

Microchip PIC18CXX2

C Routines

Copyright, Peter H. Anderson, Baltimore, MD, Jan, '02

Introduction.

This is a collection of C routines for the Microchip PIC18CXX2.

This family consists of the following closely related devices;

PIC18C242	8K X 16	22 IO, 5 Ch A/D
PIC18C442	8K X 16	33 IO, 8 Ch A/D
PIC18C252	16K X 16	22 IO, 5 Ch A/D
PIC18C452	16K X 16	33 IO, 8 Ch A/D

It also includes equivalent low power devices; e.g., PIC18LC242.

Note that all of these devices are currently available from [Digikey](#)

I have attempted to limit the focus of the routines to illustrating new features associated with the PIC18CXX2 over and above the PIC16F877. Thus, I am assuming the reader is familiar with my earlier discussions of the PIC16F877.

Development Platform.

All routines were written for the PIC18C452 running at 4.0 MHz.

The routines were debugged using an [Advanced Transdata](#) RICE17A emulator with a PB18 probe. In the main, the RICE17A performed very well, but initially it seemed to have an annoying habit of occasionally stalling when using the internal PIC timers and I am uncertain as to whether this was a loose connection or the design of the emulator. After a few weeks, this problem cleared and thus, I assume I had a loose connection. Note that other than the far more expensive Microchip ICE-2000, the RICE17A is the only emulator I am aware of for the PIC18 series. Note that Advanced Transdata developed and manufactures the Serial ICD for the PIC16F87X. They seem to be a quality company.

The testing of some features required actually programming the PICs and I spent more than a day programming and erasing three windowed versions of the PIC18C452 using my trusty PIC Start Plus (Firmware Version 2.3) under MPLAB Version 5.5 to no avail. I finally pulled out a WARP-13A Programmer from Newfound Electronics and was able to successfully program devices using the Newfound software.

As an added debugging aid, I used the PICs serial IO to directly communicate with a PC COM Port at 9600 baud using HyperTerm with a direct connection to the PC COM Port..

I used the CCS Info PCW + PCH software package, Version 3.060, and found no bugs. I continue to feel that the CCS packages are superior to the far more expensive High Tech packages. My feeling is that the big appeal of the High Tech is its price. If something is expensive, it has to be good. In my opinion, this is not true in this case.

The cost of all of this is nominally \$1200.00, a bit high for a hobbyist, but not a great deal for developing a product.

When I initially undertook this effort, I was hopeful that I would conclude that development of various modules could be done using the low cost PIC16F87X In Circuit Debugger and then the code could be ported to the PIC18 with no emulator. I now have mixed emotions. Perhaps, but it takes a full five minutes to program a PIC18 using the WARP13A programmer and it is no fun going through this exercise time after time only to have the PIC sit there doing nothing. My point is that when using these high end PICs, an emulator is a rather vital tool.

In developing this material I have attempted to present applications which use inexpensive and readily available parts; a DS275 or MAX232 to interface with a PC COM port, a few LEDs, a 10K potentiometer, a speaker, an infrared receiver and a unipolar stepping motor and associated ULN2803 driver. Exceptions are a Honeywell HIH-3605 relative humidity and an Atmel AT45DB321 4 Megabyte Flash EEPROM. I have attempted to provide abbreviated ASCII style schematics in the program descriptions.

I have attempted to illustrate most aspects of the PIC18CXX2 including use of the port latches (LAT), UART, A/D, external interrupts, Timer 0, Timer 1, Timer 2 and Timer 3, use of the capture and compare (CCP) modules in the capture, compare and PWM modes, clock switch, use of the TBLRD and TBLWT and use of the synchronous serial port as an SPI master. However, there are gaps; use of the UART to receive characters, use of the SSP as an SPI and I2C slave, low voltage detection, determining the cause of a reset and probably a few more.

Why use the PIC18CXX2.

When I first heard the PIC18C announced at a Microchip seminar, my first reaction was “perhaps in ten years” and my second was “why would I ever need such power”.

I suppose I am much like everyone. If there are 33 IO pins on a PIC, then it seems wasteful to develop a design which uses only five. Similarly, if a PIC has a full 16K X 16 bits, it would be a waste to use it in an application using a measly 2K of program memory.

However, the arithmetic is that Digikey currently sells the PIC18C452 for \$5.50 each in hundred quantities. The difference between the PIC16F876, 877 and the PIC18C452 is less than \$0.25 and I have to remind myself, that I don't really have to use all of the resources of the PIC18C452. In addition to the size of the memory, the PIC18C has a good many other features that may well justify its use.

For example;

1. The PIC18CXX2 may be clocked using an external crystal as high as 10 MHz in a four times mode so as to achieve a clock of 40 MHz or 100 ns per instruction. In addition many of the addressing, stack and mathematical features of the processor, which are used by the CCS PCH compiler and thus hidden from the user should result in greater speed. I did not run any programs on the PIC16F87X and the PIC18CXX2 to compare the speeds.
2. The PIC18CXX2 provides a means to switch between the main clock and an external crystal, usually 32.768 kHz, on T1OSC0 and T1OSC1. Thus, in power critical applications, one might idle at 32.768 kHz and switch to the 4.0 MHz clock for such tasks as interfacing with Dallas 1-wire devices or interfacing with a peripheral using RS232 serial communication.

3. Timer 0 may be configured for 16-bit operation. The Timer 0 prescaler is no longer shared with the watch dog timer and thus Timer 0 might be considered a 24-bit timer or counter.
4. In addition, a new 16-bit Timer 3 which has the same properties as Timer 1 has been added and there is also a capability of assigning one timer to one capture and compare (CCP) module and the other timer to the second CCP module which I found quite useful. With four timers, Timer 0 – 3, and two CCP modules, it is hard to imagine a timing application that cannot be implemented.
5. With earlier PICs, terminal RC1 is shared with T1OSC1 and CCP2. Thus, when using an external 32.768 kHz crystal, the CCP2 terminal was unavailable. With the PIC18CXX2, the CCP2 terminal may alternately assigned to RB3.
6. The 16-bit architecture of the PIC18C amounts to a full 32K bytes. Unused program memory might be used for data logging or for playing short .wav files.
7. Earlier PICs had but a single external interrupt on RB0. With the PIC18CXX2, there are now two additional external interrupts on RB1 and RB2.
8. The PIC18CXX2 provides a 32 level stack as opposed to the 8 level stack on most midrange processors including the PIC16F87X.

I have probably missed a few other pluses.

But, my point is that any of these features may well prove in the PIC18CXX2 over the PIC16F87X.

The Future.

Microchip has announced many processors in the PIC18 series, both one time programmable and flash. In fact, so many, one wonders if there is truly a market for each entry.

Right now, Digikey shows the PIC18F452 as being available in early March, '02. In reading literature on the Microchip site, I note that they indicate ICD support. In running MPLAB, I note the appearance of the PIC18C601 and PIC18C801 which are rom-less versions of the PIC18C on the ICD pull down menu. In a news group I noted someone indicating he had samples of the PIC18F452 and Microchip had promised him an ICD.

Not much to go on, but we do know the PIC18C452 is a reality and it appears the flash version is as well. My general feeling is that inexpensive tools are very much in Microchip's interest and I suspect we will see ICD support for the flash versions of the PIC18 series. I have no idea of whether this will be the current ICD or a new design. But, recognize, all of this is speculation.

DEFS_18C.H.

In using the CCS compiler, I avoid the blind use of the various built-in functions provided by CCS; e.g., #use RS232, #use I2C, etc as I have no idea as to how these are implemented and what PIC resources are used. One need only visit the CCS User Exchange to see the confusion.

Rather, I use a header file (DEFS_18C.H) which defines each special function register (SFR) byte and each bit within these and then use the "data sheet" to develop my own utilities. This approach is close to assembly language programming without the aggravation of keeping track of which SFR contains each bit and keeping track of the register banks. The DEFS_18C.H file was prepared from the register file map and special function register summary in Section 4 of the "data sheet".

One exception to avoiding blindly using the CCS #use routines is I do use the #int feature to implement interrupt service routines.

Snippets of DEFS_18C.H.

```
#byte TOSU      = 0xffff
#byte TOSH      = 0xFFE
#byte TOSL      = 0xFFD
#byte STKPTR     = 0xFFC
#byte PCLATU     = 0xFFB
#byte PCLATH     = 0xFFA
#byte PCL        = 0xFF9
#byte TBLPTRU    = 0xFF8
#byte TBLPTRH    = 0xFF7
#byte TBLPTRL    = 0xFF6
#byte TABLAT     = 0xFF5
#byte PRODH      = 0xFF4
#byte PRODL      = 0xFF3

#byte INTCON     = 0xFF2
#byte INTCON1    = 0xFF2
#byte INTCON2    = 0xFF1
#byte INTCON3    = 0xFF0

#bit tmr0on      = T0CON.7
#bit t08bit      = T0CON.6
#bit t0cs        = T0CON.5
#bit t0se        = T0CON.4
#bit psa         = T0CON.3
#bit t0ps2       = T0CON.2
#bit t0ps1       = T0CON.1
#bit t0ps0       = T0CON.0
```

Note that I have identified bytes using uppercase letters and bits using lower case.

Thus, an entire byte may be used;

```
TRISD = 0x00;      // make all bits outputs
LATD = 0x05;       // output 0000 0101

TRISD = 0xff;      // make all bits outputs
x = PORTD;         // read PORTD or a single bit;

tris4 = 0;         // make bit 4 an output
lat4 = 1;          // bring it to a one

tris7 = 1;         // make bit 7 an input
x = port7;         // read bit 7
```

Use of upper and lower case designations requires that you use the #case directive which causes the compiler to distinguish between upper and lower case letters.

(This has a side effect that causes problems when using some of the CCS header files where CCS has been careless in observing case. For example they may have a call to "TOUPPER" in a .h file when the function is

actually named "toupper". Simply correct CCS's code to be lower case when you encounter this type of error when compiling.)

DELAY.C.

DELAY.C (and the associated header file DELAY.H) is a file which may be #included to provide 10 us and one ms delays. Note that these are simple loops that use the execution time of the instructions. They were written for a clock of 4.0 MHz where $f_{osc}/4$ is 1 us per instruction. If an interrupt occurs during execution, the time to handle the interrupt will be added to the execution time.

I have made a few modifications in the delay routines presented in previous discussions of various processors to tighten up on their accuracy. These modifications really have nothing to do with the PIC18C.

Note that in delay_10us(), the loop consists of 10 instruction cycles. The first time the loop is executed, the loop is 6 instructions shorter to allow for the time for the calling function to store the variable in t (1), to call the function (2) and the extra machine cycle when the DECFSZ is executed (1).

Function delay_ms() is implemented by calling delay_10us(99) (990 instructions) the specified number of times. The overhead associated with each loop is nominally 10 us.

Note that these are brute force delays suitable for flashing an LED or de-bouncing a switch.

```
// delay.c
//
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02

void delay_10us(byte t)
{
    #asm
        GOTO DELAY_10US_2
    DELAY_10US_1:
        CLRWDI
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
    DELAY_10US_2:
        DECFSZ t, F
        GOTO DELAY_10US_1
    #endasm
}

void delay_ms(long t)    // delays t millisecs
{
    do
    {
        delay_10us(99);    // not 100 to compensate for overhead
    } while(--t);
}
```

Program FLASH_1.C

This program simply flashes an LED on PORTD0 when input PORTB7 is at ground. Otherwise, the LED is off. The intent is to discuss the LAT registers.

```
// FLASH_1.C
//
// Flashes LED on PORTD0 (term 19) when PORTB7 (term 40) is at ground.
//
// Uses LAT instruction.
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02
//

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>

void flash(byte t_ms);

void main(void)
{
    pspmode = 0;           // use portd as general purpose IO
    latd0 = 0;
    trisd0 = 0;

    not_rbpu = 0;          // enable weak pullup resistors on portb

    while(1)
    {
        if(!portb7)
        {
            flash(50);
        }
        else
        {
            {
                latd0 = 0;
            }
        }
    }
}

void flash(byte t_ms)
{
    latd0 = 1;
    delay_ms(t_ms);
    latd0 = 0;
    delay_ms(t_ms);
}

#include <delay.c>
```

The LAT Registers.

Note that this program uses the LATD register rather than PORTD in outputting data.

When using simply the TRIS and PORT registers, there is a potential problem. Consider;

```
PORTD = 0x00;
TRISD = 0xfe;    // upper 7 bits are inputs.  least sign bit in output

portd0 = 1;
.. ..
trisd7 = 0;      // output a zero on bit 7 - FLAWED REASONING
```

Note that PORTD is initialized to 0x00 and TRISD to 0xfe and thus a logic zero is output on the least significant bit of PORTD. The high seven bits are configured as inputs.

The least significant bit of PORTD is then brought to a logic one. However, there is a bit more going on here. In implementing the portd0 = 1 command (bsf PORTD, 0), the processor reads PORTD, ors a logic one into the least significant bit position and writes the result to the output latch. Note that in addition to the least significant bit of PORTD being a logic one, the other seven bits are now the state of the inputs on the high seven bits and not the zeros as one might assume.

Thus, in executing the trisd7 = 1 instruction, the user might expect a logic zero to appear on the output. However, the actual state will be the state of the input appearing on input bit 7 when the portd0 = 1 instruction was executed which may have been several hundred lines above.

(Note that program FLASH_1.C does not illustrate this problem).

However, Microchip has added LAT registers. Another example;

```
LATD = 0x00;
TRISD = 0xfe;    // upper 7 bits are inputs.  least sign bit in output

latd0 = 1;
.. ..
trisd7 = 1;      // output a zero on bit 7
```

The output latch is set to 0x00 and the direction register TRISD is configured such that the least significant bit is an output and the other seven bits are inputs.

Now, in executing the latd0 = 1 instruction, the processor actually reads the latch (rather than the states on inputs) and thus the high seven bits of LATD are not changed. Thus, when the trisd7 bit is configured as an output, the logic zero appears on output 7 as expected.

I have lived with only the TRIS and PORT registers for so long that I always set the port bit to the desired state prior to making a bit an output and one could continue to use only the PORT and TRIS registers. However I have forced myself to use the LAT register when outputting data.

Consider some examples;

```
LATD = 0x01;
```

Write 0x01 to the latch. Of course, these states are gated only to the pins which have been configured as outputs using the TRISD register.

```
x = PORTD;
```

Variable x will now be the states of the inputs for those bits which have been configured as inputs using the TRIS register and the states of the latch for those which have been configured as outputs. There is no change here.

```
x = LATD;
```

Note that variable x is the value of the latch, regardless of the state of the TRIS register. None of the bits are the states appearing on inputs.

```
x = LATD & 0xf0 | patt[index];
```

The state of the latch is read, the lower nibble is set to the value of patt[index].

All of this seems like more words than it worth. Simply put, the LAT register permits you to read the outputs of the actual latch. This is important in implementing a command such as latd0 = 1; as this actually involves reading the latch. No other latch bits are changed.

Emulator Note.

Note that in the above program, the weak pullup resistors associated with PORTB are enabled;

```
not_rbpu = 0;
```

However, I found the emulator did not perform this function and I had to add external pull-ups.

Serial IO.

In developing this material, a PC COM port using Hyperterm (direct to COM port, 9600 baud, 8 – N – 1, no flow) was used to observe serial output from the PIC. Note that the UART associated with the PIC generates TTL levels where a logic one is near +5 VDC while the PC is compatible with RS232 levels where a logic one is less than –3 VDC and thus an intermediate level shifter is required. I used a DS275 as illustrated below. An alternative is a MAX232.

PC COM Port (9-pin)	DS275	PIC18C452
TX (term 3) ----- 7	1 -----	RC5/RX (term 26)
RX (term 2) < ----- 5	3 -----	RC6/TX (term 25)
GRD (term 5) -----	GRD	
	GRD (term 4)	
	+5 VDC (terms 2, 8)	

Program SER_18C.C.

Only the header file is shown below. There is no functional difference between this and that used for the 16F87X.

```
// ser_18c.h
//
// copyright, Peter H. Anderson, Baltimore, MD, Dec, '01

void ser_init(void);
void asynch_enable(void);
void asynch_disable(void);
void ser_char(char ch);
void ser_newline(void);
void ser_hex_byte(byte val);
void ser_dec_byte(byte val, byte digits);
void ser_out_str(char *s);
char num_to_char(byte val);

char ser_get_ch(long t_wait);
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 function ser_init() configures the UART and must be called prior to outputting any data.

Program TST_SER1.C.

This program illustrates how to output a constant string, a float, a long and a byte.

Note that although routines ser_dec_byte, ser_hex_byte, ser_newline are provided, the same functionality may be achieved using the printf function with various format specifiers.

Note that use of the 'r' (return) and 'n' (line feed). When interfacing with the PC no delay is required after sending these characters. However, when working with a serial LCD, delays may well be required as there is a bit of overhead in handling these special characters.

Program TST_SER1.C illustrates only output functions.

```
// TST_SER1.C
//
// Illustrates the use of ser_18c.c utility routines.
//
// Initializes UART and continually displays "Hello World" and a byte in
// both decimal and hex formats, a long and a float.
//
// PIC16C452                PC COM Port
//
// RC6/TX (term 25) ---- DS275 ---> (term 2)
//
// copyright, Peter H. Anderson, Baltimore, MD, Dec, '01

#case
#device PIC18C452
```

```

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

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

    ser_init();           // configures UART

    while(1)
    {

        printf(ser_char, "Hello World\r\n");
        printf(ser_char, "%3.2f\r\n", ff); // display a float

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

        printf(ser_char, "%u %x\r\n", bb, bb);

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

#include <delay.c>
#include <ser_18c.c>

```

Program SW_CLK.C

The PIC18CXX2 provides the capability to switch from the normal clock source to a crystal on the T1OSC inputs. This might be useful in power critical applications where a wakeup from sleep interrupt is not practical.

In this program, an LED on portd0 is flashed five times using the regular system clock and then the program switches to an external 32.768 kHz crystal and flashes an LED on portd1 three times.

```

// SW_CLK.C
//
// Flashes LED on portd0 (term 19) five times using the 4.0 MHz system clock and
// then flashed LED on portd1 (term 20) three times using the 32.768 kHz clock.
//
// 32.768 KHz crystal between T1OSC0 (term 15) and T1OSC1 (term 16).
//
// Note that the 32.768 kHz oscillator does not work on the RIC17A emulator.
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02

#case
#device PIC18C452

```

```

#include <defs_18c.h>
#include <delay.h>

#define FALSE 0
#define TRUE !0

void flash_portd0(byte num_flashes);
void flash_portd1(byte num_flashes);
void delay_ms_32khz(long t);          // actually about 1.2 ms

void main(void)
{
    byte n;

    pspmode = 0;          // configure PORTD as
    tloscen = 1;          // enable external 32.768 kHz crystal osc

    scs = 0;              // use 4.0 MHz clock

    while(1)
    {
        flash_portd0(5);
        delay_ms(1000);

        scs = 1;          // system clock switch to timer 1

        flash_portd1(3);
        delay_ms_32khz(1000);

        scs = 0;          // back to 4.0 MHz XT clock
    }
}

void flash_portd0(byte num_flashes)
{
    byte n;
    for (n=0; n<num_flashes; n++)
    {
        trisd0 = 0;
        latd0 = 1;
        delay_ms(200);
        latd0 = 0;
        delay_ms(200);
    }
}

void flash_portd1(byte num_flashes)
{
    byte n;
    for (n=0; n<num_flashes; n++)
    {
        trisd1 = 0;
        latd1 = 1;
        delay_ms_32khz(200);
        latd1 = 0;
        delay_ms_32khz(200);
    }
}

```

```

    }
}

void delay_ms_32khz(long t)
{
    byte i;

    while(t--)    // 10 ~ or about 1.22 ms
    {
        }
    }
}

#include <delay.c>

```

Microchip has gone to considerable lengths to assure this clock switch doesn't happen inadvertently.

The /oscen bit in the configuration word at 0x300001 must be set to a zero. Both the emulator and the WARP-13A (and PIC Start Plus) provide this option when the unit is programmed.

Bit tloscen must be set so as to enable the external oscillator and the oscillator must actually be running. Otherwise, when the sws bit is set to transfer to the T1OSC, the switch will not occur and the processor will clear the sws bit back to a zero and continue to run using the regular system clock. The sws bit may be read by the processor to determine which clock is being used.

When using the RICE-17A emulator I simply could not get the external 32.768 kHz crystal to run. As I needed a source of nominally 32.768 kHz for other routines I programmed a PIC12C509A to continually generate nominally 33 kHz and connected this to T1CKI/T1OSC1 (term 15). However, this work around was not adequate for this routine as tloscen is at zero when using the T1CKI input. Thus, the testing of this routine required programming a PIC using the WARP-13A programmer.

Note that aside from the fact that the external crystal "might" be used as a clock source for Timer 1 (and/or Timer 3) there is no connection between this and the operation of the internal timers. Rather, this clock source is stripped off prior to the prescalers and the multiplexer associated with Timers 1 and 3.

A frequency of 32.768 kHz translates to an fosc/4 of 8192 Hz or a period of 122 us or 8.19 ticks per ms. Note that in implementing the delay_ms_32khz() routine, the code while(t--) requires 10 machine cycles and thus the delay is really in multiples of 1.22 ms rather than 1.0. However, I opted not to correct this or offer up more imaginative routines due to the effort in not having the emulator available for rapid download and debugging.

However, the idea of running the processor at the 32.768 kHz is attractive, say for measuring a temperature every hour and logging to memory and switching to the 4.0 MHz clock to upload the data to a PC.

Program AD_1.C.

In program AD_1.C, the voltage on the wiper of a potentiometer is continually measured and the A/D value and the angle of the potentiometer are displayed.

I could only find one modification from the PIC16F87X. The PIC16F87X has two bits, adcs1 and adcs0, to select the A/D clock source. The PIC18C452 adds an adcs2 bit to provide for additional A/D clock source options. However, to select the internal A/D clock, the state of the adcs2 bit is a don't care.

Note the use of a macro to combine two bytes into a long. This saved me considerable time in developing these routines.

```
// Program AD_1.C
//
// Illustrates the use of the A/D using polling of the adgo bit. Continually
// measures the voltage on potentiometer on AN0 (term 2) and displays A/D value
// and angle.
//
// Uses macro to combine two bytes into a long.
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

#define FALSE 0
#define TRUE !0

#define MAKE_LONG(h, l) (((long) h) << 8) | (l)

main()
{
    long ad_val;
    float angle;

    ser_init();

    pcf3 = 0; pcf2 = 1; pcf1 = 0; pcf0 = 0;
    // config A/D for 3/0

    adfm = 1; // right justified
    adcs2 = 0; 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 to allow channel to settle
    while(1)
    {
        adgo = 1;
        while(adgo) ; // poll adgo until zero

        ad_val = MAKE_LONG(ADRESH, ADRESL);
        angle = (float) ad_val * 270.0 / 1024.0;

        printf(ser_char, "ad_val = %ld, angle = %2.1f\n\r", ad_val, angle);

        delay_ms(3000); // three second delay
    }
}
```

```
#include <delay.c>
#include <ser_18c.c>
```

Program RH_1.C.

This work is based on developments by Nakia Collins as a part of her Senior Project at Morgan State University.

This program illustrates measuring relative humidity using a [Honeywell](#) HIH-3605-A RH to Voltage sensor. The HIH-3605 ([pdf file](#)) is simply a three terminal device with power +5 VDC (V_ref), GRD and the output which provides a voltage which is proportional to the relative humidity. Thus, measuring relative humidity is simply a matter of performing an A/D conversion and then calculating the RH. A minor correction for temperature is also performed. [Additional discussion](#) in the context of a BasicX BX24 appears on my web site.

[As of this writing, I have a supply of HIH-3605A which are three terminal SIPs, quite easy for experimenters to work with. They are \$25.00 plus shipping.]

The output of the HIH-3605 varies with the relative humidity;

$$(1) \quad RH = (V_{out} / V_{ref} - 0.16) / 0.00062$$

The output of the HIH-3605 when measured using a 10-bit A/D;

$$(2) \quad V_{out} = adval / 1024 * V_{ref}$$

Where V_ref is the nominal +5.0 VDC supply which powers both the A/D and the HIH-3605.

Substituting (2) into (1) and doing a bit of algebra;

$$(3) \quad RH = 0.157 * adval - 25.80$$

where adval is the A/D measurement which is in the range of 0 to 1023.

The relative humidity might be corrected for temperature using the expression;

$$(4) \quad RH_{corrected} = RH * 1.0 / (1.0546 - 0.00216 * TC)$$

where TC is the temperature in degrees C.

or

$$(5) \quad RH_{corrected} = RH * 1 / (1.093 - 0.0012 * TF)$$

where TF is the temperature in degrees F.

[Note that this amounts to a zero percent correction at 25 degrees C, a -6.0 percent correction at 0 degrees C and a +6.0 percent correction at 50 degrees C. Note that this is six percent of the reading. For example, an uncorrected RH of 50 percent at 50 degrees C is corrected by six percent to 53 percent RH.]

[In fact, this is quite small and in many situations, one might dispense with the correction altogether. For example, in a home, the temperature is probably in the range of 20 to 30 degrees C and the correction is but a bit over one percent. In a weather station application, for an outdoor temperature of freezing, the uncorrected

reading is six percent high. But, if the uncorrected RH reading is 16 percent, does one really care that this is six percent high and the corrected RH is in fact 15 percent.

However, measuring temperature using a negative temperature coefficient (NTC) thermistor costs less than a dollar and in many applications, you probably want to know the temperature as well.]

In program RH_1.C the uncorrected RH is calculated based on 100 A/D measurements on AN1 (terminal 3), the temperature is calculated based on 100 A/D measurements on AN2 (terminal 4). The details of calculating the temperature using an NTC thermistor is treated in my earlier discussion of the PIC16F87X.

```
// Program RH_1.C
//
// Continually measures output of a Honeywell HIH-3605 and calculates and displays
// the relative humidity.
//
// A temperature measurement is also performed and the value of RH is corrected.
// Note that an NTC thermistor in a voltage divider arrangement is used for
// measuring temperature.
//
//      HIH3605                      PIC18C452
//
//  GRD - Term 1
//  OUT - Term 2 ----- AN1 (term 3)
//  +5 VDC - Term 3
//
//
//
//      +5 VDC
//      |
//      10K
//      |----- AN2 (term 4)
//      |
//      10K NTC Thermistor
//      |
//      GRD
//
// copyright, Peter H Anderson, Baltimore, MD, Jan, '02

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

#include <math.h> // for thermistor caculation.

#define FALSE 0
#define TRUE !0

#define MAKE_LONG(h, l) (((long) h) << 8) | (l)

#define A_THERMISTOR 0.000623
// Thermistor two-point model used to calculate Temperature
#define B_THERMISTOR 0.000297
```

```

float meas_RH(byte ad_channel);
float meas_T_C(byte ad_channel);

float mult_adc(byte ad_channel, long num_samps);

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

    ser_init();

    while(1)
    {
        RH = meas_RH(1);          // meas relative humidity using A/D channel 1
        T_C = meas_T_C(2);        // meas T_C using A/D channel 2
        RH_corrected = RH/(1.0546 - 0.00216 * T_C);

        printf(ser_char, "RH = %3.2f\tT_C = %3.2f\r\n", RH_corrected, T_C);

        delay_ms(5000);
    }
}

float meas_RH(byte ad_channel)
{
    float ad_val_avg, RH;

    ad_val_avg = mult_adc(ad_channel, 100);    // compute average of 100 samples
    RH = 0.157 * ad_val_avg - 25.80;
    return(RH);
}

float meas_T_C(byte ad_channel)
{
    float ad_val_avg, r_therm, T_K, T_C;

    ad_val_avg = mult_adc(ad_channel, 100);    // A/D conversion
    r_therm = 10000.0/(1024.0/ad_val_avg-1.0); // Calculate thermistor R
    T_K = 1.0/(A_THERMISTOR + B_THERMISTOR *log(r_therm)); // T_Kelvin
    T_C = T_K-273.15;                          // T_Celcius

    return(T_C);
}

float mult_adc(byte ad_channel, long num_samps)
{
    float sum;
    long n, ad_val;

    pcf3 = 0; pcf2 = 1; pcf1 = 0; pcf0 = 0;
    // config A/D for 3/0

    adfm = 1;    // right justified
    adcs2 = 0; adcs1 = 1; adcs0 = 1; // internal RC

```

```

adon=1; // turn on the A/D

chs2=0;

switch(ad_channel)
{
    case 0:    chs1=0;    chs0=0;
               break;

    case 1:    chs1=0;    chs0=1;
               break;

    case 2:    chs1=1;    chs0=0;
               break;

    default:   break;
}

delay_10us(10); // a brief delay

sum = 0.0;

for (n=0; n<num_samps; n++)
{
    adgo = 1;
    while(adgo) /* wait */;
    ad_val = MAKE_LONG(ADRESH, ADRESL);
    sum = sum + (float) ad_val;
}

return(sum/(float)num_samps);
}

#include <delay.c>
#include <ser_18c.c>

```

PWM.

In this program, capture and control module CCP1 is set up in the PWM mode. The program continually performs an A/D conversion on AN0 and uses the most significant eight bits to control the PWM duty cycle.

With the PIC18CXX2, there is little difference from the PIC16F87X except that the two bits used for 10-bit PWM have been renamed to dc1b1 and dc1b0 (as opposed to ccp1x and ccp1y). Note that 10-bit PWM is not illustrated in this example.

```

// Program PWM_1.C
//
// Varies PWM duty using potentiometer on A/D Ch0 and outputs
// the value of "duty" to LCD.
//
// PIC18C452
//
// PORTC2 (term 17) ----- transistor ----- motor ----- +V
//

```

```

// Uses 8-bit PWM. The period is 1/256 us or about 4KHz.
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

main()
{
    byte duty;

    ser_init();

    // set up A/D converter
    pcfg3 = 0; pcfg2 = 1; pcfg1 = 0; pcfg0 = 0;
    // config A/D for 3/0
    adfm = 0; // left justified - high 8 bits in ADRESH
    adcs2 = 0; adcs1 = 1; adcs0 = 1; // internal RC
    adon=1; // turn on the A/D
    chs2=0; chs1=0; chs0=0;

    delay_10us(10); // a brief delay

    // Configure CCP1
    PR2 = 0xff; // period set to max of 256 usecs - about 4 kHz
    duty = 0x00;
    CCP1L = duty; // duty initially set to zero

    // configure CCP1 for PWM operation
    ccplm3 = 1; ccplm2 = 1; // other bits are don't care

    // Timer 2 post scale set to 1:1 - Not really necessary
    toutps3 = 0; toutps2 = 0; toutps1 = 0; toutps0 = 0;

    // Timer 2 prescale set to 1:1
    t2ckps1 = 0; t2ckps0 = 0;
    tmr2on = 1; // turn on timer #2

    latc2 = 0;
    trisc2 = 0; // make PORTC.2 an output 0

    while(1)
    {
        adgo = 1;
        while(adgo) ; // poll adgo until zero
        duty = ADRESH;
        CCP1L = duty;
        printf(ser_char, "%x\r", duty);
        delay_ms(100); // so the user can see the display
    }
}

#include <delay.c>

```

```
#include <ser_18c.c>
```

Table Write and Table Read.

The PIC18C provides the ability to write to and read from program memory. Of course, with a non-flash device, the table write is a one time thing.

The organization of the PIC18C452 program memory is 16K X 16. However, Microchip addresses in a byte fashion. Thus, address 1000 decimal is actually the high byte of word 500 and 1001 is the low byte of word 500. It took me quite a while to get this into my head.

However, as each program word is in fact 16 bits wide, the program counter advances by two with each instruction. You might look at an assembly listing to verify this.

Thus, the memory range of the 16K word PIC18C452 is 0x0000 – 0x7fff (32,768) rather than the 0x0000 – 0x3fff as one might expect. [Note, however, the WARP13A programmer provides feedback as to what memory locations it is currently programming and this display is in words. That is, the programming stops at location 0x3ff. However, aside from the possible confusion, there is no action required by the user. The CCS compiler generates a .hex file and the WARP13A takes this .hex file and programs the device.]

The PIC18C provides a three byte pointer consisting of bytes TBLPTRU, TBLPTRH and TBLPTRL, providing a theoretical addressing space of 16.77 million bytes.

Data may be written to a program location by copying the data to register TABLAT, making sure the three byte pointer contains the desired address and then executing the TBLWT assembly language command. Or, data may be read, by configuring the three byte pointer to the appropriate address and executing TBLRD. The data may then be read from the TABLAT register.

Actually, this isn't quite true. Microchip provides for pre increment, post increment or leave the pointer alone;

```
TBLWT  +*
TBLWT  *+
TBLWT  *
```

And similar command to decrement the pointer; pre decrement, post decrement and leave the pointer the same.

Although the CCS manual indicates it supports these six instructions, I found it doesn't accept anything other than;

```
#asm
    TBLWT
#endasm
```

In examining the actual assembly language listing I found that this is compiled as TBLWT +*. That is, in executing either TBLWT or TBLRD, the three byte pointer is first incremented and then the operation is performed. For example, if one desires to write to location 0x001000, the three byte pointer is set to 0x000fff.

There is one additional complication. The C compiler uses the three byte pointer for its own purposes as illustrated below.

```
0000                                00722 .....    printf(ser_char, "Hello
World\r\n");
```

0004 0F16	00723 ADDLW 16
0006 6EF6	00724 MOVWF FF6
0008 0E00	00725 MOVLW 00
000A 6EF7	00726 MOVWF FF7
000C B0D8	00727 BTFSC FD8.0
000E 2AF7	00728 INCF FF7,F
0010 0008	00729 TBLRD*
0012 50F5	00730 MOVF FF5,W
0014 0012	00731 RETURN 0
0016 6548	00732 DATA 48,65

Thus, prior to fooling with the three TBLPTRx registers, it is wise to save the compiler values, then use the registers for your purposes, avoiding any C command which may also use them, and then restore the TBLPTRx registers to the original compiler values when done. A simple concept sure gets complicated!

However, once I understood all of the anomalies, life became a bit easier;

Consider the following function which displays program data. This is about as bad as it gets.

Note that variable `program_table_save` is of type struct `TABLE` which consists of bytes `u`, `h`, and `l` corresponding to the three byte pointer and `lat` corresponding the register `TABLAT`. This is initialized to the desired program location and then decremented to compensate for the pre increment nature of the `TBLRD` instruction.

The current values of registers `TBLPTRx` and `TABLAT` are saved to variable `compiler_table_save` and the values of `program_table_save` are copied to the `TBLPTRx` registers. The `TBLRD` instruction is executed and the value of `TABLAT` is copied to variable `dat`.

However, as I desired to use a `printf` statement, I feared I might be interfering with the C compiler and thus, my values of the `TBLPTRx` were saved to `program_table_save` and the values of `compiler_table_save` are again placed back in the three `TBLPTR` registers.

```
void display_program_memory(long page, long num_bytes)
{
    byte dat;
    long n;
    struct TABLE program_table_save, compiler_table_save;

    program_table_save.ptr_u = (byte) (page >> 8);
    program_table_save.ptr_h = (byte) page;
    program_table_save.ptr_l = 0x00;

    decrement_table_ptr(&program_table_save);

    for(n=0; n<num_bytes; n++)
    {
        if (n%16 == 0)
        {
            printf("ser_char, "\n\r");
        }
        TBL_to_RAM(&compiler_table_save);    // save the compiler values
        RAM_to_TBL(&program_table_save);    // fetch the program values
    }
}
```

#asm

```

        TBLRD
#endasm
    dat = TABLAT;
    TBL_to_RAM(&program_table_save);    // save these vals
    RAM_to_TBL(&compiler_table_save);  // fetch the compiler values
    printf(ser_char, "%2x ", dat);
}
}

```

Program TAB_RD1.C.

The full program appears below.

```

// Program TAB_RD1.C
//
// Reads and displays the first 256 byte of program memory to the terminal.
//
// Then displays configuration registers beginning at 0x030000
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

struct TABLE
{
    byte ptr_u;
    byte ptr_h;
    byte ptr_l;
    byte lat;
};

void display_program_memory(long page, long num_bytes);

void TBL_to_RAM(struct TABLE *p);
void RAM_to_TBL(struct TABLE *p);
void decrement_table_ptr(struct TABLE *p);

void main(void)
{
    byte x, dat;
    long n;
    struct TABLE compiler_table_save, program_table_save;

    ser_init();

    printf(ser_char, ".....\n\r");
    display_program_memory(0x0000, 256);

    printf(ser_char, ".....\n\r");
    display_program_memory(0x0300, 10);

```

```

    while(1) /* loop */ ;
}

void display_program_memory(long page, long num_bytes)
{
    byte dat;
    long n;
    struct TABLE program_table_save, compiler_table_save;

    program_table_save.ptr_u = (byte) (page >> 8);
    program_table_save.ptr_h = (byte) page;
    program_table_save.ptr_l = 0x00;

    decrement_table_ptr(&program_table_save);

    for(n=0; n<num_bytes; n++)
    {
        if (n%16 == 0)
        {
            printf(ser_char, "\n\r");
        }
        TBL_to_RAM(&compiler_table_save);    // save the compiler values
        RAM_to_TBL(&program_table_save);    // fetch the program values

#asm
        TBLRD
#endasm
        dat = TABLAT;
        TBL_to_RAM(&program_table_save);    // save these vals
        RAM_to_TBL(&compiler_table_save);    // fetch the compiler values
        printf(ser_char, "%2x ", dat);
    }
}

void TBL_to_RAM(struct TABLE *p)
{
    p->ptr_u = TBLPTRU;
    p->ptr_h = TBLPTRH;
    p->ptr_l = TBLPTRL;
    p->lat = TABLAT;
}

void RAM_to_TBL(struct TABLE *p)
{
    TBLPTRU = p->ptr_u;
    TBLPTRH = p->ptr_h;
    TBLPTRL = p->ptr_l;
    TABLAT = p->lat;
}

void decrement_table_ptr(struct TABLE *p)
{
    --(p->ptr_l);
    if (p->ptr_l == 0xff)
    {

```

```

        --(p->ptr_h);
        if (p->ptr_h == 0xff)
        {
            --(p->ptr_u);
            p->ptr_u = p->ptr_u & 0x0f;
        }
    }
}

#include <delay.c>
#include <ser_18c.c>

```

Table Write – Program TAB_WT1.C.

This program is intended to illustrate the use of both TBLWT and TBLRD.

On boot, the processor call first_time() which checks if program location 0x001000 is blank (0xff). If so, a copyright message is written to program memory. In any event, the copyright message is read and displayed on the terminal.

In all cases, a variable of type struct TABLE is initialized to the program memory location and then decremented to compensate for the pre increment nature of the TBLRD and TBLWT instructions. The compiler values of the TBLPTRx are stored to compiler_table_save prior to loading the program_table_save values. The TBLRD or TBLWT operation is then performed and when done, the compiler_table_save values are again copied back to the TBLPTR registers.

I wrote this routine as it is easy to understand. However, it really isn't practical as one can simply printf the copyright string;

```
printf(ser_char, "copyright, Peter H. Anderson .. \n\r")
```

However, the concept might have applications where specific setup data which varies from one processor to another is input by the user during the first boot. It might also be used in a security arrangement where the serial number of a DS2401 is read and written to program memory and thereafter, the DS2401 serial number is compared with that stored in program memory. Again, this may be done with the flash memory on the PIC16F87X as well.

```

// Program TAB_WT1.C
//
// Illustrates writing to and reading from program memory using TBLWT and TBLRD.
//
// Program checks program memory location 0x001000 and if blank (0xff), writes
// copyright message beginning at location 0x001000.
//
// Reads and displays message on terminal.
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>

```

```

#include <ser_18c.h>

#define FALSE 0
#define TRUE !0

struct TABLE
{
    byte ptr_u;
    byte ptr_h;
    byte ptr_l;
    byte lat;
};

byte first_time(void);                // TRUE if location is at 0xff
void write_copyright_message(void);
void display_copyright_message(void);

void TBL_to_RAM(struct TABLE *p);
void RAM_to_TBL(struct TABLE *p);
void decrement_table_ptr(struct TABLE *p);

void main(void)
{
    byte x, dat;
    long n;

    ser_init();

    printf(ser_char, "\n\r");

    if(first_time() == TRUE)
    {
        write_copyright_message();
    }

    display_copyright_message();

    while(1)    /* loop */ ;          // to keep emulator from stalling
}

byte first_time(void)
{
    struct TABLE compiler_table_save, program_table_save;
    byte dat;

    program_table_save.ptr_u = 0x00;
    program_table_save.ptr_h = 0x10;
    program_table_save.ptr_l = 0x00;

    TBL_to_RAM(&compiler_table_save);    // save the compiler values

    decrement_table_ptr(&program_table_save);
    RAM_to_TBL(&program_table_save);
    // fetch the program values into TBL pointer

#asm
    TBLRD

```

```

#endasm
    dat = TABLAT;

    RAM_to_TBL(&compiler_table_save);
    // fetch the compiler values to TBL pointer
    // printf("ser_char, \"%2x\\n\\r\", dat);

    if(dat == 0xff)
    {
        return(TRUE);
    }
    else
    {
        return(FALSE);
    }
}

void write_copyright_message(void)
{
    struct TABLE compiler_table_save, program_table_save;
    char s[80] = {"copyright, Peter H. Anderson, Baltimore, MD\\n\\r"};
    byte s_length, n;

    program_table_save.ptr_u = 0x00;
    program_table_save.ptr_h = 0x10;
    program_table_save.ptr_l = 0x00;

    TBL_to_RAM(&compiler_table_save);           // save the compiler values

    decrement_table_ptr(&program_table_save);
    RAM_to_TBL(&program_table_save);
    // fetch the program values into TBL pointer

    for(n=0; ; n++)
    {
        TABLAT = s[n];
#asm
        TBLWT
#endasm
        if (s[n] == '\\0')
        {
            break;
        }
    }

    RAM_to_TBL(&compiler_table_save);           // restore compiler values
}

void display_copyright_message(void)
{
    struct TABLE compiler_table_save, program_table_save;

    byte dat, n;

    program_table_save.ptr_u = 0x00;
    program_table_save.ptr_h = 0x10;
    program_table_save.ptr_l = 0x00;

```

```

decrement_table_ptr(&program_table_save);

for(n=0; ; n++)
{
    TBL_to_RAM(&compiler_table_save); // save the compiler values
    RAM_to_TBL(&program_table_save); // fetch the program values
#asm
    TBLRD
#endasm
    dat = TABLAT;
    TBL_to_RAM(&program_table_save); // save these vals
    RAM_to_TBL(&compiler_table_save); // fetch the compiler values
    if (dat == '\0')
    {
        break;
    }
    printf(ser_char, "%c", dat);
}

void TBL_to_RAM(struct TABLE *p)
{
    p->ptr_u = TBLPTRU;
    p->ptr_h = TBLPTRH;
    p->ptr_l = TBLPTRL;
    p->lat = TABLAT;
}

void RAM_to_TBL(struct TABLE *p)
{
    TBLPTRU = p->ptr_u;
    TBLPTRH = p->ptr_h;
    TBLPTRL = p->ptr_l;
    TABLAT = p->lat;
}

void decrement_table_ptr(struct TABLE *p)
{
    --(p->ptr_l);
    if (p->ptr_l == 0xff)
    {
        --(p->ptr_h);
        if (p->ptr_h == 0xff)
        {
            --(p->ptr_u);
            p->ptr_u = p->ptr_u & 0x0f;
        }
    }
}

#include <delay.c>
#include <ser_18c.c>

```

Program MEM_SAVE.C.

This program illustrates how any variable (byte, long, float or structure) may be written to or read from program memory. This might be useful in storing calibration constants which are calculated at run time or as a disposable data logger. Actually, at about \$5.00 for some 32K bytes, the idea of a disposable logger is not all that outlandish.

Note that a pointer is simply an address. The reason for distinguishing whether it is a pointer to a byte, a long, a float or a structure is simply how C handles arithmetic operations on the pointer. For example,

```
++p_byte;    // add one to the pointer
++p_float;   // add four to the pointer
```

Note that incrementing the pointer to the float, the value of the pointer is actually increased by four bytes.

Thus, in this example, the address of the beginning of the variable to be saved is copied to a pointer to a byte and this pointer along with the size of the variable is passed to the function. In writing a variable to program memory, the first byte pointed to by the pointer is written to memory, the pointer is advanced such that it points to the next byte of the variable, and this is written to memory and the process continues for the size of the variable. In reading a variable, the program memory byte is read from memory and written to the first byte of the variable. The byte pointer is advanced and the next byte of the variable is read from memory and copied to the next byte of the variable.

```
// Program MEM_SAVE.C
//
// Illustrates how to save a quantity to and fetch a quantity from program
// memory.
//
// Saves a float and a struct TM to EEPROM and then fetches them and displays
// on the terminal.
//
// Note that a byte pointer which points to the beginning of the quantity and the
// number of bytes is passed to each function.
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02
```

```
#case
```

```
#device PIC18C452
```

```
#include <defs_18c.h>
```

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

```
#include <ser_18c.h>
```

```
struct TABLE
```

```
{
    byte ptr_u;
    byte ptr_h;
    byte ptr_l;
    byte lat;
};
```

```
struct LONG24
```

```
{
    byte u;
    byte h;
```

```

        byte l;
};

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

void save_to_memory(struct LONG24 *p_adr, byte *p_dat, byte num_bytes);
void read_from_memory(struct LONG24 *p_adr, byte *p_dat, byte num_bytes);

void TBL_to_RAM(struct TABLE *p);
void RAM_to_TBL(struct TABLE *p);
void decrement_table_ptr(struct TABLE *p);

void main(void)
{
    float float_1 = 1.2e-12, float_2;
    struct TM t1, t2;
    struct LONG24 adr;

    byte *ptr;

    ser_init();

    printf(ser_char, "\r\n.....\r\n");

    adr.u = 0x00;  adr.h = 0x20;  adr.l = 0x00;  // 0x002000

    ptr = (byte *) &float_1;          // ptr points to first byte of float_1
    save_to_memory(&adr, ptr, sizeof(float));  // save float_1

    t1.hr = 12;    t1.mi = 45;    t1.se = 33;

    adr.u = 0x00;  adr.h = 0x20;  adr.l = 0x04;  // 0x002004
    ptr = (byte *) &t1;
    save_to_memory(&adr, ptr, sizeof(struct TM));  // save t1

    adr.u = 0x00;  adr.h = 0x20;  adr.l = 0x00;  // 0x002000
    ptr = (byte *) &float_2;
    read_from_memory(&adr, ptr, sizeof(float));

    adr.u = 0x00;  adr.h = 0x20;  adr.l = 0x04;  // 0x002004
    ptr = (byte *) &t2;
    read_from_memory(&adr, ptr, sizeof(struct TM));

    printf(ser_char, "float = %1.3e\r\n", float_2);  // print the float

    printf(ser_char, "t2 = ");
    ser_dec_byte(t2.hr, 2);
    ser_char(':');
    ser_dec_byte(t2.mi, 2);

```

```

    ser_char(':');
    ser_dec_byte(t2.se, 2);
    ser_newline();

    while(1)
    {
#asm
        CLRWDI
#endasm
    }
}

void save_to_memory(struct LONG24 *p_adr, byte *p_dat, byte num_bytes)
{
    byte n;
    struct TABLE compiler_table_save, program_table_save;

    program_table_save.ptr_u = p_adr->u;
    program_table_save.ptr_h = p_adr->h;
    program_table_save.ptr_l = p_adr->l;

    decrement_table_ptr(&program_table_save);

    TBL_to_RAM(&compiler_table_save);        // save compiler values
    RAM_to_TBL(&program_table_save);

    for (n=0; n<num_bytes; n++)
    {
        TABLAT = *p_dat;
#asm
        TBLWT
#endasm
        ++p_dat;
    }

    RAM_to_TBL(&compiler_table_save);        // restore compiler values
}

void read_from_memory(struct LONG24 *p_adr, byte *p_dat, byte num_bytes)
{
    byte n;
    struct TABLE compiler_table_save, program_table_save;

    program_table_save.ptr_u = p_adr->u;
    program_table_save.ptr_h = p_adr->h;
    program_table_save.ptr_l = p_adr->l;

    decrement_table_ptr(&program_table_save);

    TBL_to_RAM(&compiler_table_save);        // save compiler values
    RAM_to_TBL(&program_table_save);

    for (n=0; n<num_bytes; n++)
    {
#asm
        TBLRD
#endasm

```

```

        *p_dat = TABLAT;
        ++p_dat;
    }
    RAM_to_TBL(&compiler_table_save);    // restore compiler values
}

void TBL_to_RAM(struct TABLE *p)
{
    p->ptr_u = TBLPTRU;
    p->ptr_h = TBLPTRH;
    p->ptr_l = TBLPTRL;
    p->lat = TABLAT;
}

void RAM_to_TBL(struct TABLE *p)
{
    TBLPTRU = p->ptr_u;
    TBLPTRH = p->ptr_h;
    TBLPTRL = p->ptr_l;
    TABLAT = p->lat;
}

void decrement_table_ptr(struct TABLE *p)
{
    --(p->ptr_l);
    if (p->ptr_l == 0xff)
    {
        --(p->ptr_h);
        if (p->ptr_h == 0xff)
        {
            --(p->ptr_u);
            p->ptr_u = p->ptr_u & 0x0f;
        }
    }
}

#include <delay.c>
#include <ser_18c.c>

```

Use of the #ROM Directive – Program ROM_1.C.

The CCS ROM directive permits the programmer to define values in program memory when the PIC is programmed.

The form of the directive is;

```
#rom 0x1000 = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Two points.

After compiling, this does not appear in the .lst assembly file nor the .cod debugging file. As the emulator uses the .cod file for debugging, the emulator will simply read these locations as 0xffff. (The only place this data

appears is in the .hex file). Thus, when using this directive, it is necessary to actually program a PIC to test the operation..

In using this technique, a full word (16-bits) is used to store each value. Thus, the 'H' is stored at byte location 0x1000, the 'e' at byte location 0x1002, etc.

Admittedly, with some 32K bytes, in most applications one can throw away 50 percent of the memory, but there may be situations where a considerable amount of initial data needs to be stored and this 50 percent throwaway cannot be tolerated. As of this writing, I note David Houston battling to store sunrise and sunset times in the BX24's EEPROM. I don't know just how much data is involved. Another example is to store a couple of .wav files for an impressive user interface.

I found that using the #rom directive in the following manner, I was able to store each character in a single byte. That is, two bytes per word.

```
#rom 0x1000 = { 'H' }
#rom 0x1001 = { 'e' }
#rom 0x1002 = { 'l' }
#rom 0x1003 = { 'l' );
#rom 0x1004 = { 'o' }
#rom 0x1005 = { '\0' }
```

In program ROM_1.C this technique is used to initialize a string and, display the string at run time. In program ROM_2.C (not shown in this discussion), this is extended such that multiple strings are stored in a continuous memory area. To display the nth string, the program traverses the memory counting the number of '\0' characters. That is to display string 2, the program traverses the memory until it has found two '\0' characters corresponding to string 0 and string 1, and then fetches the characters until the '\0' character associated with string 2 is found.

```
// ROM_1.C
//
// Illustrates how to initialize memory locations byte by byte using the #rom
// directive.
//
// copyright, Peter H Anderson, Baltimore, MD, Jan, '02
```

```
#case
```

```
#device PIC18C452
```

```
#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>
```

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

```
struct TABLE
{
```

```
    byte ptr_u;
    byte ptr_h;
    byte ptr_l;
    byte lat;
```

```

};

void get_str_mem(byte str_num, char *s);
void out_str(char *s);

void TBL_to_RAM(struct TABLE *p);
void RAM_to_TBL(struct TABLE *p);
void decrement_table_ptr(struct TABLE *p);

void main(void)
{
    byte n;
    char s[80];

    ser_init();
    while(1)
    {
        printf(ser_char, "\n\r.....\n\r");
        get_str_mem(0, s);
        out_str(s);
        ser_newline();
        delay_ms(500);
    }
}

void get_str_mem(byte str_num, char *s)
{
    byte n, dat;

    struct TABLE program_table_save, compiler_table_save;

    program_table_save.ptr_u = 0x00;
    program_table_save.ptr_h = 0x10;
    program_table_save.ptr_l = 0x00;

    decrement_table_ptr(&program_table_save);

    TBL_to_RAM(&compiler_table_save); // save the compiler values
    RAM_to_TBL(&program_table_save); // fetch the program values

    n = 0;
    while(1)
    {
        #asm
            TBLRD
        #endasm
        dat = TABLAT;
        s[n] = dat;
        if (s[n] == '\0')
        {
            break;
        }
        ++n;
        if (n==80) // get out if no NULL character

```

```

        {
            break;
        }
    }
    RAM_to_TBL(&compiler_table_save);    // restore the compiler values
}

void out_str(char *s)    //display the string
{
    byte n;

    n=0;
    while(s[n])    // while not NULL
    {
        ser_char(s[n]);
        ++n;
        if (n==80)
        {
            break;
        }
    }
}

void TBL_to_RAM(struct TABLE *p)
{
    p->ptr_u = TBLPTRU;
    p->ptr_h = TBLPTRH;
    p->ptr_l = TBLPTRL;
    p->lat = TABLAT;
}

void RAM_to_TBL(struct TABLE *p)
{
    TBLPTRU = p->ptr_u;
    TBLPTRH = p->ptr_h;
    TBLPTRL = p->ptr_l;
    TABLAT = p->lat;
}

void decrement_table_ptr(struct TABLE *p)
{
    --(p->ptr_l);
    if (p->ptr_l == 0xff)
    {
        --(p->ptr_h);
        if (p->ptr_h == 0xff)
        {
            --(p->ptr_u);
            p->ptr_u = p->ptr_u & 0x0f;
        }
    }
}

#rom 0x1000 = {'M'}
#rom 0x1001 = {'o'}
#rom 0x1002 = {'r'}
#rom 0x1003 = {'g'}

```

```

#rom 0x1004 = {'a'}
#rom 0x1005 = {'n'}
#rom 0x1006 = {' '}
#rom 0x1007 = {'S'}
#rom 0x1008 = {'t'}
#rom 0x1009 = {'a'}
#rom 0x100a = {'t'}
#rom 0x100b = {'e'}
#rom 0x100c = {' '}
#rom 0x100d = {'U'}
#rom 0x100e = {'n'}
#rom 0x100f = {'i'}
#rom 0x1010 = {'v'}
#rom 0x1011 = {'e'}
#rom 0x1012 = {'r'}
#rom 0x1013 = {'s'}
#rom 0x1014 = {'i'}
#rom 0x1015 = {'t'}
#rom 0x1016 = {'y'}
#rom 0x1017 = {'\0'}

#include <delay.c>
#include <ser_18c.c>

```

Although this may appear to involve a good deal of typing, the use of C on a desktop PC can, for example open a .wav file, read it to an array, and then fprintf each element of the array to a ASCII file which may then be pasted into or #included in the CCS .C file.

A sample program GEN_TBL.C written using Borland's old TurboC Version 2.01 appears below. Note that it opens "click.wav" and reads the file to an array. Each element of the array is then output to a text file in the above form. Note that this compiler is now free from [Borland Community](#). Click on "Museum".

```

/* GEN_TBLE.C - Borland TurboC - Version 2.01
**
** Opens binary file "click.wav" and reads to array and writes to text file
** "wav_dat.c" in the form.
**
** #rom 0x1000 = {0x3c}
** #rom 0x1001 = {0x45}
** etc
**
** Note that the first 100 bytes of the .wav file are discarded as these are setup
** bytes. I am actually uncertain as to the exact number.
**
** The resulting file may then either be pasted into of included in the CCS C
** file.
**
** copyright, Peter H Anderson, Baltimore, MD, Jan, '02
*/

#include <stdio.h>

void main(void)
{
    FILE *f_binary, *f_txt;

```

```

unsigned int n, address, num_bytes;
unsigned char a[32000];

if ((f_binary = fopen("click.wav", "rb")) == NULL)
{
    printf("Error opening .wav file\n");
    exit(0);
}

if ((f_txt = fopen("wav_dat.c", "wt")) == NULL)
{
    printf("Error opening output file\n");
    exit(0);
}

num_bytes = fread(a, 1, 32000, f_binary);    /* read binary file into array */

for (n=100, address = 0x1000; n<num_bytes; n++, address++)
{
    /* output each value in the array, beginning with element 100 */
    fprintf(f_txt, "#rom 0x%.4x = {0x%.2x}\n", address, a[n]);
}

fcloseall();
}

```

```
#ifdef OUTPUT
```

Sample of the output.

```
*****
```

```

#rom 0x1000 = {0x65}
#rom 0x1001 = {0xbd}
#rom 0x1002 = {0x00}
#rom 0x1003 = {0x00}
#rom 0x1004 = {0xff}
#rom 0x1005 = {0x7f}
#rom 0x1006 = {0x8f}
#rom 0x1007 = {0xbf}
#rom 0x1008 = {0x41}
#rom 0x1009 = {0x8c}
#rom 0x100a = {0x00}
#rom 0x100b = {0x34}
#rom 0x100c = {0x72}
#rom 0x100d = {0x5d}
#rom 0x100e = {0x00}

```

```
#endif
```

Thus, the PIC might have the ability to “play” a few short .wav file. Set up Timer 1 for 125 us timeout (8000 samples per second), and on interrupt, fetch from the table and output to a D/A on PORTD. However, note that even 32K bytes of program memory is none too much when it comes to .wav files.

Interrupts.

With the PIC16F87X, an interrupt enable and an interrupt flag are associated with each interrupt source. In many cases the peie bit must also be set. The processor is interrupted when the gie bit is set and an interrupt event occurs from an enabled source. On interrupt, execution of the current instruction is completed, the return address is saved on the stack and the program is directed to program location 0x004. CCS then provides code to cause the program to jump to the appropriate service routine and CCS also takes care of clearing the interrupt flag and the return from interrupt. The address is pulled off the stack and the program continues with whatever it was doing.

With the PIC18CXX2, many additional interrupt sources have been added.

However, the significant addition is the ability to accommodate two levels of interrupts. Interrupt prioritization is enabled by setting bit ipen. Virtually all interrupt sources now have an additional priority bit; e.g., tmr1ip which if set assigns the interrupt to high priority and if clear to low priority. Bit gie has been renamed gieh (high priority and the peie bit now carries a dual designation of giel (low). Thus, for the two priority level, ipen is one and interrupts are assigned as either high or low by setting or clearing their respective priority bits. High and low priority interrupts are enabled by setting gieh and giel. On interrupt, the return address is pushed on the stack and the program vectors to either program location 0x0008 high priority) or to 0x0018 (low priority).

I say, significant. In fact, it is not as CCS has no means to handle this. In examining the assembly, CCS inserts NOPs from location 0x0008 through location 0x0017 and then begin their interrupt handler. Thus, you might be deceived into believing there is prioritization, but there isn't. All interrupts go to location 0x0018.

My suggestion is to handle interrupts as with the PIC16F87X. Be sure that the ipen bit is cleared to zero and all interrupts then have the same priority and all interrupts redirect program flow to location 0x0008 and CCS then handles the code to direct the program to the appropriate interrupt service routine.

In fairness to CCS, I can see their rationale. After all, they provide a directive of the form;

```
#priority int3, timer1, timer3
```

CCS then uses this in their interrupt handler to determine the order in which the interrupt flags are examined. Thus, they have implemented pretty much the same task in software as Microchip has done in hardware.

One might argue that the Microchip hardware approach permits priorities to be changed during the course of the program, but off hand I can't think of any application where this would be a significant advantage.

External Interrupts – Program EXT_INT1.C

The number of external interrupts has been increased from one (INT0 on PORTB0) to three; INT0, INT1 and INT2. (Actually with the interrupt on change which is associated with PORTB4 – PORTB7, all but PORTB3 are now associated with interrupts and PORTB3 may be configured with a CCP2 module).

In program EXT_INT1.C, all three interrupts are configured for interrupt on the rising edge (intedg2, 1, and 0), global variables int0_occ, int1_occ and int2_occ are all set to FALSE and each interrupt is enabled and global interrupts are enabled (gieh = 1).

The program then loops and if any of the intx_occ variables are TRUE, interrupts are momentarily turned off and the LED is flashed with a delay between the flashes to distinguish the source of the interrupt.

Note that the intx_occ variables are set to TRUE in the interrupt service routines. Recall that the only way to communicate with an interrupt service routine is by using global variables.

```

// Program EXT_INT1.C
//
// Illustrates the use of external interrupts on INT0/RB0, INT1/RB1 and
// INT2/RB2.
//
// All interrupts are configured for rising edge.
//
// On interrupt, LED on PORTD0 is flashed at various speeds, depending on the
// interrupt source.
//
// copyright, Peter H Anderson, Baltimore, MD, Jan, '02

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

#define FALSE 0
#define TRUE !0

void flash_led(byte num_flashes, long delay_time);

byte int0_occ, int1_occ, int2_occ; // note globals

void main(void)
{
    not_rbp_u = 0; // internal pullups

    pspmode = 0; // configure PORTD for general purpose IO

    latd0 = 0; // LED is an output
    trisd0 = 0;

    int0_occ = FALSE; int1_occ = FALSE; int2_occ = FALSE;

    intedg0 = 1; intedg1 = 1; intedg2 = 1; // interrupt on rising edge
    ipen = 0; // disable interrupt priority mode

    while(1)
    {
        int0ie = 1; int1ie = 1; int2ie = 1; // enable ints
        gieh = 1;

        if (int0_occ)
        {
            while(gieh)
            {
                gieh = 0;
            }
            int0_occ = FALSE;

            flash_led(5, 200);
            int0if = 0;
            gieh = 1;
        }
    }
}

```

```

    }

    if (int1_occ)
    {
        while(gieh)
        {
            gieh = 0;
        }
        int1_occ = FALSE;

        flash_led(5, 100);
        int1if = 0;
        gieh = 1;
    }

    if (int2_occ)
    {
        while(gieh)
        {
            gieh = 0;
        }
        int2_occ = FALSE;

        flash_led(5, 50);
        int2if = 0;
        gieh = 1;
    }
}

void flash_led(byte num_flashes, long delay_time)
{
    byte n;

    for (n=0; n<num_flashes; n++)
    {
        latd0 = 1;
        delay_ms(delay_time);
        latd0 = 0;
        delay_ms(delay_time);
    }
}

#int_ext
int0_int_handler(void)
{
    int0_occ = TRUE;
}

#int_ext1
int1_int_handler(void)
{
    int1_occ = TRUE;
}

```

```
#int_ext2
int2_int_handler(void)
{
    int2_occ = TRUE;
}

#int_default
default_int_handler(void)
{
}

#include <delay.c>
```

Timer 1 and Timer 3.

One of the greatest improvements in the PIC18CXX2 is the addition of another 16-bit timer; Timer 3. It operates precisely the same as Timer 1 with its own prescale and clock source multiplexer. The clock source for either timer may be either the fosc/4 internal clock or the input on T13CKI (or the external crystal on T1OSCO and T1OSCI).

In addition, the two timers may be assigned to the two CCP modules, Timer 1 controlling both or Timer 1 controlling one and Timer 3 the other. I found this quite useful.

[There is an improvement in writing and reading from the timers which I did not illustrate in these sample routines.. Normally, writing to or reading from timer 1 is a two step process. For example;

```
x = TMR1H;
y = TMR1L;
```

The problem here is that as the two reads do not occur at the same time, there is a danger that there was an overflow from the low to the high byte. For example if the timer was at 0x00fe, one might well read 0x00 and 0x03 when in fact, the value was 0x01 and 0x03. This is usually solved by reading TMR1H again and if it is not the same as the previous, going back and reading the whole thing again.

With the PIC18CXX2, Timers 1 and 3 have a t1rd16 (t3rd16) bit which when set provides for a 16-bit transfer. TMR1H is then actually a holding register.

```
x = TMR1L;
y = TMR1H;
```

Note that when TMR1L is read, the hardware transfers the eight high bits into TMR1H.

Similarly,

```
TMR1H = x;
TMR1L = y;
```

Again, TMR1H is a holding register and is not loaded to the timer until TMR1L is updated.]

Program TIMER3_1.C.

I spent more time on this routine than I care to note. I finally traced the problem to the 32.768 kHz crystal on T1OSC0 and T1OSC1. When using the emulator, the crystal would not oscillate. I finally programmed at PIC12C509A to output a continuous 33 kHz and used it as a source on input T13CKI. This required that bit t1osc be cleared to zero.

[Note that this code for the PIC12C509A is included in the file directory as 32KHZ.C.]

[A parallel routine TIMER1_1.C is included in the files. It is precisely the same as TIMER3_1.C except that Timer 1 is used in place of Timer 3. This was developed out of frustration in trying to find why Timer 3 was not incrementing as noted above and I decided not to throw it away.]

In routine TIMER3_1.C Timer 3 is configured to generate an interrupt each second by preloading the timer with the value 0x8000 and on interrupt loading TMR1H with the value 0x80. Note that this is one application where the 16-bit parallel load is not desired.

After each second, the elapsed time is updated and displayed on the terminal and a speaker on PORTB3 is beeped for nominally 200 ms. In beeping the speaker, the Timer 2 prescale is set to 1:4 such that Timer 2 increments each 4 us. The period register (PR2) is set to 250 such that Timer 2's input to the postscaler is every 1.0 ms. The postscaler is configured for 1:1 and thus a Timer 2 interrupt occurs each 1.0 ms. The duration of the tone is controlled with the standard delay_ms() function. In this application the 200 ms is none too accurate as a good deal of time is spent processing the Timer 2 interrupts.

```
// Program TIMER3_1.C
//
// Illustrates the use of Timer 3 with the external 32.768 kHz crystal T1OSC0
// and T1OSC1.
//
// Each second, briefly blips the speaker and displays the elapsed time in
// seconds and in hour:minute:sec format on the terminal
//
// copyright, Peter H. Anderson, Baltimore, MD, Dec, '01

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

#define FALSE 0
#define TRUE !0

#define MAKE_LONG(h, l) (((long) h) << 8) | (l)

#define EXT_32KHZ

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

void blip_tone(void);
```

```

void increment_time(struct TM *t);

byte timer3_int_occ;    // note that this is global

main()
{
    byte duty;
    long elapsed_t;
    struct TM t;

    pcfg3 = 0; pcfg2 = 1; pcfg1 = 0; pcfg0 = 0;
    // config A/D for 3/0 - not really necessary in this application

    ser_init();

    latb3 = 0;    // make speaker an output 0
    trisb3 = 0;

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

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

    // Timer 2 prescale set to 1:4
    t2ckps1 = 0; t2ckps0 = 1;    // thus, the rollover is 4 * 256us

    // Set up timer 3
#ifdef EXT_32KHZ
    tloscen = 0;
#else
    tloscen = 1;    // enable external crystal osc circuitry
#endif
    tmr3cs = 1;    // select this as the source
    not_t3sync = 1;

    t3ckps1 = 0; t3ckps0 = 0;    // prescale of 1

    tmr3if = 0;    // kill any junk interrupt
    TMR3L = 0x00;
    TMR3H = 0x80;

    tmr3ie = 1;
    peie = 1;
    gieh = 1;

    timer3_int_occ = FALSE;

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

    tmr3on = 1;

    while(1)
    {
        if (timer3_int_occ)

```

```

    {
        timer3_int_occ = FALSE;
        ++elapsed_t;
        increment_time(&t);
        printf(ser_char, "%ld\t", elapsed_t);

        ser_dec_byte(t.mi, 2);
        ser_char(':');
        ser_dec_byte(t.se, 2);

        ser_newline();

        blip_tone();
    }
    // else do nothing
}

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

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

    tmr2ie = 0;
    tmr2on = 0;
}

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

#int_timer3
timer3_int_handler(void)
{
    timer3_int_occ = TRUE;
    TMR3H = 0x80;
}

```

```

#int_timer2
timer2_int_handler(void)
{
    latb3 = !latb3;
}

#int_default
default_int_handler(void)
{
}

#include <delay.c>
#include <ser_18c.c>

```

Timer 0.

Most PICs include this 8-bit timer / counter which may be either used as a counter of events appearing on input RA4 or a timer, using the $f_{osc} / 4$ clock. A programmable prescaler may either be assigned to the watch dog timer or to Timer 0.

With the PIC18CXX2, the watchdog prescale is incorporated in the configurations bits and thus the prescaler can no longer be assigned to the watch dog. However, to maintain backward compatibility, the psa bit is maintained. This will cause a good deal of confusion as on earlier PICs, the psa is set to one to assign it to the watchdog. With the PIC18C, setting psa to 1 removes it as a prescaler for Timer 0, and clearing the psa bit to zero inserts the prescaler. One can see the confusion as in my copy of the PIC18CXX2 data sheet, the text is correct, but the figure is wrong.

With the PIC18CXX2, Timer 0 may also be operated as a 16-bit counter / timer by setting the t08bit to a zero. This inserts an 8-bit postscaler on the tail of TMR0. Thus, one might think of this as TMR0L (same as TMR0) and the postscaler as TMR0H.

However, TMR0H is simply a holding register. On writing to timer, a value is first written to TMR0H which is held in the holding register until a value is written to TMR0L. Writing to TMR0L causes a 16-bit write to the timer. Thus, when writing, always write TMR0H first.

When reading from the timer, reading TMR0L causes the high byte of the timer to be transferred to register TMR0H. Thus, when reading, read TMR0L first.

In the following, an LED on PORTD0 is toggled each second. The $f_{osc} / 4$ clock is used with a prescale of 256 and thus, with an f_{osc} of 4.0 MHz, the timer increments each 256 us. By preloading the timer with the two's complement of 3906, the timer rolls over and causes an interrupt after $256 * 3906$ or 0.999936 seconds.

Note that it takes some time for the program to enter the timer_0 interrupt service routine, and thus Timer 0 will probably have advanced beyond 0x0000. Thus, I read the TMR0H & L, add the two's complement of 3906 and write this back to TMR0H & L. However, recognize that TMR0L may have been on the verge of incrementing when the timer is read and also that when writing to TMR0H & L, the prescaler is cleared.

```

// Program TIMER0_1.C
//
// Illustrates the use of Timer 0 in the 16-bit mode to time for very close to one
// second.
//
// Each second, toggles LED on PORTD0.

```

```

//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

#define FALSE 0
#define TRUE !0

#define MAKE_LONG(h, l) (((long) h) << 8) | (l)

main()
{
    ser_init();    // for possible debuuging

    pspmode = 0;
    latd0 = 0;    // make LED an output logic zero
    trisd0 = 0;

    // Set up timer0
    t0cs = 0;    // fosc/4 is source
    psa = 0;    // prescaler assigned to TMR0

    t0ps2 = 1; t0ps1 = 1; t0ps0 = 1; // prescale of 1:256
    t08bit = 0;    // configure for 16-bit mode

    TMR0H = (byte) ((~3906 + 1) >> 8);
    TMR0L = (byte) (~3906 + 1);

    tmr0on = 1;

    t0if = 0;    // kill any pending interrupt
    t0ie = 1;
    gieh = 1;

    while(1)
    {
    }
}

#int_timer0
timer0_int_handler(void)
{
    byte h, l;
    unsigned long t_new;

    latd0 = !latd0;

    l = TMR0L;    // important to read LSByte first
    h = TMR0H;

    t_new = MAKE_LONG(h, l) + (~3906 + 1);

```

```

    TMR0H = (byte)(t_new >> 8);    // write high byte first
    TMR0L = (byte)(t_new);
}

#int_default
default_int_handler(void)
{
    #asm
        NOP                // for debugging
        NOP
    #endasm
}

#include <delay.c>
#include <ser_18c.c>

```

Program COUNTER.C.

This program counts the number of events appearing on input T13CKI (term 15) over a period of one second. I used the output of the PIC12C509 32 kHz generator noted above.

Timer 1 is used to count the number of events (a maximum of 65525 events) and Timer 3 is used to perform the one second timing window.

Timer 3 is assigned to the CCP1 module which is configured to trigger a special event when TMR3H & L matches CCPR1H & L. When the special event occurs, Timer 3 is reset to zero. Thus, if CCPR1H & L is set to 10000, the special event occurs every 10.00 ms and Timer 3 resets to zero. Note that this happens at the time the match occurs, not in the CCP1 interrupt service routine. This feature, which may be used with both Timers 1 and 3 is extremely powerful and may make the PIC18CXX2 a viable choice, even in applications requiring only a few hundred words of program memory.

In the CCP1 interrupt service routine, global variable t_10ms is decremented.

In the main routine, t_10ms is initialized to 100 so as to perform timing for 1000 ms. When t_10ms is decremented to zero the count value in TMR1H & L is fetched and displayed.

Note that there is a bit of a deficiency here in that the count value is read perhaps 100 us after the final timeout of Timer 3. This could be corrected somewhat by testing t_10ms for zero and fetching the count in the interrupt service routine.

```

// Program COUNTER.C
//
// Counts the number of events on T13CKI (term 15) over 1.0 seconds and
// displays on the terminal.
//
// Use TIMER3 to time for 10 ms * 100
// Use TIMER1 to count events
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02

#case
#device PIC18C452

```

```

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

#define MAKE_LONG(h, l) (((long) h) << 8) | (l)

unsigned long period_time;

byte num_10ms;

void main(void)
{
    long count;

    ser_init();
    printf(ser_char, "\r\n.....\r\n");

    // set up timer3
    t3rdl6 = 0;
    t3ckps1 = 0; t3ckps0 = 0; // 1:1 prescale
    tmr3cs = 0; // internal clock - 1 usec per tick

    TMR3H = 0x00; TMR3L = 0x00;

    // set up CCP2
    CCPR2H = (byte) (10000 >> 8);
    CCPR1L = (byte) 10000;

    t3ccp2 = 0; t3ccp1 = 1; // assign timer3 to CCP2, timer1 to CCP1

    ccp2m3 = 1; ccp2m2 = 0; ccp2m1 = 1; ccp2m0 = 1;
    // special event - resets Timer 3

    // set up timer 1 as counter
    t1rdl6 = 0;
    t1ckps1 = 0; t1ckps0 = 0; // 1:1 prescale
    tloscen = 0;
    tmr1cs = 1;
    t1sync = 1; // do not sync

    TMR1H = 0x00; // number of events
    TMR1L = 0x00;

    num_10ms = 100;

    // turn on timers
    tmr3on = 1;
    tmr1on = 1;

    // enable interrupts
    peie = 1;
    ccp2ie = 1;
    gieh = 1;

    while(num_10ms) /* loop */
    {

```

```

while(gieh)
{
    gieh = 0;
}

peie = 0;
ccp2ie = 0;
ccp2if = 0;

count = MAKE_LONG(TMR1H, TMR1L);
printf(ser_char, "%lu", count);

while(1)          // loop to keep emulator from stalling
{
}
}

#int_ccp2
compare2_int_handler(void)
{
    --num_10ms;
}

#int_default
default_handler(void)
{
}

#include <delay.c>
#include <ser_18c.c>

```

Program TONES_1.C.

This program generates an alternating 500 and 440 Hz tones on a speaker on output PORTB3.

It uses Timer 3 in conjunction with CCP2 to drive PORTB3 either high or low when there is a match between Timer 3 and CCPR2H & L. Note that the output PORTB3 makes the transition when the match occurs, not in the interrupt service routine.

In the interrupt service routine, bit ccp2m0 is inverted such that when the next match occurs, the output will be driven to the opposite state. In addition, the value of CCPR2H & L is read, advanced by the number of microseconds until the next match and the result is written back to the CCPR2H & L registers. Note that Timer 3 simply free runs. Only the value of the CCPR2 registers is changed.

The 1000 ms duration of the tone is set by the delay_ms() function. Note that this will be someone longer as a considerable amount of time is spent processing the CCP2 interrupt.

Another improvement was made in the design of the PIC18CXX2 in that the pin associated with the CCP2 module may either be configured as RC1/T1OSC1/CCP2 (term 16) or RB3/CCP2 (term 36). This assignment is programmed into a configuration bit when the device is programmed.

The advantage over the PIC16F877 is that CCP2 shared one of the terminals associated with the external T1 crystal. With the PIC18CXX2, the CCP2 pin may be moved to PORTB3, permitting the use of the external crystal without sacrificing a CCP2 terminal. This observation is not applicable to the following routine. I simply assigned CCP2 to PORTB3 to verify it worked.

```
// Program TONES_1.C
//
// Generates 500 and 440 Hz tone on portb3. Illustrates the use of TMR3 in
// conjunction with CCP2.
//
// PIC18C452
//
// PORTB3 (term 17) -----+----- SPKR
//
// Set Configuration for CCP2 assigned to RB3. (CCP2MX = 0)
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

#define MAKE_LONG(h, l) (((long) h) << 8) | (l)

unsigned long period_time;

void main(void)
{
    ser_init();          // for debugging

    // set up timer3
    t3rdl6 = 0;
    t3ckps1 = 0; t3ckps0 = 0;    // 1:1 prescale
    tmr3cs = 0;    // internal clock - 1 usec per tick
    tmr3on = 1;

    // set up CCP2
    latb3 = 0;
    trisb3 = 0;

    t3ccp2 = 0; t3ccp1 = 1; // assign timer3 to CCP2, timer1 to CCP1

    ccp2m3 = 1; ccp2m2 = 0; ccp2m1 = 0; ccp2m0 = 0;
    // make an output one on match

    period_time = 1136;

    // enable interrupts
    peie = 1;
    ccp2ie = 1;
    gieh = 1;

    while(1)
```

```

    {
        printf(ser_char, ".");
        lattb5 = 1;
        trisb5 = 0;
        period_time = 1136;    // 440 Hz
        delay_ms(1000);

        latb5 = 0;
        period_time = 1000;    // 500 Hz
        delay_ms(1000);
    }
}

#int_ccp2
compare2_int_handler(void)
{
    unsigned long x;
    ccp2m0 = !ccp2m0;        // toggle for the opposite
    x = MAKE_LONG(CCPR2H, CCPR2L);
    x = x + period_time;
    CCPR2H = x >> 8;
    CCPR2L = x & 0xff;
}

#int_default
default_handler(void)
{
}

#include <delay.c>
#include <ser_18c.c>

```

Program MUSIC_1.C.

This program builds on NOTES_1.C to play a sequence of notes. Timer 3 in conjunction with CCP2 in the “trigger special event” mode is used to control the frequency. Timer 1 in conjunction with CCP1 is used to control the duration of each tone.

This program uses the enum construct.

```

enum note {E3, F3, G3, A3, B3, C3, D3, E4, F4};
enum dur {Whole, Half, Quarter, Eighth, Sixteenth};

```

This is nothing more than defining E3 to be the number 0, F3 to be the number 1, etc. Similarly for the duration Whole is the number 0, Half is the number 1, etc.

Consider;

```

const long half_periods[9] = {3034, 2864, 2551, 2273,
                             2025, 1911, 1703, 1517, 1432};
const long durations[5] = {80, 40, 20, 10, 5};

```

Note that half_periods[E1] is really half_periods[0], except that E1 is a bit more understandable. Similarly durations[Half] is simply durations[1].

Now, consider the musical sequence;

```
const byte notes_f[5] = {E3, A3, C3, D3, E4};
const long notes_d[5] = {Whole, Half, Whole, Eighth, Whole};
```

A whole note E3, followed by a half note A3, etc.

For the nth note, the half period and the duration are;

```
half_periods[notes_f[n]]
durations[notes_d[n]]
```

Thus, in this routine, CCP2H & L is loaded with half_periods[notes[n]]. The CCP2 module is configured in the “trigger special event on match” mode which causes Timer 3 to reset on match. The state of PORTB3 is toggled in the interrupt service routine.

The duration of the tone is controlled by Timer 1 in conjunction with CCP1 in the “trigger special event” mode which causes the timer to reset on match. The CCP1H & L registers are set to 12500 and thus a CCP1 interrupt occurs every 12.5 ms. In the CCP1 interrupt service routine, the global variable num_12_5ms is decremented. This is similar to the technique used in the COUNT.C routine, except Timer 1 is used.

Thus in the following snippet, interrupt CCP2 (associated with Timer 3) is toggling the state of the output and interrupt CCP1 (associated with Timer 1) is decrementing num_12_5ms.

```
while(num_12_5ms)
{
    /* loop until zero */
}
```

When, num_12_5ms decrements to zero, the program exits the while() loop, the interrupts are disabled, the timers are turned off and the program moves to the next note.

[In reviewing this program I regret using the 12.5 ms timing. It works well for whole, half, quarter, eighth and sixteenth notes, but then it fails. A fundamental timing interval of 6.25 ms might have been a wiser choice.]

```
// Program MUSIC_1.C
//
// Plays a musical sequence on speaker on PORTB3.
//
// Uses CCP2 and Timer 3 in the Trigger Special Event mode to toggle PORTB
// at the frequency of the musical note and uses CCP1 and Timer 1 in Trigger
// Special Event to provide the timing for the duration of the note.
//
// Also illustrates the use of enum.
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02
```

```
#ifdef XXX // calculation of half periods for various musical notes
```

Musical Notes		
	freq	Period (us) 1/2 Period (us)
E3	164.80	6067.961165 3033.980583
F3	174.60	5727.376861 2863.688431

```

G3      196.00      5102.040816 2551.020408
A3      220.00      4545.454545 2272.727273
B3      246.94      4049.566696 2024.783348
C3      261.63      3822.191645 1911.095822
D3      293.66      3405.298645 1702.649322
E4      329.63      3033.704457 1516.852228
F4      349.23      2863.44243  1431.721215

#endif

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

byte num_12_5ms;

void main(void)
{
    enum note {E3, F3, G3, A3, B3, C3, D3, E4, F4};
    enum dur {Whole, Half, Quarter, Eighth, Sixteenth};

    const long half_periods[9] = {3034, 2864, 2551, 2273,
                                   2025, 1911, 1703, 1517, 1432};
    const long durations[5] = {80, 40, 20, 10, 5};

    const byte notes_f[5] = {E3, A3, C3, D3, E4};
    const long notes_d[5] = {Whole, Half, Whole, Eighth, Whole};

    byte n;

    ser_init();

    // set up timer3
    t3rdl6 = 0;
    t3ckps1 = 0;    t3ckps0 = 0;        // 1:1 prescale
    tmr3cs = 0;     // internal clock - 1 usec per tick
    TMR3H = 0x00; TMR3L = 0x00;

    // setup CCP2
    t3ccp2 = 0;      t3ccp1 = 1; // assign timer3 to CCP2, timer1 to CCP1
    ccp2m3 = 1;  ccp2m2 = 0; ccp2m1 = 1;  ccp2m0 = 1;
                                   // special event - resets Timer 3

    // set up timer 1
    t1rdl6 = 0;
    t1ckps1 = 0;    t1ckps0 = 0;        // 1:1 prescale
    t1oscen = 0;
    tmr1cs = 0;

    TMR1H = 0x00; TMR1L = 0x00;

    // set up CCP1
    ccplm3 = 1;  ccplm2 = 0; ccplm1 = 1;  ccplm0 = 1;
                                   // special event - resets Timer 1

```

```

for (n=0; n<5; n++) // play each note
{
    // disable interrupts
    while(gieh)
    {
        gieh = 0;
    }
    ccplie = 0;
    ccp2ie = 0;
    tmr3on = 0;
    tmrlon = 0;

    CCPR2H = (byte) (half_periods[notes_f[n]] >> 8);
    CCPR2L = (byte) (half_periods[notes_f[n]]);

    CCPR1H = (byte) (12500 >> 8); // 12.5 ms
    CCPR1L = (byte) (12500);

    num_12_5ms = durations[notes_d[n]];

    tmr3on = 1;
    tmrlon = 1;

    // enable interrupts
    ccplif = 0;
    ccp2if = 0;

    peie = 1;
    ccplie = 1;
    ccp2ie = 1;
    gieh = 1;

    while(num_12_5ms)
    {
        /* loop until zero */
    }
}

// disable interrupts
while(gieh)
{
    gieh = 0;
}
ccplie = 0;
ccp2ie = 0;
tmr3on = 0;
tmrlon = 0;

while(1)
{
    /* loop to keep emulator from stalling */
}
}

#int_ccpl
compare1_int_handler(void)

```

```

{
    --num_12_5ms;
}

#int_ccp2
compare2_int_handler(void)
{
    trisb3 = 0;                // toggle portb3
    latb3 = latb3;
}

#int_default
default_handler(void)
{
}

#include <delay.c>
#include <ser_18c.c>

```

Stepper Motor Control.

A unipolar stepper consists of four windings which I identify as PHI0, PHI1, PHI2 and PHI3. Turning the motor is a matter of energizing one winding at a time.

```
const patts[4] = {0x01, 0x02, 0x04, 0x08};
```

By traversing the array and outputting each state, the motor is advanced one step. The direction of the motor is a matter of whether the array is traversed up (0x01, 0x02, 0x04, 0x08, 0x01, etc) or down (0x01, 0x08, 0x04, etc). The speed of the motor is determined by the delay in outputting each pattern.

As an aside, a variant of this full step which provides more torque is to energize two windings at a time.

```
const patts[4] = {0x03, 0x06, 0x0c, 0x09};
```

Greater resolution is achieved by “half stepping”. That is, energizing one winding, then both this winding and the adjacent, followed by the adjacent only, etc.

```
const patts[8] = {0x01, 0x03, 0x02, 0x06, 0x04, 0x0c, 0x08, 0x09};
```

In the following discussions I use the half step technique.

Program STEP_1.C.

In this routine, a stepper is turned in one direction or the other depending on the state of input PORTB7.

Timer 1 is used in conjunction with CCP1 in the compare – trigger special event to perform the 3333 us timing between outputting each new half step.

Note that the constant array of patts and the index within that array is global.

The index with the pattern array is either incremented or decremented depending on the state of PORTB7. It is wrapped, if it falls outside the range of 0 – 7.

When Timer 1 matches CCPR1H & L, the timer resets and a CCP1 interrupt occurs. The pattern is output on the lower nibble of PORTB.

```
// Program STEP_1.C
//
// Turns stepper at 300 pulses per second (3333 usec per step) in direction
// indicated by switch on PORTB7. Uses Timer 1 in conjunction with CCP1 in
// Compare - Trigger Special Event mode which resets the timer to 0 on a match of
// TMR1H & L and CCP1PR1H & L
//
//
//          PIC18C452                      ULN2803                      Stepper
//
// PORTB3 (term 36) ----- 1          18 ----- PHI3
// PORTB2 (term 35) ----- 2          17 ----- PHI2
// PORTB1 (term 34) ----- 3          16 ----- PHI1
// PORTB0 (term 33) ----- 4          15 ----- PHI0
//
//
//                      GRD - Term 9
//                      Vdiode - Term 10
//
//                      PIC18C452
//
// GRD ----- \----- PORTB7 (term 40)
//
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02
```

```
#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>      // for possible debugging

#define FALSE 0
#define TRUE !0

const byte pattns[8] = {0x01, 0x03, 0x02, 0x06, 0x04, 0x0c, 0x08, 0x09};
byte ccpl_int_occ, index;

void main(void)
{
    ccpl_int_occ = FALSE;
    index = 0;

    not_rbpu = 0; // enable weak pullups
    LATB = 0x0f;  // lower nibble at logic one
    TRISB = 0xf0; // lower nibble outputs

    // set up timer1
    t1rd16 = 1;
    t1ckps1 = 0;    t3ckps0 = 0;          // 1:1 prescale
    tmr1cs = 0;     // internal clock - 1 usec per tick
```

```

TMR1H = 0x00; TMR1L = 0x00;

// setup CCP1
t3ccp2 = 0;          t3ccp1 = 1; // assign timer3 to CCP2, timer1 to CCP1
ccp1m3 = 1; ccp1m2 = 0; ccp1m1 = 1; ccp1m0 = 1;
                                // special event - resets Timer 1
CCPR1H = (byte) (3333 >> 8);
CCPR1L = (byte) 3333;

// turn on timer
tmr1on = 1;

// enable ints
ccplif = 0;
ccplie = 1;
peie = 1;
gieh = 1;

while(1)
{
    if (ccp1_int_occ)
    {
        ccp1_int_occ = FALSE;
        if (portb7)
        {
            ++index;    // note that this is global
            if (index > 7)
            {
                index = 0;
            }
        }
        else
        {
            --index;
            if (index == 0xff)
            {
                index = 7;
            }
        }
    }
} // of while(1)
}

#int_ccp1
ccp1_int_handler(void)
{
    LATB = (LATB & 0xf0) | patts[index];
    ccp1_int_occ = TRUE;
}

#int_default
default_int_handler(void)
{
}

#include <delay.c>

```

```
#include <ser_18c.c>
```

Program STEP_2.C.

This program builds on the previous in providing the ability to accelerate (or decelerate) over a fixed number of steps.

For example;

```
accelerate(CW, 10000, 1000, 300);
```

Accelerate from 10000 us between half steps (100 half steps per sec) to 1000 us between half steps (1000 half steps per sec) over the course of 300 steps. Note that this involves calculating the delay as;

```
delay = start_delay - (start_delay - stop_delay)* n / num_steps;
```

However, note that $(\text{start_delay} - \text{stop_delay}) * n$ will overflow a long; e.g., $(10000 - 1000) * 299$, and I finally settled for;

```
delay = start_delay - (start_delay - stop_delay)/128* n / num_steps * 128;
```

This really requires more study. I left this alone as I didn't desire to get bogged down in a matter that really doesn't have much to do with the PIC18C processor.

The value of delay is then copied to the CCP1H & L.

The value of accelerating, running and decelerating over a fixed number of half steps is that usually one wishes to advance a stepper a certain distance, rather than run the motor for a certain period time. Thus, the sequence;

```
accelerate(CW, 10000, 1000, 300);  
run(CW, 1000, 5000);  
decelerate(CW, 1000, 10000, 300);
```

advances the motor in a clockwise direction 5600 half steps.

[I also used floating point numbers to perform the delay calculation, but could not convince myself that the calculation was completed by the next timeout.]

[Note that I really am not an expert on accelerating steppers. I thought it to be an interesting problem.]

```
// Program STEP_2.C  
//  
// Illustrates accelerating, running and decelerating a stepper for a defined  
// number of steps.  
//  
// From rest the stepper is linearly accelerated from 10000 us per half step to  
// 1000 us per half step over 300 half steps of travel. The motor is then run at  
// 1000 us per half step for 5000 half steps and then decelerated over 300 half  
// steps. This is repeated in the other direction.  
//  
//  
// PIC18C452                                ULN2803                                Stepper
```

```

//
//  PORTB3 (term) ----- 1      18 ----- PHI3
//  PORTB2 (term )----- 2      17 ----- PHI2
//  PORTB1 (term )----- 3      16 ----- PHI1
//  PORTB0 (term )----- 4      15 ----- PHI0
//
//
//                      GRD - Term 9
//                      Vdiode - Term 10
//
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

#define FALSE 0
#define TRUE !0

#define CW 0
#define CCW 1

//#define FLOAT

void run(byte dir, unsigned long run_delay, unsigned long num_steps);
void accelerate(byte dir, unsigned long start_delay, unsigned long stop_delay,
               unsigned long num_steps);
void decelerate(byte dir, unsigned long start_delay, unsigned long stop_delay,
               unsigned long num_steps);

const byte patts[8] = {0x01, 0x03, 0x02, 0x06, 0x04, 0x0c, 0x08, 0x09};
byte ccpl_int_occ, index;

void main(void)
{
    ccpl_int_occ = FALSE;
    index = 0;

    LATB = 0x0f;
    TRISB = 0xf0;

    while(1) // continually
    {
        accelerate(CW, 10000, 1000, 300);
        run(CW, 1000, 5000);
        decelerate(CW, 1000, 10000, 300);

        accelerate(CCW, 10000, 1000, 300);
        run(CCW, 1000, 5000);
        decelerate(CCW, 1000, 10000, 300);

        delay_ms(1000);
    }
}

```

```

void run(byte dir, unsigned long run_delay, unsigned long num_steps)
{
    unsigned long n;

    // set up timer1
    t1rd16 = 0;
    t1ckps1 = 0;    t3ckps0 = 0;        // 1:1 prescale
    tmrlcs = 0;     // internal clock - 1 usec per tick
    TMR1H = 0x00; TMR1L = 0x00;

    // setup CCP1
    t3ccp2 = 0;      t3ccp1 = 1; // assign timer3 to CCP2, timer1 to CCP1
    ccplm3 = 1; ccplm2 = 0; ccplm1 = 1; ccplm0 = 1;
                // special event - resets Timer 1

    tmrlon = 1;
    ccplif = 0;
    ccplie = 1;
    peie = 1;
    gieh = 1;

    CCPR1H = (byte) (run_delay >> 8);
    CCPR1L = (byte) run_delay;

    ccpl_int_occ = FALSE;

    for (n=0; n<num_steps; n++)
    {
        while(!ccpl_int_occ)
        {
            /* loop */
        }

        ccpl_int_occ = FALSE;

        if (dir == CW)
        {
            ++index; // note that this is global
            if (index > 7)
            {
                index = 0;
            }
        }
        else
        {
            --index;
            if (index == 0xff)
            {
                index = 7;
            }
        }
    }

    while(gieh)
    {

```

```

    gieh = 0;
}

ccplif = 0;    // kill any interrupt
tmrlon = 0;
ccplie = 0;
}

void accelerate(byte dir, unsigned long start_delay, unsigned long stop_delay,
                unsigned long num_steps)
{
    unsigned long d, n, delay;
    float x, y;

    // set up timer1
    t1rd16 = 0;
    t1ckps1 = 0;    t1ckps0 = 0;    // 1:1 prescale
    t1mlcs = 0;    // internal clock - 1 usec per tick
    TMR1H = 0x00; TMR1L = 0x00;

    // setup CCP1
    t3ccp2 = 0;    t3ccp1 = 1; // assign timer3 to CCP2, timer1 to CCP1
    ccplm3 = 1; ccplm2 = 0; ccplm1 = 1; ccplm0 = 1;
    // special event - resets Timer 1

    tmrlon = 1;
    ccplif = 0;
    ccplie = 1;
    peie = 1;
    gieh = 1;

    ccpl_int_occ = FALSE;

    for (n=0; n<num_steps; n++)
    {
#ifdef FLOAT
        x = (float) (start_delay - stop_delay);
        y = (float) (n) / (float) (num_steps);
        x = x * y;

        d = (unsigned long) x;

        delay = start_delay - d;
#else
        delay = start_delay - (start_delay - stop_delay)/128 * n / num_steps * 128;
#endif
        CCPR1H = (byte) (delay >> 8);
        CCPR1L = (byte) delay;

        while(!ccpl_int_occ)
        {
            /* loop */
        }

        ccpl_int_occ = FALSE;

        if (dir == CW)

```

```

    {
        ++index; // note that this is global
        if (index > 7)
        {
            index = 0;
        }
    }

    else
    {
        --index;
        if (index == 0xff)
        {
            index = 7;
        }
    }
}

while(gieh)
{
    gieh = 0;
}

ccplif = 0;    // kill any interrupt
tmrlon = 0;
ccplie = 0;
}

void decelerate(byte dir, unsigned long start_delay, unsigned long stop_delay,
                unsigned long num_steps)
{
    unsigned long d, n, delay;
    float x, y;

    // set up timer1
    tlrdl6 = 0;
    tlckps1 = 0;    tlckps0 = 0;    // 1:1 prescale
    tmrlcs = 0;    // internal clock - 1 usec per tick
    TMR1H = 0x00; TMR1L = 0x00;

    // setup CCP1
    t3ccp2 = 0;    t3ccp1 = 1; // assign timer3 to CCP2, timer1 to CCP1
    ccplm3 = 1;    ccplm2 = 0; ccplm1 = 1;    ccplm0 = 1;
                                // special event - resets Timer 3

    tmrlon = 1;
    ccplif = 0;
    ccplie = 1;
    peie = 1;
    gieh = 1;

    ccpl_int_occ = FALSE;

    for (n=0; n<num_steps; n++)
    {
#ifdef FLOAT

```

```

    x = (float) (stop_delay - start_delay);
    y = (float) (n) / (float) (num_steps);
    x = x * y;

    d = (unsigned long) x;

    delay = start_delay + d;
#else
    delay = start_delay + (stop_delay - start_delay)/128 * n / num_steps * 128;
#endif

    CCPR1H = (byte) (delay >> 8);
    CCPR1L = (byte) delay;

    while(!ccpl_int_occ)
    {
        /* loop */
    }

    ccpl_int_occ = FALSE;

    if (dir == CW)
    {
        ++index; // note that this is global
        if (index > 7)
        {
            index = 0;
        }
    }

    else
    {
        --index;
        if (index == 0xff)
        {
            index = 7;
        }
    }
}

while(gieh)
{
    gieh = 0;
}

ccplif = 0;    // kill any interrupt
tmrlon = 0;
ccplie = 0;
}

#int_ccpl
ccpl_int_handler(void)
{
    LATB = (LATB & 0xf0) | patts[index];
    ccpl_int_occ = TRUE;
}

```

```
#int_default
default_int_handler(void)
{
}
```

```
#include <delay.c>
#include <ser_18c.c>
```

Program STEP_3.C.

In this program a potentiometer with the wiper input to A/D channel AN0 (term 2) is used to control the stepper. When at mid scale, the stepper is at its slowest speed. If the wiper is slightly above or below mid scale the motor turns slowly in one direction or the other. As the wiper is moved further and further from mid scale the motor turns faster and faster. [If I had it to do again, I would have stopped the motor altogether in the region of mid scale].

As in the previous routines, Timer 1 operates in conjunction with CCP1 configured in the compare – trigger special event mode. When Timer 1 matches CCPR1H & L, the timer resets. This is used to control the delay between outputting the half steps.

In this routine, Timer 3 operates in conjunction with CCP2 which is also configured in the same mode. However, with CCP2, the “special event” also includes an A/D conversion if the A/D converter is configured. Note that CCP2 is configured to timeout every 10,000 us. Thus, every 10 ms, an A/D conversion is performed and the setting of the potentiometer is mapped into new CCP1H & L values to control the speed of the motor.

The speed of the motor varies from about 3000 us per half step (330 half steps per sec) to 1000 us (1000 half steps per sec) at the limits.

```
// Program STEP_3.C
//
// Uses a potentiometer on AN0 (term 2) to control the direction and speed of
// a stepper on the lower four bits of PORTB.
//
// Uses Timer 3 in conjunction with the CCP2 module configured to trigger a
// special event (rest of timer plus perform an A/D conversion). This is set to
// 10 ms such that an A/D conversion is performed every 10 ms.
//
// Uses Timer 1 with CCP1 to trigger a special event (reset timer). This
// is used to control the delay between outputting half steps.
//
//      PIC18C452                ULN2803                Stepper
//
// PORTB3 (term 36) ----- 1      18 ----- PHI3
// PORTB2 (term 35) ----- 2      17 ----- PHI2
// PORTB1 (term 34) ----- 3      16 ----- PHI1
// PORTB0 (term 33) ----- 4      15 ----- PHI0
//
//                                GRD - Term 9
//                                Vdiode - Term 10
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02
```

```
#case
```

```

#define PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

#define FALSE 0
#define TRUE !0

#define CW 0
#define CCW 1

#define MAKE_LONG(h, l) (((long) h) << 8) | (l)

const byte patts[8] = {0x01, 0x03, 0x02, 0x06, 0x04, 0x0c, 0x08, 0x09};
byte ccp1_int_occ, ccp2_int_occ, index;

void main(void)
{
    byte dir;
    unsigned long x, step_delay_time, ad_sample_time;

    ccp1_int_occ = FALSE;
    ccp2_int_occ = FALSE;

    index = 0;
    step_delay_time = 20000; // 20 ms
    dir = CW;
    ad_sample_time = 10000; // every 10 ms

    LATB = 0x0f;
    TRISB = 0xf0;

    // configure A/D
    pcf3 = 0; pcf2 = 1; pcf1 = 0; pcf0 = 0; // config A/D for 3/0

    adfm = 1; // right justified
    adcs2 = 0; adcs1 = 1; adcs0 = 1; // internal RC

    adon=1; // turn on the A/D
    chs2=0; chs1=0; chs0=0;

    // config timer 1
    t1rd16 = 1;
    t1ckps1 = 0; t3ckps0 = 0; // 1:1 prescale
    t1rlcs = 0; // internal clock - 1 usec per tick
    TMR1H = 0x00; TMR1L = 0x00;

    // config timer 3
    t3rd16 = 1;
    t3ckps1 = 0; t3ckps0 = 0; // 1:1 prescale
    t3r3cs = 0; // internal clock - 1 usec per tick
    TMR3H = 0x00; TMR3L = 0x00;

    // assign timers
    t3ccp2 = 0; t3ccp1 = 1; // assign timer3 to CCP2, timer1 to CCP1

```

```

// configure CCP1
ccp1m3 = 1; ccp1m2 = 0; ccp1m1 = 1; ccp1m0 = 1;
// special event - resets Timer 1
CCPR1H = (byte) (step_delay_time >> 8);
CCPR1L = (byte) (step_delay_time);

// config ccp2
ccp2m3 = 1; ccp2m2 = 0; ccp2m1 = 1; ccp2m0 = 1;
// special event - resets Timer 3 and initiates A/D
CCPR2H = (byte) (ad_sample_time >> 8);
CCPR2L = (byte) (ad_sample_time);

// turn on timers and config interrupts
tmr1on = 1;
tmr3on = 1;

ccplif = 0;
ccp2if = 0;
ccplie = 1;
ccp2ie = 1;
peie = 1;
gieh = 1;

while(1)
{
    if (ccp1_int_occ)
    {
#asm
        CLRWDT
#endasm
        ccp1_int_occ = FALSE;
        CCPR1H = (byte) (step_delay_time >> 8);
        CCPR1L = (byte) step_delay_time;

        // either increment or decrement
        if (dir == CW)
        {
            ++index; // note that this is global
            if (index > 7)
            {
                index = 0;
            }
        }
        else
        {
            --index;
            if (index == 0xff)
            {
                index = 7;
            }
        }
    }

    if (ccp2_int_occ)

```

```

    {
#asm
        CLRWDI
#endasm
        ccp2_int_occ = FALSE;
        x = MAKE_LONG(ADRESH, ADRESL);
        if (x >= 0x0200)
        {
            x = x - 0x0200; // in range of 0 to 0x01ff
            step_delay_time = 4 * (0x0200 - x) + 1000;
            dir = CW;
        }

        else
        {
            step_delay_time = 4 * x + 1000;
            dir = CCW;
        }
    }
} // end of while 1
}

#int_ccp1
ccp1_int_handler(void)
{
    LATB = (LATB & 0xf0) | patts[index];
    ccp1_int_occ = TRUE;
}

#int_ccp2
ccp2_int_handler(void)
{
    ccp2_int_occ = TRUE;
}

#int_default
default_int_handler(void)
{
}

#include <delay.c>
#include <ser_18c.c>

```

Program STEP_4.C.

This program is functionally similar to STEP_2.C. However, rather than accelerating, decelerating or running over a fixed number of half steps, this programs accelerates or decelerates over a specified period of time. That is;

```

accelerate(CW, 10000, 1500, 25); // accelerate over 25 ms
run(CW, 1500, 5000); // run for 5000 ms
decelerate(CW, 1500, 10000, 25); // decelerate over 25 ms

```

Timer 3 is used in conjunction with CCP2 in the special event trigger mode such that the timer resets on match which is set for 1000 us. Thus, a CCP2 interrupt occurs every ms and this is used to perform the duration timing.

Timer 1 is used in conjunction with CCP1 in the compare mode to perform the timing between steps.

There is a subtle difference here from the STEP_2.C program. In the earlier program, Timer 1 was configured to reset when there is a match between TMR1H & L and CCPR1H & L. In either accelerate() and decelerate() the fact that an interrupt occurred is noted and a new value of CCPR1H & L is calculated. There is no possibility that Timer 1 would have advanced so far that the new value of CCPR1H & L is actually less than the current value of Timer 1. This would have the undesired effect of Timer 1 having to roll over and counting up to the new match. As I say, this isn't a problem as a CCP1 match occurs which resets Timer 1 and causes an interrupt which set the ccp1_int_occ flag which causes the program to calculate a new value of CCPR1H & L. Thus a CCP1 match directly causes a new value for the next match.

However, with STEP_4.C, we have a much different situation, where Timer 3 causes the update of the CCPR1H & L values, and Timer 3 is in no way related to the value of Timer 1. Thus, we could get into this kind of situation.

Timer 3 in conjunction with CCP2 causes a match and a CCP2 interrupt occurs. Assume that at that time CCPR1H & L and Timer 1 are as shown;

```
CCPR1H & L at      4000
Timer 1 at         3900      // note that a CCP1 match is about to occur
```

On seeing the CCP2 interrupt has occurred, the accelerate() function calculates the new value of CCPR1H & L as 2000.

This has the undesired effect that Timer 1 will now count up to 65535, roll over and count up to 2000, taking upwards of 65 ms before the CCP1 match occurs. A full 65 ms between outputting half steps.

{Note that this can actually happen in program STEP_3.C as the A/D sampling time is in no way related to Timer 3. Unfortunately, I didn't see this.}

Thus, in the accelerate() and decelerate() functions, Timer 1 in conjunction with CCP1 is not configured for the "trigger special event" on match, but rather to "generate a software interrupt" on match. Timer 1 is not reset, but continues to advance. An analogy. Rather than sending the runner back to "go", let her keep running. The trick now is to set a new finish line further down the track.

Bang. A Timer 3 / CCP2 match occurs. The current value of CCPR1H & L is then read and the new step value is added and written back to CCPR1H & L. As a CCP2 interrupt occurs every 1000 us, and the minimum delay between half steps is 1200 us, the new value of CCPR1H & L will always be ahead of the current state of Timer 1. The analogy. Keep the finish line ahead of the runner.

If you are thoroughly confused, don't feel too bad. Just note that the CCP modules may be configured for a "match – generate software" interrupt and a new match may be set by adding to the current value of CCPR1H & L.

Note that in the run() routine, the value of CCPR1H & L is constant and thus Timer 1 in conjunction with CCP1 is configured in the match – trigger special event mode to time between outputting half steps.

```

// Program STEP_4.C
//
// Accelerates, runs and decelerates a stepping motor over a period of time.
//
// Accelerates from 10000 us between half steps to 1500 us between half steps over
// a time interval of 25 ms.  Runs for 5000 ms with 1500 us between half steps.
// Decelerates to 10000 us over a time of 25 ms.
//
// This is repeated in the opposite direction.
//
//      PIC18C452                      ULN2803                      Stepper
//
//  PORTB3 (term) ----- 1          18 ----- PHI3
//  PORTB2 (term )----- 2          17 ----- PHI2
//  PORTB1 (term )----- 3          16 ----- PHI1
//  PORTB0 (term )----- 4          15 ----- PHI0
//
//
//                      GRD - Term 9
//                      Vdiode - Term 10
//
//
// copyright, Peter H. Anderson, Baltimore, MD, Jan, '02

```

```

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

#define FALSE 0
#define TRUE !0

#define CW 0
#define CCW 1

#define MAKE_LONG(h, l)  (((long) h) << 8) | (l)

void run(byte dir, unsigned long run_delay, unsigned long time_ms);
void accelerate(byte dir, unsigned long start_delay, unsigned long stop_delay,
               unsigned long time_ms);
void decelerate(byte dir, unsigned long start_delay, unsigned long stop_delay,
               unsigned long time_ms);

const byte patts[8] = {0x01, 0x03, 0x02, 0x06, 0x04, 0x0c, 0x08, 0x09};
byte ccp1_int_occ, ccp2_int_occ, index;

void main(void)
{
    ccp1_int_occ = FALSE;
    ccp2_int_occ = FALSE;

    index = 0;

    LATB = 0x0f;
    TRISB = 0xf0;

```

```

while(1)
{
    accelerate(CW, 10000, 1500, 25);          // accelerate over 25 ms
    run(CW, 1500, 5000);                      // run for 5000 ms
    decelerate(CW, 1500, 10000, 25);         // decelerate over 25 ms

    accelerate(CCW, 10000, 1500, 250);
    run(CCW, 1500, 5000);
    decelerate(CCW, 1500, 10000, 25);

    delay_ms(1000);
}
}

void run(byte dir, unsigned long run_delay, unsigned long time_ms)
{
    // set up timer1
    t1rdl6 = 0;
    t1ckps1 = 0;    t1ckps0 = 0;          // 1:1 prescale
    tmr1cs = 0;     // internal clock - 1 usec per tick
    TMR1H = 0x00; TMR1L = 0x00;

    // set up timer3
    t3rdl6 = 0;
    t3ckps1 = 0;    t3ckps0 = 0;          // 1:1 prescale
    tmr3cs = 0;     // internal clock - 1 usec per tick
    TMR3H = 0x00; TMR3L = 0x00;

    // assign timers
    t3ccp2 = 0;      t3ccp1 = 1; // assign timer3 to CCP2, timer1 to CCP1

    // setup CCP1
    ccplm3 = 1; ccplm2 = 0; ccplm1 = 1; ccplm0 = 1;
    // special event - resets Timer 1 - used for step delay
    CCPR1H = (byte) (run_delay >> 8);
    CCPR1L = (byte) (run_delay);

    // setup CCP2
    ccp2m3 = 1; ccp2m2 = 0; ccp2m1 = 1; ccp2m0 = 1;
    // special event - resets Timer 3 - one ms
    CCPR2H = (byte) (1000 >> 8);
    CCPR2L = (byte) 1000;

    // turn on timers
    tmr1on = 1;
    tmr3on = 1;

    ccplif = 0;      // kill any pending interrupts
    ccp2if = 0;

    ccplie = 1;
    ccp2ie = 1;
    peie = 1;
    gieh = 1;

```

```

ccp1_int_occ = FALSE;
ccp2_int_occ = FALSE;

while(time_ms)
{
    if(ccp1_int_occ) // step delay
    {
        ccp1_int_occ = FALSE;
        if (dir ==CW)
        {
            ++index; // note that this is global
            if (index > 7)
            {
                index = 0;
            }
        }
        else
        {
            --index;
            if (index == 0xff)
            {
                index = 7;
            }
        }
    }

    if (ccp2_int_occ) // time delay
    {
        ccp2_int_occ = FALSE;
        --time_ms;
    }
}

while(gieh) // turn off interrupts and timers
{
    gieh = 0;
}

ccp1if = 0; // kill any interrupt
ccp2if = 0;

tmrlon = 0; // turn off timers
tmr3on = 0;

ccplie = 0;
ccp2ie = 0;
}

void accelerate(byte dir, unsigned long start_delay, unsigned long stop_delay,
                unsigned long time_ms)
{
    unsigned long n, d, delay, new_match;
    float x, y;

    n = 0;

```

```

// set up timer1
t1rd16 = 1;
t1ckps1 = 0;   t1ckps0 = 0;           // 1:1 prescale
tmr1cs = 0;    // internal clock - 1 usec per tick
TMR1H = 0x00; TMR1L = 0x00;

// set up timer3
t3rd16 = 1;
t3ckps1 = 0;   t3ckps0 = 0;           // 1:1 prescale
tmr3cs = 0;    // internal clock - 1 usec per tick
TMR3H = 0x00; TMR3L = 0x00;

// assign timers
t3ccp2 = 0;          t3ccp1 = 1; // assign timer3 to CCP2, timer1 to CCP1

// setup CCP1
ccp1m3 = 1;  ccp1m2 = 0; ccp1m1 = 1;  ccp1m0 = 0;
           // software interrupt - used for step delay
CCPR1H = (byte) (start_delay >> 8);
CCPR1L = (byte) (start_delay);

// setup CCP2
ccp2m3 = 1;  ccp2m2 = 0; ccp2m1 = 1;  ccp2m0 = 1;
           // trigger special event - reset timer 1
CCPR2H = (byte) (1000 >> 8);
CCPR2L = (byte) 1000;

// turn on timers
tmr1on = 1;
tmr3on = 1;

ccp1if = 0;    // kill any pending interrupts
ccp2if = 0;

ccp1ie = 1;
ccp2ie = 1;
peie = 1;
gieh = 1;

ccp1_int_occ = FALSE;
ccp2_int_occ = FALSE;

while(n<time_ms)
{
    if(ccp1_int_occ)
    {
        ccp1_int_occ = FALSE;

        if (dir ==CW)
        {
            ++index;    // note that this is global
            if (index > 7)
            {
                index = 0;
            }
        }
    }
}

```

```

        else
        {
            --index;
            if (index == 0xff)
            {
                index = 7;
            }
        }
    }

    if (ccp2_int_occ)
    {
        ccp2_int_occ = FALSE;
        ++n;
#ifdef FLOAT
        // update CCPR1H & L
        x = (float) (start_delay - stop_delay);
        y = (float) (n) / (float) (time_ms);
        x = x * y;

        d = (unsigned long) x;
#else
        d = (start_delay - stop_delay)/128 * n / time_ms;
#endif
        delay = start_delay - d;
        new_match = MAKE_LONG(CCPR1H, CCPR1L) + delay;
        CCPR1H = (byte) (new_match >> 8);
        CCPR1L = (byte) new_match;
    }
}

while(gieh)
{
    gieh = 0;
}

ccplif = 0;    // kill any interrupt
ccp2if = 0;

tmr1on = 0;    // turn off timers
tmr3on = 0;

ccplie = 0;
ccp2ie = 0;
}

void decelerate(byte dir, unsigned long start_delay, unsigned long stop_delay,
                unsigned long time_ms)
{
    unsigned long n, d, delay, new_match;
    float x, y;

    n = 0;

    // set up timer1

```

```

t1rd16 = 1;
t1ckps1 = 0;   t1ckps0 = 0;           // 1:1 prescale
tmr1cs = 0;    // internal clock - 1 usec per tick
TMR1H = 0x00; TMR1L = 0x00;

// set up timer3
t3rd16 = 1;
t3ckps1 = 0;   t3ckps0 = 0;           // 1:1 prescale
tmr3cs = 0;    // internal clock - 1 usec per tick
TMR3H = 0x00; TMR3L = 0x00;

// assign timers
t3ccp2 = 0;          t3ccp1 = 1; // assign timer3 to CCP2, timer1 to CCP1

// setup CCP1
ccp1m3 = 1;  ccp1m2 = 0; ccp1m1 = 1;  ccp1m0 = 0;
           // software interrupt - used for step delay
CCPR1H = (byte) (start_delay >> 8);
CCPR1L = (byte) (start_delay);

// setup CCP2
ccp2m3 = 1;  ccp2m2 = 0; ccp2m1 = 1;  ccp2m0 = 1;
           // trigger special event - reset timer 1
CCPR2H = (byte) (1000 >> 8);
CCPR2L = (byte) 1000;

// turn on timers
tmr1on = 1;
tmr3on = 1;

ccp1if = 0;    // kill any pending interrupts
ccp2if = 0;

ccp1ie = 1;
ccp2ie = 1;
peie = 1;
gieh = 1;

ccp1_int_occ = FALSE;
ccp2_int_occ = FALSE;

while(n<time_ms)
{
    if(ccp1_int_occ)
    {
        ccp1_int_occ = FALSE;

        if (dir ==CW)
        {
            ++index;    // note that this is global
            if (index > 7)
            {
                index = 0;
            }
        }
    }
    else

```

```

        {
            --index;
            if (index == 0xff)
            {
                index = 7;
            }
        }
    }

    if (ccp2_int_occ)
    {
        ccp2_int_occ = FALSE;
        ++n;
    }

#ifdef FLOAT
    x = (float) (stop_delay - start_delay);
    y = (float) (n) / (float) (time_ms);
    x = x * y;

    d = (unsigned long) x;
#else
    d = (start_delay - stop_delay)/128 * n / time_ms;
#endif

    delay = start_delay + d;

    new_match = MAKE_LONG(CCPR1H, CCPR1L) + delay;
    CCPR1H = (byte) (new_match >> 8);
    CCPR1L = (byte) new_match;
}
}
while(gieh)
{
    gieh = 0;
}

ccplif = 0;    // kill any interrupt
ccp2if = 0;

tmr1on = 0;    // turn off timers
tmr3on = 0;

ccplie = 0;
ccp2ie = 0;
}

#ifdef int_ccp1
ccp1_int_handler(void)
{
    LATB = (LATB & 0xf0) | pattps[index];
    ccp1_int_occ = TRUE;
}
#endif

#ifdef int_ccp2
ccp2_int_handler(void)
{

```

```

    ccp2_int_occ = TRUE;
}

#int_default
default_int_handler(void)
{
}

#include <delay.c>
#include <ser_18c.c>

```

Input Capture.

The following set of routines illustrate the concept of input capture in the setting of decoding an infrared TV remote control. The idea of input capture is that a timer is running and when an event occurs on the CCPx input, the value of the timer is transferred to the CCPRxH & L and a CCP interrupt occurs. Note that the transfer from the timer to the CCPR registers occurs at the time of the “event”, not in the interrupt service routine. An event may be defined as a rising or falling pulse on the CCP input, every 4th rising or every 16th rising. In these applications I used the rising edge and falling edge.

[Note. Over a year ago, I encountered a problem with the CCS compiler is working with arrays of longs in functions. The pointer arithmetic in the function was being handled as if the array was an array of bytes. As the following routines uses arrays of longs, I first developed a simple routine, ARRY_TST.C, to check that arrays of longs are being properly handled. It appears this problem has been resolved.]

Sony Remote Control.

This material deals with interfacing a Sony TV remote control unit with the PIC. It was developed by Timothy Wallace as a part of his Senior Project (2 credits) in the Dept of Electrical and Computer Engineering at Morgan State University.

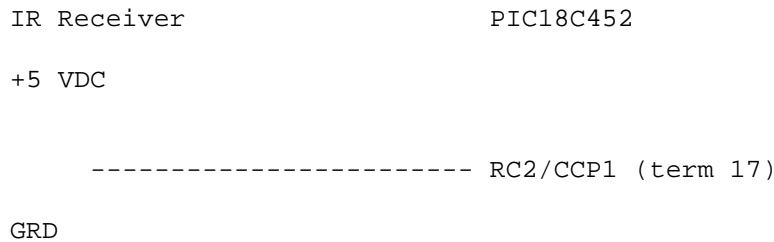
The TV remote control technology is mature and inexpensive and an infrared remote control might be used in many PIC applications in place of a keypad or other means of inputting data. One need only look to the home entertainment industry to see that consumers insists on remote control and for the industry, they save on all of the fancy and expensive mechanics associated with knobs and buttons, usually providing but a bare minimum of push buttons for local control.

Obvious applications are movement, e.g., turn a motor one way or another, turn it faster or slower or turn it 23 clicks in the defined direction. However, the remote might be used for virtually any data entry, e.g., perform a temperature measurement on sensor 1, or set a thermostat threshold to 28 degrees C.

A Sony TV remote was used in developing this material, simply because I happened to have a Sony TV remote control lying around. The following routine were also tested with a Sony CD remote (number keys only) and with two "One For All" universal remotes of different vintages using TV programming code 000. However, this material was developed by reverse engineering the Sony remote. That is, we observed the operation of the remote on a high quality storage scope as opposed to consulting authoritative literature (if such exists). The danger here is assuming our observations extend to all Sony TV remotes. However, with a "Universal", you should be able to duplicate our results.

The only hardware that is required is a Sony TV remote or a "Universal" and a small 38 kHz three terminal infrared receiver which is available from [Jameco](http://www.jameco.com) as their numbers #139889 or #131908. These are nominally \$3.00 each.

This is configured as shown;



When idle, the output of the IR receiver is high. When the receiver detects a 38 KHz burst of infrared, the output goes low for the duration of the burst.

We looked at the output of the IR receiver using a good quality storage scope and reached a number of conclusions.

1. When a button on the remote is depressed, a code consisting of a start pulse, immediately followed by twelve bits is sent. As long as the button is depressed, this is repeated with a substantial idle time between each sequence.
2. A start pulse is a burst of nominally 2.5 ms. I assume one reason for this length is to assure the automatic gain control (AGC) associated with the IR receiver has sufficient time to adjust to the proper level. In program IRRem_2.C, a duration of greater than 2.0 ms is assumed to be a valid start pulse.
3. A zero bit consists of no burst for nominally 0.5 ms followed by a burst having a duration of nominally 0.75 ms. That is, the output of the IR receiver is high for 0.5 ms and then low for 0.75 ms. In program IRRem_2.C, a no burst time of 0.3 to 0.6 ms is assumed to be valid. If the low time is less than 1.0 ms, the bit is assumed to be a zero.
4. A one bit consists of no burst for nominally 0.5 ms followed by a burst having a duration of nominally 1.25 ms. In program IRRem_2.C, a no burst time of 0.3 to 0.6 ms is assumed to be valid. If the low time is greater than 1.0 ms, the bit is assumed to be a one.
5. The bits are sent, least significant bit first.

Program IRREM_1.C.

The first step is to capture a pulse train.

In function input_capture(), timer 1 is set up and the CCP1 module is configured for input capture;

```
ccp1m3 = 0; ccp1m2 = 1; ccp1m1 = 0; ccp1m0 = start_state;
```

Note that if variable start_state is a zero, the first capture occurs on the first negative transition on input RC2/CCP1.

When the negative going transition occurs, the value of Timer 1 is copied to CCPR1H & L. In the CCP1 interrupt service routine, the ccp1m0 bit is inverted such that the next capture will occur on a positive going

transition. On noting the interrupt, the value of CCPR1H & L are copied to the array a[]. This process continues for each of the num_transitions.

The values of each of the transitions are then displayed on the terminal.

```
// Program IRRem_1.C
//
// Illustrates input capture using Timer 1 in conjunction with CCP1. Function
// input_capture() camps on RC2/CCP1 (term 17) and when a transition of the
// specified start state occurs, the times at which the specified number of
// transitions occur are saved in array a.
//
// This program captures a pulse train and displays the result.
//
//      IR Receiver ----- RC2/CCP1 (term 17)
//
// copyright, Peter H. Anderson, Baltimore, Jan, '02

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

#define FALSE 0
#define TRUE !0

#define MAKE_LONG(h, l) (((long) h) << 8) | (l)

void input_capture(unsigned long *a, byte start_state, byte num_transitions);
void print_array(unsigned long *a, byte num_ele);

byte ccpl_int_occ;

void main(void)
{
    unsigned long t_times[18];

    ser_init();
    while(1)
    {
        input_capture(t_times, 0, 18);
        print_array(t_times, 18);
    }
}

void input_capture(unsigned long *a, byte start_state, byte num_transitions)
{
    byte n;
    // set up timer1
    tloscen = 0;
    tmrlcs = 0;    // internal 1 usec clock
    tlckps1 = 0;  tlckps0 = 0;
    tmrlon = 1;
```

```

// set up ccpl for input capture
t3ccp2 = 0; t3ccp1 = 0; // timer 1 is clock source for both CCP modules
ccplm3 = 0; ccplm2 = 1; ccplm1 = 0; ccplm0 = start_state;

// enable interrupts
ccpl_int_occ = FALSE;
peie = 1;
ccplie = 1;
gieh = 1;

for (n=0; n<num_transitions; n++)
{
    while(!ccpl_int_occ) /* loop */ ;
    while(gieh)
    {
        gieh = 0;
    }

    ccpl_int_occ = FALSE;
    a[n] = MAKE_LONG(CCPR1H, CCPR1L);

    gieh = 1;
}

while(gieh)
{
    gieh = 0;
}
ccplie = 0;
peie = 0;
}

void print_array(unsigned long *a, byte num_ele)
{
    byte n;

    printf(ser_char, "*****\r\n");

    for (n=0; n<num_ele; n++)
    {
        printf(ser_char, "%2x%2x\r\n", (byte) (a[n] >> 8), (byte) (a[n] & 0xff));
    }
}

#int_ccpl
ccpl_int_handler(void)
{
    // invert the m0 bit
    ccplm0 = !ccplm0;
    ccpl_int_occ = TRUE;
}

#int_default
default_int_handler(void)
{

```

```

}

#include <delay.c>
#include <ser_18c.c>

```

Program IRREM_2.C.

This builds on the previous routine to actually decode the IR pulse train and either flash an LED a number of times, corresponding to the value of a number key, increase or decrease the flash rate (up or down volume) or change from one LED to another (up or down channel).

In function `fetch_IR_code()`, an array of transition times (`t_time[]`) is captured. Note that these are times that transitions actually occurred. These are converted to the widths of the pulses by subtracting element `n` from element `n+1` in function `convert_to_widths()`.

The widths are then checked using two criterion. That the start pulses (`widths[0]`) is 2 ms or larger and that are odd elements (inter-burst time) are in the range nominally 0.5 ms (0.30 to 0.60 ms). If either of these conditions is not met, another pulse train is captured.

However, if these conditions are met, the even elements, other than 0 (2, 4, 6, etc) of the widths are examined using 1.0 ms as a threshold. If a width is less than 1.0 ms, the bit is assumed to be a zero and if greater than 1.0 ms, the bit is assumed to be a one. The “code” is built, beginning with the least significant bit. For example, if `widths[2]` is 600 us, it is assumed to be a zero and it is the least significant bit of variable code.

In `main()`, if a code is less than 10, it is assumed to be a number key and the designated LED is flashed `code + 1` times at a designated speed. Note that code 0 corresponds to key 1, code 1 to key 2, etc. If the code is 16 (up channel) or 17 (down channel), the designated LED is switched from one to another. If the code is 18 (up volume) or 19 (down volume), the flash rate is either increased or decreased. Note that if the code is 16 – 19, the LED is quickly flashed one time to provide the user with some feedback that something was received from the remote control.

```

// Program IRREM_2.C
//
// Fetches a code from a Sony Remote Control. If the code is less than 10
// it is interpreted as a number key and an LED is flashed the number of times
// of the key value. If the code is up volume (18) or down volume (19) the flash
// rate is either increased or decreased. If the code is up channel (16) or down
// channel, a different LED is flashed.
//
//
// This program captures a pulse train and displays the result.
//
//      IR Receiver ----- RC2/CCP1 (term 17)
//
//
// copyright, Peter H. Anderson, Baltimore, Jan, '02

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

```

```

#define FALSE 0
#define TRUE !0

#define MAKE_LONG(h, l) (((long) h) << 8) | (l)

byte fetch_IR_code(void);

void input_capture(unsigned long *a, byte start_state, byte num_transitions);
void convert_to_widths(unsigned long *a, unsigned long *widths, byte num_ele);
void print_array(unsigned long *a, byte num_ele);
void put_bit(byte *p, byte bit_pos, byte bit_state);

byte flash_led(byte led_num, byte num_flashes, long delay_time);

byte ccpl_int_occ;

void main(void)
{
    byte code, led_num;
    long delay_time = 200;

    led_num = 0;

    ser_init();
    while(1)
    {
        code = fetch_IR_code();
        printf(ser_char, "Code = %d\r\n", code);
        if (code < 10)
        {
            flash_led(led_num, code+1, delay_time); // add one to the key value
        }

        else if (code == 16) // up channel
        {
            (led_num) ? (led_num = 0) : (led_num = 1);
            flash_led(led_num, 1, 50); // provide some feedback
        }

        else if (code == 17)
        {
            (led_num) ? (led_num = 0) : (led_num = 1);
            flash_led(led_num, 1, 50); // provide some feedback
        }

        else if (code == 18) // up volume
        {
            delay_time = delay_time - 50;
            if (delay_time < 50)
            {
                delay_time = 50; // max speed
            }
            flash_led(led_num, 1, 50); // provide some feedback
        }

        else if (code == 19) // up volume
    }
}

```

```

    {
        delay_time = delay_time + 50;
        if (delay_time > 500)
        {
            delay_time = 500; // min speed
        }
        flash_led(led_num, 1, 50); // provide some feedback
    }

    else
    {
        // valid code but not defined
    }
}

byte fetch_IR_code(void)
{
    unsigned long t_times[18], widths[17];
    byte n, code, valid;

    do
    {
        input_capture(t_times, 0, 18);
        // print_array(pulses, 18);
        convert_to_widths(t_times, widths, 17);
        print_array(widths, 17);

        valid = TRUE;

        if (widths[0] < 2000)
        {
            valid = FALSE;
        }

        for (n=0; n<8; n++)
        {
            if ((widths[2*n+1] < 300) || (widths[2*n+1] > 600))
            {
                valid = FALSE;
            }
        }
    } while(!valid);

    printf(ser_char, "\n\rSuccess\n\r");

    code = 0x00;

    for (n=0; n<7; n++)
    {
        if (widths[2*(n+1)] < 1000)
        {
            put_bit(&code, n, 0);
        }
        else
        {
            put_bit(&code, n, 1);
        }
    }
}

```

```

    }
}

return(code);
}

void input_capture(unsigned long *a, byte start_state, byte num_transitions)
{
    byte n;

    // set up timer1
    tloscen = 0;
    tmr1cs = 0;    // internal 1 usec clock
    tickps1 = 0;  tickps0 = 0;
    tmr1on = 1;

    // set up ccpl for input capture
    t3ccp2 = 0;  t3ccp1 = 0;  // timer 1 is clock source for both CCP modules
    ccplm3 = 0; ccplm2 = 1; ccplm1 = 0;  ccplm0 = start_state;

    // enable interrupts
    ccpl_int_occ = FALSE;
    peie = 1;
    ccplie = 1;
    gieh = 1;

    for (n=0; n<num_transitions; n++)
    {
        while(!ccpl_int_occ)  /* loop */ ;

        while(gieh)
        {
            gieh = 0;
        }

        ccpl_int_occ = FALSE;
        a[n] = MAKE_LONG(CCPR1H, CCPR1L);

        gieh = 1;
    }

    while(gieh)
    {
        gieh = 0;
    }
    ccplie = 0;
    peie = 0;
}

void convert_to_widths(unsigned long *a, unsigned long *widths, byte num_ele)
{
    byte n;

    for (n=0; n<num_ele; n++)
    {
        if (a[n+1] > a[n])

```

```

        {
            widths[n] = a[n+1] - a[n];
        }

        else
        {
            widths[n] = a[n+1] - a[n];
        }
    }
}

void put_bit(byte *p, byte bit_pos, byte bit_state)
{
    const byte mask_1[8] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};

    if (bit_state)
    {
        *p = *p | mask_1[bit_pos];
    }

    else
    {
        *p = *p & (~mask_1[bit_pos]);
    }
}

void print_array(unsigned long *a, byte num_ele)
{
    byte n;

    printf(ser_char, "*****\r\n");

    for (n=0; n<num_ele; n++)
    {
        printf(ser_char, "%ld\r\n", a[n]);
    }
}

byte flash_led(byte led_num, byte num_flashes, long delay_time)
{
    const byte mask_1[2] = {0x01, 0x02};
    byte n;

    pspmode = 0;           // portd configured as general purpose IO
    LATD = LATD & 0xfc;
    TRISD = TRISD & 0xfc;   // be sure lower two bits on portd are output zeros

    for (n=0; n<num_flashes; n++)
    {
        LATD = (LATD & 0xfc) | mask_1[led_num];
        delay_ms(delay_time);
        LATD = LATD & 0xfc;
        delay_ms(delay_time);
    }
}

#include <int_ccpl>

```

```

ccpl_int_handler(void)
{
    // invert the m0 bit
    ccplm0 = !ccplm0;
    ccpl_int_occ = TRUE;
}

#int_default
default_int_handler(void)
{
}

#include <delay.c>
#include <ser_18c.c>

```

Program IRREM_3.C.

Note that in the previous examples, when the `input_capture()` function is executed, there is a very real problem that if no pulse train is detected, the processor will simple loop forever in this function.

Although, one might rely on the watch dog timer to force a reset, resetting the processor is less than elegant.

Program IRREM_3.C differs from the previous only in that in the `input_capture()` routine, Timer 3 running from the external 32.768 kHz crystal is turned on and preloaded with 0x8000 so as to cause an interrupt if no pulse train is detected within one second. If a timeout occurs, FAILURE is returned to the calling function, `fetch_IR_code()` which returns the code 0xff to `main()`.

However, if a pulse train is detected prior to the one second timeout, the timer is turned off and interrupts are disabled and SUCCESS is returned to the calling function.

[Again, note that I simply could not get the 32.768 kHz crystal to oscillate when using the RICE-17A emulator and used the programmed PIC12C509A on input T13CKI. Of course, in this application one might use the `fosc/4` as a source for Timer 3 as illustrated in other routines and thus eliminate the need of the 32.768 kHz crystal].

```

// Program IRRem_3.C
//
// This program is the same as IRRem_2.C except that Timer 3 is used to
// prevent lockup in the input_capture() function. If a one second timeout
// occurs, function input_capture() returns FAILURE and a timeout message is
// displayed to the terminal.
//
//
//      IR Receiver ----- RC2/CCP1 (term 17)
//
//
// copyright, Peter H. Anderson, Baltimore, Dec, '01

#case
#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

```

```

#define FALSE 0
#define TRUE !0

#define FAILURE 0
#define SUCCESS !0

#define EXT_32KHZ

#define MAKE_LONG(h, l) (((long) h) << 8) | (l)

byte fetch_IR_code(void);

byte input_capture(unsigned long *a, byte start_state, byte num_transitions);
void convert_to_widths(unsigned long *a, unsigned long *widths, byte num_ele);
void print_array(unsigned long *a, byte num_ele);
void put_bit(byte *p, byte bit_pos, byte bit_state);

byte flash_led(byte led_num, byte num_flashes, long delay_time);

byte ccpl_int_occ, tmr3_int_occ;

void main(void)
{
    byte code, led_num;
    long delay_time = 200;

    led_num = 0;

    ser_init();
    while(1)
    {
        code = fetch_IR_code();
        printf(ser_char, "Code = %u\r\n", code);
        if (code < 10)
        {
            flash_led(led_num, code+1, delay_time); // add one to the key value
        }

        else if (code == 16) // up channel
        {
            (led_num) ? (led_num = 0) : (led_num = 1);
            flash_led(led_num, 1, 50); // provide some feedback
        }

        else if (code == 17)
        {
            (led_num) ? (led_num = 0) : (led_num = 1);
            flash_led(led_num, 1, 50); // provide some feedback
        }

        else if (code == 18) // up volume
        {
            delay_time = delay_time - 50;
            if (delay_time < 50)
            {

```

```

        delay_time = 50; // max speed
    }
    flash_led(led_num, 1, 50); // provide some feedback
}

else if (code == 19) // up volume
{
    delay_time = delay_time + 50;
    if (delay_time > 500)
    {
        delay_time = 500; // min speed
    }
    flash_led(led_num, 1, 50); // provide some feedback
}

else if (code == 0xff)
{
    printf(ser_char, "\r\nTime Out\r\n");
}

else
{
}

}
}

```

```

byte fetch_IR_code(void)
{
    unsigned long t_times[18], widths[17];

    byte n, code, valid;

    do
    {
        if (!input_capture(t_times, 0, 18)) // if failure
        {
            return(0xff);
        }
        // print_array(pulses, 18);
        convert_to_widths(t_times, widths, 17);
        print_array(widths, 17);

        valid = TRUE;

        if (widths[0] < 2000)
        {
            valid = FALSE;
        }

        for (n=0; n<8; n++)
        {
            if ((widths[2*n+1] < 320) || (widths[2*n+1] > 600))
            {
                valid = FALSE;
            }
        }
    }
}

```

```

    } while(!valid);

    printf(ser_char, "\n\rSuccess\n\r");

    code = 0x00;

    for (n=0; n<7; n++)
    {
        if (widths[2*(n+1)] < 1000)
        {
            put_bit(&code, n, 0);
        }
        else
        {
            put_bit(&code, n, 1);
        }
    }

    return(code);
}

byte input_capture(unsigned long *a, byte start_state, byte num_transitions)
{
    byte n;
    // set up timer1

    tmrlcs = 0;    // internal 1 usec clock
    t1ckps1 = 0;   t1ckps0 = 0;
    tmrlon = 1;

    // set up timer 3
    //t3rd16 = 1;
    t3ckps1 = 0;   t3ckps0 = 0;        // 1:1 prescale

    TMR3H = 0x80; TMR3L = 0x00;    // prime for one second

#ifdef EXT_32KHZ
    tloscen = 0;
#else
    tloscen = 1;
#endif
    tmr3cs = 1;    // external clock - 32.768
    tmr3on = 1;

    // set up ccpl for input capture
    t3ccp2 = 0;   t3ccp1 = 0;    // timer 1 is clock source for both CCP modules
    ccplm3 = 0; ccplm2 = 1; ccplm1 = 0;   ccplm0 = start_state;

    // enable interrupts
    ccpl_int_occ = FALSE;
    tmr3_int_occ = FALSE;
    ccplif = 0;    // kill any pending interrupts
    tmr3if = 0;
    peie = 1;
    ccplie = 1;
    tmr3ie = 1;

```

```

gieh = 1;

for (n=0; n<num_transitions; n++)
{
    while(!ccpl_int_occ)
    {
        if (tmr3_int_occ) // timer 3 is used as a timeout
        {
            while(gieh)
            {
                gieh = 0;
            }
            tmr3on = 0;
            tmr3ie = 0;           // turn off interrupts
            ccplie = 0;
            peie = 0;

            tmr3if = 0;
            ccplif = 0;
            return(FAILURE);
        }
    }

    while(gieh)
    {
        gieh = 0;
    }

    ccpl_int_occ = FALSE;
    a[n] = MAKE_LONG(CCPR1H, CCPR1L);

    gieh = 1;
}

while(gieh)
{
    gieh = 0;
}
tmr3on = 0;
ccplie = 0;
tmr3ie = 0;
peie = 0;

tmr3if = 0;
ccplif = 0;
return(SUCCESS);
}

// Other functions have been deleted in this discussion. The full routine appears
// in the source code files.

```

SPI Master.

The following routines all deal with the use of the PIC18CXX2 as an SPI Master controlling a Microchip 25LC640 (8K X 8) and an Atmel AT45DB321 (4Meg X 8) Flash EEPROM.

My primary intent in developing these routines was to illustrate the operation of the Atmel AT45DB321 massive 4Meg Byte EEPROM. I decided to implement this using the PIC18CXX2 to verify there were no surprises in working with the PICs synchronous serial port. There were none.

I did waste better than a day fooling with the 25LC640. In reading from the device I was obtaining erratic results which varied with my hand position and pressure on the 25LC640 and the emulator cable and I was certain I had found a problem with the emulator. In fact, I found that the /HOLD lead on the 25LC640 must be pulled high though a pullup resistor for normal operation. I have worked with this device in the past and do not recall having encountered this.

Program 25_60BB.C.

A portion of this routine appears below. Functionally, it differs very little from the bit-bang version which was presented for the PIC16F87X.

However, I did tinker with the two bank protect bits (BP1 and BP0) in the 25LC640's status register. In one case, these two non-volatile bits are cleared to zero such that none of the EEPROM memory is protected from writes. Data is written to the EEPROM and then read back and displayed to the terminal. The two bank protect bits are then set to ones so as to protect all EEPROM locations. New data is written, but reading and displaying the data indicates that the new data was not programmed into the EEPROM. Rather, the previous data is retained.

This routine uses software bit bang rather than the hardware SSP module. Frankly, I see no advantage in one over the other. Again and again, I see posting to news groups by users seeking to control both I2C and SPI devices and moaning that the larger PICs have but one synchronous serial port. Bit-bang one or the other or both.

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

    ser_init();

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

    _25_640_setup_SPI();

    _25_640_write_status_reg(0x00); // no block protection
    dat = _25_640_read_status_reg();
    printf(ser_char, ".....\r\n%2x\r\n", dat);
                                // display status register

    printf(ser_char, "\n\r");

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

```

}

for (n=0, adr = 0x0700; n<10; n++, adr++)
{
    dat = _25_640_read_byte(adr);          // now read back the data
    ser_hex_byte(dat);                     // and display
    ser_char(' ');
    delay_ms(250);
}

_25_640_write_status_reg(0x0c);           // block protect all memory
dat = _25_640_read_status_reg();          // display status register
printf(ser_char, ".....\r\n%2x\r\n", dat);

printf(ser_char, "\n\r");
for (n=0, adr = 0x0700; n<10; n++, adr++)
{
    _25_640_write_byte(adr, n+20);         // write 20, 21, 22, etc
    ser_char('!');                         // to verify that something is happening
}

for (n=0, adr = 0x0700; n<10; n++, adr++)
{
    dat = _25_640_read_byte(adr);          // now read back the data
    ser_hex_byte(dat);                     // and display. Note that this will be 10, 11, 12 from previous write
    ser_char(' ');
    delay_ms(250);
}

while(1)                ;                // continual loop
}

```

Programs 25_640_1.C and 25_640_2.C.

These programs use the PIC's SSP Module in writing to a reading from EEPROM, one byte at a time and eight bytes at a time, sequential write and read. Again, these are similar to those previously developed for the PIC16F87X.

However, I did make one improvement. At least, it improves the clarity of the code. I am uncertain, it is any more efficient.

Note that a pointer to an array of bytes is passed to the function and each element of the array is output and the value received is placed in that element.

```

void spi_io(byte *io, byte num_bytes)
{
    byte n;

    for(n=0; n<num_bytes; n++)
    {
        SSPBUF = io[n];
        while(!stat_bf)    /* loop */        ;
        io[n] = SSPBUF;
    }
}

```

```
}
```

For example;

```
void _25_640_read_seq_bytes(long adr, byte *read_dat, byte num_bytes)
{
    byte io[19], n;

    io[0] = SPI_READ;
    io[1] = (byte) (adr >> 7);
    io[2] = (byte) adr;

    latb7 = 0;           // CS low - begin sequence
    spi_io(io, num_bytes + 3);
    latb7 = 1;

    for (n=0; n<num_bytes; n++)
    {
        read_dat[n] = io[n+3];
    }
}
```

Note that the command for a memory read, the high and low bytes of the address are copied to elements 0, 1 and 2. The values in the other elements really do not matter as the 25LC640 stops listening after receiving the address. Beginning with byte 3, data is read from the 25LC640. These bytes are then copied to array read_data.

Note that a total of 19 bytes are sent and 19 bytes are received. For the first three bytes, the data is read by the 25LC640. The three bytes received by the PIC from the 25LC640 are simply ignored. For the next num_bytes, the value of the bytes sent to the 25LC640 really do not matter. They are sent only to read num_bytes from the EEPROM.

ATMEL AT45DB321 (4.0 Meg Byte Flash EEPROM).

Much of this effort was done by Lisa Mickens as a part of her Senior Design Project.

It seems as if every time I visit the [Atmel](#) site, they have a few new devices that are twice the size of their previous offerings.

The Atmel AT45DB321 is a 3.0 VDC device and thus in interfacing with a PIC or similar processor having TTL (5V) logic outputs, it is necessary to supply the EEPROM with 3.0 VDC, scale the levels on the /CS, SCK and Master Out Slave In (MOSI) from 5 Volt to 3 Volt logic and also boost the level of the Master In Slave Out (MISO) from the EEPROM from 3 V to 5 V logic.

Rather than going through the agony of trying to find the AT45DB321, mount this surface mount device and fool with these interface issues, we opted for a [Rabbit Semiconductor](#) SF1010 Serial Flash Expansion EEPROM which is marketed as an add-on for the Rabbit. Rabbit also has another model which is configured with a AT45DB642 which provides for 8 Megabytes of EEPROM.

For some applications where the \$99.00 cost is not a factor, the Rabbit board may be a viable path. It is a small standalone board which provides a connector and a number of through hole connector pads which we used for inserting wires to access +5 VDC, GRD, SCK, MOSI, MISO and /CS. The board includes a 3.0 VDC regulator and uses resistor dividers to scale the TTL to 3.0 VDC on SCK, MOSI and /CS. The boost from 3 to 5 VDC logic is implemented using a FET.

There is an issue to consider if one is attempting to operate other SPI devices on the same bus (SCK, MOSI and MISO). In fact, I don't think it is possible.

The 3.0 to 5.0 VDC boost on the MISO is implemented using a FET with the output being an open with a 1K pullup to +5 VDC when the EEPROM is idle. This is not quite the high impedance typically associated with the MISO lead. I assume other slave devices are not capable of pulling the MISO to near ground through the 1K resistor. I might have gone for a 220K pullup, rather than expecting other slave devices to sink 5 mA. For example, the 25LC640 sink current is rated at 400 uA.

Note that the FET also has the effect of inverting the data on the MISO lead. That is, when reading, a logic one is interpreted as a zero, and a logic zero as a one. Thus, in the attached routines, you will note that a ones compliment is performed on each byte which is read from the AT45DB331.

Program AT45DB_1.C.

The AT45DB321 provides two 528 byte RAM buffers. Note that 528 bytes requires a 10-bit address.

This routine illustrates how to write 16 bytes at a time to one of the buffers. The data is then read and displayed to the terminal.

In writing, the command code 0x84 (buffer 1) or 0x87 (buffer 2), followed by the high and low bytes of the address, followed by an idle byte to allow time for the AT45 EEPROM to set up. The sixteen bytes are then sent. In reading, either the command code 0xd4 or 0xd6 is sent followed by the high and low bytes of the address, followed by an idle byte. The sixteen bytes are then read.

Note that the data sheet indicates the commands for reading from the buffer may be either 0xd4 or 0x54 (or 0xd6 or 0x56 for buffer 2). In fact, I found the 0x54 command did not function properly.

```
// AT45DB_1.C
//
// Illustrates the use of the SSP module in interfacing with an AT45DB321 Flash
// EEPROM using the SPI protocol. Illustrates writing to and reading from RAM
// buffer 1.
//
// PIC18F452
//
// RC5/SDO (term 24) ----- MOSI -----> SI
// RC4/SDI (term 23) <----- MISO ----- SO
// RC3/SCK (term 18) ----- SCK -----> SCK
// RB7/CS (term 40) -----> /CS
//
// Copyright, Peter H. Anderson, Baltimore, MD, Jan, '02
```

```
#case
```

```
#device PIC18C452
```

```
#include <defs_18c.h>
```

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

```
#include <ser_18c.h>
```

```
#define TRUE !0
```

```

#define FALSE 0

#define B1_WRITE 0x84
#define B2_WRITE 0x87

#define B1_READ 0xd4    // see text
#define B2_READ 0xd6

#define DONT_CARE 0x00

void at45_setup_SPI(void);

void at45_buffer_write_seq_bytes(byte buffer, unsigned long adr, byte *write_dat,
                                byte num_bytes);
void at45_buffer_read_seq_bytes(byte buffer, unsigned long adr, byte *read_dat,
                                byte num_bytes);
void display(byte *read_dat, byte num_bytes);

void spi_io(byte *io, byte num_bytes);

void main(void)
{
    unsigned long adr;
    byte dat[16], n;

    ser_init();
    printf("ser_char, "\r\n.....\r\n");

    at45_setup_SPI();

    for(adr=0x000; adr < 0x0200; adr+=16)        // write a block of eight bytes
    {
        for (n=0; n<16; n++)
        {
            dat[n] = (adr % 10) + n;
        }
        at45_buffer_write_seq_bytes(1, adr, dat, 16);
    }

    for (adr=0x000; adr < 0x0200; adr+=16)
    {
        at45_buffer_read_seq_bytes(1, adr, dat, 16);
        display(dat, 16);
    }

    while(1)
    {
    }
}

void at45_setup_SPI(void)
{
    ssplen = 0;
    ssplen = 1;
    sspm3 = 0; sspm2 = 0; sspm1 = 1; sspm0 = 0;
                                // Configure as SPI Master, fosc / 64
    ckp = 0;                    // idle state for clock is zero
}

```

```

    stat_cke = 1;          // data transmitted on rising edge
    stat_smp = 1;          // input data sampled at end of clock pulse

    latc3 = 0;
    trisc3 = 0;    // SCK as output 0

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

    latb7 = 1;          // CS for AT45DB321
    trisb7 = 0;
}

void at45_buffer_write_seq_bytes(byte buffer, unsigned long adr, byte *write_dat,
                                byte num_bytes)
{
    byte io[20], n;

    if (buffer == 1)
    {
        io[0] = B1_WRITE;
    }

    else
    {
        io[0] = B2_WRITE;
    }

    io[1] = DONT_CARE;

    io[2] = (byte) (adr >> 8);
    io[3] = (byte) adr;

    for (n=0; n<num_bytes; n++)
    {
        io[n+4] = write_dat[n];
    }

    latb7 = 0;
    spi_io(io, num_bytes + 4);
    latb7 = 1;
}

void at45_buffer_read_seq_bytes(byte buffer, long adr, byte *read_dat,
                                byte num_bytes)
{
    byte io[21], n;

    if (buffer == 1)
    {
        io[0] = B1_READ;
    }
    else
    {
        io[0] = B2_READ;
    }
    io[1] = DONT_CARE;

```

```

io[2] = (byte) (adr >> 8);
io[3] = (byte) adr;
io[4] = DONT_CARE;

latb7 = 0;          // CS low - begin sequence
spi_io(io, num_bytes + 5);
latb7 = 1;

for (n=0; n<num_bytes; n++)
{
    read_dat[n] = ~io[n+5];    // note one's comp
}

}

void display(byte *read_dat, byte num_bytes)
{
    byte n;

    printf(ser_char, "\n\r");

    for (n=0; n<num_bytes; n++)
    {
        ser_hex_byte(read_dat[n]);
        ser_char(' ');

        if (((n+1)%16) == 0)    // 16 values per line
        {
            printf(ser_char, "\n\r");
        }
    }
}

void spi_io(byte *io, byte num_bytes)
{
    byte n;

    for(n=0; n<num_bytes; n++)
    {
        SSPBUF = io[n];
        while(!stat_bf)    /* loop */          ;
        io[n] = SSPBUF;
    }
}

#include <delay.c>
#include <ser_18c.c>

```

Program AT45DB_2.C.

This program illustrates how the two RAM buffers may be used to continually log data.

With an EEPROM having a single RAM buffer, data is written to the buffer and then the buffer is transferred to EEPROM which may take as long as 25 ms. This may be tolerable in applications where the data is being logged slowly, but not in applications requiring say 1000 samples per second. [Actually, the PIC18CXX2 probably has enough RAM to buffer the data, but this could get a bit complex].

The AT45DB321 provides two RAM buffers. Thus, one may write to Buffer 1, and when full, a command is issued to transfer this to the main EEPROM memory. While this data is being burned into EEPROM, one may write to Buffer 2, and when full, transfer this to the main memory, and then return to write to Buffer 1.

Data is written to memory, by first writing to a RAM buffer as illustrated in the previous program. When full, the buffer is transferred to EEPROM by sending either command 0x83, for Buffer 1, or 0x86 for Buffer 2. This is followed by the high and low bytes of the page address, followed by an idle byte.

Note that the AT45DB321 provides 8192 pages which requires 13 address bits. It's cousin, the AT45DB642 provides 16,384 pages, requiring 14 bits. Recall, that each page consists of 528 bytes which requires 10 address bits. Thus, each byte has its unique 24 bit address. Atmel has developed their protocol such that this may be transferred in three bytes.

```
0,  P12, P11, P10, P9, P8, P7, P6    (Byte 1)
P5, P4,  P3,  P2,  P1, P0, B9, B8    (Byte 2)
B7, B6,  B5,  B4,  B3, B2, B1, B0    (Byte 3)
```

Where P12, P11, etc is the page address and B9, B8, etc is the byte address within that page.

Note that the page address is shifted up two places such that P12 is in the bit 14 position. Byte 1 is then the high byte of the result. Byte 2 is the high byte of the result, ored with the high byte of the buffer address and byte 3 is the low byte of the buffer address .

Of course, when transferring a buffer to an EEPROM page, only the page address is of interest. The lower two bits of Byte 2 and all of Byte 3 are "don't care" bits.

Data may be read by transferring from a page to a buffer and then reading the buffer. This might be useful in cases where only a few bytes are to be changed in the main memory. Transfer the page to the RAM buffer, modify the bytes and transfer the buffer back to the main memory. I did not do a routine to illustrate this.

However, data may also be read directly by issuing the 0xd2 command, the page address and the address within that page as illustrated above, followed by four idle bytes and then reading the data.

In this program, sixteen bytes of data are obtained by calling function `fetch_data_16bytes()`. Note that this is a stub which uses a static byte along with a mod 23 operation to generate data which is readily understood, but different from one fetch to another.

Blocks of sixteen bytes are continually fetched and written to RAM Buffer 1 until 512 bytes have been written and the buffer is then transferred to memory page 0. While this operation is being performed, the program continues by writing to RAM Buffer 2 until 512 bytes have been written and the buffer is then transferred to the next memory page. The program then returns to fill RAM Buffer 1, etc.

Note that I included a 1 ms delay after writing each 16 bytes to the RAM buffer such that by the time one buffer is full (about 32 ms), the other buffer has finished transferring to the main memory which requires about 25 ms.

Note that although the RAM buffers and the pages are 528 bytes wide, I used only 512 bytes. My feeling is that the additional 16 bytes are more appropriately used for administration. I didn't do anything with this, but administration that comes to mind is a CRC or checksum or a time stamp.

After the data has been “logged”, it is fetched and displayed, sixteen bytes at a time from the specified page and the specified address within that page.

```
// AT45DB_2.C
//
// Illustrates the use of the SSP module in interfacing with an AT45DB311
// EEPROM using the SPI protocol.
//
// Writes 512 bytes data, 16 bytes at a time to Buffer 1 and then writes the
// buffer to EEPROM page 0x0000. While the EEPROM write is being performed, the
// program writes 512 bytes to Buffer 2, 16 bytes at a time and then writes buffer
// 2 to EEPROM page 0x0001. Continues to Buffer 1, etc.
//
// The program then reads directly from program memory and displays to the
// terminal.
//
//      PIC18F452                      AT45DB321
//
// RC5/SDO (term 24) ----- MOSI -----> SI
// RC4/SDI (term 23) <----- MISO ----- SO
// RC3/SCK (term 18) ----- SCK -----> SCK
// RB7/CS (term 40) -----> /CS
//
// Copyright, Peter H. Anderson, Baltimore, MD, Jan, '02
```

```
#case
```

```
#device PIC18C452
```

```
#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>
```

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

```
#define B1_WRITE 0x84
#define B2_WRITE 0x87
```

```
#define B1MEM_WT 0x83
#define B2MEM_WT 0x86
#define MEM_RD 0xd2
```

```
#define DONT_CARE 0x00
```

```
void at45_setup_SPI(void);
```

```
void at45_buffer_to_memory(byte buffer, unsigned long page);
void at45_buffer_write_seq_bytes(byte buffer, unsigned long adr, byte *write_dat,
                                byte num_bytes);
void at45_memory_read_seq_bytes(unsigned long page, unsigned long adr,
                                byte *read_dat, byte num_bytes);
```

```
void spi_io(byte *io, byte num_bytes);
void fetch_data_16bytes(byte *dat);
```

```

void main(void)
{
    unsigned long page, adr;
    byte dat[16], n, buffer;

    ser_init();
    printf(ser_char, "\r\n.....\r\n");

    at45_setup_SPI();

    for (page = 0x0001; page < 0x0003; page++) // 3 * 512 bytes
    {
        if ((page%2) == 0) // if even page, use Buffer 1
        {
            buffer = 1;
        }
        else
        {
            buffer = 2; // for odd pages, use Buffer 2
        }

        for (adr = 0x000; adr < 0x200; adr+=16)
        {
            fetch_data_16bytes(dat);
            at45_buffer_write_seq_bytes(buffer, adr, dat, 16);
            // write to buffer
            delay_ms(1);
        }
        at45_buffer_to_memory(buffer, page); // buffer programmed to EEPROM
    }

    delay_ms(25); // allow for final page to be programmed

    for (page = 0x0001; page < 0x0003; page++) // now, read and display
    {
        for (adr = 0x000; adr < 0x0200; adr+=16)
        {
            at45_memory_read_seq_bytes(page, adr, dat, 16);
            printf(ser_char, "%4lx:%4lx: ", page, adr);
            for (n=0; n<16; n++)
            {
                printf(ser_char, "%2x ", dat[n]);
            }
            printf(ser_char, "\r\n");
        }
    }
}

void at45_setup_SPI(void)
{
    sspcn = 0;
    sspcn = 1;
    sspm3 = 0; sspm2 = 0; sspm1 = 1; sspm0 = 0;
    // Configure as SPI Master, fosc / 64
    ckp = 0; // idle state for clock is zero
    stat_cke = 1; // data transmitted on rising edge
    stat_smp = 1; // input data sampled at end of clock pulse
}

```

```

latc3 = 0;
trisc3 = 0;           // SCK as output 0

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

latb7 = 1;           // CS for AT45DB321
trisb7 = 0;
}

void at45_buffer_write_seq_bytes(byte buffer, unsigned long adr, byte *write_dat,
                                byte num_bytes)
{
    byte io[20], n;

    if (buffer == 1)
    {
        io[0] = B1_WRITE;
    }

    else
    {
        io[0] = B2_WRITE;
    }
    io[1] = DONT_CARE;
    io[2] = (byte) (adr >> 8);
    io[3] = (byte) adr;

    for (n=0; n<num_bytes; n++)
    {
        io[n+4] = write_dat[n];
    }

    latb7 = 0;
    spi_io(io, num_bytes + 4);
    latb7 = 1;
}

void at45_buffer_to_memory(byte buffer, unsigned long page)
{
    byte io[4];
    unsigned long x;

    if (buffer == 1)
    {
        io[0] = B1MEM_WT;
    }
    else
    {
        io[0] = B2MEM_WT;
    }

    x = page << 2;       // 0 Pal2, Pal1
    io[1] = (byte) (x>>8); // high 7 bits
    io[2] = (byte) (x);
    io[3] = DONT_CARE;
}

```

```

    latb7 = 0;
    spi_io(io, 4);
    latb7 = 1;
}

void at45_memory_read_seq_bytes(unsigned long page, unsigned long adr,
                                byte *read_dat, byte num_bytes)
{
    byte io[25], n;
    unsigned long x;

    io[0] = MEM_RD;        // read from program memory
    x = page << 2;         // 0 Pa12, Pa11
    io[1] = (byte) (x>>8); // high 7 bits
    io[2] = ((byte) (x)) + ((byte) (adr >> 8) & 0x03);
    io[3] = (byte) (adr);
    io[4] = DONT_CARE;
    io[5] = DONT_CARE;
    io[6] = DONT_CARE;
    io[7] = DONT_CARE;

    latb7 = 0;
    spi_io(io, num_bytes + 8);
    latb7 = 1;

    for (n=0; n<num_bytes; n++)
    {
        read_dat[n] = ~io[n+8];
    }
}

void spi_io(byte *io, byte num_bytes)
{
    byte n;

    for(n=0; n<num_bytes; n++)
    {
        SSPBUF = io[n];
        while(!stat_bf) /* loop */ ;
        io[n] = SSPBUF;
    }
}

void fetch_data_16bytes(byte *dat) // this is a stub that generates 16 values
{
    static byte m = 0x00;
    byte n;

    for (n=0; n<16; n++, m++)
    {
        dat[n] = m % 23;
    }
}

#include <delay.c>
#include <ser_l8c.c>

```

Program AT45DB_3.C.

This program is a simple extension of the previous routine which logs A/D conversions to the AT45DB321 at 1000 samples per second to EEPROM and also fetches the data from EEPROM and displays it to the terminal.

On boot, the program reads PORTB7 and if at a logic one, the program logs 1536 bytes of data and idles. If, on boot, PORTB7 is at a logic zero, the program reads from the AT45DB321 EEPROM and displays the data.

Timer 3 is used in conjunction with CCP2 configured in the “trigger special event” on match mode. On match, Timer 3 resets and also performs an A/D conversion on AN0 (term 2).

Thus, in logging data, Timer 3 and CCP2 are configured for a 1000 us timeout. The A/D module is configured to perform an A/D conversion on AN0 with the result left justified such that the most significant byte is in ADRESH. The timer is turned on and interrupts are enabled.

Note that the interrupt service routine communicates with the program using global variables; an array of bytes, `ad_buff[16]` and byte `ad_buff_index`. In the CCP2 interrupt service routine, ADRESH is copied to the array and the `ad_buff_index` is incremented.

In function `fetch_data_16bytes()`, the program waits until 16 conversions have been performed; i.e., `ad_buff_index` is 16. Byte array `ad_buff` is then copied to byte array `dat` and this data is written to the RAM buffer. When the buffer is full, it is transferred to the main memory and while the EEPROM is being burned, the other buffer is used as in the previous routine.

In developing this routine, I was concerned as to whether the program might be dropping samples. That is, if the time to write the data bytes to RAM, plus the time to command to transfer the buffer to EEPROM exceeded the time to perform 16 A/D conversions, the `ad_buff` array would overflow. Note that in the interrupt service routine, I provided a `#ifdef` to copy the index (actually 16 – index) to the array, rather than the value of ADRESH. This allowed me to verify the dumped data. Of course, I was quite sure this situation would not occur at 1000 sample per second as this allows a whopping 16 ms to write the sixteen bytes of data to the RAM buffer.

I did not experiment to see just how fast I could sample. One constraint is the time to service an interrupt, which, with the CCS overhead is about 100 us. The other constraint is that the time to write 32 sixteen byte chunks of data to the RAM must be greater than the time for the other buffer to complete the buffer to EEPROM transfer (25 ms) which suggests a maximum sample time of 50 us per sample. I suspect that even with the 4.0 MHz clock, one can sample and save 8000 samples per second. But, as I say, I did not tinker with this.

```
// AT45DB_3.C
//
// This is an example of a data logger using the AT45DB321.
//
// On boot, the program reads input PORTB7 and if at one, performs
// A/D conversions (1000/sec) and saves to the AT45 EEPROM.
//
// If, on boot, PORTB7 is at zero, the content of the AT45 EEPROM is
// read and displayed to the terminal.
//
// Uses Timer 3 in conjunction with CCP2 in the "trigger special event" mode
// to reset the timer every 1000 us and also perform an A/D conversion on AN0.
// Note that only the high byte of the A/D conversion is saved to EEPROM
```

```

//
//
// PIC18F452
//
// RC5/SDO (term 24) ----- MOSI -----> SI
// RC4/SDI (term 23) <----- MISO ----- SO
// RC3/SCK (term 18) ----- SCK -----> SCK
// RB7/CS (term 40) -----> /CS
//
// Copyright, Peter H. Anderson, Baltimore, MD, Jan, '02

#case

#device PIC18C452

#include <defs_18c.h>
#include <delay.h>
#include <ser_18c.h>

#define TRUE !0
#define FALSE 0

#define B1_WRITE 0x84
#define B2_WRITE 0x87

#define B1MEM_WT 0x83
#define B2MEM_WT 0x86
#define MEM_RD 0xd2

#define DONT_CARE 0x00

void at45_setup_SPI(void);

void at45_buffer_to_memory(byte buffer, unsigned long page);
void at45_buffer_write_seq_bytes(byte buffer, unsigned long adr, byte *write_dat,
                                byte num_bytes);
void at45_memory_read_seq_bytes(unsigned long page, unsigned long adr,
                                byte *read_dat, byte num_bytes);

void spi_io(byte *io, byte num_bytes);
void fetch_data_16bytes(byte *dat);

byte ad_buff[16], ad_buff_index;    // global variables

void main(void)
{
    unsigned long page, adr;
    byte dat[16], n, buffer;

    ser_init();
    printf(ser_char, "\r\n.....\r\n");

    at45_setup_SPI();

    not_rbpu = 0;

    if (!portb7)    // is at zero, dump the data to the terminal

```

```

{
    for (page = 0x0000; page < 0x0003; page++)
    {
        for (adr = 0x000; adr < 0x0200; adr+=16)
        {
            at45_memory_read_seq_bytes(page, adr, dat, 16);
            printf(ser_char, "%4lx:%4lx: ", page, adr);
            for (n=0; n<16; n++)
            {
                printf(ser_char, "%2x ", dat[n]);
            }
            printf(ser_char, "\r\n");
        }
    }
}

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

    adfm = 0;    // left justified
    adcs2 = 0; adcs1 = 1; adcs0 = 1; // internal RC

    adon=1;    // turn on the A/D
    chs2=0; chs1=0; chs0=0;

    // config timer 3
    t3rdl6 = 1;
    t3ckps1 = 0;    t3ckps0 = 0;    // 1:1 prescale
    tmr3cs = 0;    // internal clock - 1 usec per tick
    TMR3H = 0x00; TMR3L = 0x00;

    // assign timers
    t3ccp2 = 0;    t3ccp1 = 1; // assign timer3 to CCP2, timer1 to CCP1

    // config ccp2
    ccp2m3 = 1; ccp2m2 = 0; ccp2m1 = 1; ccp2m0 = 1;
    // special event - resets Timer 3 and initiates A/D
    CCPR2H = (byte) (1000 >> 8);
    CCPR2L = (byte) (1000);

    // turn on timers and configs interrupts
    tmr3on = 1;

    ccp2if = 0;
    ccp2ie = 1;
    peie = 1;
    gieh = 1;

    ad_buff_index = 0;

    for (page = 0x0000; page < 0x0003; page++)    // 256 * 512 bytes
    {
        if ((page%2) == 0)

```

```

    {
        buffer = 1;
    }
    else
    {
        buffer = 2;
    }

    for (adr = 0x000; adr < 0x200; adr+=16)
    {
        fetch_data_16bytes(dat);
        at45_buffer_write_seq_bytes(buffer, adr, dat, 16);
    }
    at45_buffer_to_memory(buffer, page);
}

while(gieh) // clean up
{
    gieh = 0;
}

tmr3on = 0; // turn off the timer
ccp2ie = 0;
ccp2if = 0;

printf(ser_char, "\r\nDone!!!!!!!!!!!!\r\n");
} // end of if

while(1)
{
}
}

void at45_setup_SPI(void)
{
    sslen = 0;
    sslen = 1;
    sspm3 = 0; sspm2 = 0; sspm1 = 1; sspm0 = 0;
    // Configure as SPI Master, fosc / 64
    ckp = 0; // idle state for clock is zero
    stat_cke = 1; // data transmitted on rising edge
    stat_smp = 1; // input data sampled at end of clock pulse

    latc3 = 0;
    trisc3 = 0; // SCK as output 0

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

    latb7 = 1; // CS for AT45DB321
    trisb7 = 0;
}

void at45_buffer_write_seq_bytes(byte buffer, unsigned long adr, byte *write_dat,
                                byte num_bytes)
{

```

```

byte io[20], n;

if (buffer == 1)
{
    io[0] = B1_WRITE;
}

else
{
    io[0] = B2_WRITE;
}
io[1] = DONT_CARE;
io[2] = (byte) (adr >> 8);
io[3] = (byte) adr;

for (n=0; n<num_bytes; n++)
{
    io[n+4] = write_dat[n];
}

latb7 = 0;
spi_io(io, num_bytes + 4);
latb7 = 1;
}

void at45_buffer_to_memory(byte buffer, unsigned long page)
{
    byte io[4];
    unsigned long x;

    if (buffer == 1)
    {
        io[0] = B1MEM_WT;
    }
    else
    {
        io[0] = B2MEM_WT;
    }

    x = page << 2;          // 0 Pa12, Pa11
    io[1] = (byte) (x>>8);  // high 7 bits
    io[2] = (byte) (x);
    io[3] = DONT_CARE;

    latb7 = 0;
    spi_io(io, 4);
    latb7 = 1;
}

void at45_memory_read_seq_bytes(unsigned long page, unsigned long adr,
                                byte *read_dat, byte num_bytes)
{
    byte io[25], n;
    unsigned long x;

    io[0] = MEM_RD;
    x = page << 2;          // 0 Pa12, Pa11

```

```

    io[1] = (byte) (x>>8);    // high 7 bits
    io[2] = ((byte) (x)) + ((byte) (adr >> 8) & 0x03);
    io[3] = (byte) (adr);
    io[4] = DONT_CARE;
    io[5] = DONT_CARE;
    io[6] = DONT_CARE;
    io[7] = DONT_CARE;

    latb7 = 0;
    spi_io(io, num_bytes + 8);
    latb7 = 1;

    for (n=0; n<num_bytes; n++)
    {
        read_dat[n] = ~io[n+8];
    }
}

void spi_io(byte *io, byte num_bytes)
{
    byte n;

    for(n=0; n<num_bytes; n++)
    {
        SSPBUF = io[n];
        while(!stat_bf)    /* loop */        ;
        io[n] = SSPBUF;
    }
}

void fetch_data_16bytes(byte *dat)
{
    byte n;
    while (ad_buff_index < 16)    // wait for 16 samples
    {
    }

    ad_buff_index = 0;

    for (n=0; n<16; n++)
    {
        dat[n] = ad_buff[n];
    }
}

#ifdef int_ccp2
ccp2_int_handler(void)
{
#ifdef TRIAL
    ad_buff[ad_buff_index] = 16 - ad_buff_index;    // for testing
#else
    ad_buff[ad_buff_index] = 16 - ADRESH;
#endif
    ++ad_buff_index;
}
#endif

#ifdef int_default

```

```
default_int_handler(void)
{
}
```

```
#include <delay.c>
#include <ser_18c.c>
```