption that the master is sending the correct sequences and there are no communication errors. My concern is that the processor will hang on trouble conditions. For example, it will continually loop if no ACK is received from the master after sending a byte.

I would be more comfortable with these routines if, after receipt of the I2C address byte, a timer was started and if timeout occurred prior to completion of the sequence, the program is forced back to the "home" state of looping, waiting for an I2C address match.

Note that I did not extend this to verify the operation of a PIC in the I2C slave mode with either other I2C devices or with other PICs on the I2C bus.

SPI Slave.

A BX24 was configured as an SPI Master to test the operation of a PIC16F877 in the SPI slave mode.

Program TST_SPI.Bas.

The BX24 is used as a master. The SPI routines are implemented using the bit bang technique previously discussed in the context of a PIC.

In function SPISetup(), MOSI and SCK are configured as output logic zeros, CS is configured as an output logic one and MISO as an input.

A byte is clocked out while reading the incoming byte in function SPI_IO(). Note that SCK is normally low. Output data is setup on output MOSI prior to a positive transition on output SCK. MISO is read after the negative edge of the clock.

The BX24 brings CS low and sends a one byte command of the form 0x8X where X specifies the DS1820 on which to perform a temperature measurement. The nine byte result of the previous command is received, CS is brought high and the nine byte result is displayed on the serial LCD. Note that the result obtained when issuing the first command has no meaning and is not displayed.

```
' TST_SPI.Bas (BX24)
'
' Used to test PIC16F877 in SPI slave mode.
'
' Brings CS output low and sends &H80 plus the DS1820 to be used in
' making a temperature measurement.  Receives the nine byte result of
' the previous measurement and displays on serial LCD.
'
' Compile with SerCom3.Bas, SerCom1.Bas and if using the BasicX Serial
' LCD with LCDCntrl.Bas.
'
'     BX24                          PIC16F877
'
' MOSI (term 15) -----------------> RC4/SDI (term 23
' MISO (term 16) <----------------- RC5/SDO (term 24)
' SCK (term 17) ------------------> RC3/SCK (term 18)
' CS (term 18) -------------------> RA5/AN4/SS (term 7)
'
' (Term 13) ---------------------------------------------> LCD (term 2)
'
' copyright, Peter H. Anderson, Georgetown, SC, Mar, '01

Const SO as Byte = 15
Const SI as Byte = 16
Const SCK as Byte = 17
Const CS as Byte = 18

Sub Main()

   Dim PreviousResult(1 to 9) as Byte, Channel as Byte
   Dim First as Boolean

   Call DefineCom3(0, 13, &H08) ' noninverted, no parity, 8 data bits
                        ' input, output

   Call OpenSerialPort_3(9600)
   Call LCDInit()
   First = TRUE

   Call SPISetUp()
   Do
      Call SPIMakeTemperatureReading(0, PreviousResult)
      If (First) Then
         First = FALSE
      Else
         Call DisplayResult(PreviousResult)
      End If
      Call Sleep(1.5)
   Loop
End Sub

Sub SPIMakeTemperatureReading(ByVal Ch as Byte, ByRef PreviousResult() as Byte)
    Dim Dummy as Byte
    Dim N as Integer

    Call PutPin(CS, 0)
    Dummy = SPI_IO(&H80+Ch)

    For N = 1 to 9
         PreviousResult(N) = SPI_IO(&H00) ' fetch the 9 bytes
    Next
    Call PutPin(CS, 1)
End Sub

Sub SPISetUp()
    Call PutPin(CS, 1)
    Call PutPin(SO, 0)  ' could be either a zero or one
    Call PutPin(SI, 2)  ' make it an input
    Call PutPin(SCK, 0)
End Sub

Function SPI_IO(ByVal X as Byte) as Byte
' assumes SO has been configured as an output, SCK as an output 0, SI as an
' input, CS output low

   Dim N as Byte

   For N = 1 to 8
      If (GetBit(X, 7) = 1) Then
         Call PutPin(SO, 1)
      Else
         Call PutPin(SO, 0)
      End If

      Call PutPin(SCK, 1)
      Call ShortDelay()
      Call PutPin(SCK, 0)
      Call ShortDelay()

      If (GetPin(SI) = 1) Then
         X = X * 2 + 1
      Else
         X = X * 2
      End If
   Next
   SPI_IO = X
End Function

Sub ShortDelay()
   Sleep(0.001)
End Sub

Sub DisplayResult(ByRef Vals() as Byte)
   Dim N as Integer
   Call LCDClearAll()

   For N = 1 to 4
      Call PutHexB_3(Vals(N))
      Call PutByte_3(Asc(" "))
   Next

   Call LCDSetCursorPosition(20)    ' beginning of second line

   For N = 5 to 8
      Call PutHexB_3(Vals(N))
      Call PutByte_3(Asc(" "))
   Next

   Call LCDSetCursorPosition(40)    ' beginning of third line
   Call PutHexB_3(Vals(9))
End Sub

Sub PutHexB_3(ByVal X as Byte)
   Dim Y as Byte

   Y = X\16
   Y = ToHexChar(Y)
   Call PutByte_3(Y)
   Y = X MOD 16
   Y = ToHexChar(Y)
   Call PutByte_3(Y)
End Sub

Function ToHexChar(ByVal X as Byte) as Byte
   Dim ReturnVal as Byte
   If (X < 10) Then
      ReturnVal = X + Asc("0")
   Else
      ReturnVal = X - 10 + Asc("A")
   End If
   ToHexChar = ReturnVal
End Function
```

Program SPI_SLV2.C.

The PIC16F877 is configured as an SPI slave in SPI_setup_slave(). Note that /SS is shared with an A/D input and thus it is necessary to configure the A/D to some configuration which does not use AN4. SDI, SCK and /SS are configured as inputs and SDO as an output. The module is configured as an SPI slave with slave select (SS) enabled and the SSP module is enabled.

In main(), the SSP interrupt is enabled and the program loops waiting for an interrupt which will occur when input /SS is brought low and a byte is received. SSPBUF is read to variable command and the nine bytes are successively sent to the master by writing them to SSPBUF and waiting for an SSP interrupt.

I now see that the first byte could have been sent at the same time as the command is received, but I didn’t see it at the time I was testing and debugging this routine.

Interrupts are then turned off and the PIC performs a temperature measurement on the specified DS1820 and then returns to the “home” state, awaiting the first byte of the next sequence.

In my mind, this is a whole lot less mysterious than the I2C slave. However, as with the I2C slave, it is a sequential state machine and I would still be tempted to start a timer after receipt of the first byte and if timeout occurs, abort the sequence and return to the “home” state.

Note that multiple PICs in the slave mode or a PIC in the slave mode in conjunction with other devices on the SPI bus were not tested.

```
// SPI_SLV2.C (PIC16F877)
//
// Illustrates an implementation of a PIC in an SPI slave application.
//
// Program loops, awaiting SSP interrupt when input SS is low and a byte has
// been received from the master.  Sends the nine byte result of the previous
// measurement.
//
// Processor then performs a temperature measurement on the specified DS1820.
//
//    BX24                          PIC16F877
//
// MOSI (term 15) -----------------> RC4/SDI (term 23
// MISO (term 16) <----------------- RC5/SDO (term 24)
// SCK (term 17) ------------------> RC3/SCK (term 18)
// CS (term 18) -------------------> RA5/AN4/SS (term 7)
//
// copyright, Peter H. Anderson, Georgetown, SC, Mar, '01

#case

#device PIC16F877 *=16 ICD=TRUE

#include <defs_877.h>
#include <lcd_out.h>
#include <_1_wire.h>
```

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

void SPI_setup_slave(void);
void DS1820_make_temperature_meas(byte sensor, byte *buff);

byte ssp_int_occurred;

void main(void)
{
   byte buff[9];
   byte n, command, channel, dummy;

   lcd_init();

   SPI_setup_slave();

   while(1)
   {
       sspif = 0; // kill any pending interrupt
       peie = 1;
       sspie = 1;
       ssp_int_occurred = FALSE;
       gie = 1;

       SSPBUF = 0x00;         // could have been the first byte
       while(!ssp_int_occurred)     /* loop */  ;
       command = SSPBUF;

       ssp_int_occurred = FALSE;

       for (n=0; n<9; n++)
       {
          SSPBUF = buff[n];
          while(!ssp_int_occurred)  /* loop */ ;
          dummy = SSPBUF;
          ssp_int_occurred = FALSE;
       }

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

       channel = command - 0x80;
       DS1820_make_temperature_meas(channel, buff);   // perform a new
                                    // temperature measurement
   }
}

void SPI_setup_slave(void)
{
    sspen = 0;

    pcfg3 = 0;  pcfg2 = 1;  pcfg1 = 0;  pcfg0 = 0;    // config A/Ds as 3/0
    sspm3 = 0;  sspm2 = 1;  sspm1 = 0;  sspm0 = 0;    // Configure as SPI Slave,
                                                      // SS is enabled
    ckp = 0; // idle state for clock is zero

    stat_cke = 0;              // data transmitted on falling edge
    stat_smp = 0;        // for slave mode

    trisc3 = 1;   // SCK as input
    trisc4 = 1;   // SDI as input
    trisc5 = 0;   // SDO as output
    trisa5 = 1; // SS an input

    sspen = 1;
}

void DS1820_make_temperature_meas(byte sensor, byte *buff)
{
   byte n;

   pspmode = 0;   // configure parallel slave port as general purpose port

   _1w_init(sensor);
   _1w_out_byte(sensor, 0xcc);  // skip ROM

   _1w_out_byte(sensor, 0x44);  // perform temperature conversion
   _1w_strong_pull_up(sensor);

   _1w_init(sensor);
   _1w_out_byte(sensor, 0xcc);  // skip ROM

   _1w_out_byte(sensor, 0xbe);

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

#int_ssp ssp_int_handler(void)
{
      ssp_int_occurred = TRUE;
}

#int_default default_interrupt_handler(void)
{
}

#include <lcd_out.c>
#include <_1_wire.c>
```

PIC16F628 (18-pin DIP).

I finally obtained some PIC16F628s. This is first flash Microchip device in an 18-pin DIP since the PIC16F84 and it makes the F84 appear pretty lame and yet costs considerably less ($2.05 in hundred quantities from Digikey).

It features 2K of program memory, over 200 bytes of RAM, a USART, one capture and compare (CCP) module, Timers 0, 1 and 2 and 128 bytes of EEPROM. There is no synchronous serial port (SSP), parallel slave port (PSP) nor A/D converters. However, it does feature comparators and a programmable voltage reference.

In addition, it includes virtually every clock option I can possibly imagine, including an internal oscillator mode which can be changed from 32 kHz (low power idle) to 4.0 MHz (operational).

There is no emulator to support this new device and my experience has been that emulator (or debugger) support tends to lag new devices by a few years.

However, by using the In Circuit Debugger to first debug code on a PIC16F877, I have found it a simple matter to change port identities and modify some configuration bits and port the debugged code over to the 16F628, and then program the 16F628. In fact, I was able to port the serial routines and the Dallas 1-wire routines to the 628 in less than four hours and a quarter of that time was spent debugging a problem which turned out to be no power on the PIC. Note that this approach differs from the blind "program, burn, try" technique that can be very frustrating and time consuming as the bulk of the programming is done in an environment using the ICD. Aside from saving time, the ICD approach gives the user a bit of confidence and encourages experimenting with different implementations, the result being better code.

I suggest downloading the PIC16F628 data sheet from the Microchip website as all of the I/O pins are used for multiple functions and thus various bits must be configured to use the various features.

File DEFS_F628.H

All definitions of the special function registers (bytes) and the bits in these registers are contained in the DEFS_F628.H file. As with the DEFS_877.H file, I have used upper case to denote byte registers and lower case for bits. Note that where there is commonality between the PIC16F877 and the F628, the bytes and bits have been defined exactly the same.

File DELAY.C.

This file is intended to be #included in a main file and provides functions delay_10us() and delay_ms(). The file is precisely the same as that used for the PIC16F877.

File FLASH_1.C.

This routine flashes an LED on PORTB.7 when the input at PORTB.0 is at ground. The intent is to illustrate defining the device and the use of files defs_628.h, delay.h and delay.c

```
// FLASH_1.C (PIC16F628)
//
// Flashes an LED on PORTB.7 when pushbutton on PORTB.0 is depressed.
//
// copyright, Peter H. Anderson, Baltimore, MD, Apr, '01

#case

#device PIC16F628 *=16

#include <defs_628.h>
#include <delay.h>
```

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

void main(void)
{
   rb7 = 0;
   trisb7 = 0;          // make LED an output 0
   not_rbpu = 0;        // enable weak pull- ups

   while(1)       // continually
   {
      while(!rb0) // if at logic zero
      {
         rb7 = 1;
         delay_ms(200);
         rb7 = 0;
         delay_ms(200);
      }
   }
}

#include <delay.c>
```

File SER_628.C.

This file is functionally the same as SER_87X.C discussed above except that the UART is connected to PORTB.2 and PORTB.1. Thus, the implementation is precisely the same as SER_87X.C except in function aysnch_enable, the trisb2 and trisb1 bits are configured rather than trisc6 and trisc7 on the PIC16F877.

Note that when spen is set, PORTB bits 2 and 1 are connected to the UART.

```
 void asynch_enable(void)
{
   trisb2 = 1; // make rx input
   trisb1 = 1; // tx

   sync = 0;   // asynchronous
   brgh = 1 ;  // baud rate generator high speed
   SPBRG = 25;  // 9600 with 4.0 MHz clock
            // SPGRG = 129 for 9600 baud with 20.0 MHz clock

   spen = 1;    // serial port enabled

   txen = 1;  // as appropriate
   cren = 1;
}
```

Program TST_SER2.C.

This program illustrates how to output to a serial LCD or similar and is precisely the same as tst_ser2.c for the PIC16F877.

Program SER_ADC.C.

The BasicX Serial LCD+ provides eight 10-bit A/D converters, each of which may be configured using straps for measuring 0 – 5 VDC, 0 – 10 VDC, resistance or for a 4 – 20 mA current loop. The PIC sends command code 0x18 followed by the desired channel (1-8) and receives the two byte result.

This routine continually loops performing an A/D conversion on channel 1 and displaying the result in four digit hexadecimal format.

The intent of this routine is to illustrate how to receive characters from the Serial LCD+ using ser_get_ch().

```
// SER_ADC.C (PIC16F628)
//
// Interfaces with BasicX Serial LCD+ to perform A/D conversion.
//
// PIC sends the control code 0x16 followed by the channel (1-8)
// and then receives the two byte result.  Note that the format of
// the returned data is low byte followed by high byte.
//
// Displays the result in 4-nibble hexadecimal.
//
// PIC16F628                  Serial LCD+
//
//  RB2/TX (term 8) ----------> (term 2)
//  RB1/RX (term 7) <---------- (term 1)
//
// copyright, Peter H. Anderson, Baltimore, MD, April, '01

#case

#device PIC16F628 *=16

#include <defs_628.h>
#include <ser_628.h>
#include <delay.h>

long ser_lcd_adc_meas(byte channel);

void main(void)
{
   long ad_val;

   ser_init();
   ser_lcd_init();
   ser_lcd_set_beep_freq(100);

   while(1)
   {
      ser_lcd_clr_line(0);                // clear line 0
      ad_val = ser_lcd_adc_meas(1); // perform A/D measurement
      ser_lcd_beep();
      ser_hex_byte(ad_val >> 8);    // display high byte in hex
      ser_hex_byte(ad_val & 0xff);  // followed by the low byte
      delay_ms(500);
   }
}

long ser_lcd_adc_meas(byte channel)
```

```
{
   byte ch, ad_lo, ad_hi, n;
   long ad_val;

   ch = RCREG;                            // be sure UART is clear
   ch = RCREG;

   ser_char(0x16);                        // send command
   ser_char(channel);

   if ((ad_lo = ser_get_ch(100)) == 0xff)
   {
      return(0x7fff);               // if no response
   }
   if ((ad_hi = ser_get_ch(100)) == 0xff)
   {
      return(0x7fff);               // if no second byte received
   }

   /* else */
   ad_val = ad_hi;
   ad_val = (ad_val << 8) | ad_lo;
   return(ad_val);
}

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