

Low End Microchip PICs

C Routines

Copyright, Peter H. Anderson, Baltimore, MD, August, '01

Introduction.

This discussion focuses on Microchip's low-end PICs. It includes discussions and sample routines for the following processors;

PIC12C508/509	8-pin, 12-bit core
PIC12CE518/519	12C508/509 with 16-byte EEPROM
PIC16C505	14-pin, 12-bit core
PIC16HV540	18-pin DIP, 12-bit core, on-board regulator
PIC12C671/672	8-pin, 14-bit core, A/D
PIC12CE673/674	12C671/672 with 16-byte EEPROM

The primary significance of these processors is price and size. They range in price from \$0.90 to \$1.75 in modest 100 quantities and most have eight pins and most have reasonably accurate internal RC clocks. The 14-pin 16C505 provides a few additional IO pins. The 18-pin 16HV540 features an on-board selectable 3 or 5 VDC voltage regulator, 12 IO pins and a few other rather interesting features, but it does not have an accurate internal clock. The 14-bit core PIC12C672 provides up to four 8-bit A/D converters and with 2K of memory is considerably more powerful than the 12-bit core devices.

Although price drives hobbyists, these devices are not flash devices like the popular PIC16F84, F628 and the PIC16F87X family and with many of these powerful flash devices costing less than \$3.00, why would a hobbyist much care to go through the pain of using a windowed EEPROM to use a ultimately use a \$0.90 device. Rather, these devices are best applied to applications where either space is at a premium or the price difference of a few dollars really does make a difference. In my own case, we ship a logic "probe" with every kit we sell and a thousand units per year over a period of five years adds up to real money.

I had great reservations in undertaking this discussion for this very reason. Most people who use PICs really are not all that interested in these small devices and thus if I want to make money, it is best to write about something popular. However, I do consider this discussion as being important as for a few people, using these small PICs translates into big dollar savings.

However, for a few more, these little devices present a challenge. Fooling with a design to cram it into the mere 512 bytes of the PIC16HV540 and finally doing so with three program words to spare leaves one with a bit of satisfaction.

Debugging Tools.

When I began this effort, my feeling was that one could simply use the PIC16F87X platform and debug using an In Circuit Debugger and then carefully port the code to the target device and use a windowed EEPROM to clean up any problems. I have changed my mind, at least for the 12-bit core devices. My suggestion is an Advanced Transdata RICE17A with a PB505A Emulator probe. I continue to feel that one can use the PIC16F87X ICD to debug applications for the PIC12C672, but you will save a good deal of time with a RICE17A with a PB67X Probe.

In developing this material I used the following tools.

PIC12C509 - RF Solutions ICE-PIC with a PB5X Personality Module. I used this because I had it. My general feeling is that one can work with only an emulator for the 16C505 or the 12C509 and then quickly map their design over to the other processor and then use windowed EEPROM devices. Another alternative is to use the RICE17A with the PB505 and buy a special emulator header for the 12C509 (about \$65).

PIC12CE518/519. Routines for this PIC are confined to working with the on-board EEPROM. I used the windowed EEPROM approach.

PIC16C505 - Advanced Transdata RICE17-A with a PB-505A Probe Card.

PIC16HV540. I don't know of an emulator for this device. I used the 16C505 to develop and debug each application and then carefully mapped it over to this device using a windowed EEPROM device using the "blow and go" technique.

PIC12C671/672/CE673/CE674 – RICE17A with a PB-1267X Probe. However, I purchased this probe specifically for this discussion. I have done many designs using the PIC12C672 in the past using a PIC16F87X with an In Circuit Debugger and then carefully modified the code for the 12C672

Software.

I used the CCS PCB compiler for the 12-bit core devices (12C5XX, 16C505 and 16HV540) and the CCS PCM compiler for the 14-bit core 12C67X.

At the time this was written, CCS had released a Version 3.0 of their compiler with a new version being released every other day. I stuck with the most recent Version 2.XXX.

One note. I happened to upgrade to Version 3.XXX and was disheartened to find it simply would not run under Windows ME. I posted this to the CCS Users Forum and noted a few others in the same boat. But, the day after my neighbor ran over my buried five pair telephone cable, I was wiring this and that PC just died (not a good weekend). I did take steps to find a replacement running Windows 98 SE.

In the past I have always felt that the "bugs" associated with the CCS compiler were associated with the "frills" that I encourage you not to use. However, with the introduction of Version 3.XXX, I began to see strange things related to such fundamental things as handling a TMR interrupt and there was no way I was going to download the latest release every few days and test each and every one of the 90 routines in this discussion. You really don't want to position yourself on the job in this risky business.

My very strong suggestion is to resist the human tendency to "upgrade" during a job. Usually, an upgrade simply adds another processor or resolves some problem with a frill you are not even using and the upgrade just may introduce a bug that bites you. See the job through with the same version right to the end and if you are doing this professionally where you have to continually maintain the code, save this version that is, the SetUp.Exe file, along with your source code. Actually, I am beginning to look to an upgrade with dread rather than the breathe of fresh air we are supposed to associate with something new. This is not limited to the CCS compiler.

Upgrading to the latest version of MPLAB is probably safe. After all, MPLAB is simply calling on the CCS compiler to generate the files.

Note that the Advanced Transdata RICE17A emulator is not supported by Microchip's MPLAB. They supply their own software which I found to be quite intuitive. The package provides an integrated environment which permits editing, compiling using the CCS compiler from within the IDE and then debugging using the emulator.

The RF Solutions ICEPIC provides a package which runs within MPLAB. However, I have had problems with this in the past and continued to have problems some five years later and I used their standalone ICEPIC-32 integrated development environment which is very well done.

Presentation Technique.

In developing this material I took a very simple design definition which was presented to me a few weeks before;

A Frost Alarm. This is a simple device that gardeners or farmers might use to warn of a potential frost. The device continually monitors and displays the temperature and if this less than an alarm temperature, a sonalert is pulsed to warn the user of a potential frost condition.

The user may set the alarm threshold using a potentiometer and while setting the threshold, its value is displayed.

I decided to implement this for each of the processors with the idea of eliminating confusion in presenting many different design problems.

Note that this design involves a 1-wire interface with a Dallas DS18S20, some means of measuring the value of a potentiometer and some means of displaying either the alarm threshold or the current temperature. I found that in developing each of the modules and then combining all of the pieces to realize the final design I was able to cover most of the features of each the processors. Thus, for each processor, there are routines to measure temperature using a DS18S20, measuring the value of a potentiometer using an RC network and displaying quantities using 9600 baud serial, flashing an LED or outputting a series of tones. For some processors having adequate IO pins, programs include directly driving a Hitachi HD44780 type text LCD, sequentially outputting digits to single 7-segment LED

Additional routines were also written to illustrate various other features of each processor that did not fit into the frost alarm design.

Aside from the Dallas 1-W interface, I opted to not discuss interfacing with specific devices using the Philips I2C and Motorola SPI protocols. The reasons for using these PICs is space and cost and I'm not sure it makes all that much sense to go through the pain of using a \$0.90 PIC to interface with a \$7.00 Texas Instruments TLC2543 11 channel 12-bit A/D, \$7.00 MAX3100 SPI – UART or a \$7.00 MAX7219 Eight Digit LED Driver. One might do these jobs better and cheaper by using PICs which cost marginally more.

Routines.

All routines presented or referred to in this discussion are also contained in a .zip file which is organized with subdirectories 12C509, 16C505, 12HV540 and 12C672.

All routines were tested. However, they were developed for this educational effort and I may have failed to recognize some situation where the program will fail. That is, don't bet the farm by blindly using these and then burning 10 million PICs for an automobile application.

And, although they were tested, sometimes one manages to save an earlier copy of a file right over the top of a fully debugged file. I have a high confidence level that I was able to avoid this, but offer no guarantee.

Programming Technique.

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. Actually, one can actually get rather panicky, wondering if these folks are writing the code for a braking system you may be using in a few years.

Rather, I use a header file for the particular PIC I am using; e.g., (defs_505.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_xxx.h files were prepared from the register file map and special function register summary in the "data sheet" for each device.

Thus, there is a defs file for each device presented in this discussion.

PIC12C508/509/CE518/CE519 – defs_509.h

PIC16C505 – defs_505.h

PIC16HV540 – defs_540.h

PIC12C671/672/CE673/CE674 – defs_672.h

12-bit Core PICs.

PIC12C509.

DEFS_509.H.

The defs_509.h file is presented below followed by a discussion of some of the anomalies of 12-bit core devices.

```
// DEFS_509.H
//
// Standard definitions for 12C508, 12C509, 12CE518, 12CE519
//
// Particularly note the declaration of static int DIRS and OPTIONS
//
// Peter H. Anderson, July, '01

#define byte unsigned int

#define W 0
#define F 1

//----- Register Files -----

#define INDF      =0x00
#define TMR0     =0x01
#define PCL      =0x02
#define STATUS   =0x03
#define FSR      =0x04
#define OSCCAL   =0x05
#define GPIO     =0x06

static int DIRS, OPTIONS; // note global definitions

#define sda_in =0x06.6 // PIC12CE518/CE519 only
#define gp5    =0x06.5
#define gp4    =0x06.4
#define gp3    =0x06.3
#define gp2    =0x06.2
#define gp1    =0x06.1
#define gp0    =0x06.0

// Direction Bits
// Note that DIRS is a file containing the directions.  Actually
```

```

// setting the directions requires moving DIRS to W and then TRIS GPIO

#bit dirs5    =DIRS.5
#bit dirs4    =DIRS.4
#bit dirs3    =DIRS.3
#bit dirs2    =DIRS.2
#bit dirs1    =DIRS.1
#bit dirs0    =DIRS.0

//----- STATUS Bits -----
#bit gpwuf    =0x03.7
#bit pa0      =0x03.5
#bit not_to   =0x03.4
#bit not_pd   =0x03.3

//----- OPTION Bits -----
// OPTION Bits
// Note that PTIONS is a file containing the options.  Actually
// setting the options requires moving OPTIONS to W and then executing OPTION

#bit not_gpwu =OPTIONS.7
#bit not_gppu =OPTIONS.6
#bit t0cs     =OPTIONS.5
#bit t0se     =OPTIONS.4
#bit psa      =OPTIONS.3
#bit ps2      =OPTIONS.2
#bit ps1      =OPTIONS.1
#bit ps0      =OPTIONS.0

// for assembly language
#define SCL 7
#define SDA 6
#define GP5 5
#define GP4 4
#define GP3 3
#define GP2 2
#define GP1 1
#define GP0 0

#define Z 2
#define CY 0

```

Discussion.

Note that “byte” is defined as an unsigned int, which for the CCS compiler is eight bits. I prefer to use the type “byte”, as “char” seems confusing and “int” is ambiguous for those who wish to port the code to other compilers where an “int” may be 16 bits. Using this approach, one need only modify the definition of a “byte”.

Using the DEFS_509.H, a quantity may be output either as a byte or a bit;

```

GPIO = 0x02;
gp1 = 1;

```

Note that care must be used with the byte output (GPIO = 0x02) as all bits which are outputs will assume the defined value, in this case; 00 0010. By using the bit approach, gp1 = 1, only bit 1 of the GPIO is set.

Upper case letters are used to define each byte and lower case letters to define a bit. This requires the use of the #case directive in the program which leads to other minor problems which are discussed below.

IO Port Direction.

For the 12-bit core devices (12C5XX, 16C505, 16HV540), there is no TRIS register to define the direction of an IO bit. Thus, in the DEFS_505.H file, I have defined a global variable DIRS which one might think of as similar to the TRIS registers associated with the 14-bit core devices.

Each bit within DIRS has also be defined;

```
#bit dirs5    =DIRS.5
#bit dirs4    =DIRS.4
#bit dirs3    =DIRS.3
#bit dirs2    =DIRS.2
#bit dirs1    =DIRS.1
#bit dirs0    =DIRS.0
```

Thus, making GP1 an output is implemented;

```
        dirs1 = 0;
#asm
        MOVF DIRS, W
        TRIS GPIO
#endasm
```

OR

```
#asm
        BCF DIRS, 1           // clear bit 1
        MOVF DIRS, W
        TRIS GPIO
#endasm
```

Option Register.

Along the same lines, there is no OPTION register. Thus, in DEFS_509.H, I have defined another global variable OPTIONS and each of the bits within this register;

```
#bit not_gpwu =OPTIONS.7
#bit not_gppu =OPTIONS.6
#bit t0cs     =OPTIONS.5
#bit t0se     =OPTIONS.4
#bit psa      =OPTIONS.3
#bit ps2      =OPTIONS.2
#bit ps1      =OPTIONS.1
#bit ps0      =OPTIONS.0
```

Assume one wishes to enable the internal weak pull up resistors;

```
        not_gppu = 0;
#asm
        MOVF OPTIONS, W
        OPTION
#endasm
```

FLASH_1.C.

The following program reads GP3 and if at zero, flashes an LED on output GP1 five times, followed by a delay.

```
// FLASH_1.C (PIC12C509) CCS PCB
//
// Flashes an LED on GP1 in bursts of five flashes if input on GP3 is at logic zero.
//
// Note that DIRS and OPTIONS are defined in defs_509.h
//
```

```
//  
// GRD ----- \----- GP3 (term 4) (weak pullups enabled)  
// GP1 (term 6) ----- 330 ----->|----- GRD  
//  
//  
// copyright, Peter H. Anderson, Elmore, VT, July, '01
```

#case

#device PIC12C509 *=8

#include <defs_509.h>

```
void flash(byte num_flashes);  
void delay_10us(byte t);  
void delay_ms(long t);
```

void main(void)

```
{  
    DIRS = 0x3f;  
    dirsl = 0;    // make gp1 an output
```

```
#asm  
    MOVF DIRS, W  
    TRIS GPIO  
#endasm
```

not_gppu = 0; // enable weak pull-up resistors

```
#asm  
    MOVF OPTIONS, W  
    OPTION  
#endasm
```

```
    while(1)  
    {  
        while(gp3)  
        {  
            // loop until at logic zero  
            flash(5);  
            delay_ms(3000);  
        }  
    }
```

void flash(byte num_flashes)

```
{  
    byte n;  
    for (n=0; n<num_flashes; n++)  
    {  
        gp1 = 1;  
        delay_ms(500);  
        gp1 = 0;  
        delay_ms(500);  
    }  
}
```

void delay_10us(byte t)

```
{  
#asm  
DELAY_10US_1:  
    CLRWDT  
    NOP  
    NOP  
    NOP  
    NOP  
    NOP  
    NOP  
    DECFSZ t, F  
    GOTO DELAY_10US_1  
#endasm  
}
```

```

void delay_ms(long t)    // delays t millisecs
{
    do
    {
        delay_10us(100);
    } while(--t);
}

```

Note that bit 1 of DIRS is cleared and then “trised” as described above. The not_gppu bit of OPTIONS is similarly cleared and then moved to the W register followed by the assembly language command OPTION.

Two Level Stack.

The stack associated with the 12-bit core devices, other than the 16HV540, is limited to two levels. However, note that in the above, there appears to be a nesting level of three. main() calls flash() which calls delay_ms() which calls delay_10us(). For the C compiler, this is a no-brainer. As flash() is only called once, it is placed in-line. Similarly, in delay_ms(), delay_10us() is called only one time and thus it is also coded in-line in the delay_ms() function. Thus, the nesting level is only one.

In program FLASH_2.C (not presented in this discussion, but, in the 12C509 directory, main() was modified to call flash two times; e.g., flash(3) and flash(2), and delay_ms() was modified to call delay_10us() two times; e.g., delay_10us(50) and delay_10us(50). Thus, the compiler was forced to do something more interesting as flash(), delay_ms() and delay_10us() are all called at least twice. In fact, the compiler implemented flash() in-line, in a rather interesting fashion such the passed argument was either initialized to 3 or to 2 and delay_ms() and delay_10us() were each implemented using calls. Thus, a depth of two.

When I began this effort I had hopes of fully exploring how the compiler handled the two level stack problem in depth, but I really didn't. I was under the impression that the compiler implemented a stack in software and it may do so, but I never saw this.

My suggestion is to simply code and then compile. The compiler is well aware of the two level stack limitation and it will try to implement your code. If it runs out of program memory, it will let you know.

However, it is important to recognize that the compiler seems to have an absolute rule of coding a function in-line if it is called only one time. Later in this discussion, I illustrate how this got me into trouble and I was forced to learn and use the #separate directive.

In the final rewrite, I realize there is one issue I did not fully investigate and I don't really know when I will revisit the problem. There is a question in my mind as to whether the stack limitation is observed when one is two levels out in a function and one implements a constant array. For example;

```

byte const patts[4] = {0x03, 0x06, 0xc0, 0x09};           // full step stepping motor patterns
. . .
    GPIO = patts[n]; // output the pattern

```

Note that the constant array is implemented as a call;

```

    MOVF    N, W
    CALL LOOK
    . . .

LOOK:
    ADDWF  PCL, F
    RETLW 0x03
    RETLW 0x06
    RETLW 0x0c
    RETLW 0x09

```

I am uncertain the compiler handles this situation. That is, main() calls foo() which in turn calls bar() which uses a constant array. Unfortunately, when the CALL LOOK is executed, the return address in main() is lost.

The easiest way to possibly get into trouble is to use the innocent printf(ser_char, "Hello") in a function. Note that the printf is implemented using a constant array.

Program FLASH_3.C.

This is simply a rework of program FLASH_1.C except that the prototypes and implementations for the delay functions are in files delay.h and delay.c, respectively.

The CCS compile does not support a user library. However, I have found the ability to #include files to be adequate. One nice feature of the CCS compiler is that it compiles only those functions which are actually used. This is not true of the far more costly High Tech compiler. With the High Tech compiler, one is forced to limit the program file (and libraries) to only those functions which are actually used.

```
// FLASH_3.C (PIC12C509) CCS PCB
//
// This is a simple rework of FLASH_1.C. Files delay.h and delay.c are #included.
//
// Flashes an LED on GP1 in bursts of five flashes if input on GP3 is at logic zero.
//
// Note that DIRS and OPTIONS are defined in defs_509.h
//
//
// GRD ----- \----- GP3 (term 4) (weak pullups enabled)
//                                     GP1 (term 6) ----->|----- GRD
//
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

#case

#device PIC12C509 *=8

#include <defs_509.h>
#include <delay.h>

void flash(byte num_flashes);

void main(void)
{
    DIRS = 0x3f;
    dirsl = 0;      // make gp1 an output
#asm
    MOVF DIRS, W
    TRIS GPIO
#endasm

    not_gppu = 0;
#asm
    MOVF OPTIONS, W
    OPTION
#endasm

    while(1)
    {
        while(gp3)
        {
            // loop until at logic zero
            flash(5);
            delay_ms(3000);
        }
    }
}
```

```

void flash(byte num_flashes)
{
    byte n;
    for (n=0; n<num_flashes; n++)
    {
        gp1 = 1;
        delay_ms(500);
        gp1 = 0;
        delay_ms(500);
    }
}

#include <delay.c>

```

Serial Routines (SER_509.H and SER_509.C).

None of the processors in this discussion have a UART and thus one is forced to implement the routines using the “bit bang” technique where the bit timing is implemented by carefully adjusting the number of executed instructions.

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

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

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

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

Thus, for true RS232, the TTL idle level is a logic one. A character is then sent, starting with the start bit, a logic zero, and then each bit is sent at its true logic levels, beginning with the least significant bit and then back to the idle logic one level. For 9600 baud, each bit time is 1/9600 or 104 us.

For inverted RS232, the idle level is a TTL logic zero. A character is sent, beginning with the start bit, a TTL logic one, and each bit is then sent at its inverted logic level and then back to the idle zero condition.

File SER_509.C contains the implementations of low level routines; ser_init() and ser_char() for both true and inverted applications. Note that if inverted is desired, the using program must define INV and also define which IO is being used, TxData. That is;

```

#define INV                // use inverted logic
#define TxData 0          // use GP0

```

Routine ser_init() is implemented;

```
void ser_init(void) // sets TxData in idle state
{
#ifdef INV
#asm
    BCF GPIO, TxData // idle at logic zero
    BCF DIRS, TxData
    MOVF DIRS, W
    TRIS GPIO
#endasm
#else
#asm
    BSF GPIO, TxData // idle at logic 1
    BCF DIRS, TxData
    MOVF DIRS, W
    TRIS GPIO
#endasm
#endif
    ser_char(0x0c); // for PIC-n-LCD from BG Micro <<<<<<< see text
    delay_ms(250);
}
```

Routine ser_char() is used to send a byte ch. The CY bit in the STATUS register is first set to zero (the start bit). If the logic levels are defined as inverted, the CY bit is tested and the opposite state is output on TxData. The eight data bits are sent by successively rotating the variable ser_char_ch to the right with the least significant bit being shifted into the CY bit and the opposite state is output. Finally, the TxData output is returned to the idle state, a TTL logic zero.

The same technique is used if the logic levels are not defined as inverted, except that the same state of the CY bit is output and finally, the output is returned to the TTL logic one idle state.

Not that the number of instructions from the point that SER_CHAR_1 is entered to the next time it is entered is 104 (104 us when using a 4.0 MHz clock).

```
void ser_char(byte ch) // serial output 9600 baud
{
    //byte n, dly;
    ser_char_ch = ch; // copy to global
    // start bit + 8 data bits
#ifdef INV
#asm
    MOVLW 9
    MOVWF ser_char_n
    BCF STATUS, CY
SER_CHAR_1:
    BTFSS STATUS, CY
    BSF GPIO, TxData
    BTFSC STATUS, CY
    BCF GPIO, TxData
    MOVLW 32
    MOVWF ser_char_dly
SER_CHAR_2:
    DECFSZ ser_char_dly, F
    GOTO SER_CHAR_2
    RRF ser_char_ch, F
    DECFSZ ser_char_n, F
    GOTO SER_CHAR_1

    BCF GPIO, TxData
    CLRWDT
    MOVLW 96
    MOVWF ser_char_dly
#endif
}
```

```

SER_CHAR_3:
    DECFSZ ser_char_dly, F
    GOTO SER_CHAR_3
    CLRWDT
#endasm

#else // true

#asm
    MOVLW 9
    MOVWF n
    BCF STATUS, CY

SER_CHAR_1:

    BTFSS STATUS, CY
    BCF GPIO, TxData
    BTFSC STATUS, CY
    BSF GPIO, TxData
    MOVLW 32
    MOVWF dly

SER_CHAR_2:
    DECFSZ dly, F
    GOTO SER_CHAR_2
    RRF ch, F
    DECFSZ n, F
    GOTO SER_CHAR_1

    BSF GPIO, TxData
    CLRWDT
    MOVLW 96
    MOVWF dly

SER_CHAR_3:
    DECFSZ dly, F
    GOTO SER_CHAR_3
    CLRWDT
#endasm
#endif
}

```

Note that variables `ser_char_ch`, `ser_char_dly` and `ser_char_n` are defined globally to force them to be located in RAM bank 0. The reason is that the compiler handles variables in the higher banks somewhat differently which results in additional instructions being added to the code which changes the critical 104 us timing. This is discussed below.

Indirect Addressing.

With the 12C509, general purpose RAM memory is at addresses 0x07 – 0x0f and 0x10 – 0x1f in Bank 0 and at addresses 0x30 – 0x3f in Bank 1.

In handling RAM in the higher bank, the CCS Compiler uses the FSR and INDF SFRs to implement indirect addressing. Examples of the use of the FSR and INDF registers are illustrated below.

Note that the lowest five bits (0 – 4) of the FSR register are used to address RAM within one bank. The bank bit (bit 5) is used to address the bank.

```

    MOVLW 0x10
    MOVWF FSR           // FSR now points to location 0x10

    MOVLW 0x3E
    BSF FSR, 5         // switch to bank 1
    MOVF INDF          // location 0x30 now contains 0x3e
    BCF FSR, 5         // back to bank 0

```

```

. . .
BSF FSR, 5
DECF INDF, F // value in location pointed to by FSR is decremented
BCF FSR, 5

```

The CCS compiler defaults to using only Bank 0. Use of the #device *=8 is required to access other banks.

Thus;

```
#device PIC12C509 *=8
```

The problem that I had when my serial routines used local variables was that on larger programs, the local variables, ch, dly and n were being assigned to bank 1 and thus the compiler was using indirect addressing to implement the time critical task of sending a character. The result is that the setting and clearing of the bank bits in the FSR register caused my carefully calculated 104 us timing to be a bit longer and ser_char() did not work.

My solution, and there may be better solutions, was to use global variables. The CCS compiler seems to assign variables as they appear and thus globals are assigned first and this will in all likelihood be in the range of 0x07 – 0x1f (RAM bank 0).

As I write this, I realize that this observation is probably also true of my timing routines delay_10us() and delay_ms(). I do recall one application where there was a five second delay which seemed to be more like seven seconds and although I make no claim as to the precision of these routines, common sense suggests they just are not this inaccurate. My guess is that the local variables were being assigned to a RAM bank other than bank 0 and thus additional instructions were being added to my 10 us timing loop. However, at this juncture, I am not going to return to the some 90 routines I have written and correct this by using global variables. Rather, I leave it to you to either use global variables or adjust the timing to meet your requirement. That is, if a delay_ms(5000) is resulting in a 7000 ms delay, scale back on the 5000 or modify the implementation of delay_ms().

Program TST_SER.C (12C509).

Program TST_SER.C is intended to illustrate the use of the various routines in SER_509.C.

Noteworthy points include;

1. Use of the *=8 to permit the compiler to use the higher RAM banks. (Only bank 1 in the case of the 12C509).
2. The header file string.h is included to implement the strcpy function. Note that in string.h, CCS carelessly refers to an “isalpha()” routine using uppercase “ISALPHA()”. In using the #case directive, this will cause an error as the compiler has no idea of what ISALPHA() is. If and when you get this error, take a good look and simply modify the CCS routine to use lower case letters.
3. Header file SER_509.H includes the prototypes for the functions in SER_509.C and also declares global variables ser_char_ch, ser_char_dly and ser_char.
4. The pin used for TxData and whether the serial in to be inverted is #defined in the using routine, in this case, TST_SER.C.

The routine is intended to illustrate how to use strcpy and such functions as ser_init(), ser_char(), ser_out_str(), ser_hex_byte(), ser_dec_byte() and ser_new_line().

```

// TST_SER.C (PIC12C509), CCS PCB
//
//
// Illustrates the use of various serial output functions contained in "ser_509.c".

```

```

//
// PIC12C509
//
// GP0 (term 7) ----- Serial LCD or PC Com Port
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

#case

#device PIC12C509 *=8

#include <defs_509.h>
#include <string.h> // for strcpy

#include <delay.h>
#include <ser_509.h>

#define TxData 0 // use GP0
#define INV // send inverted RS232

void main(void)
{
    byte s[8], n;
    n = 150;

    while(1)
    {
        DIRS=0x3f;

        ser_init();
        strcpy(s, "Morgan");
        // note that CONST string is copied to RAM string
        ser_out_str(s);
        strcpy(s, " State");
        ser_out_str(s);
        ser_new_line();
        strcpy(s, "Univer");
        ser_out_str(s);
        strcpy(s, "sity");
        ser_out_str(s);
        ser_new_line();
        ser_hex_byte(n);
        ser_char(' ');
        ser_dec_byte(n, 3); // display in dec, three places

        delay_ms(500);
        ++n;
    }
}

#include <delay.c>
#include <ser_509.c>

```

Serial – Other Points.

In debugging these routines, I used the “PIC-n-LCD” serial LCD from B. G. Micro and thus, there are aspects of the serial routines that were tailored for this unit. You will probably have to modify some aspects for operation with whatever RS232 text display you are using

In `ser_init()`, the control character 0x0c is sent to the “PIC-n-LCD”, followed by a substantial delay. The control character 0x0c is used to clear the LCD and place the cursor in the upper left position. In `ser_new_line()`, the control characters for CR and LF are sent, with a 10 ms delay between each as the implementation of these commands by the processor in the PIC-n-LCD unit may involve taking what is currently on line 1 and moving it to line 0, line 2 to 1 and line 3 to 2 and positioning the cursor at the beginning of line 3 and this takes time. In `ser_char()`, a five ms delay is provided as I have found I get framing errors when providing little or no delay between characters.

OSCCAL and the Internal RC Clock.

On boot, the PIC12C508 (509) begins execution at the highest memory location; 0x1ff (or 0x3ff) where Microchip has implemented an instruction of the form;

```
MOVLW xxxxxxxx
```

Where xxxxxxxx is a calibration constant. The program counter then wraps to location 0x000 and the CCS compiler automatically uses the command;

```
MOVWF OSCCAL
```

Thus, Microchip and CCS have done all of the work in calibrating the clock for you. At least with development one time programmable (OTP) parts. Windowed EEPROMs are discussed below.

The question then arises, is the internal clock sufficiently accurate to assure the asynchronous transfer of data . The answer is a nutshell is, "I don't know". This is one of those many things where it is not valid to assume that if it works okay in the lab, it will work for every unit in the field. My main concern is the drift with temperature. I have fielded many kits using the 12C508 and have not cared to take the chance and have always used an external 4.0 MHz resonator. But, I did field a design which used the internal clock on the PIC12C671 with the general idea that I could always pull the kit off the market if this proved to be a problem and after fielding many hundreds, I have yet to have a complaint. But, note that in my case, I really had no appreciable money nor moral qualms riding on my decision to use the internal clock.

My intuition is that if there is money or liability riding on the decision, go with an accurate external clock. If there are not a sufficient number of pins, go with the PIC12C505 in place of the 12C508 or 509.

Note that in developing all of the routines in this discussion, I used the internal clock. The one exception is the 12HV540 which does not have an internal RC clock.

Internal Clock Calibration Constant for Windowed EEPROM Parts.

After erasing a windowed EEPROM, all of the program memory will be at 0xffff which is the op code for XORLW 0xff. Thus, if a previously erased EEPROM is used, on boot, the PIC will begin at the highest address, take the unknown state in W and invert it and then wrap to program memory location 0x000 and this unknown value will be written to the OSCCAL register. At least the program will run, which is not true of the 14-bit core PIC12C672, but the value of the calibration constant has been lost.

I might suggest that prior to using a windowed EEPROM part, you use a programmer to read the highest word in program memory. It will be of the form;

```
1100 kkkk kkkk
```

where 1100 is the op code for MOVLW and kkkk kkkk is the calibration constant.

For example, this might appear in hexadecimal format as;

```
0xC2D
```

Note that 0x2D is the calibration constant which you might write on the bottom of the PIC.

When working with the windowed part, you may insert code near to beginning of main() to write this value to the OSCCAL register.

```
#ifdef _EEPROM
    OSCCAL = CAL_CONSTANT;
#endif
```

where _EEPROM is #defined and CAL_CONSTANT is #defined as 0x2D. When ready for production, simply undefined _EEPROM.

Note then, that the CCS compiler is writing the garbage value to the OSCCAL register and you are following up with writing the correct value.

In developing routines for the 12C509 and 16C505, I was working with an emulator which had a very accurate internal clock which was not affected by OSCCAL and thus I did not include this type of code in the routines.

Program FLSH_Q_1.C (PIC12C509).

In realizing the “Frost Alarm”, it probably isn’t realistic to expect the gardener or farmer to have either a PIC-n-LCD or to cable up a PC to view the alarm setting and the current temperature and with the limited number of pins on the 12C509 one is limited to interfacing with the user using either blips of an LED or using beeps of a speaker.

In program FLSH_Q_1.C, a bi-color LED is used on outputs GP5 and GP4 to display either the alarm setting or the current temperature.

When the user depresses a pushbutton switch on input GP3, the routine continually displays the alarm setting on the ALM LED. The idea here is that the user might be turning a potentiometer to set the alarm threshold.

When the pushbutton is released, the program continually displays the temperature on the TEMP LED.

Some notes;

1. In this routine, the alarm threshold was set at 34. The various temperatures were dummied up in a constant array such that I could test a number of situations’ e.g., a negative value, zero, a single digit, and a two digit quantity of the form 10 or 20 where the LED must be flashed ten times to represent the trailing zero.
2. Note that the quantity is tested as to whether it is negative by ascertaining if the most significant bit is a logic one (T_C & 0x80) and if so, the two’s complement operation is performed. The minus sign is “displayed” as a long flash. Thus, -5 is displayed as “long”, “blip”, “blip”, “blip”, “blip”, “blip”.
3. Note that the pin number of the ALM and TEMP LEDs is #defined as a number; 4, or 5 and this value is passed to flash_q. I don’t know of a simple technique in C, short of constant arrays, which aren’t all that simple, to specify a bit position as a variable. My point is that it pays to know a bit of trivial assembly as such commands as the following makes this very easy and are clearly very compact.

```
BCF GPIO, LED_ALM
BCF DIRS, LED_ALM
MOVF DIRS, W
TRIS GPIO
```

```
// FLSH_Q.C (PIC12C509), CCS PCB
//
// When input GP3 is at ground, T_threshold is output to Bicolor LED on GP5.
// When input GP3 is not at ground, the current value of T_C is output to
// the bicolor LED on GP4. Note that readings are dummied in a constant array.
//
// In outputting the quantity, a long flash indicates a minus. Each digit is
```

```

// output as a series of 250 ms flashes. Interdigit time of 1 sec. Five seconds
// between outputting each quantity.
//
//
// PIC12C509
//
// GRD ---- \----- GP3 (term 4)
//
//
// Bicolor LED
// GP4 (term 3) ----->|----|----- 330 ---- GRD
// GP5 (term 2) ----->|----|
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

#case

#device PIC12C509

#include <defs_509.h>
#include <delay.h>

#define TRUE !0
#define FALSE 0

#define LED_ALM 5 // use GP5 and GP4
#define LED_TEMP 4

void flash_q(byte LED, byte q, byte minus_flag);

void main(void)
{
    byte T_threshold = 34, T_C, minus_flag, n;
    char const T_C_array[5] = {-5, 0, 1, 25, 70};

    DIRS = 0x3f;

#asm
    BCF GPIO, LED_ALM // make LED pins output logic zeros
    BCF GPIO, LED_TEMP
    BCF DIRS, LED_ALM
    BCF DIRS, LED_TEMP
    MOVF DIRS, W
    TRIS GPIO
#endasm

    not_gppu = 0; // enable internal weak pull-ups
#asm
    MOVF OPTIONS, W
    OPTION
#endasm

    while(1)
    {
        if(!gp3) // if switch at ground
        {
            flash_q(LED_ALM, T_threshold, FALSE); // display the T_thresh on LED on GP5
            delay_ms(5000);
        }

        else
        {
            for (n = 0; n < 5; n++)
            {
                T_C = T_C_array[n];
                if (T_C & 0x80) // if its negative
                {
                    minus_flag = TRUE;
                    T_C = (~T_C) + 1; // 2's comp
                }
            }
            else

```

```

        {
            minus_flag = FALSE;
        }
        flash_q(LED_TEMP, T_C, minus_flag); // display on the LED on GP4
    }
}
delay_ms(5000);
}
}

```

```

void flash_q(byte LED, byte q, byte minus_flag)
{
    byte n, digit;

```

```

#asm
    BCF GPIO, LED_ALM    // make LED pins output logic zeros
    BCF GPIO, LED_TEMP
    BCF DIRS, LED_ALM
    BCF DIRS, LED_TEMP
    MOVF DIRS, W
    TRIS GPIO
#endasm

```

```

    if (minus_flag)
    {
        if (LED == LED_ALM)
        {

```

```

#asm
        BSF GPIO, LED_ALM
#endasm

```

```

        }
        else
        {

```

```

#asm
            BSF GPIO, LED_TEMP
#endasm

```

```

        }

```

```

        delay_ms(500); // long delay to indicate minus

```

```

#asm
        BCF GPIO, LED_ALM
        BCF GPIO, LED_TEMP
#endasm

```

```

        delay_ms(1000);
    }

```

```

    digit = q/10; // number of tens
    if (digit)    // if non zero
    {
        for (n=0; n<digit; n++)
        {

```

```

            if (LED == LED_ALM)
            {

```

```

#asm
                BSF GPIO, LED_ALM
#endasm

```

```

            }
            else
            {

```

```

#asm
                BSF GPIO, LED_TEMP
#endasm

```

```

            }
            delay_ms(250); // long delay to indicate minus

```

```

#asm
                BCF GPIO, LED_ALM
                BCF GPIO, LED_TEMP
#endasm

```

```

            delay_ms(250);
        }
    }
}

```

```

    delay_ms(1000); // separation between digits
}

digit = q%10;
if (!digit) // if its zero, make it ten
{
    digit = 10;
}

for (n=0; n<digit; n++)
{
    if (LED == LED_ALM)
    {
#asm
        BSF GPIO, LED_ALM
#endasm
    }
    else
    {
#asm
        BSF GPIO, LED_TEMP
#endasm
    }
    delay_ms(250); // long delay to indicate minus
#asm
    BCF GPIO, LED_ALM
    BCF GPIO, LED_TEMP
#endasm
    delay_ms(250);
}

    delay_ms(1000); // separation between digits
}

#include <delay.c>

```

Program TONE_Q.C (PIC12C509).

This program is functionally similar to the FLSH_Q routine except that it uses beeps on a speaker rather than flashes of an LED to “display” a quantity. Only portions of the routine are presented below.

In fact, the beep could have been simply implemented as;

```

void beep(long ms)
{
    long n;
    for (n = 0; n < ms/2; n++)
    {
        gp0 = 1;
        delay_ms(1);
        gp0 = 1;
        delay_ms(1);
    }
}

```

However, my desire was to illustrate the use of TMR0. Thus, in function beep(), TMR0 is configured with the clock source being the system $f_{osc} / 4$ clock (1.0 MHz) with the prescaler set to 1:4 and assigned to TMR0. Thus, TMR() is incremented each 4 us and rolls over every 1.024 ms which was close enough to 1.0 ms for me.

The 12-bit core devices have no interrupt capability and thus the program must constantly monitor TMR0 to ascertain if there was a rollover.

```

while(ms)
{

```

```

    tmr0_new = TMR0;
    if ((tmr0_new < 0x80) && (tmr0_old >= 0x80)) // if there was a rollover
    {
        gp0 = !gp0;        // toggle speaker output
        --ms;
    }
    tmr0_old = tmr0_new;
}

```

As I now review this, I can see that one might modify the timing by adding an offset to TMR0 in the if statement as shown below.

```

while(n)
{
    tmr0_new = TMR0;
    if ((tmr0_new < 0x80) && (tmr0_old >= 0x80)) // if there was a rollover
    {
        TMR0 = TMR0 + 56    // <<<<<<<<<<
        gp0 = !gp0;        // toggle speaker output
        --n;
    }
    tmr0_old = tmr0_new + 56;    // <<<<<<<<<<
}

```

Note that in adding 56, the periodicity of TMR0 is now 200 counts time 4 us or 800 us. One might use this to generate different tones, in this case, one for the display of the alarm temperature threshold and another for the current temperature. Or, one might use this to play a simple melody. I did not try this and in all honesty it is not high on my list of things to do.

```

// TONE_Q.C (PIC16C505), CCS PCB
//
// Intended for possible use with frost alarm in place of serial output
// to serial LCD or to PC Com Port.
//
// When input GP3 is at ground, T_threshold is sounded on speaker on output
// GP0. When input GP3 is not at ground, the current value of T_C is output on
// the speaker.
//
// In sounding the quantity, a long 500 Hz tone indicates a minus. Each digit is
// sounded as a series of 250 ms beeps with an interdigit delay of 1 second.
//
//
// GRD --- \---- GP3 (internal weak pull-up) (term 4)
//
//           GP0 (term 7)  -----||--- SPKR --- GRD
//                               + 47 uFd
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

```

```

#case

#device PIC12C509

#include <defs_509.h>
#include <delay.h>

#define TRUE !0
#define FALSE 0

void beep(long ms);
void beep_q(byte q, byte minus_flag);

void main(void)
{
    byte T_threshold = 34, T_C, minus_flag, n;
    byte const T_C_array[5] = {-5, 0, 1, 25, 70}; // some dummy temperatures

    DIRS = 0x3f;

```

```

    not_gppu = 0;
#asm
    MOVF OPTIONS, W
    OPTION
#endasm

while(1)
{
    if(!gp3) // if switch at ground
    {
        beep_q(T_threshold, FALSE);
        delay_ms(5000);
    }

    else
    {
        for (n = 0; n < 5; n++) // beep each temperature
        {
            T_C = T_C_array[n];
            if (T_C & 0x80) // its negative
            {
                minus_flag = TRUE;
                T_C = (~T_C) + 1;
            }
            else
            {
                minus_flag = FALSE;
            }
            beep_q(T_C, minus_flag);
            delay_ms(1000);
        }
        delay_ms(5000);
    }
}

void beep_q(byte q, byte minus_flag)
{
    byte n, digit;

    if (minus_flag)
    {
        beep(500);
        delay_ms(1000); // long delay to indicate minus
    }

    digit = q/10; // number of tens
    if (digit) // if non zero
    {
        for (n=0; n<digit; n++)
        {
            beep(250);
            delay_ms(250);
        }

        delay_ms(1000); // separation between digits
    }

    digit = q%10;
    if (!digit)
    {
        digit = 10;
    }

    for (n=0; n<digit; n++)
    {
        beep(250);
        delay_ms(250);
    }
}

```

```

    delay_ms(1000); // separation between digits
}

void beep(long ms)
{
    byte tmr0_old, tmr0_new;

    gp0 = 0;
    dirs0 = 0;
#asm
    MOVF DIRS, W
    TRIS GPIO
#endasm

    t0cs = 0;    // internal fosc / 4
    psa = 0;
    ps2 = 0;    // prescale of 1:4, thus rollover every ms
    ps1 = 0;
    ps0 = 1;

#asm
    MOVF OPTIONS, W
    OPTION
#endasm
    TMR0 = 0x00;
    tmr0_old = 0;

    while(ms)
    {
        tmr0_new = TMR0;
        if ((tmr0_new < 0x80) && (tmr0_old >= 0x80)) // if there was a rollover
        {
            gp0 = !gp0;    // toggle speaker output
            --ms;
        }
        tmr0_old = tmr0_new;
    }
    gp0 = 0;
}

#include <delay.c>

```

Program RCTIME.C (PIC12C509).

This program measures the discharge time of an external RC network and might be used to measure R or C. A diagram appears in the program description.

The capacitor is charged to near +5 VDC through a 330 Ohm current limiting resistor using output GP2. GP2 is then made a high impedance input and the time is measured from the time the capacitor begins discharge until it is seen by input GP2 as a logic zero.

Assuming the voltage is nominally +5 VDC this time is;

$$t_{rc} = RC * \ln (5.0 / (5.0 - V_{thresh}))$$

where V_{thresh} is the point at which an input is seen as a logic zero.

For example, if V_{thresh} is 2.5 VDC

$$t_{rc} = RC * \ln(2) \text{ or } RC * 0.693$$

However, if V_thresh is 1.5 VDC

$$t_{rc} = RC * \ln(3.333) \text{ or } RC * 1.20$$

Thus, it is clear that the time can vary wildly depending on the V_threshold. My own finding is that V_threshold is close to 1.5 VDC, but this is another one of those areas where what is in the lab can't be guaranteed in the field where devices vary from one to the next.

Thus, for the network I used, when the potentiometer is at zero,

$$t_{rc} = 10K * 1.0 \text{ uFd} * 1.20 \text{ or } 12,000 \text{ us or nominally } 0x3000 \text{ us}$$

When the pot is at the maximum;

$$t_{rc} = 20K * 1.0 \text{ uFd} * 1.2 \text{ or } 24,000 \text{ us or } 0x6000 \text{ us}$$

On reflection, given the time, I would rework this such that the series resistance was say 1K and thus the variation of the potentiometer from 0 to 10K produced a far greater swing.

On another note, it would have been better to continually maintain the +5 VDC on GP2, that is, maintain a charge on the external electrolytic. When electrolytic capacitors are continually charged, the actual capacitance is close to the specification and it will have a longer life than if it is left to discharge for hours or days and then suddenly hit with a voltage. After a capacitor has been discharged for a day, the electrolyte will degrade and the effective capacitance may be half or less of the specified value. A design tidbit I know well; when using an electrolytic, keep it charged, but I overlooked it when developing the routines for this discussion.

However, right now, I do not have the time to rework the material.

In the routine, TMR0 is configured for the $f_{osc} / 4$ (1.0 MHz) and the prescaler is assigned to the watch dog timer and thus, TMR0 increments with each clock pulse. Thus, the periodicity of TMR0 is 256 us.

```
t0cs = 0; // fosc / 4 is clock source
psa = 1; // prescale assigned to WDT
ps2 = 0; // 1:1 prescale
ps1 = 0;
ps0 = 0;
```

```
#asm
MOVF OPTIONS, W
OPTION
```

In measuring the time for the RC network to decay such that a logic zero is seen by GP2, bytes counter_hi and counter_lo are initialized to 0x00. Each time there is a rollover variable counter_hi is incremented. If and when the voltage decays to the point it is seen as a logic zero, the residual value of TMR0 is copied to counter_lo and the RC time is the value of counter_hi and counter_lo. However, if counter_hi rolls over as would be the case if the voltage did not decay within nominally 65 ms, the program breaks with both counter_hi and counter_lo at 0xff to signal to the calling routine that the value has no meaning.

```
while(1) // wait for cap to discharge
{
    tmr0_new = TMR0;
    if ((tmr0_new < 0x80) && (tmr0_old >= 0x80)) // there was a roll over
    {
        ++count_hi;
        if (count_hi == 0) // no zero crossing with 65 ms
        {
```

```

        count_hi = 0xff;
        count_lo = 0xff;
        break;
    }
}
if (!gp2) // if capacitor discharged below zero crossing
{
    count_lo = tmr0_new;
    break;
}
tmr0_old = tmr0_new;
}

```

In this program, RC times are continually measured and the result is displayed in hexadecimal format on a serial LCD or PC COM port.

Note that the intent of this program in the context of the “frost alarm” is to use the potentiometer to permit the user to set the alarm threshold temperature. That is, the RCTime is mapped into a temperature alarm value in the range of 34 to 44 degrees.

```

// RCTIME.C (PIC12C509), CCS-PCB
//
// Charges capacitor in parallel with a resistor on GP2 for one second.
// GP2 is then made an input and capacitor discharges through capacitor and
// and the time for detection of a one to zero transition is measured.
//
// Result in number of 1 usec ticks is displayed in hex on serial LCD
// on GP0.
//
// Illustrates use of TMR0
//
//.
// GP2 (term 5) --- 330 -----
//
//                |           |
//                1.0 uFd     10K Pot
//                |           |
//                |           10K Resistor
//                |           |
//                GRD         GRD
//
// GP0 (term 7) ----- To Serial LCD or PC COM Port
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

```

```

#case

#device PIC12C509  *=8

#include <defs_509.h>
#include <delay.h>
#include <ser_509.h>

#define TxData 0
#define INV

void main(void)
{
    byte count_hi, count_lo, tmr0_old, tmr0_new;

    DIRS = 0x3f;
    ser_init();

    while(1)
    {

        t0cs = 0; // fosc / 4 is clock source
        psa = 1; // prescale assigned to WDT
    }
}

```

```

    ps2 = 0; // 1:1 prescale
    ps1 = 0;
    ps0 = 0;

#asm
    MOVF OPTIONS, W
    OPTION
#endasm

    dir2 = 0; // output
#asm
    MOVF DIRS, W
    TRIS GPIO
#endasm
    gp2 = 1; // charge capacitor

    delay_ms(10);

    count_hi = 0;
    count_lo = 0;
    tmr0_old = 0x00;
    TMR0 = 0x00;
#asm
    BSF DIRS, 2
    MOVF DIRS, W
    TRIS GPIO
#endasm
while(1) // wait for cap to discharge
{
    tmr0_new = TMR0;
    if ((tmr0_new < 0x80) && (tmr0_old >= 0x80)) // there was a roll over
    {
        ++count_hi;
        if (count_hi == 0) // no zero crossing with 65 ms
        {
            count_hi = 0xff;
            count_lo = 0xff;
            break;
        }
    }
    if (!gp2) // if capacitor discharged below zero crossing
    {
        count_lo = tmr0_new;
        break;
    }
    tmr0_old = tmr0_new;
}
ser_init();
ser_hex_byte(count_hi);
ser_hex_byte(count_lo);
delay_ms(1000);
}
}

#include <delay.c>
#include <ser_509.c>

```

Program DS1820.C (PIC12C509).

This program illustrates an interface with two Dallas DS18S20 1-wire thermometers.

A detailed discussion of the 1-W interface appears in my “C Routines for the PIC16F87X”. However, the implementation is briefly presented below.

Low Level Routines.

A communication session is initiated by the PIC bringing the DQ lead low for 500 us and then back to a high impedance. Note that in this implementation, I did not look for the presence pulse generated by the DS1820 shortly after DQ is brought to a high Z.

```
#asm
    BSF DIRS, 1 // high impedance
    MOVF DIRS, W
    TRIS GPIO

    BCF GPIO, 1 // bring DQ low for 500 usecs
    BCF DIRS, 1
    MOVF DIRS, W
    TRIS GPIO
#endasm
delay_10us(50);
#asm
    BSF DIRS, 1 // high Z
    MOVF DIRS, W
    TRIS GPIO
#endasm
delay_10us(50);
```

A logic one is sent to 1-W device by briefly winking the DQ lead low and then back to a high Z such that when the DS1820 reads the DQ lead some 15 us later, it sees the +5 VDC through the 4.7K resistor and interprets this as a logic one.

A logic zero is sent by bringing the DQ lead low for 60 us such that when the DS1820 reads the line some 15 us after the negative transition, it sees a logic zero.

```
    if (d&0x01)
    {
#asm
        // send a 1
        BCF GPIO, 1 // wink low and high and wait 60 usecs
        BCF DIRS, 1
        MOVF DIRS, W
        TRIS GPIO

        BSF DIRS, 1
        MOVF DIRS, W
        TRIS GPIO
#endasm
        delay_10us(6);
    }
    else
    {
#asm
        // send a zero
        BCF GPIO, 1 // bring low, 60 usecs and bring high
        BCF DIRS, 1
        MOVF DIRS, W
        TRIS GPIO
#endasm
        delay_10us(6);
#asm
        BSF DIRS, 1
        MOVF DIRS, W
        TRIS GPIO
#endasm
    }
```

A byte is transmitted starting with the least significant bit. The least significant bit of the byte is tested and the appropriate state as shown above is sent and the byte is shifted to the right such that the next bit is in the least significant bit position. This is repeated for each of the eight bits.

```

void _lw_out_byte(byte sensor, byte d)
{
    byte n;
    if (sensor==1)
    {
        for(n=0; n<8; n++)          // for each of the eight bits
        {
            if (d&0x01)            // test the least significant bit
            {
#asm
                BCF GPIO, 1 // wink low and high and wait 60 usecs
                BCF DIRS, 1
                MOVF DIRS, W
                TRIS GPIO

                BSF DIRS, 1
                MOVF DIRS, W
                TRIS GPIO
#endasm
                delay_10us(6);
            }
            else
            {
#asm
                BCF GPIO, 1 // bring low, 60 usecs and bring high
                BCF DIRS, 1
                MOVF DIRS, W
                TRIS GPIO
#endasm
                delay_10us(6);
#asm
                BSF DIRS, 1
                MOVF DIRS, W
                TRIS GPIO
#endasm
            }
            d=d>>1;                // next bit to least significant bit position
        } // end of for
    }
    else // sensor 2

```

Bits are read from the external 1-W device by the processor momentarily winking the DQ lead low and then reading the state of the DQ lead nominally 10 us later. A byte is reading by repeating this process for each of the eight bits. Note that the DS18S20 sends the data beginning with the least significant bit. Thus, as each bit is read, the result is shifted to the right and the bit is inserted into the most significant bit position. Thus, after reading eight bits, the first bit received is in the least significant bit position.

```

byte _lw_in_byte(byte sensor)
{
    byte n, i_byte, temp;

    for (n=0; n<8; n++)
    {
        if (sensor==1)
        {
#asm
            BCF GPIO, 1 // wink low and read
            BCF DIRS, 1
            MOVF DIRS, W
            TRIS GPIO

```

```

        BSF DIRS, 1
        MOVF DIRS, W
        TRIS GPIO

        CLRWDT
        NOP
        NOP
        NOP
        NOP
#endasm
    temp=GPIO; // now read
    if (temp & 0x02) // if GP1
    {
        i_byte=(i_byte>>1) | 0x80; // least sig bit first
    }
    else
    {
        i_byte=i_byte >> 1;
    }
    delay_10us(6);
}
else

```

When performing a temperature conversion (or writing to the DS18S20's EEPROM), the DS1820 requires more current than is available from the +5 VDC through the 4.7K pull-up resistor. Function `_1_w_strong_pullup` brings the DQ lead to a hard logic one (+5VDC) for 750 ms.

```

void _1w_strong_pull_up(byte sensor) // bring DQ to strong +5VDC
{
    if (sensor ==1)
    {
#asm
        BSF GPIO, 1 // output a hard logic one
        BCF DIRS, 1
        MOVF DIRS, W
        TRIS GPIO
#endasm

        delay_ms(750);
#asm
        BSF DIRS, 1
        MOVF DIRS, W
        TRIS GPIO
#endasm
    }
    else

```

High Level Commands.

Most 1-wire devices manufactured by Dallas provide a 64-bit unique serial number which permits the interfacing processor to specifically address a unique device on the bus. The nature of the commands dealing with this "ROM" address are;

Read "ROM". This operation might be performed to determine the 64-bit serial number when there is only one device on the bus. Match "ROM". This command is followed by the unique 64-bit serial number and then the command as to what operation is to be performed. Search "ROM". This provides for an algorithmic search for each 64-bit address of every device on the bus.

And, finally, the simplest, Skip "ROM". The interpretation is that there is only one device on the bus, and thus, there is no need to specifically address it. The skip "ROM" command code is 0xcc.

Thus, in performing a temperature measurement, the skip ROM command is sent followed by the command to begin a temperature measurement 0x44. This is followed by 750 ms of strong pull-up to provide adequate current for the measurement. The temperature is then read with the skip ROM command followed by the read temperature command 0xbb. Bytes are then read from the DS18S20.

In fact, nine bytes are returned to the processor. In the following, I read only the first two. The first is the T_C times two and the second is the sign.

```

    _lw_init(sensor);
    _lw_out_byte(sensor, 0xcc); // skip ROM

    _lw_out_byte(sensor, 0x44); // perform temperature conversion
    _lw_strong_pull_up(sensor);

    _lw_init(sensor);
    _lw_out_byte(sensor, 0xcc); // skip ROM
    _lw_out_byte(sensor, 0xbe); // read the result

    T_C = _lw_in_byte(sensor);
    sign = _lw_in_byte(sensor);

```

Note that if sign is zero, the temperature times two is in T_C and in routine 1820_1.C, I simply divided this by two. If sign is not zero, the T_C is negative and I performed a two's complement and then divided by two and displayed the result with a leading minus sign.

Program 1820_1.C interfaces with two DS18S20's, one on GP1 and the other on GP2. Temperatures are read and displayed as whole numbers.

Note that in implementing the low level routines, I opted to use the somewhat inefficient technique of implementing the code twice, once for GP1 and again for GP2.

```

// 1820_1.C, CCS - PCB (PIC12C509)
//
// Illustrates an implementation of Dallas 1-wire interface.
//
// Configuration. DS18S20 on GP.1 and GP.2. Note a 4.7K pullup to +5V
// is required on each DQ lead. DS18S20s are configured in parasite power
// mode. That is, VCC connected to ground.
//
// Reads and displays temperature and displays the result on serial LCD
// (or PC COM Port) connected to PORTC.0.
//
// PIC12C509
//
// GP2 (term 5) ----- DQ of DS18S20
// GP1 (term 6) ----- DQ of DS18S20
// GP0 (term 7) ----- To Ser LCD or PC Com Port
//
// 4.7K Pullup Resistors to +5 VDC on DQ leads of each DS18S20.
//
// Debugged using RF Solutions ICEPIC Emulator, July 27, '01.
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

#case

#device PIC12C509

#include <defs_509.h>

#include <delay.h>
#include <ser_509.h>

```

```

#define TxData 0 // use GP0
#define INV // send inverted RS232

// 1-wire prototypes
void _lw_init(byte sensor);
int _lw_in_byte(byte sensor);
void _lw_out_byte(byte sensor, byte d);
void _lw_strong_pull_up(byte sensor);

void main(void)
{
    byte sensor, T_C, sign;
    DIRS=0x3f;

    while(1)
    {
        for (sensor=1; sensor<3; sensor++) // sensors 1 and 2 only
        {

            _lw_init(sensor);
            _lw_out_byte(sensor, 0xcc); // skip ROM

            _lw_out_byte(sensor, 0x44); // perform temperature conversion
            _lw_strong_pull_up(sensor);

            _lw_init(sensor);
            _lw_out_byte(sensor, 0xcc); // skip ROM
            _lw_out_byte(sensor, 0xbe); // read the result

            T_C = _lw_in_byte(sensor);
            sign = _lw_in_byte(sensor);

            ser_init();
            ser_dec_byte(sensor, 1); // display the sensor number
            ser_new_line();

            if (sign) // negative
            {
                T_C = ~T_C + 1;
                ser_char('-');
            }

            T_C = T_C / 2;

            if (T_C > 99)
            {
                ser_dec_byte(T_C, 3);
            }
            else if (T_C > 9)
            {
                ser_dec_byte(T_C, 2);
            }
            else
            {
                ser_dec_byte(T_C, 1);
            }
            delay_ms(2000);
        }
    }
}

// The following are standard 1-Wire routines.
void _lw_init(byte sensor)
{
    if (sensor==1) // there probably is a more efficient technique
        // my goal here was clarity
    {
        #asm
            BSF DIRS, 1 // high impedance

```

```

    MOVF DIRS, W
    TRIS GPIO

    BCF GPIO, 1 // bring DQ low for 500 usecs
    BCF DIRS, 1
    MOVF DIRS, W
    TRIS GPIO
#endasm
    delay_10us(50);
#asm
    BSF DIRS, 1           // high Z
    MOVF DIRS, W
    TRIS GPIO
#endasm
    delay_10us(50);
}

else // its channel 2
{
#asm
    BSF DIRS, 2
    MOVF DIRS, W
    TRIS GPIO

    BCF GPIO, 2 // bring DQ low for 500 usecs
    BCF DIRS, 2
    MOVF DIRS, W
    TRIS GPIO
#endasm
    delay_10us(50);
#asm
    BSF DIRS, 2
    MOVF DIRS, W
    TRIS GPIO
#endasm
    delay_10us(50);
}
}

byte _lw_in_byte(byte sensor)
{
    byte n, i_byte, temp;

    for (n=0; n<8; n++)
    {
        if (sensor==1)
        {
#asm
            BCF GPIO, 1 // wink low and read
            BCF DIRS, 1
            MOVF DIRS, W
            TRIS GPIO

            BSF DIRS, 1
            MOVF DIRS, W
            TRIS GPIO

            CLRWDT
            NOP
            NOP
            NOP
            NOP
#endasm
            temp=GPIO; // now read
            if (temp & 0x02) // if GP1
            {
                i_byte=(i_byte>>1) | 0x80; // least sig bit first
            }
            else
            {
                i_byte=i_byte >> 1;
            }
        }
    }
}

```

```

    }
    delay_10us(6);
}
else
{
#asm
    BCF GPIO, 2
    BCF DIRS, 2
    MOVF DIRS, W
    TRIS GPIO

    BSF DIRS, 2
    MOVF DIRS, W
    TRIS GPIO

    CLRWDT
    NOP
    NOP
#endasm
    temp=GPIO;
    if (temp & 0x04) // GP.2
    {
        i_byte=(i_byte>>1) | 0x80; // least sig bit first
    }
    else
    {
        i_byte=i_byte >> 1;
    }
    delay_10us(6);
}
}
return(i_byte);
}

void _lw_out_byte(byte sensor, byte d)
{
    byte n;
    if (sensor==1)
    {
        for(n=0; n<8; n++)
        {
            if (d&0x01)
            {
#asm
                BCF GPIO, 1 // wink low and high and wait 60 usecs
                BCF DIRS, 1
                MOVF DIRS, W
                TRIS GPIO

                BSF DIRS, 1
                MOVF DIRS, W
                TRIS GPIO
#endasm
                delay_10us(6);
            }
            else
            {
#asm
                BCF GPIO, 1 // bring low, 60 usecs and bring high
                BCF DIRS, 1
                MOVF DIRS, W
                TRIS GPIO
#endasm
                delay_10us(6);
#asm
                BSF DIRS, 1
                MOVF DIRS, W
                TRIS GPIO
#endasm
            }
        }
    }
}

```

```

        d=d>>1;
    } // end of for
}
else // sensor 2
{
    for(n=0; n<8; n++)
    {
        if (d&0x01)
        {
#asm
            BCF GPIO, 2
            BCF DIRS, 2
            MOVF DIRS, W
            TRIS GPIO

            BSF DIRS, 2
            MOVF DIRS, W
            TRIS GPIO
#endasm
            delay_10us(6);
        }

        else
        {
#asm
            BCF GPIO, 2
            BCF DIRS, 2
            MOVF DIRS, W
            TRIS GPIO
#endasm
            delay_10us(6);
#asm
            BSF DIRS, 2
            MOVF DIRS, W
            TRIS GPIO
#endasm
        }
        d=d>>1;
    } // end of for
}

void _lw_strong_pull_up(byte sensor) // bring DQ to strong +5VDC
{
    if (sensor ==1)
    {
#asm
        BSF GPIO, 1 // output a hard logic one
        BCF DIRS, 1
        MOVF DIRS, W
        TRIS GPIO
#endasm

        delay_ms(750);
#asm
        BSF DIRS, 1
        MOVF DIRS, W
        TRIS GPIO
#endasm
    }
    else
    {
#asm
        BSF GPIO, 2 // output a hard logic one
        BCF DIRS, 2
        MOVF DIRS, W
        TRIS GPIO
#endasm

        delay_ms(750);
#asm

```

```

        BSF DIRS, 2
        MOVF DIRS, W
        TRIS GPIO
#endasm
    }
}

#include <delay.c>
#include <ser_509.c>

```

Program FRST_ALM.C (PIC12C509).

This routine was the ultimate goal of the previously presented routines.

The user program continually loops, measuring the alarm temperature setting of the potentiometer using the RC time approach. I used only the high byte of the time count which ranged from 48 for a zero setting on the potentiometer to 96 for a full setting. This value was mapped into a temperature in the range of 0 to 21 degrees C. However, if the potentiometer is near its full setting, a high byte count of greater than 90, a T_threshold of 30 degrees C was returned to permit testing of the unit in a toasty environment.

If the pushbutton on GP3 is depressed, the value of T_threshold is “displayed by flashing the portion of the bi-color LED on GP5.

Otherwise, the temperature of a DS18S20 on GP1 is measured and the value is displayed on by flashing the portion of the LED on GP4. Note that the 1-wire routines were revised for GP1 only. That is, no value of “sensor” is passed to the various low level 1-wire routines. Note that in function meas_temperature(), the value of T_C is returned as a byte with the sign bit in the most significant bit position followed by the seven magnitude bits in two’s complement format

If the measured temperature is negative or less that the T_threshold setting, a sonalert on GP0 is briefly pulsed on each pass through the loop.

```

// FRST_ALM.C (PIC12C509), CCS-PCB
//
// Frost Alarm.
//
// Continually measures temperature using a Dallas DS18S20. If the temperature
// is negative (degrees C) or if T_C < T_threshold then a Sonalert on PORTC2
// is pulsed five times.
//
// The alarm threshold temperature T_thresh is measured using the RC configuration
// shown below. If the time required for the capacitor to discharge to a logic zero
// exceeds nominally 65 ms as might be the case if the potentiometer is not present
// the T_threshold is 2 degrees C. Otherwise, the RC time is mapped into T_threshold
// values in the range of 0 to 14 degrees C and 30 degrees C. The 30 degrees C is
// provided for testing of the piezo alarm.
//
// The program continually loops, displaying the values of T_threshold or T_C on
// Alarm LED or on Temperature LED.
//
// PIC16C509
//
// GP4 (term 3) ----->|---- |----- 330 ---- GRD
// GP5 (term 2) ----->|---- |
//
// GP2 (term 5) --- 330 -----
//
//
//          |          |
//          1.0 uFd    10K Pot
//          |          |
//          |          10K Resistor
//          |          |
//          GRD        GRD
//
//          +5VDC

```

```

//
//
//          |
//          4.7K
//          |
// GP1 (term 6) ----- DS18S20
//
// GP0 (term 7) ----- Sonalert ---- GRD
//
// GRD ---- \----- GP3 (term 4) (internal pullup resistor)
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

```

```
#case
```

```
#device PIC12C509
```

```
#include <defs_509.h>
```

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

```
#define TRUE !0
```

```
#define FALSE 0
```

```
#define _1W_PIN 1
```

```
#define LED_ALM 5
```

```
#define LED_TEMP 4
```

```
void flash_q(byte LED, byte q, byte minus_flag);
```

```
byte meas_threshold(void);
```

```
byte meas_temperature(void);
```

```
void alarm(void);
```

```
// 1-wire prototypes
```

```
void wl_init(void);
```

```
int wl_in_byte(void);
```

```
void wl_out_byte(byte d);
```

```
void wl_strong_pull_up(void);
```

```
void main(void)
```

```
{
    byte T_C, T_threshold, minus_flag;
```

```
    DIRS = 0x3f;
```

```
#asm
```

```
BCF GPIO, LED_ALM // make LED pins output logic zeros
```

```
BCF GPIO, LED_TEMP
```

```
BCF DIRS, LED_ALM
```

```
BCF DIRS, LED_TEMP
```

```
MOVF DIRS, W
```

```
TRIS GPIO
```

```
#endasm
```

```
    not_gppu = 0; // enable internal weak pull-ups
```

```
#asm
```

```
MOVF OPTIONS, W
```

```
OPTION
```

```
#endasm
```

```
while(1)
```

```
{
```

```
    // measure the alarm threshold
```

```
    T_threshold = meas_threshold();
```

```
    if (!gp3)
```

```
    {
```

```
        flash_q(LED_ALM, T_threshold, FALSE);
```

```
        delay_ms(2000);
```

```
    }
```

```
    else
```

```
    {
```

```
        T_C = meas_temperature();
```

```

        if (T_C & 0x80) // if negative
        {
            minus_flag = TRUE;
            T_C = (~T_C) + 1;
        }
        else
        {
            minus_flag = FALSE;
        }
        flash_q(LED_TEMP, T_C, minus_flag);

        if ((minus_flag) || (T_C < T_threshold))
        {
            alarm();
        }
        else
        {
            delay_ms(1000);
        }
    }
}

void flash_q(byte LED, byte q, byte minus_flag)
{
    byte n, digit;

#asm
    BCF GPIO, LED_ALM // make LED pins output logic zeros
    BCF GPIO, LED_TEMP
    BCF DIRS, LED_ALM
    BCF DIRS, LED_TEMP
    MOVF DIRS, W
    TRIS GPIO
#endasm

    if (minus_flag)
    {
        if (LED == LED_ALM)
        {
#asm
            BSF GPIO, LED_ALM
#endasm
        }
        else
        {
#asm
            BSF GPIO, LED_TEMP
#endasm
        }

        delay_ms(500); // long delay to indicate minus
#asm
        BCF GPIO, LED_ALM
        BCF GPIO, LED_TEMP
#endasm
        delay_ms(1000);
    }

    digit = q/10; // number of tens
    if (digit) // if non zero
    {
        for (n=0; n<digit; n++)
        {
            if (LED == LED_ALM)
            {
#asm
                BSF GPIO, LED_ALM
#endasm
            }
        }
    }
}

```

```

    }
    else
    {
#asm
        BSF GPIO, LED_TEMP
#endasm
    }
    delay_ms(250); // long delay to indicate minus
#asm
        BCF GPIO, LED_ALM
        BCF GPIO, LED_TEMP
#endasm
    delay_ms(250);
    }

    delay_ms(1000); // separation between digits
}

digit = q%10;
if (!digit)
{
    digit = 10;
}

for (n=0; n<digit; n++)
{
    if (LED == LED_ALM)
    {
#asm
        BSF GPIO, LED_ALM
#endasm
    }
    else
    {
#asm
        BSF GPIO, LED_TEMP
#endasm
    }
    delay_ms(250); // long delay to indicate minus
#asm
        BCF GPIO, LED_ALM
        BCF GPIO, LED_TEMP
#endasm
    delay_ms(250);
    }

    delay_ms(1000); // separation between digits
}

void alarm(void) // pulse sonalert five times
{
    byte n;
    gp0 = 0;
    dirs0 = 0;
#asm
    MOVF DIRS, W
    TRIS GPIO
#endasm

    for (n=0; n<5; n++)
    {
        gp0 = 1;
        delay_ms(100);
        gp0 = 0;
        delay_ms(100);
    }
}

byte meas_threshold(void)
{
    byte count_hi, count_lo, tmr0_old, tmr0_new, T_thresh;

```

```

t0cs = 0; // fosc / 4 is clock source
psa = 1; // prescale assigned to WDT
ps2 = 0; // 1:1 prescale
ps1 = 0;
ps0 = 0;

#asm
    MOVF OPTIONS, W
    OPTION
#endasm

    dirs2 = 0; // output
#asm
    MOVF DIRS, W
    TRIS GPIO
#endasm

    gp2 = 1; // charge capacitor
    delay_ms(10);

    count_hi = 0;
    count_lo = 0;

    tmr0_old = 0x00;
    TMR0 = 0x00;
#asm
    BSF DIRS, 2
    MOVF DIRS, W
    TRIS GPIO
#endasm
while(1) // wait for cap to discharge
{
    tmr0_new = TMR0;
    if ((tmr0_new < 0x80) && (tmr0_old >= 0x80)) // there was a roll over
    {
        ++count_hi;
        if (count_hi == 0) // no zero crossing with 65 ms
        {
            return(2);
        }
    }

    if (!gp2) // if capacitor discharged below zero crossing
    {
        count_lo = tmr0_new;
        break;
    }
    tmr0_old = tmr0_new;
}
if (count_hi < 48)
{
    count_hi = 48;
}
else if (count_hi > 96)
{
    count_hi = 96;
}

if (count_hi > 90)
{
    return(30); // this is to test the alarm
}

else
{
    return((count_hi - 48)/2); // in the range of 0 to 21 degrees C
}
}

byte meas_temperature(void)

```

```

{
    byte T_C, sign;

    wl_init();
    wl_out_byte(0xcc); // skip ROM
    wl_out_byte(0x44); // perform temperature conversion

    wl_strong_pull_up();

    wl_init();
    wl_out_byte(0xcc); // skip ROM
    wl_out_byte(0xbe); // read the result

    T_C = wl_in_byte();
    sign = wl_in_byte();

    if (sign) // if negative, change to pos for divide by two
    {
        T_C = ~T_C + 1;
    }

    T_C = T_C / 2;

    if (sign) // negative
    {
        T_C = ~T_C + 1;
    }
    return(T_C);
}

```

// The following are standard 1-Wire routines.

```

void wl_init(void)
{
    #asm
        BSF DIRS, _1W_PIN // high impedance
        MOVF DIRS, W
        TRIS GPIO

        BCF GPIO, _1W_PIN // bring DQ low for 500 usecs
        BCF DIRS, _1W_PIN
        MOVF DIRS, W
        TRIS GPIO
    #endasm
    delay_10us(50);
    #asm
        BSF DIRS, _1W_PIN
        MOVF DIRS, W
        TRIS GPIO
    #endasm
    delay_10us(50);
}

byte wl_in_byte(void)
{
    byte n, i_byte, temp;

    for (n=0; n<8; n++)
    {
        #asm
            BCF GPIO, _1W_PIN // wink low and read
            BCF DIRS, _1W_PIN
            MOVF DIRS, W
            TRIS GPIO

            BSF DIRS, _1W_PIN
            MOVF DIRS, W
            TRIS GPIO

            CLRWDI

```

```

        NOP
        NOP
        NOP
        NOP
#endasm
temp = GPIO; // now read
if (temp & (0x01 << _1W_PIN))
{
    i_byte=(i_byte>>1) | 0x80; // least sig bit first
}
else
{
    i_byte=i_byte >> 1;
}
delay_10us(6);
}

return(i_byte);
}

void w1_out_byte(byte d)
{
    byte n;

    for(n=0; n<8; n++)
    {
        if (d & 0x01)
        {
#asm
            BCF GPIO, _1W_PIN // wink low and high and wait 60 usecs
            BCF DIRS, _1W_PIN
            MOVF DIRS, W
            TRIS GPIO

            BSF DIRS, _1W_PIN
            MOVF DIRS, W
            TRIS GPIO
#endasm
            delay_10us(6);
        }
        else
        {
#asm
            BCF GPIO, _1W_PIN // bring low, 60 usecs and bring high
            BCF DIRS, _1W_PIN
            MOVF DIRS, W
            TRIS GPIO
#endasm
            delay_10us(6);
#asm
            BSF DIRS, _1W_PIN
            MOVF DIRS, W
            TRIS GPIO
#endasm
        }
        d=d>>1;
    } // end of for
}

void w1_strong_pull_up(void) // bring DQ to strong +5VDC
{
#asm
    BSF GPIO, _1W_PIN // output a hard logic one
    BCF DIRS, _1W_PIN
    MOVF DIRS, W
    TRIS GPIO
#endasm

    delay_ms(750);
}

```

```

#asm
    BSF DIRS, _1W_PIN
    MOVF DIRS, W
    TRIS GPIO
#endasm
}

```

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

Internal EEPROM (PIC12CE518/519).

The 12CE518 and CE519 provide an internal 16 byte high impedance EEPROM. I believe that Microchip simply added a 24LC00 and use GPIO bits 6 and 7 to control the EEPROM using the Philips Inter IC (I2C) protocol.

This discussion does not include a discussion of the I2C protocol. It is treated in some detail in my “PIC16F87X Routines”.

My intent in developing this routine in the context of the frost alarm was to save the value of the alarm threshold. That is, when the user depressed the pushbutton, the program would continually measure and display the potentiometer setting and save the alarm threshold to EEPROM. In the normal mode, with the pushbutton released, the program would then read the alarm threshold from EEPROM.

In addition, one must deal with the situation where the user never sets the alarm threshold. Unfortunately, there is no way to initialize the EEPROM when the program is programmed into the PIC. Rather, one must determine if this is the first time the processor has booted and if so, set a dedicated byte to a default alarm temperature, say, 34. Detecting, a first time boot, might be implemented by ascertaining if four bytes in EEPROM agree with four specified values and if there is not agreement, setting these EEPROM bytes to the specified values and setting another EEPROM byte to the default 34 degrees. Thereafter, when the PIC boots, it reads the four first time bytes and on finding agreement with the specified values, it leaves the alarm threshold byte alone.

A lofty goal! My emulator did not include the ability to write to and read from the on-board EEPROM, and not having any CerDIP Windowed devices, I managed to go through some 50 one time programmable devices just to get these few routines to work. By the time, I received the Windowed devices, I was off looking at another processor and never returned to the PIC12CE519. However, the emulator I used for debugging the PIC12CE673 and CE674 did provide the capability of writing to and reading from the on-board EEPROM. Thus, I was able to do a great deal more with the PIC12CE674 which you may map over to the PIC12CE519. I also managed to clear up in my mind why I was having so much trouble with the on-board EEPROM.

Microchip assigned the SCL and SDA leads to bits 7 and 6 of GPIO, respectively. However, there are no corresponding TRIS bits.

In executing a simple command such as;

```
BCF GPIO, 6
```

There is a bit more than meets the eye. GPIO is read by the processor, bit 6 of the value is set to a one and the result is output. Unfortunately, with no TRIS bits, if when reading GPIO, bit 7 is read as a logic one, it will be output as a one. Thus, if bit 7 had been a logic zero, this simple operation changed not only bit 6, but it also changed bit 7 as well. I am uncertain I have explained this very well, and the reason is simply that I am uncertain I still fully understand it.

However, it has become clear to me that GPIO bits 6 and 7 must be treated together as two bits. That is, one bit cannot be cleared as illustrated above as it may affect the other bit.

My approach was to define a global variable `hi_two_bits`. Note that bits 7 and 6 are “scl” and “sda”, respectively. Thus, in bringing SDA and SCL either high or low, the approach is to modify the bit in `hi_two_bits` and then output this to GPIO.

For example,

```
void i2c_internal_high_sda(void)
{
    high_two_bits = high_two_bits | 0x40; // X1
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}
```

Note that in using this approach, the five lower bits of GPIO remain unchanged. However, the two high SCL and SDA bits are always output simultaneously.

All and all, using the internal EEPROM is not trivial and one wonders just how many people are using it.

In program `INTEEPRM.C`, data is written to four locations in EEPROM and is then read and displayed.

Use of the #Separate Directive.

Note that a function, including `main()` must be accommodated in a single 512 word page. Recall, also, that the compiler seems to have an absolute rule that if a function is called only one time, the function is implemented in-line.

Thus, in this case, functions `i2c_internal_eeprom_random_read()` and `i2c_internal_eeprom_random_write()` are called only one time in `main` and thus they are placed in the `main()` function and the net effect is that `main()` exceed the 512 byte limit.

Use of the `#separate` directive forces the function to be called and is thus not a part of the calling function. The implementation is illustrated below;

```
#separate byte i2c_internal_eeprom_random_read(byte adr);
#separate void i2c_internal_eeprom_random_write(byte adr, byte dat);
```

Note however, that by using this one stack level, the compiler is forced to implement these functions using only the one remaining stack level. This can be a bit unnerving as `i2c_internal_eeprom_random_read()` involves calls many functions including `i2c_internal_in_byte()`, which in turn calls such functions as `i2c_internal_low_scl(void)` which in turn calls `delay_10us()`. My only advice is to use a bit of common sense to limit the size of the nesting, but don't get paranoid as to whether you can cram it into the PIC. Rather, keep fooling with it.

```
// Program INTEEPRM.C, (PIC12CE519) CCS PCB
//
// Illustrates how to write to and read from internal EEPROM on the PIC12CE519.
//
// Note that I was unable successfully implement the MicroChip
// routine FL51XINC.ASM and as I don't have an emulator for the
// 12CE519, I managed to burn up some 30 12CE519's before abandoning
// that approach.
//
// This routine does probably use a bit more program memory but does
// work.
//
```

```

// Serial LCD or PC Com Port is connected to GP.0.  Serial data is 9600 baud, inverted.
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

#case

#device PIC12CE519

#include <defs_509.h>

#include <delay.h>
#include <ser_509.h>

#define TxData 0 // use GP0
#define INV // send inverted RS232

#separate byte i2c_internal_eeprom_random_read(byte adr);
#separate void i2c_internal_eeprom_random_write(byte adr, byte dat);

// standard I2C routines for internal EEPROM
byte i2c_internal_in_byte(byte ack);
void i2c_internal_out_byte(byte o_byte);
void i2c_internal_start(void);
void i2c_internal_stop(void);
void i2c_internal_high_sda(void);
void i2c_internal_low_sda(void);
void i2c_internal_high_scl(void);
void i2c_internal_low_scl(void);

byte high_two_bits; // bits 7 and 6 of GPIO

main(void)
{
    byte mem_adr, dat, m, n;

    high_two_bits = 0xc0; // bits 7 and 6 at one
    GPIO = GPIO & 0x3f | high_two_bits;

    DIRS = 0x3f;

    while(1)
    {
        ser_init();
        mem_adr=0x00;
        for(n=0; n<4; n++)
        {
            dat = 0x10 + n;
            i2c_internal_eeprom_random_write(mem_adr, dat);
            ++mem_adr;
        }

        // now, read the data back and display
        mem_adr=0x00;
        for(n=0; n<4; n++)
        {
            dat = i2c_internal_eeprom_random_read(mem_adr);
            ser_hex_byte(dat);
            ser_char(' ');
            ++mem_adr;
        }
        delay_ms(500);
    }
}

#separate byte i2c_internal_eeprom_random_read(byte adr)
{
    byte dat;
    i2c_internal_start();
    i2c_internal_out_byte(0xa0);
    i2c_internal_out_byte(adr);
}

```

```

i2c_internal_start();
i2c_internal_out_byte(0xa1);

dat = i2c_internal_in_byte(0); // no ack prior to stop
i2c_internal_stop();
return(dat);
}

#separate void i2c_internal_eeprom_random_write(byte adr, byte dat)
{
    i2c_internal_start();
    i2c_internal_out_byte(0xa0);
    i2c_internal_out_byte(adr);
    i2c_internal_out_byte(dat);
    i2c_internal_stop();
    delay_ms(25); // wait for byte to burn
}

byte i2c_internal_in_byte(byte ack)
{
    byte i_byte, n;
    i2c_internal_high_sda();
    for (n=0; n<8; n++)
    {
        i2c_internal_high_scl();
        if (sda_in)
        {
            i_byte = (i_byte << 1) | 0x01; // msbit first
        }
        else
        {
            i_byte = i_byte << 1;
        }
        i2c_internal_low_scl();
    }
    if (ack)
    {
        i2c_internal_low_sda(); // ack slave with zero
    }
    i2c_internal_high_scl();
    i2c_internal_low_scl();
    i2c_internal_high_sda(); // be sure to exit with SDA high
    return(i_byte);
}

void i2c_internal_out_byte(byte o_byte)
{
    byte n;
    for(n=0; n<8; n++)
    {
        if(o_byte&0x80)
        {
            i2c_internal_high_sda();
            //ser_char('1'); // used for debugging
        }
        else
        {
            i2c_internal_low_sda();
            //ser_char('0'); // used for debugging
        }
        i2c_internal_high_scl();
        i2c_internal_low_scl();
        o_byte = o_byte << 1;
    }
    i2c_internal_high_sda();
    //ser_new_line(); // for debugging
    i2c_internal_high_scl(); // allow for slave to ack
    i2c_internal_low_scl();
}

void i2c_internal_start(void)

```

```

{
    i2c_internal_low_scl();
    i2c_internal_high_sda();
    i2c_internal_high_scl(); // bring SDA low while SCL is high
    i2c_internal_low_sda();
    i2c_internal_low_scl();
}

void i2c_internal_stop(void)
{
    i2c_internal_low_scl();
    i2c_internal_low_sda();
    i2c_internal_high_scl();
    i2c_internal_high_sda(); // bring SDA high while SCL is high
    // idle is SDA high and SCL high
}

void i2c_internal_high_sda(void)
{
    high_two_bits = high_two_bits | 0x40; // X1
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}

void i2c_internal_low_sda(void)
{
    high_two_bits = high_two_bits & 0x80; // X0
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}

void i2c_internal_high_scl(void)
{
    high_two_bits = high_two_bits | 0x80; // 1X
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}

void i2c_internal_low_scl(void)
{
    high_two_bits = high_two_bits & 0x40; // 0X
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}

#include <delay.c>
#include <ser_509.c>

```

Summary (PIC12C509).

PIC16C505.

The PIC16C505 might be characterized as a PIC12C509 with six additional IO pins and nominally 30 more bytes of RAM. Otherwise, it appears to me to be functionally identical to the PIC12C509. The cost is nominally \$1.10 in 100 quantities, about \$0.15 more than the PIC12C509.

The IO bits are identified as the six lower bits of PORTB and PORTC. In general, PORTB might be thought of as parallel to the GPIO on the 12C509. That is, the weak pull-up resistors may be enabled on PORTB, wakeup from SLEEP is caused on pin change is associated with PORTB, an external oscillator uses bits 5 and 4 of PORTB, an external /MCLR is associated with bit 3 and PORTB3 may only be configured as an input.

One notable exception is that the TMR0 input (TOCKI) is assigned to PORTC5 rather than GP2 for the PIC12C509.

I may have missed something in this cursory analysis, but the close similarity should permit you to quickly port code from one to the other. I happened to use two different emulators, one for the 12C509 and another for the 16C505, but only because I had them. My suggestion is that only one is required, and I would suggest the 16C505. Debug on the 16C505 and then change a few references for the 12C509.

DEFS_505.H.

The definitions closely follow those of the PIC12C50X.

However, note that there are now two ports, PORTB and PORTC. Note that individual bits within these may either be referred to as either rb5 or portb5 and rc5 or portc5.

As there are two IO ports, I have declared two global variables, DIRB and DIRC. Individual bits within these are of the form dira2 or dirc5.

Three other bits differ from the 12C509; rbwuf, not_rbwu and not_rbpw as opposed to gpwuf, not_gpwu and not_gppu for the 12C509.

As with the 12C509, modifying the direction of an IO requires that the bit in DIRB or DIRC be modified and then execution of the TRIS command as shown in the following examples.

```
        DIRB = 0x30           // bits 3, 2, 1 and 0 outputs
#asm
        MOVF DIRB, W
        TRIS PORTB
#endasm

        dirb5 = 0;           // make rb5 an output
#asm
        MOVF DIRB, W
        TRIS PORTB
#endasm

        dircl = 0;           // make rcl and output
#asm
        MOVF DIRC, W
        TRIS PORTC
#endasm

#asm
        BCF DIRB, 5           // make rb5 an output
        MOVF DIRB, W
        TRIS PORTB
#endasm
```

The OPTION register is handled exactly as with the PIC12C509.

```
        not_rbpw = 0;        // enable weak pull-ups
#asm
        MOVF OPTIONS, W
        OPTION
#endasm

// DEFS_505.H
//
// Standard definitions for PIC16C505
//
// Particularly note the declaration of static byte DIRB, DIRC and OPTIONS
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01
```

```

#define byte unsigned int

#define W 0
#define F 1

//----- Register Files -----

#define INDF =0x00
#define TMR0 =0x01
#define PCL =0x02
#define STATUS =0x03
#define FSR =0x04
#define OSCCAL =0x05
#define PORTB =0x06
#define PORTC =0x07

static byte DIRB, DIRC, OPTIONS; // note global definition

#define dirb5 =DIRB.5
#define dirb4 =DIRB.4
#define dirb3 =DIRB.3
#define dirb2 =DIRB.2
#define dirb1 =DIRB.1
#define dirb0 =DIRB.0

#define dirc5 =DIRC.5
#define dirc4 =DIRC.4
#define dirc3 =DIRC.3
#define dirc2 =DIRC.2
#define dirc1 =DIRC.1
#define dirc0 =DIRC.0

#define rb5 =PORTB.5
#define rb4 =PORTB.4
#define rb3 =PORTB.3
#define rb2 =PORTB.2
#define rb1 =PORTB.1
#define rb0 =PORTB.0

#define rc5 =PORTC.5
#define rc4 =PORTC.4
#define rc3 =PORTC.3
#define rc2 =PORTC.2
#define rc1 =PORTC.1
#define rc0 =PORTC.0

#define portb5 =PORTB.5
#define portb4 =PORTB.4
#define portb3 =PORTB.3
#define portb2 =PORTB.2
#define portb1 =PORTB.1
#define portb0 =PORTB.0

#define portc5 =PORTC.5
#define portc4 =PORTC.4
#define portc3 =PORTC.3
#define portc2 =PORTC.2
#define portc1 =PORTC.1
#define portc0 =PORTC.0

//----- STATUS Bits -----

#define rbwuf =0x03.7
#define pa0 =0x03.5
#define not_to =0x03.4
#define not_pd =0x03.3
// #bit z =0x03.2 // probably not required in C
// #bit dc =0x03.1 // but if they are, come up with
// #bit c =0x03.0 // something more unique than "z" and "c"

```

```
//----- OPTION Bits -----
// OPTION Bits
// Note that PTIONS is a file containing the options. Actually
// setting the options requires a call to set_options which moves
// DIRS to W and then OPTION

#bit not_rbwu      =OPTIONS.7
#bit not_rbpw     =OPTIONS.6
#bit t0cs         =OPTIONS.5
#bit t0se         =OPTIONS.4
#bit psa          =OPTIONS.3
#bit ps2          =OPTIONS.2
#bit ps1          =OPTIONS.1
#bit ps0          =OPTIONS.0
```

```
#define ZZ 2
#define CY 0
```

Parallel Routines for the PIC16C505.

The following routines parallel those which were written for the PIC12C509. Some of the details may vary, but only because I wrote them at different times, but I see no point in including them in this discussion.

delay.c and delay.h. Same as for the PIC12C509.

ser_505.c and ser_505.h. Uses a bit on PORTC to output 9600 baud serial, either true or inverted. TxData must be defined in the using program.

tst_ser.c. Illustrates the various routines in ser_505 to display a string and byte in both decimal and hex format.

rctime.c. Measures RCTime using IO portc1.

flash_q.c. Flashes LED on portc0 to “display” a value.

tone_q.c. Beeps a speaker on portc0 to “sound” a value.

1820_1.c. Interfaces with a Dallas DS18S20 on portb0 and displays result on serial LCD.

frst_alm.c. Permits the user to set the alarm threshold by varying a potentiometer. Continually reads temperature using the DS18S20 and displays on serial LCD. If the measured temperature is less than the alarm threshold, a sonalert is pulsed. Note that the values could have been “displayed” using the pulsing of an LED as in flash_q.c.

Program 7_SEG_1.C.

In the design of the frost alarm, the limited number of pins on the 12C509 limited the interface with the user to flashing an LED, beeping a speaker or interfacing with a serial LCD.

However, the extra pins associated with the 16C505 suggests the use of a 7-segment LED to display either the alarm threshold temperature or the current temperature.

In the following routine, either of these are displayed on a single 7-segment LED.

Note that the patterns to display various numbers and characters are defined in a constant array, where a one indicates a lit segment. In addition, a “blank” is #defined as 0x00 and a minus sign as 0x01, segment g only. Note that in this routine, the characters A – F and H, L, P are defined but are not used.

```

byte const hex_digit_patts[16]=
//   0   1   2   3   4   5   6   7
   {0x7e, 0x30, 0x6d, 0x79, 0x33, 0x5b, 0x5f, 0x70,
//   8   9   A   B   C   D   E   F
   0x7f, 0x7b, 0x77, 0x1f, 0x4e, 0x3d, 0x4f, 0x47};

byte const help_patts[4] = {0x37, 0x4f, 0x0f, 0x67}; // info only
                           //   H   E   L   P

```

Subroutine display_q() outputs the value by sequentially briefly displaying each digit and possibly a leading minus sign with a blank display between each digit.

Note that the seven bits associated with the 7-segment LED are not contiguous on the same port. Rather the five lower bits are on PORTC0 - PORTC4 and the high two bits are PORTB4 and PORTB5. Note that I left PORTC5 unused as this IO serves the important alternate function of being the input to TMR0.

Thus, in outputting a pattern, the pattern is first inverted as a logic zero actually lights a segment. The resulting patt is split into the lower five and the upper two bits and the five lower bits and two highest bits of PORTC and PORTB are configured as outputs and segs_low_five are output on the lowest five bits of PORTC and segs_high_two on the highest two bits of PORTB. Note that in the implementation, the directions and states of the other bits on PORTB and PORTC are left undisturbed.

```

patt = ~patt; // convert to negative logic
segs_low_five = patt & 0x1f;
segs_high_two = (patt >> 5) & 0x03;

DIRC = DIRC & 0x20; // make lowest five bits outputs
DIRB = DIRB & 0x0f; // make bits 5 and 4 outputs
#asm
  MOVF DIRC, W
  TRIS PORTC
  MOVF DIRB, W
  TRIS PORTB
#endasm

PORTC = (PORTC & 0x20) | segs_low_five; // output the low five bits
PORTB = (PORTB & 0x0f) | (segs_high_two << 4);

// _7_SEG_1.C (PIC16HV540), CCS PCB
//
// PIC16C505          Common Anode 7-Seg LED (MAN72A or LSD3221-11, Jameco #24740)
//
// PORTB5 (term 2) ----- 330 ----- a seg (term 1) ----|< ----- +5 VDC (terms 3, 14)
// PORTB4 (term 3) ----- 330 ----- b seg (term 13)
//
// PORTC4 (term 6) ----- 330 ----- c seg (term 10)
// PORTC3 (term 7) ----- 330 ----- d seg (term 8)
// PORTC2 (term 8) ----- 330 ----- e seg (term 7)
// PORTC1 (term 9) ----- 330 ----- f seg (term 2)
// PORTC0 (term 10) ----- 330 ----- g seg (term 11)
//
//           a // Layout of 7-seg LED
//          f  b
//           g
//          e  c
//           d
//
// copyright, Peter H. Anderson, Baltimore, MD, July, '01

#case

```

```

#define PIC16C505 *=8

#include <defs_505.h>
#include <delay.h>

#define TRUE !0
#define FALSE 0

#define MINUS_SIGN 0x01 // segment g only
#define BLANK 0x00 // no segments

void display_q(byte q, byte minus_flag);
void display_patt(byte patt);

void main(void)
{
    byte T_threshold = 34, T_C, minus_flag, n;
    char const T_C_array[5] = {-5, 0, 1, 25, 70};

    DIRC = 0x3f;
    DIRB = 0x3f;

    not_rbpu = 0; // enable weak pullups on PORTB
#asm
    MOVF OPTIONS, W
    OPTION
#endasm

    while(1)
    {
        if(!portb3) // if switch at ground
        {
            display_q(T_threshold, FALSE);
            delay_ms(2000);
        }
        else
        {
            for (n = 0; n < 5; n++)
            {
                T_C = T_C_array[n];
                if (T_C & 0x80)
                {
                    minus_flag = TRUE;
                    T_C = (~T_C) + 1;
                }
                else
                {
                    minus_flag = FALSE;
                }
                display_q(T_C, minus_flag);
                delay_ms(2000);
            }
        }
    }
}

void display_q(byte q, byte minus_flag)
{
    byte const hex_digit_patts[16]=
    //  0   1   2   3   4   5   6   7
    {0x7e, 0x30, 0x6d, 0x79, 0x33, 0x5b, 0x5f, 0x70,
    //  8   9   A   B   C   D   E   F
    0x7f, 0x7b, 0x77, 0x1f, 0x4e, 0x3d, 0x4f, 0x47};

    byte const help_patts[4] = {0x37, 0x4f, 0x0f, 0x67}; // info only
    //      H      E      L      P

    byte n, digit;

    if (minus_flag)

```

```

    {
        display_patt(MINUS_SIGN);
        delay_ms(500);
        display_patt(BLANK);
        delay_ms(500);
    }

    digit = q/10; // number of tens

    display_patt(hex_digit_patts[digit]);
    delay_ms(500);
    display_patt(BLANK);
    delay_ms(500);

    digit = q%10;

    display_patt(hex_digit_patts[digit]);
    delay_ms(500);
    display_patt(BLANK);
    delay_ms(500);
}

void display_patt(byte patt)
{
    byte segs_low_five, segs_high_two;

    patt = ~patt; // convert to negative logic
    segs_low_five = patt & 0x1f;
    segs_high_two = (patt >> 5) & 0x03;

    DIRC = DIRC & 0x20; // make lowest five bits outputs
    DIRB = DIRB & 0x0f; // make bits 5 and 4 outputs
#asm
    MOVF DIRC, W
    TRIS PORTC
    MOVF DIRB, W
    TRIS PORTB
#endasm

    PORTC = (PORTC & 0x20) | segs_low_five; // output the low five bits
    PORTB = (PORTB & 0x0f) | (segs_high_two << 4);
}

#include <delay.c>

```

Program LCD_OUT.C, LCD_OUT.H (PIC16C505).

Another technique for displaying data is to directly interface with a text LCD with a standard Hitachi interface.

```

// PORTC5 (term 5) ----- EN (CLK) (term 6)
// PORTC4 (term 6) ----- RS (DAT/CMD) (term 4)
//                               GRD ----- RW (term 5)
//
// PORTC3 (term 7) ----- DB7 (term 14)
// PORTC2 (term 8) ----- DB6 (term 13)
// PORTC1 (term 9) ----- DB5 (term 12)
// PORTC0 (term 10) ----- DB4 (term 11)

```

Note that I used all of PORTC. However, this might be modified to use another IO for the CLK function and leave T0CKI (PORTC5) available as an input to TMR0.

Whenever I look at data sheets for the Hitachi compatible LCDs, I can only speculate that the Asian engineers looked at various designs and concluded the “EN” was an English abbreviation for “CLK” and “RS”, an abbreviation for whether

the data appearing on DB4 – DB7 is data to be displayed or a command, which I call DAT/CMD. In this discussion, I use CLK and DAT/CMD.

A command byte might be to select a font, format of the cursor, cursor position or clearing the LCD. A data byte is simply the ASCII value of the character to be displayed. A two step process is required to transfer first the high and then the low nibble. Note that I have named the function which transfers a data byte `lcd_char()` and a command byte `lcd_cmd_byte`. Note that the only difference between the two functions is whether `lcd_data_nibble()` or `lcd_cmd_nibble()` is called.

```
void lcd_char(byte c)      // displays ASCII character c to LCD
{
    lcd_data_nibble(c>>4); // high byte followed by low
    lcd_data_nibble(c&0x0f);
    delay_ms(1);
}

void lcd_cmd_byte(byte c) // used for sending byte commands
{
    lcd_cmd_nibble(c>>4); // high byte followed by low
    lcd_cmd_nibble(c&0x0f);
    delay_ms(1);
}
```

The implementations of `lcd_data_nibble()` and `lcd_cmd_nibble()`;

```
void lcd_data_nibble(byte c) // RS is at logic one for data
{
    PORTC = c | 0x10;          // DAT/CMD high
    delay_ms(1);
    PORTC = c | 0x10 | 0x20; // CLK high
    delay_ms(1);
    PORTC = c | 0x10;          // CLK back to zero
}

void lcd_cmd_nibble(byte c) // RS is at logic zero for commands
{
    PORTC = c;
    delay_ms(1);
    PORTC = c | 0x20;          // CLK high
    delay_ms(1);
    PORTC = c;                 // and then low
}
```

Note that with `lcd_cmd_nibble`, the nibble is output on the 4-bit bus with DAT/CMD low and CLK is brought high and then low. The implementation of `lcd_data_nibble()` is the same except the DAT/CMD lead is high.

In `lcd_init()`, all six bits on PORTC are made outputs and the LCD is configured in the 4-bit transfer mode by sending the nibble 0x3, three times. The command byte 0x0f selects the 5 X 8 font, cursor on and the cursor type. Command byte 0x01, clears the LCD and sets the cursor in the upper left position.

```
void lcd_init(void)
{
    PORTC = 0x00;
    DIRC = 0x00;
#asm
    MOVF DIRC, W
    TRIS PORTC
#endasm
```

```

    lcd_cmd_nibble(0x03);          // configure LCD in 4-bit transfer mode
    delay_ms(5);
    lcd_cmd_nibble(0x03);
    delay_ms(5);
    lcd_cmd_nibble(0x03);
    delay_ms(5);
    lcd_cmd_nibble(0x02);
    delay_ms(5);
    lcd_cmd_byte(0x0f);
    lcd_cmd_byte(0x01);
}

```

LCD_OUT.C also provides routines to clear a specified line, clear the LCD, position the cursor, display a character, display a quantity in decimal or in hexadecimal format.

Note that LCD_OUT.C is simply a collection of routines which may be included in a routine, much like the ser_505 routines. The delay.h and delay.c files must also be included as these functions are used in the implementation of the most routines in LCD_OUT.C.

One can only marvel at the genius of a compiler that manages to implement this with the two stack limitation. Consider that a call to lcd_dec_byte() requires a call to lcd_char() which requires a call to lcd_data_nibble() which in turn calls delay_ms() which in turn, calls delay_10us(). I'm not sure just how far one can push this, but I have found the CCS compiler will attempt it and either it will all fit or you will get an "out of ROM" error. Could I do all of this in assembly. Yes, but probably not as compactly as the CCS compiler. Would I want to? No.

Note that I found it was necessary to declare the lcd_dec_byte() and lcd_hex_byte() functions as #separate as their implementation was all being implemented in my main() which caused the main() to exceed the 512 byte limit. This may vary, depending on the exact nature of your code.

```

// Program LCD_OUT.C (PIC16C505), CCS PCB
//
// This collection of routines provides a direct interface with a 20X4 Optrex
// DMC20434 LCD to permit the display of text. This uses PIC outputs PORTC,
// bits 0 - 5
//
// Routine lcd_init() places the LCD in a 4-bit transfer mode, selects
// the 5X8 font, blinking block cursor, clears the LCD and places the
// cursor in the upper left.
//
// Routine lcd_char(byte c) displays ASCII value c on the LCD. Note that
// this permits the use of printf statements; printf(lcd_char, "T=%f", T_F).
//
// Routine lcd_dec_byte() displays a quantity with a specified number of
// digits. Routine lcd_hex_byte() displays a byte in two digit hex format.
// Routine lcd_str() outputs the string. In many applications, these may
// be used in place of printf statements.
//
// Routine lcd_clr() clears the LCD and locates the cursor at the upper
// left. lcd_clr_line() clears the specified line and places the cursor
// at the beginning of that line. Lines are numbered 0, 1, 2, 3.
//
// Routine lcd_cmd_byte() may be used to send a command to the lcd.
//
// Routine lcd_cursor_pos() places the cursor on the specified line (0-3)
// at the specified position (0 - 19).
//
// The other routines are used to implement the above.
//
// lcd_data_nibble() - used to implement lcd_char. Outputs the
// specified nibble.
//
// lcd_cmd_nibble() - used to implement lcd_cmd_byte. The difference
// between lcd_data_nibble and lcd_cmd_nibble is that with data, LCD
// input RS is at a logic one.

```

```

//
// num_to_char() - converts a digit to its ASCII equivalent.
//
// PIC16C505          DMC20434
//
// PORTC5 (term 5) ----- EN (CLK) (term 6)
// PORTC4 (term 6) ----- RS (DAT/CMD) (term 4)
//          GRD ----- RW (term 5)
//
// PORTC3 (term 7) ----- DB7 (term 14)
// PORTC2 (term 8) ----- DB6 (term 13)
// PORTC1 (term 9) ----- DB5 (term 12)
// PORTC0 (term 10) ----- DB4 (term 11)
//
//          +5 VDC ----- VCC (term 2)
//          |
//          4.7K
//          |_____ VEE (term 3)
//          |
//          330
//          |
//          GRD ----- GRD (term 1)
//
// copyright, Peter H. Anderson, Brattleboro, VT, July, '01

void lcd_char(byte c) // displays ASCII character c to LCD
{
    lcd_data_nibble(c>>4); // high byte followed by low
    lcd_data_nibble(c&0x0f);
    delay_ms(1);
}

void lcd_data_nibble(byte c) // RS is at logic one for data
{
    PORTC = c | 0x10;
    delay_ms(1);
    PORTC = c | 0x10 | 0x20;
    delay_ms(1);
    PORTC = c | 0x10;
}

void lcd_cmd_byte(byte c) // used for sending byte commands
{
    lcd_cmd_nibble(c>>4); // high byte followed by low
    lcd_cmd_nibble(c&0x0f);
    delay_ms(1);
}

void lcd_cmd_nibble(byte c) // RS is at logic zero for commands
{
    PORTC = c;
    delay_ms(1);
    PORTC = c | 0x20;
    delay_ms(1);
    PORTC = c;
}

void lcd_init(void)
{
    PORTC = 0x00;
    DIRC = 0x00;
#asm
    MOVF DIRC, W
    TRIS PORTC
#endasm

    lcd_cmd_nibble(0x03); // configure LCD in 4-bit transfer mode
    delay_ms(5);
    lcd_cmd_nibble(0x03);
    delay_ms(5);
}

```

```

    lcd_cmd_nibble(0x03);
    delay_ms(5);
    lcd_cmd_nibble(0x02);
    delay_ms(5);
    lcd_cmd_byte(0x0f);
    lcd_cmd_byte(0x01);
}

void lcd_clr(void)    // clear LCD and cursor to upper left
{
    lcd_cmd_byte(0x01);
}

void lcd_clr_line(byte line) // clear indicated line and leave
                             // cursor at the beginning of the line
{
    byte n;
    lcd_cursor_pos(line, 0);
    for (n=0; n<20; n++)
    {
        lcd_char(' ');
    }
    lcd_cursor_pos(line, 0);
}

void lcd_cursor_pos(byte line, byte pos)
                             // position cursor on line 0 .. 3, pos 0 .. 19
{
    const byte a[4] = {0x80, 0xc0, 0x94, 0xd4};
    lcd_cmd_byte(a[line]+pos);
}

void lcd_str(char *s)
{
    byte n=0;
    while(s[n])
    {
        lcd_char(s[n]);
        ++n;
    }
}

#separate void lcd_dec_byte(byte val, byte digits)
// displays byte in decimal as either 1, 2 or 3 digits
{
    byte d;
    char ch;
    if (digits == 3)
    {
        d=val/100;
        ch=num_to_char(d);
        lcd_char(ch);
    }
    if (digits >1) // take the two lowest digits
    {
        val=val%100;
        d=val/10;
        ch=num_to_char(d);
        lcd_char(ch);
    }
    if (digits == 1) // take the least significant digit
    {
        val = val%100;
    }

    d=val % 10;
    ch=num_to_char(d);
    lcd_char(ch);
}

#separate void lcd_hex_byte(byte val)

```

```

{
    byte d;
    char ch;
    d = val >> 4;
    ch = num_to_char(d); // high nibble
    lcd_char(ch);
    d = val & 0xf;
    ch = num_to_char(d); // low nibble
    lcd_char(ch);
}

char num_to_char(byte val) // converts val to hex character
{
    char ch;
    if (val < 10)
    {
        ch=val+'0';
    }
    else
    {
        val=val-10;
        ch=val + 'A';
    }
    return(ch);
}

```

Program TST_LCD2.C (PIC16C505).

This program illustrates the use of the various functions in LCD_OUT.C. It illustrates the initialization of the LCD, outputting a string and displaying a byte in decimal and hexadecimal formats.

```

// Program TST_LCD.C (PIC16C505), CCS, PCB
//
// Direct Interface with Optrex DMC20434.
//
// Illustrates the various features of the routines in LCD_OUT.C
//
// Displays a string, byte in decimal format and in hex format.
//
// copyright, Peter H. Anderson, Brattleboro, VT, July, '01

#case

#device PIC16C505 *=8

#include <defs_505.h>
#include <lcd_out.h>
#include <delay.h>

void main(void)
{
    byte n, q;
    byte const str_const[20] = {" Hello World"};

    DIRC = 0x3f;
    DIRB = 0x3f;

    lcd_init();
    q = 0;
    while(1)
    {

        lcd_clr_line(0); // beginning of line 0
        n=0;
        while (str_const[n])
        {
            lcd_char(str_const[n]);
            ++n;
        }
    }
}

```

```

    lcd_clr_line(1);
    lcd_dec_byte(q, 3);
    lcd_cursor_pos(1, 10); // line 1, position 10
    lcd_hex_byte(q);

    ++q; // modify the displayed quantity

    delay_ms(1000);
}
}

#include <lcd_out.c>
#include <delay.c>

```

Program TST_LCD2.C (PIC16C505).

This program continually measures the temperature using a Dallas DS18S20 and displays the result on the Hitachi style text LCD. It is not treated in this detailed discussion. But, it is interesting to note that these PICs with very limited resources can, in fact, perform some rather complex tasks.

Program EVENT.C. (PIC16C505).

As previously noted, the 12-bit core devices do not have interrupt capability. I managed to get around this in using TMR0 as a timer by continually sampling the timer and determining if it used to be 0x80 or greater and is now less than 0x80. Of course, this requires that the program continually look at the value of TMR0.

But, the clock source for TMR0 may be configured as an input on PORTC5 which might be used to count events over a period of time. This is discussed below.

However, this external clocking ability may also be used to detect an event (pulse on PORTC5) without the processor camping on a pin. That is, the processor may set TMR0 to 0x00 and then perform other tasks and when convenient, check TMR0 and if it is non zero, an event happened and the processor can perform the defined task. Much like detecting the rollover of TMR0 in the timer mode, this isn't quite as robust as an interrupt as the response is dependent on how often TMR0 is read, but it is a viable alternative to an external interrupt.

Note that this concept is also applicable to the PIC12C509.

As I write this, I am uncertain program EVENT.C is the best example as the routine is not really performing any task when idle. However, hopefully, you get the point.

TMR0 is configured for PORTC5 as the clock source and the prescaler is assigned to the watch dog timer which provides an effective 1:1 prescale for TMR0. TMR0 is set to zero, and when the program detects that its value is non zero, the program flashes an LED ten times.

Note that as the T0CKI input is on PORTC5, there is no internal weak pull-up and thus if the nature of the external source is one of open and ground, an external pull-up resistor is required.

```

// EVENT.C (PIC16C505), CCS-PCB
//
// Normally, an LED is on. If there was one or more events on T0CKI, the
// LED is flashed 10 times at 10 pulses per second.
//
//
//          +5 VDC
//          |
//          10K
//          |
// GRD ----- \----- T0CKI/PORTC5 (term 5)
//                  PORTC4 (term 6) ----- 330 ----->| ----- GRD
//

```

```

//
// Tested using RICE-17A on July 22, '01
//
// copyright, Peter H. Anderson, Baltimore, MD, July, '01

#case

#device PIC16C505 *=8

#include <defs_505.h>
#include <delay.h>

void main(void)
{
    byte n;

    DIRB = 0x3f;
    DIRC = 0x3f;

    t0cs = 1; // external event
    psa = 1; // prescale assigned to WDT
    ps2 = 0; // 1:1 prescale
    ps1 = 0;
    ps0 = 0;

#asm
    MOVF OPTIONS, W
    OPTION
#endasm
    TMR0 = 0x00;

    while(1)
    {
        portc4 = 1;
        dirc4 = 0;
#asm
        MOVF DIRC, W
        TRIS PORTC
#endasm

        if (TMR0)
        {
            TMR0 = 0x00;
            for (n = 0; n<10; n++)
            {
                portc4 = 0;
                delay_ms(50);
                portc4 = 1;
                delay_ms(50);
            }
        }
    }

#include <delay.c>

```

Program COUNT.C. (PIC16C505).

In this program the number of transitions on input T0CKI is counted over a period of five seconds and the resulting RPM is displayed on a serial LCD or PC COM Port.

In function count(), TMR0 is configured for transition on input T0CKI and the prescaler is assigned to the watch-dog timer.

The number of ms to count is broken into the number of hundreds of ms and then the number of remaining ms. The program then loops for the number of hundreds of ms, delaying for 100 ms, and then reading TMR0. If there is a rollover,

counter_hi is incremented. This is repeated for the number of remaining ms. The value of counter_hi and the residual count are combined into a long.

The idea in breaking the timing interval into smaller intervals, in this case 100 ms, is to avoid TMR0 from rolling over twice prior to each read of TMR0. For example, if the input were a maximum of 1000 pulses per second, the count in TMR0 would not advance more than 100 during each 100 ms timing interval and thus TMR0 would not rollover twice during the 100 ms timing. However, if the input were 3000 pulses per second, 100 ms would be inadequate.

In testing this routine, I was surprised (and confused) to find that the 5000 timing interval was closer to seven seconds. Clearly, there is some overhead in addition to the delay_ms(100) and as previously noted, I can't attest to the precision of these crude timing routines. However, this simply couldn't explain the 7000 ms compared to 5000 ms. I now see that my problem was that the variables in my timing routines were assigned to RAM banks other than bank 0 and thus the compiler was setting and clearing the higher order bits in the FSR register. This was previously discussed in the context of the serial routines associated with the PIC12C509 discussion. The solution is to either globally declare the timing variables or adjust the timing routines such that delay_ms(100) does in fact provide a 100 ms delay.

Note that the concepts presented in this routine are applicable to the PIC12C509 as well.

```
// COUNT.C (PIC16C505), CCS-PCB
//
// Counts the number of pulses on input T0CKI over a period of time and displays events per
// minute on serial LCD or PC Com Port, 9600 baud, inverted.
//
//          +5 VDC
//          |
//          10K
//          |
// 100 PPS ----- T0CKI/PORTC5 (term 5)
//                   PORTC0 (term 10) ----- To Serial LCD or PC Com Port
//
// Tested using RICE-17A on July 23, '01
//
// copyright, Peter H. Anderson, Baltimore, MD, July, '01

#case

#device PIC16C505 *=8

#include <defs_505.h>
#include <delay.h>
#include <ser_505.h>

#define TxData 0
#define INV

long count(long ms);

void main(void)
{
    long events, RPM;
    byte hi, lo;

    DIRB = 0x3f;
    DIRC = 0x3f;
    ser_init();

    while(1)
    {
        events = count(5000);
        RPM = events * (60 / 5);           // 60 seconds / 5 sec sample time
        hi = (byte) (RPM / 100);         // split into high and low bytes
        lo = (byte) (RPM % 100);
    }
}
```

```

        ser_init();

        ser_dec_byte(hi, 3);
        ser_dec_byte(lo, 2);
    }
}

long count(long ms)
{
    byte ms_100, ms_1, count_hi, count_lo, tmr0_old, tmr0_new, n;
    long cnt;

    t0cs = 1; // external clock source
    psa = 1; // prescale assigned to WDT
    ps2 = 0; // 1:1 prescale
    ps1 = 0;
    ps0 = 0;
#asm
    MOVF OPTIONS, W
    OPTION
#endasm
    ms_100 = (byte) (ms / 100);
    ms_1 = (byte) (ms % 100);

    tmr0_old = 0;
    count_hi = 0;
    count_lo = 0;

    TMR0 = 0x00;
    for(n = 0; n < ms_100; n++)
    {
        delay_ms(100);
        tmr0_new = TMR0;

        if ((tmr0_new < 0x80) && (tmr0_old >= 0x80)) // there was a roll over
        {
            ++count_hi;
        }
        tmr0_old = tmr0_new;
    }

    if (ms_1)
    {
        delay_ms(ms_1);
        tmr0_new = TMR0;
        if ((tmr0_new < 0x80) && (tmr0_old >= 0x80)) // there was a roll over
        {
            ++count_hi;
        }
    }
    count_lo = tmr0_new;
    cnt = count_hi;
    cnt = (cnt << 8) | count_lo;
    return(cnt);
}

#include <delay.c>
#include <ser_505.c>

```

PIC16HV540.

Introduction.

The PIC16HV540 is an interesting device in that it includes an internal 3.0 or 5.0 VDC programmable regulator and may be powered from a source in the range of 3.5 to 15.0 VDC. Note that the value of the internal regulator may be set in software to either 3.0 or 5.0 VDC and thus, for 5.0 VDC operation, I assume the supply voltage must be in the range of 5.5 to 15.0 VDC. In the following, I assume the regulator voltage is set to 5.0 VDC.

The 15.0 VDC supply suggests that the 16HV540 may be used in an automobile environment, but I would be inclined to protect the PIC against spikes with an upfront zener diode. This is a bit trickier than it might seem as the zener should be in the range of 13.6 to 15.0 VDC, but it strikes me that somewhere out there in the world must be a standard diode for this type of application.

The PIC16HV540 is a bit more than an upfront regulator driving a PIC.

Rather, eight of the IO pins (PORTB) are sourced in the output mode at the supply voltage. For example, the output states are near ground for a logic zero and near V_{supply} for a logic one. Inputs are TTL compatible but may also be used with levels up to V_{supply}. That is, a logic zero is near ground and a logic one is anything above nominally 2.0 VDC, up to V_{supply}.

The other four IO pins (PORTA) are regulated at either +5 or +3 VDC. In addition to performing normal input and output functions, these may be used to power external +5 VDC peripherals. The inputs are TTL. The input voltage to these IOs is limited to +5.0 VDC.

There is an independent T0CKI input to TMR0. This is a Schmidt Trigger input where a logic one is greater than $0.85 * 5.0$ VDC and a logic zero is less than $0.1 * 5.0$ VDC. However, note that this input can also be interfaced directly with ground and voltages up to V_{supply}.

The four lower bits of PORTB may be used for wakeup on change. In addition, PORTB7 may be used for a slow wakeup on change.

The software stack level is four rather than two for all other 12-bit core devices that I am aware of. This results in less code being compiled “in-line” which helps to save program memory. However, the program memory is limited to 512 bytes and the RAM memory to 25 bytes. But, even then, I was amazed with the amount of code I could cram into the device.

There is provision for enabling weak pull-up resistors.

One real disappointment is that there is no accurate internal RC oscillator. An external RC oscillator may be used, but its frequency varies considerably with the supply voltage. An interesting feature is that when the RC oscillator is used, a programmable output which uses the TMR0 prescaler is available on OCS2/CLKOUT.

All and all, at about \$1.00 in 100 quantities, I found this an amazing device.

Development Tools.

I really don't know of an emulator for the PIC16HV540. In the main, I used my emulator for the 16C505 to develop and debug small modules and then carefully moved them to the 16HV540 using a handful of Windowed EEPROM type devices.

Oscillator Options on OTP Parts.

I also used 20 MHz one time programmable (OTP) devices and was quite surprised to find that the configuration word's oscillator option had been preprogrammed for an HS oscillator.

Bits f_{osc1} and f_{osc2} in the configuration word are used to select the oscillator type;

11 – RC Oscillator

10 – HS (high speed) Oscillator
01 – XT Osc
00 – LP Osc

Note that in my case, these bits had been preprogrammed with the 10 configuration. Recall that in programming a PIC, a one is the erased condition and thus I was able to use these devices in either the HS (10) mode or the LP (00) mode (32 kHz oscillator). However, I could not configure for the RC or XT modes. I was surprised to discover this as I always assumed a 20 MHz part was simply a selected part capable of operating at 20 MHz, but also capable of being run in any slower mode.

Out of curiosity, I also purchased some 4 MHz devices and found the bits were in the RC (11) mode as I would expect and thus they could be configured in any of the oscillator modes.

The windowed CERDIP EEPROM versions could of course also be configured for any of the clock modes.

I do feel that Digikey is a reputable supplier and really don't know if my experience with the 20 MHz OTP parts being preprogrammed for the HS (10) mode was a fluke. I posted a message to the PIC list, but there were no answers.

Thus, if your application requires operation much above 4.0 MHz, buy the 20 MHz version. However, if you are planning to use either the external RC network or a 4.0 MHz resonator (XT mode), buy the 4 MHz version.

DEFS_540.H (PIC16HV540).

The definitions closely follow those of the PIC12C50X and 16C505

However, note that there are two ports, PORTA and PORTB. Note that individual bits within these may either be referred to as either ra3 or porta3 and rb5 or portb5.

As there are two IO ports, I have declared two global variables, DIRA and DIRB. Individual bits within these are of the form dira2 or dirb5.

As with the 12C509, modifying the direction of an IO requires that the bit in DIRB or DIRC be modified and then execution of the TRIS command as shown in the following examples.

```
DIRB = 0xf0           // bits 3, 2, 1 and 0 outputs
#asm
    MOVF DIRB, W
    TRIS PORTB
#endasm

dirb7 = 0;           // make rb7 an output
#asm
    MOVF DIRB, W
    TRIS PORTB
#endasm

dira1 = 0;           // make ra1 an output
#asm
    MOVF DIRA, W
    TRIS PORTA
#endasm

#asm
    BCF DIRA, 1       // make ra1 an output
    MOVF DIRA, W
    TRIS PORTA
```

```
#endasm
```

The PIC16HV540 has two OPTION registers and thus, in the defs_540 file, I have declared two global variables, one for each; OPTIONS1 and OPTIONS2.

The OPTIONS1 register is used for the configuration of TMR0.

```
t0cs = 0;           // use internal osc as source for TMR0
psa = 1;           // assign prescaler to watch dog timer
```

```
#asm
    MOVF OPTIONS1, W
    OPTION
#endasm
```

The OPTIONS2 register consists of bits to configure the operation of the PIC;

not_pcwu. 1 enables pin change wakeup from sleep

not_swdtten. 1 turns off the watch dog timer. 0 turns it on (only if the WDT timer has been enabled in the configuration word)

rl. 1 selects a regulator level of +5 VDC. 0 selects +3 VDC

sl. 1 selects a sleep regulator of +5 VDC. 0 selects +3 VDC when the processor is in SLEEP.

bodl. 1 selects brownout detect level of RL when active and SL when in SLEEP. 0 selects 3.0 VDC.

not_boden. 1 disables brown out detect circuitry.

Note that there is a provision to turn the watch dog timer off and again on in software, but only if the watch dog timer has been enabled in the configuration word (or fuses). I usually specify this when I actually “burn” the chip.

In the routines for the 16HV540, I have usually configured the processor in a function;

```
void config_processor(void) // configure OPTION2 registers
{
    not_pcwu = 1; // wakeup disabled
    not_swdtten = 1;
    rl = 1; // regulated voltage is 5V
    sl = 1; // sleep level same as RL
    not_boden = 1; // brownout disabled
#asm
    MOVF OPTIONS2, W
    TRIS 0x07
#endasm
}
```

Note that when dealing with OPTIONS2, the byte is loaded into the W register followed by TRIS 0x07. Microchip did not give a name to SFR location 0x07 and I followed this convention.

Note that OPTIONS1 dealing with TMR0 is handled quite differently from OPTIONS2. One use the OPTION command and the other uses the TRIS 0x07 command.

```
#asm
    MOVF OPTIONS1, W
    OPTION
#endasm
```

```
#asm
    MOVF OPTIONS2, W
    TRIS 0x07
#endasm
```

```

// DEFS_540.H
//
// Standard definitions for PIC16HV540
//
// Particularly note the declaration of static byte DIRA, DIRB and OPTIONS1
// and OPTIONS2
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

#define byte unsigned int

#define W 0
#define F 1

//----- Register Files -----

#define INDF      =0x00
#define TMR0     =0x01
#define PCL      =0x02
#define STATUS   =0x03
#define FSR      =0x04

#define PORTA    =0x05
#define PORTB    =0x06

static byte DIRA, DIRB, OPTIONS1, OPTIONS2; // note global definition

// Direction Bits
// Note that DIRA and DIRB are files containing the directions of PORTA and PORTB.  Actually
// setting the directions requires a call to set_dirs which moves
// DIRS to W and then TRIS PORTA or TRIS PORTB

#define dira3 =DIRA.3
#define dira2 =DIRA.2
#define diral =DIRA.1
#define dira0 =DIRA.0

#define dirb7 =DIRB.7
#define dirb6 =DIRB.6
#define dirb5 =DIRB.5
#define dirb4 =DIRB.4
#define dirb3 =DIRB.3
#define dirb2 =DIRB.2
#define dirbl =DIRB.1
#define dirb0 =DIRB.0

#define ra3 =PORTA.3
#define ra2 =PORTA.2
#define ral =PORTA.1
#define ra0 =PORTA.0

#define rb7 =PORTB.7
#define rb6 =PORTB.6
#define rb5 =PORTB.5
#define rb4 =PORTB.4
#define rb3 =PORTB.3
#define rb2 =PORTB.2
#define rbl =PORTB.1
#define rb0 =PORTB.0

#define porta3 =PORTA.3 // alternate definitions
#define porta2 =PORTA.2
#define porta1 =PORTA.1
#define porta0 =PORTA.0

#define portb7 =PORTB.7
#define portb6 =PORTB.6
#define portb5 =PORTB.5
#define portb4 =PORTB.4
#define portb3 =PORTB.3
#define portb2 =PORTB.2

```

```

#bit portb1 =PORTB.1
#bit portb0 =PORTB.0

//----- STATUS Bits -----
#bit not_pcwuf =0x03.7
#bit pa0 =0x03.5
#bit not_to =0x03.4
#bit not_pd =0x03.3
//#bit z =0x03.2 // probably not required in C
//#bit dc =0x03.1 // but if they are, come up with
//#bit c =0x03.0 // something more unique than "z" and "c"

//----- OPTION Bits -----
// OPTIONS1 Bits
// Note that OPTIONS1 is a file containing the options.
// Setting the options requires moving OPTIONS1 to W and then
// executing OPTION
#bit t0cs =OPTIONS1.5
#bit t0se =OPTIONS1.4
#bit psa =OPTIONS1.3
#bit ps2 =OPTIONS1.2
#bit ps1 =OPTIONS1.1
#bit ps0 =OPTIONS1.0

//----- OPTION2 Bits
// OPTIONS2 Bits
// File OPTIONS2 is moved to W and then the command TRIS 0x07 is
// executed.
#bit not_pcwu =OPTIONS2.5
#bit not_swkten =OPTIONS2.4
#bit rl =OPTIONS2.3
#bit sl =OPTIONS2.2
#bit bodl =OPTIONS2.1
#bit not_boden =OPTIONS2.0

#define ZZ 2
#define CY 0

```

Programs Previously Discussed. (PIC16HV540).

The following routines were developed but are not discussed in detail as they are very similar to previous routines.

DELAY.C and DELAY.H are the same as those which have previously been discussed.

SER_540.C and SER_540.C implements the bit bang serial on the bit TxData associated with regulated PORTA.

TST_SER.C. Illustrates the use of the various routines in SER_540.C. Use regulated PORTA0.

FLSH_Q.C. Displays a quantity by flashing an LED on regulated PORTA0.

7SEG_1.C. Displays a quantity on a common anode 7-segment display on unregulated PORTB.

RCTIME.C. An implementation of measuring the decay time of an RC network. Uses regulated PORTA2.

1820_1.C. Illustrates an interface with the Dallas DS18S20 to continually perform temperature measurements and display the result on a serial LCD or PC COM port. The DQ lead is on regulated bit PORTA1. Power for the DS18S20 is provided by PORTA3.

Program FRST_ALM.C.

Many of these routines are brought together in an implementation of the frost alarm. The design measures the RC time (PORTA2) and maps this into an alarm threshold. If the user depresses a pushbutton on PORTB7, the alarm threshold is displayed on a 7 segment LED on the low seven bits of PORTB prefaced by the character "A" as in alarm. However, If

the pushbutton is not depressed, a temperature measurement is performed using a DS18S20 on PORTA1 and the result is displayed on the 7-segment LED. Note that the DS18S20 is powered by PORTA3. If the temperature is negative or is less than the alarm threshold, a sonalert on PORTA0 is pulsed.

I am particularly pleased with this implementation as there was a good deal of pain in managing to get it all into the 512 program words. Clear programming and compact code are opposed to one another and getting this into 512 words required a bit of tinkering. But, I would always suggest that folks start by coding clearly and then massage to make the program fit. However, I feel the result is clear enough to present.

Note that the measurement of the RC time and the DS18S20 temperature measurement are coded in main(). This is a practice that would not warrant my approval with my students, and in fact, there were originally separate functions. However, I was surprised to find that coding them in main() actually saved 30 program words.

The dreaded “goto” is used a number of times in mapping the RC time to an alarm threshold. It is used when there is a counter overflow. In this case, the program is in a while loop and one would normally break and later test for a counter overflow. In this case, the value was copied to T_thresh followed by a goto MEAS_THRESH_DONE. A goto also proved fruitful in place of an if, else if, else if, else.

The one regret I have in this design is that the alarm and the current temperature values are displayed in degrees C and people in the United States simply don’t like this. With only 42 program words scattered all over the map, I doubt that I can convert to the preferred degrees F.

```
// FRST_ALM.C (PIC16HV540), CCS PCB
//
// Frost Alarm.
//
// Continually measures temperature using a Dallas DS18S20. The temperature is displayed
// by sequentially displaying each digit on a common anode 7-segment LED.
//
// If the temperature is less than T_threshold, a Sonalert is pulsed. T_threshold is set
// using a potentiometer in an RC network.
//
// When input PORTB7 is at ground, the T_threshold is displayed on the 7-seg LED.
//
// PIC16HV540
//
// PORTA2 (term 1) --- 330 -----
//
//                |           |
//                1.0 uFd     10K Pot
//                |           |
//                |           10K Resistor
//                |           |
//                GRD         GRD
//
// PORTA3 (term 2) ---- ( +5 VDC From PORTA3)
//
//                |
//                4.7K
//                |
// PORTA1 (term 18) ----- DS18S20
// PORTA0 (term 17) ----- To Sonalert
//
//
// PORTB6 (term 12) ---- 1K ----- a seg (term 1) ----- +12 VDC (term 3, 14)
// PORTB5 (term 11) ---- 1K ----- b seg (term 13)
// PORTB4 (term 10) ---- 1K ----- c seg (term 10)
// PORTB3 (term 9) ---- 1K ----- d seg (term 8)
// PORTB2 (term 8) ---- 1K ----- e seg (term 7)
// PORTB1 (term 7) ---- 1K ----- f seg (term 2)
// PORTB0 (term 6) ---- 1K ----- g seg (term 11)
//
//
//                +12 VDC
//                |
//                10K
```

```

//
// GRD ----- \----- PORTB7 (term 13), Ground to display T_thresh.
//
//      a // Layout of 7-seg LED
//      f b
//      g
//      e c
//      d
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

#case

#device PIC16HV540

#include <defs_540.h>
#include <delay.h>

#define TRUE !0
#define FALSE 0

#define _1W_PIN 1

#define MINUS_SIGN 0x01 // segment g only
#define BLANK 0x00 // no segments

void config_processor(void);

void alarm(void);

void display_q(byte q, byte minus_flag);
void display_patt(byte patt);

// 1-wire prototypes
void _lw_init(void);
int _lw_in_byte(void);
void _lw_out_byte(byte d);
void _lw_strong_pull_up(void);

void main(void)
{
    byte T_C, T_F, sign, T_threshold, minus_flag, count_hi, count_lo, tmr0_new, tmr0_old;

    DIRA = 0x0f;
    DIRB = 0xff;

    config_processor(); // configure brownout, WDT, etc

    porta3 = 1; // apply power to DS18S20 via RA3
    dira3 = 0;
#asm
    MOVF DIRA, W
    TRIS PORTA
#endasm
    while(1)
    {
        porta0 = 0; // be sure sonalert is off

        // measure the alarm threshold
        t0cs = 0; // fosc / 4 is clock source
        psa = 1; // prescale assigned to WDT
        ps2 = 0; // 1:1 prescale
        ps1 = 0;
        ps0 = 0;

#asm
        MOVF OPTIONS1, W
        OPTION
#endasm

```

```

    dira2 = 0;    // output
#asm
    MOVF DIRA, W
    TRIS PORTA
#endasm
    porta2 = 1; // charge capacitor

    delay_ms(10);

    count_hi = 0;
    tmr0_old = 0x00;

    TMR0 = 0x00;
#asm
    BSF DIRA, 2
    MOVF DIRA, W
    TRIS PORTA
#endasm
    while(1) // wait for cap to discharge
    {
#asm
        CLRWDT
#endasm
        tmr0_new = TMR0;
        if ((tmr0_new < 0x80) && (tmr0_old >= 0x80)) // there was a roll over
        {
            ++count_hi;
            if (count_hi == 0) // no zero crossing with 65 ms
            {
                T_threshold = 2;
                goto MEAS_THRESH_DONE;
            }
        }
        if (!porta2) // if capacitor discharged below zero crossing
        {
            break;
        }
        // else
        tmr0_old = tmr0_new;
    }

    if (count_hi < 48)
    {
        count_hi = 48;
    }

    if (count_hi > 90)
    {
        T_threshold = 30; // this is to test the alarm
        goto MEAS_THRESH_DONE;
    }

    T_threshold = ((count_hi - 48)/2); // in the range of 0 to 21 degrees C

MEAS_THRESH_DONE:

    if (!portb7) // simply display alarm threshold
    {
        display_patt(0x77); // display an "A" as in alarm
        delay_ms(500);
        display_patt(BLANK);
        delay_ms(500);
        display_q(T_threshold, FALSE);
    }

    else // otherwise, make a temperature meas and display
    {
        _lw_init();
        _lw_out_byte(0xcc); // skip ROM
    }

```

```

    _lw_out_byte(0x44); // perform temperature conversion
    _lw_strong_pull_up();

    _lw_init();
    _lw_out_byte(0xcc); // skip ROM
    _lw_out_byte(0xbe); // read the result

    T_C = _lw_in_byte();
    sign = _lw_in_byte();

    if (sign) // if negative
    {
        T_C = (~T_C) + 1;
    }

    T_C = T_C / 2;
    display_q(T_C, sign);

    if ((sign) || (T_C < T_threshold))
    {
        alarm();
    }
}
delay_ms(1000);
}
}

// The following are standard 1-Wire routines.
void _lw_init(void)
{
#asm
    BSF DIRA, _1W_PIN // high impedance
    MOVF DIRA, W
    TRIS PORTA

    BCF PORTA, _1W_PIN // bring DQ low for 500 usecs
    BCF DIRA, _1W_PIN
    MOVF DIRA, W
    TRIS PORTA
#endasm
    delay_10us(50);
#asm
    BSF DIRA, _1W_PIN
    MOVF DIRA, W
    TRIS PORTA
#endasm
    delay_10us(50);
}

byte _lw_in_byte(void)
{
    byte n, i_byte, temp;

    for (n=0; n<8; n++)
    {
#asm
        BCF PORTA, _1W_PIN // wink low and read
        BCF DIRA, _1W_PIN
        MOVF DIRA, W
        TRIS PORTA

        BSF DIRA, _1W_PIN
        MOVF DIRA, W
        TRIS PORTA

        CLRWDT
        NOP
        NOP
        NOP

```

```

        NOP
#endasm
    temp = PORTA; // now read
    if (temp & (0x01 << _1W_PIN))
    {
        i_byte=(i_byte>>1) | 0x80; // least sig bit first
    }
    else
    {
        i_byte=i_byte >> 1;
    }
    delay_10us(6);
}

return(i_byte);
}

void _lw_out_byte(byte d)
{
    byte n;

    for(n=0; n<8; n++)
    {
        if (d&0x01)
        {
#asm
            BCF PORTA, _1W_PIN // wink low and high and wait 60 usecs
            BCF DIRA, _1W_PIN
            MOVF DIRA, W
            TRIS PORTA

            BSF DIRA, _1W_PIN
            MOVF DIRA, W
            TRIS PORTA
#endasm
            delay_10us(6);
        }

        else
        {
#asm
            BCF PORTA, _1W_PIN // bring low, 60 usecs and bring high
            BCF DIRA, _1W_PIN
            MOVF DIRA, W
            TRIS PORTA
#endasm
            delay_10us(6);
#asm
            BSF DIRA, _1W_PIN
            MOVF DIRA, W
            TRIS PORTA
#endasm
        }
        d=d>>1;
    } // end of for
}

void _lw_strong_pull_up(void) // bring DQ to strong +5VDC
{
#asm
    BSF PORTA, _1W_PIN // output a hard logic one
    BCF DIRA, _1W_PIN
    MOVF DIRA, W
    TRIS PORTA
#endasm

    delay_ms(750);
#asm
    BSF DIRA, _1W_PIN
    MOVF DIRA, W
    TRIS PORTA

```

```

#endasm
}

void display_q(byte q, byte minus_flag)
{
    byte const hex_digit_patts[10] = {0x7e, 0x30, 0x6d, 0x79, 0x33, 0x5b, 0x5f, 0x70, 0x7f, 0x7b};
    byte digit;

    if (minus_flag)
    {
        display_patt(MINUS_SIGN);
        delay_ms(500);
        display_patt(BLANK);
        delay_ms(500);
    }

    digit = q/10; // number of tens

    display_patt(hex_digit_patts[digit]);
    delay_ms(500);
    display_patt(BLANK);
    delay_ms(500);

    digit = q%10;

    display_patt(hex_digit_patts[digit]);
    delay_ms(500);
    display_patt(BLANK);
    delay_ms(500);
}

void display_patt(byte patt)
{
    patt = (~patt) & 0x7f; // convert to negative logic

    DIRB = DIRB & 0x80; // make lowest seven bits outputs

#asm
    MOVF DIRB, W
    TRIS PORTB
#endasm

    PORTB = (PORTB & 0x80) | patt;
}

void alarm(void) // pulse sonalert five times
{
    byte n;

#asm
    BCF PORTA, 0
    BCF DIRA, 0
    MOVF DIRA, W
    TRIS PORTA
#endasm

    for (n=0; n<10; n++)
    {
        porta0 = !porta0;
        delay_ms(100);
    }
}

void config_processor(void) // configure OPTION2 registers
{
    not_pcwu = 1; // wakeup disabled
    not_swkten = 1;
    rl = 1; // regulated voltage is 5V
    sl = 1; // sleep level same as RL
    not_boden = 1; // brownout disabled
}

```

```

#asm
    MOVF OPTIONS2, W
    TRIS 0x07
#endasm

}

#include <delay.c>

```

Program FLSH_ALM.C (PIC16HV540).

This is a simple design to alert a driver in a noisy truck when a turn signal has been left "on" for an extended period of time.

Normally, the +12V for the flashers on each side of the vehicle is at an open and thus the input to the onboard counter (TMR0) is stable at a logic zero (100K pull-down resistor) and the program loops, delaying for nominally one second and then testing to see if the count in TMR0 is zero.

When either flasher is activated, the count in TMR0 over a one second period is not zero and the program exits the "no signal condition loop" and enters a second loop which similarly delays for nominally one second and if the count in TMR0 over the course of a second is non zero, variable secs is incremented.

When variable secs is greater than a defined TIMEOUT, the program exits this second loop and enters a third loop which continually beeps a speaker at 500 Hz for nominally one second.

Note that if the signal condition disappears (TMR0 at zero over one second), the program returns to the "no signal present" loop using a GOTO TOP.

There are flaws in the design. For example, if the driver has the right directional on for 35 seconds and switches to the left signal, TIMEOUT may well be exceeded and an alarm will be generated. However, the alarm is simply a noise maker, not an ejection seat, and the alarm will be terminated when the driver turns off the left signal.

A second flaw is that this system has no way of distinguishing directional signals from emergency flashers where the flashers may well be on for longer than TIMEOUT. However, again, the alarm is simply noise.

Note that the right and left flasher signals are "ored" using diodes.

This program uses nominally 135 of 512 program words in the PIC16HV540.

```

// FLSH_ALM.C (PIC16HV540)
//
//                                     PIC16HV540
//
// Right Flash ----->|----- T0CK1 (term 3)
// Left Flash  ----->|
//
//          100K          PORTB0 (term 6) -----| (---- SPKR ---- GRD
//          |                                     + 47 uFd
//          GRD
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

#case

#device PIC16HV540

#include <defs_540.h>
#include <delay.h>

#define TIMEOUT 45 // adjust timeout in seconds as necessary

```

```

void beep_sec(void);
void setup_tmr0(void);
void config_processor(void);

void main(void)
{
    byte secs;

TOP:
    while(1) // "no turn signal" condition
    {
#asm
        CLRWDT
#endasm
        setup_tmr0();
        TMR0 = 0x00;
        secs = 0;
        delay_ms(1000);
        if (TMR0 != 0)
        {
            break; // turn signal is on
        }
    }

    while(1) // "turn signal on" condition
    {
        setup_tmr0();
#asm
        CLRWDT
#endasm
        TMR0 = 0x00;
        delay_ms(1000);
        if (TMR0 == 0) // condition is not present
        {
            goto TOP;
        }

        // else
        ++secs;
        if (secs > TIMEOUT)
        {
            break; // go to alarm state
        }
    }

    while(1) // alarm condition
    {
        setup_tmr0();
#asm
        CLRWDT
#endasm
        TMR0 = 0x00;
        beep_sec(); // beep for one second
        if (TMR0 == 0) // condition is no longer present
        {
            goto TOP;
        }
        // else, stay in the loop
    }
}

void beep_sec(void)
{
    byte n;
    dirb0 = 0;
#asm
    MOVF DIRB, W // make rb0 an output
    TRIS PORTB
#endasm
    for (n = 0; n<125; n++) // beep 500 Hz for 250 ms

```

```

    {
        portb0 = 1;
        delay_ms(1);
        portb0 = 0;
        delay_ms(1);
    }

    delay_ms(250);

    for (n = 0; n<125; n++) // beep 500 Hz for 250 ms
    {
        portb0 = 1;
        delay_ms(1);
        portb0 = 0;
        delay_ms(1);
    }

    delay_ms(250); // and off for 250 ms
}

void setup_tmr0(void) // configure OPTION2 register
{
    t0se = 1; // rising edge
    t0cs = 1; // source is external TOCK1
    psa = 1; // prescale assigned to WDT
    ps2 = 0; // prescale for WDT set to 1:1
    ps1 = 0;
    ps0 = 0;
#asm
    MOVF OPTIONS1, W
    OPTION
#endasm
}

void config_processor(void) // configure OPTION2 registers
{
    not_pcwu = 0; // wakeup enabled
    not_swkten = 1;
    rl = 1; // regulated voltage is 5V
    sl = 1; // sleep level same as RL
    not_boden = 1; // brownout disabled
#asm
    MOVF OPTIONS2, W
    TRIS 0x07
#endasm
}

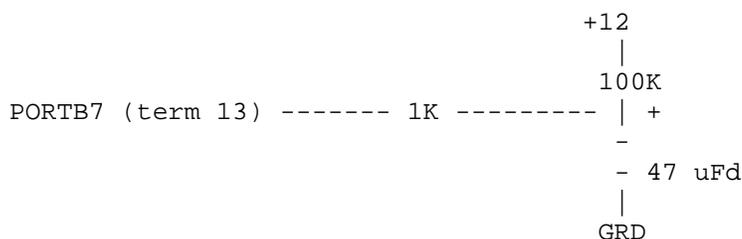
#include <delay.c>

```

Program SL_WAKE.C. (PIV16HV540).

The intent of this program is to illustrate the wake up from SLEEP on slow transition on PORTB7. It also illustrates how the various status bits might be read to determine the cause of a reset.

An external RC network is configured on PORTB7 as illustrated.



The idea is to output a logic zero on PORTB7 and thus discharge the capacitor through the 1K resistor and maintain the ground while performing a task. Upon finishing the task, the processor might make the IO an input (high impedance) and go to sleep.

The capacitor then charges from ground toward +12V and upon crossing some threshold causes a wakeup on slow change. The processor might then ground PORTB7, perform its task and then again go to sleep.

Input PORTB7 seems a bit complex and I am uncertain Microchip has offered many clues on just exactly what the characteristics are for this wake up on slow change implementation. Recall that as an input, all bits on PORTB are TTL compatible with less than 1.0 VDC being an guaranteed logic zero and above 2.0 VDC being a logic one and there is step down circuitry that permits voltages up to V_{supply} to be applied.

However, my observation of the amount of time for each wakeup to occur indicates that the crossover threshold is much higher for the wakeup feature and in looking at the block diagram of the PORTB7 pin in the data sheet, I see unexplained separate circuitry associated with the wakeup on slow pin change. My assumption is that it is a Schmidt Trigger referenced to the supply voltage and the threshold is nominally $0.85 * V_{supply}$.

Thus, the charge time of the above network is;

$$\begin{aligned}t &= R * C * \ln (1.0 / (1.0 - 0.85)) \\ &= R * C * 1.90 \\ &= 100K * 47 \text{ uFd} * 1.90 \\ &= 9.0 \text{ seconds}\end{aligned}$$

My observation does agree with my theory, that, indeed, the above arrangement did cause a wakeup nominally every nine seconds. However, before you go out and field a design on a production run of 25 million devices, it probably would be good to contact Microchip and pin them down as to the characteristics of the wakeup on slow change.

Note that if the above resistor is changed to 1 Meg, the timeout should be nominally 90 seconds.

Thus, in the case of the frost alarm, the design might be powered off a nine volt battery and wakeup every 90 seconds, go through its paces and if there is no alarm, go back to sleep for another 90 seconds.

The advantage of this over the using the watchdog timer to wakeup from sleep is that assuming the watchdog timeout is 30 ms and assuming the prescaler is set to its maximum of 1:128, the maximum timeout is nominally 4.0 seconds. But, with the very simple circuitry shown above, timeouts of 90 seconds and more are easily obtained with current drains of less than 12 uA, far less than running the watch dog timer.

I offer this up as a possible use for the slow wakeup on pin change feature, but of course, it could be used to wake up the processor on any slowly changing signal.

In the following program, the capacitor is discharged and PORTB7 is then made an input and the program goes to sleep. On wake-up, the program reads the not_pcwuf, not_to and not_pd bits to determine the cause of the reset and flashes one of four LEDs to indicate the cause of the reset. The capacitor is then discharged and PORTB7 is made an input.

Thus, if left alone, the wakeup on slow pin change should occur about every nine seconds and the corresponding LED briefly flashed. However if power is removed and restored, the power up LED will flash. If the reset is caused by a momentary ground on /MCLR, the wakeup on /MCLR LED will flash.

Note that aside from the wakeup on slow change, use of the not_to and not_pd bits to determine the cause of a reset are applicable to the other 12-bit core PICs which have been discussed.


```

#endasm
#asm
    SLEEP
#endasm
}

else if (!not_pcwuf && not_to && !not_pd) // entered sleep mode and pin change flag is at zero
{
    // wakeup on PORTB
    portb7 = 0; // ground external RC network
    dirb7 = 0;
#asm
    MOVF DIRB, W
    TRIS PORTB
#endasm
do_wakeup_pin_change_task();

    dirb7 = 1; // remove ground on external RC
#asm
    MOVF DIRB, W
    TRIS PORTB
#endasm
#asm
    SLEEP
#endasm
}

else if (not_pcwuf && not_to && !not_pd)
{
    // MCLR wakeup from sleep
    portb7 = 0; // ground external RC network
    dirb7 = 0;
#asm
    MOVF DIRB, W
    TRIS PORTB
#endasm
do_mclr_from_sleep_task();
    dirb7 = 1; // remove ground on external RC
#asm
    MOVF DIRB, W
    TRIS PORTB
#endasm
#asm
    SLEEP
#endasm
}

else
{
    portb7 = 0; // ground external RC network
    dirb7 = 0;
#asm
    MOVF DIRB, W
    TRIS PORTB
#endasm
do_unknown_reset_task();

    dirb7 = 1; // remove ground on external RC
#asm
    MOVF DIRB, W
    TRIS PORTB
#endasm
#asm
    SLEEP
#endasm
}
}

void do_power_on_task(void)
{
    byte n;

```

```

    porta0 = 0;
    dira0 = 0;
#asm
    MOVF DIRA, W
    TRIS PORTA
#endasm
    for (n = 0; n<10; n++)
    {
        porta0 = 1;
        delay_ms(100);
        porta0 = 0;
        delay_ms(100);
    }
}

void do_mclr_from_sleep_task(void)
{
    byte n;
    portal = 0;
    diral = 0;
#asm
    MOVF DIRA, W
    TRIS PORTA
#endasm
    for (n = 0; n<10; n++)
    {
        portal = 1;
        delay_ms(100);
        portal = 0;
        delay_ms(100);
    }
}

void do_wakeup_pin_change_task(void)
{
    byte n;
    porta2 = 0;
    dira2 = 0; // make an output
#asm
    MOVF DIRA, W
    TRIS PORTA
#endasm

    for (n=0; n<10; n++)
    {
        porta2 = 1;
        delay_ms(100);
        porta2 = 0;
        delay_ms(100);
    }
}

void do_unknown_reset_task(void)
{
    byte n;
    porta3 = 0;
    dira3 = 0;
#asm
    MOVF DIRA, W
    TRIS PORTA
#endasm
    for (n = 0; n<10; n++)
    {
        porta3 = 1;
        delay_ms(100);
        porta3 = 0;
        delay_ms(100);
    }
}

```

```

void config_processor(void) // configure OPTION2 registers
{
    not_pcwu = 0; // wakeup enabled
    not_swtdten = 1;
    rl = 1; // regulated voltage is 5V
    sl = 1; // sleep level same as RL
    not_boden = 1; // brownout disabled
#asm
    MOVF OPTIONS2, W
    TRIS 0x07
#endasm
}

#include <delay.c>

```

Program CLK_OUT.C (PIC16HV540).

The PIC16HV540 does not have an accurate internal RC clock. However, an external RC network on OSC1 / CLKIN may be used as a clock. Microchip provides a number of graphs for suggested R and C values for a specific frequency at a specific V_{supply}. Note that the frequency varies linearly as a function of V_{supply}.

My thinking is that such critical timing tasks as bit bang serial just can't be done with the external RC network. However, I would think the Dallas 1-W could be implemented if the V_{supply} did not vary by more than 25 percent.

When using the external RC network, a clock output is provided at OSC2/CLKOUT which is actually the same as the f_{osc} / 4 input to TMR0 after being prescaled. That is, the frequency may be adjusted using the 8-bit prescaler associated with TMR0.

In the following, the RC network values are such as to provide a nominal f_{osc} of 1000 kHz. Thus, f_{osc} / 4 = 250 kHz. By assigning the prescaler to TMR0 and setting the prescale value to 256, the result is a nominal 1000 Hz signal at CLKOUT.

This tone is turned on by assigning the f_{osc} / 4 as the input to the TMR0 prescaler and turned off by assigning T0CKI as the input source to the prescaler. An alternative would be to adjust the prescale value to 1:2 which would cause an output of 125 kHz which is inaudible to humans.

CLK_OUT.C is used to “sound” out a quantity on a speaker.

```

// CLK_OUT.C (PIC16HV540), CCS PCB
//
// Illustrates the use of CLKOUT to gate tones to "sound out" a quantity.
// Note that the RC Timer configuration on OSC1. I used R = 22K and C = 100 pFd.
// f_osc is nominally 1.0 MHz at +12 VDC. The TMR0 prescaler is set for a prescale
// of 1:256. Thus, f_out = f_osc / 4 / 256 or about 1000 Hz.
//
// When input PORTB7 is at ground, T_threshold is sounded on speaker on output
// CLKOUT. When input PORTB7 is not at ground, the current value of T_C is output on
// the speaker.
//
// In sounding the quantity, a long 500 Hz tone indicates a minus. Each digit is
// sounded as a series of 250 ms beeps with an inter digit delay of 1 second.
//
// PIC16HV540
//
// OSC2/CLKOUT (term 15) -----| (----- SPKR ----- GRD
//                               + 47 uFd
//
//                               + 12 VDC
//                               |
//                               10K
//                               |
// GRD ----- \----- PORTB7 (term 13)

```

```

//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

#case

#device PIC16HV540

#include <defs_540.h>
#include <delay.h>

#define TRUE !0
#define FALSE 0

#define ON !0
#define OFF 0

void config_processor(void);
void beep(byte state);
void beep_q(byte q, byte minus_flag);

void main(void)
{
    byte T_threshold = 34, T_C, minus_flag, n;
    char const T_C_array[5] = {-5, 0, 1, 25, 70};

    DIRA = 0x0f;
    DIRB = 0xff;

    config_processor();

    while(1)
    {
        if(!portb7) // if switch at ground
        {
            beep_q(T_threshold, FALSE);
        }

        else
        {
            for (n = 0; n < 5; n++)
            {
                T_C = T_C_array[n];
                if (T_C & 0x80) // negative
                {
                    minus_flag = TRUE;
                    T_C = (~T_C) + 1;
                }
                else
                {
                    minus_flag = FALSE;
                }
                beep_q(T_C, minus_flag);
            }
            delay_ms(5000/4);
        }
    }
}

void beep_q(byte q, byte minus_flag)
{
    byte n, digit;

    if (minus_flag)
    {
        beep(OFF);
        delay_ms(500/4); // long delay to indicate minus
        beep(ON);
        delay_ms(500/4);
    }
}

```

```

digit = q/10; // number of tens
if (digit) // if non zero
{
    for (n=0; n<digit; n++)
    {
        beep(ON);
        delay_ms(250/4);
        beep(OFF);
        delay_ms(250/4);
    }

    delay_ms(1000/4); // separation between digits
}

digit = q%10;
if (!digit)
{
    digit = 10;
}

for (n=0; n<digit; n++)
{
    beep(ON);
    delay_ms(250/4);
    beep(OFF);
    delay_ms(250/4);
}

delay_ms(1000/4); // separation between digits
}

void beep(byte state)
{
    if (state == ON)
    {
        t0cs = 0; // internal clock enabled
    }
    else
    {
        t0cs = 1;
    }
    ps2 = 1; // prescale of 1:512 plus divide by 2 in TMR0
    ps1 = 1;
    ps0 = 1;
    psa = 0;
#asm
    MOVF OPTIONS1, W
    OPTION
#endasm
}

void config_processor(void) // configure OPTION2 registers
{
    not_pcwu = 1; // wakeup disabled
    not_swtdten = 1;
    rl = 1; // regulated voltage is 5V
    sl = 1; // sleep level same as RL
    not_boden = 1; // brownout disabled
#asm
    MOVF OPTIONS2, W
    TRIS 0x07
#endasm
}

#include <delay.c>

```

PIC12C671/672/CE673/CE674.

These are all 8-pin devices with a 14-bit core which provides for an eight level stack. The 12C671 provides 1024 program words and the 672 provides 2048 words. Both provide 128 bytes of RAM.

They offer four 8-bit analog to digital converters, an accurate internal RC oscillator, TMR0, and weak internal pull-up resistors on GP0, GP1 and GP3.

GP3 may only be used as an input. All others IOs may be configured as either inputs or outputs.

Unlike the previously discussed 12-bit core devices, the PIC12C67X devices provide interrupt capability including interrupt on pin change (GP0, GP1, GP3), interrupt on A/D conversion complete, interrupt on TMR0 rollover and an external interrupt on GP2.

The CE673 and CE674 devices include an internal 16 byte EEPROM, much like the 12CE518 and 519.

It is of course, ridiculous to have a “favorite PIC” as one uses the PIC that best does the job at hand. However, I have had a good deal of fun implementing some very complex designs using these devices which are nominally \$2.30 in 100 quantities.

Development Tools.

In developing this material I used an Advanced Transdata RICE-17A emulator with a PB67X probe. One nice thing about this probe is that it does provide the capability to emulate the internal EEPROM associated with the CE673 and CE674 devices.

However, in the past, I have used and fielded many designs by first writing code for the PIC16F877 and debugging using the low cost Serial In Circuit Debugger (ICD) and then carefully porting my code over to the 12C672. This technique does not include the ability to emulate the internal EEPROM as the implementation of this on the 12CE673/CE674 is very different than the EEPROM associated with the PIC16F87X family (and the PIC16F84 and PIC16F627/628).

I will note however, that using the RICE-17A was a real pleasure.

DEFS_672.H. (PIC12C672).

The PIC12C672 is a 14-bit core device and thus IO directions and the configuration of the OPTION register is unlike the previously discussed 12-bit devices, but rather the same as the PIC16F87X family. That is, the TRIS (or TRISIO) and OPTION_REG may be operated on directly.

The directions for all of the GPIO pins may be defined in a byte operation;

```
TRISIO = 0x34; // 11 1000      make lower three GPIO outputs
```

Or, individual bits may be defined

```
tris5 = 0;    // GP5 is an output
tris4 = 1;    // GP4 is an input
```

The OPTION_REG includes bits for enabling the weak pull-up resistors, defining the transition of an external interrupt on GP2 and various bits for configuring TMR0. Usually, it is far clearer to implement what is desired bit by bit.

```
not_gppu = 0;          // enable weak pullup resistors. This is a part of the OPTION_REG
```

Note that unlike the 12-bit devices, there is no “shadow variable” which must be moved to the W register followed by either an OPTION or TRIS command.

```
// DEFS_672.H
//
// Definition of registers and bits for PIC12C67X
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

#define byte unsigned int

#define W 0
#define F 1

//----- Register Files-----

#define INDF      =0x00
#define TMR0     =0x01
#define PCL      =0x02
#define STATUS   =0x03
#define FSR      =0x04
#define GPIO     =0x05
#define PCLATH   =0x0a
#define INTCON   =0x0b
#define PIR1     =0x0c
#define ADRES    =0x1e
#define ADCON0   =0x1f

#define OPTION_REG =0x81
#define TRISIO    =0x85
#define TRIS     =0x85
#define PIE1      =0x8c
#define PCON     =0x8e
#define OSCCAL    =0x8f
#define ADCON1   =0x9f

// ----- I/O Bits
// lower case for C, uppercase for assembly

#define sda_in =0x05.6 // for 12CE673 and CE674
#define gp5    =0x05.5
#define gp4    =0x05.4
#define gp3    =0x05.3
#define gp2    =0x05.2
#define gp1    =0x05.1
#define gp0    =0x05.0

// ---- TRIS bits
// lower case for C

#define tris5 =0x85.5
#define tris4 =0x85.4
#define tris3 =0x85.3
#define tris2 =0x85.2
#define tris1 =0x85.1
#define tris0 =0x85.0

//----- STATUS Bits -----
// Used in assembly language
#define IRP      7
#define RP1     6
#define RP0     5
#define NOT_TO  4
#define NOT_PD  3
#define Z       2
#define DC      1
#define CY      0

//----- ADCON0 Bits -----
// lower case used by C, upper case by assembly
```

```

#bit adcs1 =0x1f.7
#bit adcs0 =0x1f.6
#bit chs1 =0x1f.4
#bit chs0 =0x1f.3

#bit adgo =0x1f.2
#bit go_done =0x1f.2
#bit adon =0x1f.0

#define ADCS1 7
#define ADCS0 6

#define CHS1 4
#define CHS0 3

#define GO_DONE 2
#define ADON 0

// ----- INTCON Bits -----
// lower case used for C, uppercase for assembly

#bit gie =0x0b.7
#bit peie =0x0b.6
#bit t0ie =0x0b.5
#bit inte =0x0b.4
#bit gpie =0x0b.3
#bit t0if =0x0b.2
#bit intf =0x0b.1
#bit gpif =0x0b.0

#define GIE 7
#define PEIE 6
#define T0IE 5
#define INTE 4
#define GPIE 3
#define T0IF 2
#define INTF 1
#define GPIF 0

// ----- PIR1 Bits -----
// lower case used for C, uppercase for assembly
#bit adif =0x0c.6

#define ADIF 6

// ----- OPTION Bits -----

#bit not_gppu =0x81.7
#bit intedg =0x81.6
#bit t0cs =0x81.5
#bit t0se =0x81.4
#bit psa =0x81.3
#bit ps2 =0x81.2
#bit ps1 =0x81.1
#bit ps0 =0x81.0

#define NOT_GPPU 7
#define INTEDG 6
#define T0CS 5
#define T0SE 4
#define PSA 3
#define PS2 2
#define PS1 1
#define PS0 0

// ----- PIE1 Bits -----
#bit adie =0x8c.6
#define ADIE 6

// ----- PCON Bits -----

```

```

#bit not_por =0x8e.1
#define NOT_POR 1

// ----- OSCCAL Bits -----
#define CAL3    7
#define CAL2    6
#define CAL1    5
#define CAL0    4
#define CALFST  3
#define CALSLW  2

// ----- ADCON1 Bits -----

#bit pcfg2 =0x9f.2
#bit pcfg1 =0x9f.1
#bit pcfg0 =0x9f.0

#define PCFG2  2
#define PCFG1  1
#define PCFG0  0

```

DELAY.C and SER_672.C. (PIC12C672).

These files and the associated header files implement the delay functions and the bit bang serial output at 9600 baud. The implementation is very similar to those discussed previously for the 12-bit core devices.

There is one important difference. With the 12-bit core devices there is limited RAM in bank 0 and thus, the variables `dly`, `ch` and `n` in the bit bang serial implementation were declared as global to insure they were assigned to bank 0. This avoided the extra instructions associated with manipulating bits in the FSR register which would corrupt the critical timing.

However, with the 12C672, there are 96 bytes of RAM in Bank 0 and 32 bytes in Bank 1. Thus, there is a high probability these variables will be assigned to Bank 0 and there may be no need to declare them globally. But, in the past, I have pushed the 12C672 to the limit and did have the problem of these variables being assigned in Bank 1. This then caused the compiler to switch banks by manipulating the RP0 bit in the STATUS register which corrupted the timing.

Declaring global variables as was done in previous implementations is one solution.

Another is to `#include <delay.c>`, not at the end as I usually do for clarity, but just after the implementation of `main()`. The compiler appears to assign variables as they appear in the program and by locating the implementation higher up in your code, hopefully, the variables will be assigned to Bank 0. Its a good idea to check the symbol table.

Program TST_SER.C.

This program illustrates the use of the various functions in `SER_672.C`.

Internal RC Calibration.

One big reason for including this file in the detailed discussion was to illustrate how the internal RC oscillator is calibrated using the 12C672. It differs from the 12C509 and 16C505, is not done automatically by the compiler and can get be troublesome when using an EEPROM.

With the 12C67X family, Microchip has preprogrammed an instruction in the highest program memory address;

```
RETLW xxxxxxxx
```

Where xxxxxxxx is the calibration constant.

For the PIC12C671 and CE673, the highest address is 0x1ff and for the 12C672 and CE674, it is 0x3ff. Thus, the programmer must call this address and then move the value to the OSCCAL register;

```
void calibrate(void)
{
#asm
    CALL 0x03ff // or 0x1ff for the 12C671/CE673
    MOVWF OSCCAL
#endasm
}
```

There is a problem here in using windowed EEPROM parts as after the first erasure, all program addresses will be cleared to 0x3fff which is the opcode for ADDLW 0xff.

Thus, in erasing an EEPROM part, not only have you lost the calibration constant, but the above implementation of the calibration will fail. The above routine will call the code at the highest program memory location, but the ADDLW 0xFF is not a return.

There are many utilities to read the calibration constant from an windowed EEPROM device, but I usually simply put the EEPROM in my PIC programmer and read the value of the highest memory location. It should be of the form;

```
11 01xx kkkk kkkk
```

where x may be either a 0 or a 1 and kkkk kkkk is the eight bit calibration constant.

Thus, you may read;

```
0x372D
```

Note that 0x37 is the op code for RETLW and 0x2D is the calibration constant.

While using this particular EEPROM, you may then temporarily implement the configuration routine as;

```
void calibrate(void)
{
#ifdef _EEPROM
#asm
    MOVLW CAL_CONSTANT
    MOVWF OSCCAL
#endasm
#else
#asm
    CALL 0x03ff // or 0x1ff for the 12C671/CE673
    MOVWF OSCCAL
#endasm
#endif
}
```

where _EEPROM is #defined and CAL_CONSTANT is #defined as 0x2d. Once ready the design is ready for production, one time programmable devices, remember to undefined _EEPROM.

Note that in debugging these routines, my emulator had a precise internal oscillator and thus note that I never call the calibrate routine.

Configuring the A/D Converters.

The PIC12C67X/CE67X family includes four 8-bit A/D converters which are shared with IO bits GP0, GP1, GP2 and GP4. Microchip has provided a considerable amount of flexibility in permitting the user to configure the PIC with no A/Ds, where all of the pins may be used as regular IO, one A/D where the other three pins are used as IO, etc to using all four pins for A/D.

Unfortunately, the PIC boots with the configuration of all four pins configured as A/D inputs. This can cause a bit of wasted time as without changing this configuration, one simply cannot read inputs on GP0, GP1, GP2 and GP4.

Thus, I make it a practice of always setting the A/D configuration early in my main(). In most of these routines I have set the configuration as one A/D on GP0 and the other three pins as standard IO.

```
pcfg2 = 1; // configure A/D for AN0 (GP0) - Not used in this example
pcfg1 = 1; // others as IO
pcfg0 = 0;

// TST_SER.C (PIC12C672), CCS PCB
//
// Illustrates the use of various serial output functions contained in "ser_672.c".
//
// PIC12C672
//
// GP1 (term 6) ----- Serial LCD or PC Com Port
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

#case

#device PIC12C672

#include <defs_672.h>
#include <string.h> // for strcpy

#include <delay.h>
#include <ser_672.h>

#define TxData 1 // use GP1
#define INV // send inverted RS232

#define TRUE !0
#define FALSE 0

void calibrate(void);

void main(void)
{
    byte s[20], n;
    n = 150;

    calibrate(); // no calibrate when using emulator

    pcfg2 = 1; // configure A/D for AN0 (GP0) - Not used in this example
    pcfg1 = 1; // others as IO
    pcfg0 = 0;

    while(1)
    {
        ser_init();
        strcpy(s, "Morgan State");
        // note that CONST string is copied to RAM string
    }
}
```

```

    ser_out_str(s);
    ser_new_line();
    strcpy(s, "University");
    ser_out_str(s);

    ser_new_line();
    ser_hex_byte(n);
    ser_char(' ');
    ser_dec_byte(n, 3); // display in dec, three places

    delay_ms(500);
    ++n;
}
}

void calibrate(void)
{
#asm
    CALL 0x03ff          // 0x1fff for 12C671 and CE673
    MOVWF OSCCAL
#endasm
}

#include <delay.c>
#include <ser_672.c>

```

Program FLSH_Q.C (12C672).

This program illustrates the display of a quantity by flashing an LED. It is similar to that discussed in the context of previously discussed 12-bit core devices.

Program TONE_Q.C (12C672).

This program is functionally similar to that discussed for the 12-bit core devices, but it uses the TMR0 interrupt in implementing the beep() function.

In function beep(), TMR0 is configured for the $f_{osc} / 4$ as the source and the prescaler is assigned to TMR0 with a prescale value of 1:4. Thus, TMR0 is incremented each 4 us and rolls over after $256 * 4$ us or nominally one ms. TMR0 is set to zero, the TMR0 interrupt is enabled ($t0ie = 1$) and general interrupts are enabled ($gie = 1$). The program is interrupted on rollover and in the interrupt service routine, the state of GP1 is toggled and global variable ms is decremented.

In beep(), the program loops until variable ms is at zero and interrupts are then turned off.

Note that the output on GP1 is 1024 us high and 1024 us low resulting in a frequency of 1/2048 us or nominally 488 Hz. Note that the frequency could be changed by adding an offset in the interrupt service routine, for example;

```
TMR0 = TMR0 + 0x80
```

would result in an output on GP1 which is high and then low for 512 us resulting in a tone of 976 Hz. Of course, the means of achieving the specified duration would have to be modified.

Note that variable ms is defined globally so that it is seen by both function beep() and by the TMR0 interrupt service routine.

```

// TONE_Q.C (PIC12C672), CCS PCM
//
// Intended for possible use with frost alarm in place of serial output
// to serial LCD or to PC Com Port.

```

```

//
// When input GP3 is at ground, T_threshold is sounded on speaker on output
// GP1. When input GP3 is not at ground, the current value of T_C is output on
// the speaker.
//
// In sounding the quantity, a long 500 Hz tone indicates a minus. Each digit is
// sounded as a series of 250 ms beeps.
//
//
// GRD --- \---- GP3 (internal weak pull-up)
//
//           GP1 -----||--- SPKR --- GRD
//                   + 47 uFd
//
// Use internal RC oscillator.
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

```

```
#case
```

```
#device PIC12C672
```

```
#include <defs_672.h>
```

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

```
#define TRUE !0
```

```
#define FALSE 0
```

```
void beep(long ms);
void beep_q(byte q, byte minus_flag);
void calibrate(void);
```

```
long ms_count; // used in TMR0 ISR
```

```
void main(void)
```

```
{
    byte T_threshold = 34, T_C, minus_flag, n;
    char const T_C_array[5] = {-5, 0, 1, 25, 70};

    calibrate();

    pcf2 = 1; // configure A/D for AN0 (GP0) - Not used in this example
    pcf1 = 1; // others as IO
    pcf0 = 0;

    while(1)
    {
        not_gppu = 0;

        if(!gp3) // if switch at ground
        {
            beep_q(T_threshold, FALSE);
        }

        else
        {
            for (n = 0; n < 5; n++)
            {
                T_C = T_C_array[n];
                if (T_C & 0x80)
                {
                    minus_flag = TRUE;
                    T_C = (~T_C) + 1;
                }
                else
                {
                    minus_flag = FALSE;
                }
                beep_q(T_C, minus_flag);
            }
        }
    }
}
```

```

        delay_ms(1000);
    }
}

void beep_q(byte q, byte minus_flag)
{
    byte n, digit;

    if (minus_flag)
    {
        beep(500);
        delay_ms(500);
    }

    digit = q/10;        // number of tens
    if (digit)          // if non zero
    {
        for (n=0; n<digit; n++)
        {
            beep(250);
            delay_ms(250);
        }

        delay_ms(500); // separation between digits
    }

    digit = q%10;
    if (!digit)
    {
        digit = 10;
    }

    for (n=0; n<digit; n++)
    {
        beep(250);
        delay_ms(250);
    }

    delay_ms(500);      // separation between digits
}

void beep(long ms)
{
    gpl = 0;
    trisl = 0;

    // configure TMR0
    t0cs = 0; // use internal f_osc
    ps2 = 0; // prescale 1:4, thus, rollover every 256 * 4 usec
    ps1 = 0;
    ps0 = 0;
    psa = 1;
    t0if = 0; // kill any pending interrupt;
    t0ie = 1; // enable TMR0 interrupt
    gie = 1;

    ms_count = ms;
    while(ms_count)
    {
#asm
        CLRWDT
#endasm
    }

    while(gie)
    {
        gie = 0;
    }

    t0ie = 0; // book keeping

```

```

    t0if = 0;

    gp1 = 0;
}

void calibrate(void)
{
#asm
    CALL 0x03ff // 0x3ff for 12C672/CE674, 0x1ff for 12C671/CE673
    MOVWF OSCCAL
#endasm
}

#int_rtcc tmr0_int_handler(void)
{
    gp1 = !gp1;
    --ms_count;
}

#int_default default_int_handler(void)
{
}

#include <delay.c>

```

Program RCTIME.C.

This program is functionally similar to those previously discussed. However, it is implemented quite differently using the TMR0 overflow and external interrupts.

The capacitor is charged via output GP2 through the 330 current limiting resistor.

TMR0 is configured for the $f_{osc} / 4$ clock (1.0 us) source with the prescaler assigned to the watchdog timer resulting in a effective prescale of 1:1. TMR0 interrupt is enabled ($t0ie = 1$). Further, the external interrupt is configured for interrupt on the falling edge ($intedg = 0$) and the external interrupt is enabled ($inte = 1$). GP2 is then made an input, TMR0 is cleared and general interrupts are enabled ($gie = 0$).

Thus, the voltage on the capacitor is falling toward the logic zero threshold (about 1.5 VDC) and TMR0 is counting each us. When TMR0 rolls over, it causes a TMR0 interrupt which increments `count_hi`. `Main()` continually checks `count_hi` and if at 0xff, interrupts are turned off and the program breaks. The assumption is that no negative going transition will occur.

When the capacitor discharges such that the voltage seen by GP2 is interpreted as a logic zero, an external interrupt occurs which sets global variable `ext_int_occ` to TRUE and copies the value of TMR0 to `count_lo`.

When `main()` sees that variable `ext_int_occ` is TRUE, it disables interrupts and breaks with the count in `count_hi` and `count_lo`. These values are then displayed on the serial LCD or PC COM port.

Note that variables `ext_int_occ`, `count_hi` and `count_lo` are declared globally to permit the interrupt service routines to operate on them.

```

// RCTIME.C (PIC12C672), CCS-PCM
//
// Charges capacitor in parallel with a resistor on GP2/INT for one second.
// PORTC1 is then made an input and capacitor discharges through capacitor and
// and the time for detection of a one to zero transition is measured.
//
// Result in number of 1 usec ticks is displayed in hex on serial LCD
// on GP0.

```



```

gie = 1;

while(1) // wait for cap to discharge
{
    if(count_hi == 0xff) // no negative going transition
    {
        while(gie)
        {
            gie = 0;
        }
        count_lo = 0xff;
        break;
    }
    if (ext_int_occ) // there was a negative going transition
    {
        while(gie)
        {
            gie = 0;
        }
        break;
    }
}
ser_init();
ser_hex_byte(count_hi);
ser_hex_byte(count_lo);
delay_ms(1000);
}

#int_rtcc tmr0_int_handler(void)
{
    ++count_hi;
}

#int_ext ext_int_handler(void)
{
    count_lo = TMR0;
    ext_int_occ = TRUE;
}

#int_default default_int_handler(void)
{
}

#include <delay.c>
#include <ser_672.c>

```

Files _1_WIRE.C and _1_WIRE.H. (PIC12C672).

For the 12C672, the low level 1-wire routines were implemented in a separate file that may be #included by the using routine.

Although these are functionally the same as those presented previously, the 12C672 has considerably more program memory and one can afford to write more robust code that permits the IO pin to be passed to each of the low level functions. Of course, when one runs out of program memory, it might be time to rethink whether robust code is an end in itself.

In each function, the specified IO pin is used to calculate a mask_one and a mask_zero byte.

For examples, if the specified IO pin is 2, mask_one is calculated as binary 0000 0100. That is, a one only in the bit 2 position. Similarly, mask_zero is calculated as 1111 1011, a logic zero only in the bit 2 position.

This permits the specified bit to be forced to either a logic one or to a logic zero without affecting the other bits by either oring the current value with mask_one or anding the current value with mask_zero.

For example, using the specified IO of 2;

```
GPIO = GPIO & mask_0;           // make GP2 a zero
TRIS = TRIS & mask_0;           // make GP2 an output
TRIS = TRIS | mask_1;           // make GP2 an input
```

In addition, a specified byte may be read as to whether it is a logic one or zero;

```
if (TEMP & mask_one)
{
    // it's a one
}
else
{
    // it's a zero
}
```

Note that in this implementation, mask_one and mask_zero are calculated;

```
mask_1 = 0x01<<io;
mask_0 = ~mask_1;
```

Another approach which I used in the PIC16F87X Routines was to use constant arrays to define the values of mask_one and mask_zero which is perhaps a marginally better approach.

```
// _1_wire.c (PIC16C672), CCS PCM
//
// Standard 1-Wire routines.
//
// copyright, Peter H. Anderson, Baltimore, MD, Aug, '00

void _lw_init(byte io)
{
    _lw_pin_hi(io);    // be sure DQ is high
    _lw_pin_low(io);
    delay_10us(50);    // low for 500 us

    _lw_pin_hi(io);
    delay_10us(50);
}

byte _lw_in_byte(byte io)
{
    byte n, i_byte, temp, mask_1, mask_0;
    mask_1 = 0x01<<io;
    mask_0 = ~mask_1;

    for (n=0; n<8; n++)
    {
        GPIO = GPIO & mask_0;
        TRIS = TRIS & mask_0;           // momentary low
        TRIS = TRIS | mask_1;
#asm
        NOP
        NOP
        NOP
        NOP
#endasm
        temp = GPIO;           // read port
        if (temp & mask_1)
        {
```

```

        i_byte=(i_byte>>1) | 0x80;    // least sig bit first
    }
    else
    {
        i_byte=i_byte >> 1;
    }
    delay_10us(6);
}
return(i_byte);
}

void _lw_out_byte(byte io, byte d)
{
    byte n, mask_1, mask_0;
    mask_1 = 0x01 << io;
    mask_0 = ~mask_1;

    for(n=0; n<8; n++)
    {
        if (d&0x01)
        {
            GPIO = GPIO & mask_0;
            TRIS = TRIS & mask_0;           // momentary low
            TRIS = TRIS | mask_1;         // and then high for 60 us
            delay_10us(6);
        }

        else
        {
            GPIO = GPIO & mask_0;           // low for 60 us
            TRIS = TRIS & mask_0;
            delay_10us(6);
            TRIS = TRIS | mask_1;
        }
        d=d>>1;
    }
}

void _lw_pin_hi(byte io)
{
    byte mask_1;
    mask_1 = 0x01 << io;
    TRIS = TRIS | mask_1;
}

void _lw_pin_low(byte io)
{
    byte mask_0;
    mask_0 = ~(0x01 << io);
    GPIO = GPIO & mask_0;    // 0 in bit sensor
    TRIS = TRIS & mask_0;
}

void _lw_strong_pull_up(byte io)
{
    byte mask_1, mask_0;
    mask_1 = 0x01 << io;
    mask_0 = ~mask_1;

    GPIO = GPIO | mask_1;    // output a hard logic one
    TRIS = TRIS & mask_0;

    delay_ms(750);
    TRIS = TRIS | mask_1;
}

```

Program 1820.C (PIC12C672).

The intent in presenting this routine is to illustrate how the various functions in `_1_WIRE.C` are used to interface with a DS18S20 in measuring a temperature.

Note that although the approach taken in the implementation of the one-wire routines permits multiple IO pins to each be interfaced with a DS18S20, only one DS18S20 on GP2 is used in this program.

```
// 1820_1.C PIC12C672, CCS PCM
//
// Measures temperature using a Dallas DS1820 on GP2 and displays result
// using RS232 serial (9600 inverted) on GP1.

// PIC12C672
//
//                               +5VDC
//                               |
//                               4.7K
//                               |
// GP2 (term 5) ----- DQ of DS18S20
// GP1 (term 6) ----- To Ser LCD or PC Com Port
//
// Debugged using RICE-17A Emulator, July 19, '01.
//
// copyright, Peter H. Anderson, Elmore. VT, July, '01

#case

#device PIC12C672

#include <defs_672.h>

#include <delay.h>
#include <ser_672.h>
#include <_1_wire.h>

#define TxData 1 // use GP1
#define INV // send inverted RS232

#define TRUE !0
#define FALSE 0

#define _1W_PIN 2

void calibrate(void);

void main(void)
{
    byte T_C, sign;

    // calibrate(); // do not use this function during emulation

    pcf2 = 1; // configure A/D for AN0 (GP0)- Not used in this example
    pcf1 = 1; // others as IO
    pcf0 = 0;

    while(1)
    {
        ser_init();
        _1w_init(_1W_PIN);
        _1w_out_byte(_1W_PIN, 0xcc); // skip ROM
        _1w_out_byte(_1W_PIN, 0x44); // perform temperature conversion

        _1w_strong_pull_up(_1W_PIN);

        _1w_init(_1W_PIN);
        _1w_out_byte(_1W_PIN, 0xcc); // skip ROM
        _1w_out_byte(_1W_PIN, 0xbe);

        T_C = _1w_in_byte(_1W_PIN);
        sign = _1w_in_byte(_1W_PIN);
    }
}
```

```

    if (sign) // negative
    {
        T_C = ~T_C + 1;
        ser_char('-');
    }

    T_C = T_C / 2;

    if (T_C > 99) // unlikely
    {
        ser_dec_byte(T_C, 3);
    }
    else if (T_C > 9)
    {
        ser_dec_byte(T_C, 2);
    }
    else
    {
        ser_dec_byte(T_C, 1);
    }
    delay_ms(1000);
}

void calibrate(void)
{
    #asm
        CALL 0x03ff          // 0x3fff for 12C672/CE674, 0x1fff for 12C671/CE673
        MOVWF OSCCAL
    #endasm
}

#include <delay.c>
#include <ser_672.c>
#include <_1_wire.c>

```

Program AD.C (PIC12C672).

The intent of this program is to illustrate an A/D conversion using the A/D interrupt.

In function `ad_meas()`, the A/D module is configured for an A/D on GP0 and digital IO on GP1, GP2 and GP4 by setting the `pcfg` bits to 110. The A/D is configured to internal clock source which is different that the `f_osc / 4` clock by setting the `adcs` bits to 11. The A/D is turned on and the channel is selected using the `chs` bits.

In this example, I opted to use interrupts. Thus, the A/D interrupt is enabled (`adie = 1`). Note that this is a peripheral interrupt and Microchip provided a means to either enable or disable all of the peripheral interrupts which can be a real time waster if you forget to set the `peie` bit (`peie = 1`). The `gie` bit is set to one. The A/D conversion is initiated by setting bit `go_done` (or `adgo`) and the processor enters the SLEEP mode.

When the interrupt occurs, the processors wakes, turns off interrupts and reads the result from `ADRESH`.

The advantage in using the interrupt to wake the processor from SLEEP approach is that while the A/D conversion is being performed the processor is effectively turned off which eliminates switching noise. This does require that the internal oscillator be used as the external or internal `f_osc / 4` clock is turned off when entering the SLEEP mode.

Although it is not true in this routine, I have wasted many hours on other routines where I was performing an A/D conversion using this SLEEP technique while also running a timer. Unfortunately, the SLEEP turns off the timer.

```

// Program A_D.C, PIC12C672, CCS PCM
//
// Continually perform A/D conversions on AN0 on GP0.

```

```

// Displays results on Serial LCD or PC at 9600 baud using GP1.
//
//   +5                               PIC16C672
//   |
// 10K Pot <----- GP0/AN0 (term 7)
//   |                               GP1 (term 6) -----> To Ser LCD or PC COM Port
// GRD
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

#case

#device PIC12C672

#include <defs_672.h>
#include <string.h> // for strcpy

#include <delay.h>
#include <ser_672.h>

#define TxData 1 // use GP1
#define INV // send inverted RS232

#define TRUE !0
#define FALSE 0

byte ad_meas(void);
void calibrate(void);

main()
{
    byte ad_val;

    not_gppu = 0; // enable weak pullups
    // calibrate(); // do not use for emulation
    ser_init();
    while (1) // continually
    {
        ad_val = ad_meas();
        ser_hex_byte(ad_val);
        ser_new_line();
        delay_ms(1000);
    }
} // end of main

byte ad_meas(void)
{
    pcf2=1; // config for 1 analog channel on GP0
    pcf1=1;
    pcf0=0;

    adcs1=1;
    adcs0=1; // internal RC

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

    chs1 = 0; // channel 0
    chs0 = 0;

    delay_ms(1);

    adif=0; // reset the flag
    adie=1; // enable interrupts
    peie=1;
    gie=1;
    delay_10us(10);
    go_done = 1;
#asm
    CLRWDT
    SLEEP // turn of most of PIC to reduce noise during conversion

```

```

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

    adie=0; // not really necessary, but good practice.
    peie=0;
    return(ADRES);
}

void calibrate(void)
{
#asm
    CALL 0x03ff
    MOVWF OSCCAL
#endasm
}

#int_ad a_d_int_handler(void)
{
}

#int_default default_int_handler(void)
{
}

#include <delay.c>
#include <ser_672.c>

```

Program FRST_ALM.C (PIC12C672).

This is an implementation of the frost alarm.

Note that unlike other implementations which used RC time to determine the setting of a potentiometer and then maps this into an alarm threshold, this implementation uses an A/D on GP0. Thus, only a potentiometer is required.

For A/D readings greater than 0xf0, the alarm threshold is set at 81 degrees F to permit testing of the sonalert at room temperatures. Otherwise, the A/D reading is mapped into a temperature in the range of 32 to 44 degrees F.

Unlike previous implementations, this displays the alarm threshold and the current temperature in degrees F. There sure is a more efficient way to convert from C to F than using floats as I did, but , if one has the program memory, what's the point. Compact code becomes important when one runs out of program memory.

Note that in measuring the temperature using the DS18S20, the value of the temperature T_C and sign is passed by reference. That is, pointers. In this case, there really is no advantage to this approach, the measurement function could well have returned the temperature as a float.

This is the fourth implementation of the frost alarm. Its always hard to say which is the best design as one must usually trade off cost, the number of parts, and the quality of the IO. In this implementation, the IO suffers in being simply a flashing LED, but points are scored, at least in the United States as the output is in degrees F. I happen to like this implementation as it is simple with a bare minimum of parts.

```

// FRST_ALM.C PIC12C672, CCS PCM
//
// Frost Alarm.
//
// When pushbutton on GP3 is at ground, a pot on GP0 is read. This is mapped into a
// T_threshold in the range of 32 - 42 degrees or 81 degrees F and the value is displayed
// by flashing LED_ALM on GP5.
//

```

```

// Otherwise, the temperature is measured using a DS18S20 on GP2 and this is displayed
// by flashing LED_TEMP on GP4.
//
// If the measured temperature is less than the alarm temperature, a sonarlert is pulsed.
//
//
//          +5                PIC16C672
//          |
//          10K Pot <----- GP0/AN0 (Term 7)  (Alarm Threshold Adjust)
//          |
//          GRD
//
//                      +5V
//                      |
//                      4.7K
//                      |
//          GP2 (term 5) ----- DS18S20
//
//                      BiColor LED
//          GP4 (term 3) ----->|-----
//                      |_____ 330 ____ GRD
//                      |
//          GP5 (term 2) ----->|----
//
//          GP1 (term 6) -----> Sonalert
//
//          GRD ----- \--- GP3 (term 4)
//
//
// copyright, Peter H. Anderson, Baltimore, MD, Aug, '01

```

```

#case

#device PIC12C672

#include <defs_672.h>

#include <delay.h>
#include <_1_wire.h>

#define TRUE !0
#define FALSE 0

#define _1W_PIN 2
#define LED_ALM 5
#define LED_TEMP 4

void calibrate(void);

void alarm(void);
byte meas_threshold(void);
void meas_temperature(byte *p_T_C, byte *p_sign);

void flash_q(byte LED, byte q, byte minus_flag);

void main(void)
{
    byte T_threshold, T_C, T_F, sign;
    float T_C_float, T_F_float;

//    calibrate(); // not when using emulator

    pcf2 = 1; // configure A/D for AN0 (GP0) - Not used in this example
    pcf1 = 1; // others as IO
    pcf0 = 0;

    while(1)
    {
        not_gppu = 0;
        gp1 = 0; // be sure sonalert is off

        T_threshold = meas_threshold();

```

```

if (!gp3) // set the alarm threshold
{
    flash_q(LED_ALM, T_threshold, FALSE);
    delay_ms(1000);
}

else
{
    meas_temperature(&T_C, &sign);

    if (sign) // its negative
    {
        T_C = (~T_C) + 1;
        T_C_float = (float) T_C * -0.5;
    }
    else
    {
        T_C_float = (float) T_C * 0.5;
    }
    T_F_float = 1.8 * T_C_float + 32.0;
    T_F = (byte) T_F_float;

    flash_q(LED_TEMP, T_F, FALSE);

    if (T_F < T_threshold) // pulse sonalert and LED
    {
        alarm();
    }

    else
    {
        delay_ms(2000);
    }
} // end of else
} // end of while 1
}

byte meas_threshold(void)
{
    byte T_thresh, adval;

    pcf2=1; // config for 1 analog channel on GP0
    pcf1=1;
    pcf0=0;

    adcs1=1;
    adcs0=1; // internal RC

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

    chs1 = 0; // channel 0
    chs0 = 0;

    delay_ms(1);

    adif=0; // reset the flag
    adie=1; // enable interrupts
    peie=1;
    gie=1;
    delay_10us(10);
    adgo = 1;
#asm
    CLRWDI
    SLEEP // turn of most of PIC to reduce noise during conversion
#endasm
    while(gie) // turn off interrupts
    {
        gie=0;
    }

    adie=0; // not really necessary, but good practice.

```

```

    peie=0;
    adval = ADRES;

    if (adval > 0xf0) // this is for testing at room temperature
    {
        return(81); // degrees F
    }

    else
    {
        T_thresh = (0xf0 - adval) / 20 + 32; // 32 - 44 degrees
        return(T_thresh);
    }
}

void meas_temperature(byte *p_T_C, byte *p_sign)
{
    _lw_init(_1W_PIN);
    _lw_out_byte(_1W_PIN, 0xcc); // skip ROM
    _lw_out_byte(_1W_PIN, 0x44); // perform temperature conversion

    _lw_strong_pull_up(_1W_PIN);

    _lw_init(_1W_PIN);
    _lw_out_byte(_1W_PIN, 0xcc); // skip ROM
    _lw_out_byte(_1W_PIN, 0xbe);

    *p_T_C = _lw_in_byte(_1W_PIN);
    *p_sign = _lw_in_byte(_1W_PIN);
}

void alarm(void)
{
    byte n;
    tris1 = 0;

    for (n = 0; n < 5; n++)
    {
        gp1 = 1;
        delay_ms(100);
        gp1 = 0;
        delay_ms(100);
    }
}

void flash_q(byte LED, byte q, byte minus_flag)
{
    byte n, digit;

    #asm
        BCF GPIO, LED_ALM // make LED pins output logic zeros
        BCF GPIO, LED_TEMP
        BCF TRISIO, LED_ALM
        BCF TRISIO, LED_TEMP
    #endasm

    if (minus_flag)
    {
        if (LED == LED_ALM)
        {
            #asm
                BSF GPIO, LED_ALM
            #endasm
        }
        else
        {
            #asm
                BSF GPIO, LED_TEMP
            #endasm
        }
    }
}

```

```

        delay_ms(500); // long delay to indicate minus
#asm
    BCF GPIO, LED_ALM
    BCF GPIO, LED_TEMP
#endasm
    delay_ms(1000);
}

    digit = q/10; // number of tens
    if (digit) // if non zero
    {
        for (n=0; n<digit; n++)
        {
            if (LED == LED_ALM)
            {
#asm
                BSF GPIO, LED_ALM
#endasm
            }
            else
            {
#asm
                BSF GPIO, LED_TEMP
#endasm
            }
            delay_ms(250);
#asm
                BCF GPIO, LED_ALM
                BCF GPIO, LED_TEMP
#endasm
            delay_ms(250);
        }

        delay_ms(1000); // separation between digits
    }

    digit = q%10;
    if (!digit)
    {
        digit = 10;
    }

    for (n=0; n<digit; n++)
    {
        if (LED == LED_ALM)
        {
#asm
            BSF GPIO, LED_ALM
#endasm
        }
        else
        {
#asm
            BSF GPIO, LED_TEMP
#endasm
        }
        delay_ms(250); // long delay to indicate minus
#asm
            BCF GPIO, LED_ALM
            BCF GPIO, LED_TEMP
#endasm
        delay_ms(250);
    }

    delay_ms(1000); // separation between digits
}

void calibrate(void)
{
#asm

```

```

CALL 0x03ff
MOVWF OSCCAL
#endasm
}

#int_ad a_d_int_handler(void)
{
}

#int_default default_int_handler(void)
{
}

#include <delay.c>
#include <_1_wire.c>

```

Program PWM_1.C (PIC12C672).

The 12C67X family does not have a capture and compare module, but it is quite easy to implement a continuous 8-bit “back ground” PWM using TMR0 and the TMR0 interrupt..

This involves bringing the IO output to a logic one and writing the two's complement of the duty cycle to TMR0 and on interrupt, bringing the IO output to a logic zero and writing the duty cycle to TMR0.

For example, assume the duty cycle is 32.. The two's complement is $256 - 32$ or 224. Thus, the output is brought to a logic one and TMR0 is preloaded with 224 such that 32 TMR0 clicks later, the TMR0 rolls over and causes an interrupt. The output is brought to a logic zero and 32 is loaded into TMR0 such that 224 TMR0 clicks later, TMR0 again rolls over, etc.

In this program I opted to configure the TMR0 prescaler for 1:4. Thus, with a $f_{ocs} / 4$ of 1.0 MHz, the input to TMR0 is 4 us and the periodicity is $256 * 4$ us or nominally one ms.

There is a problem with very low or very high duty cycles. For example, if the duty is one, TMR0 would be loaded with 255 such that interrupt would occur after one clock tick. However, this is but 4 us and it takes far longer than 4 us for the processor to execute the code which eventually lands it in the TMR0 interrupt service routine and during this latency, the IO is at a logic one. Thus, rather than being an output high for 4 us, you may well find it is high for 60 us. Similarly, if the duty is 255, the off time would be but one 4 us clock tick and although the intent might be for the IO to be low for 4 us, it would, in fact, be low for the 4 us plus the time to handle the interrupt which is considerably longer than 4 us.

Thus, in program PWM_1.C, for duty cycles less than 0x10, I assume the user wants the IO to be at a constant logic zero and if the duty is greater than 0xf0 the user desires the IO to be a constant logic one.

In this program, the A/D converter on GP0 is continually read. If the A/D value is less than 0x10 or if input GP3 is open, PWM output on GP1 is at a continuous logic zero. If the A/D value is greater than 0xf0 the PWM output is held at a continual logic one. Otherwise, a PWM output having a duty equal to the A/D value is output.

Note that I used this design to control large pumps at a landfill. Prior to this, they were adjusting the 8-amp pumps using a potentiometer in series with the battery and the pump!

```

// PWM_1.C (PIC12C672), CCS PCM
//
// Generates PWM at nominally 1 kHz on GP1 controlled by the setting of a
// potentiometer on GP0 / AN0.
//
// If input GP3 is at logic one (or open) output on GP1 is zero.
//
// For A/D results lower than 0x10, the output on GP1 is maintained at zero.
// Similarly, for A/D results above 0xf0, the output is maintained at a constant

```

```

// logic one.
//
// +5                PIC16C672
// |
// 10K Pot <----- GP0/AN0 (term 7)
// |
// GRD
//
// GRD ---- \----- GP3 (term 4) (if open, PWM output is zero)
//
//                GP1 (term 6) (PWM output) ----- To FET or similar
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

```

```
#case
```

```
#device PIC12C672
```

```
#include <defs_672.h>
```

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

```
#define TRUE !0
```

```
#define FALSE 0
```

```
void calibrate(void);
void setup_ad(void);
void setup_tmr0(void);
```

```
byte on_time, off_time;
```

```
void main(void)
```

```
{
  byte adval;
  // calibrate(); // do calibrate when using emulator
  not_gppu = 0;
```

```
  gp1 = 0; // PWM off
  trisl = 0;
```

```
  setup_ad();
  setup_tmr0();
```

```
  while(1)
```

```
{
#asm
  CLRWDT
```

```
#endasm
  adgo = 1;
  while(adgo)
  {
  }
```

```
  adval = ADRES;
  off_time = adval;
  on_time = (~adval) + 1;
```

```
  if ((adval < 0x10) || (gp3)) // if low value or if gp3 is high
```

```
{
  while(gie)
  {
    gie = 0;
  }
  gp1 = 0; // off
  t0ie = 0;
  t0if = 0;
}
```

```
else if (adval > 0xf0)
{
  while(gie)
```

```

        {
            gie = 0;
        }
        gp1 = 1; // full on
        t0ie = 0;
        t0if = 0;
    }

    else
    {
        t0ie = 1;
        gie = 1;
    }
}

void setup_ad(void)
{
    pcfg2=1; // config for 1 analog channel on GP0
    pcfg1=1;
    pcfg0=0;

    adcs1=1;
    adcs0=1; // internal RC

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

    chs1 = 0; // channel 0
    chs0 = 0;
    delay_ms(1);
}

void setup_tmr0(void)
{
    t0cs = 0; // f_osc / 4
    psa = 0; // prescaler assigned to TMR0
    ps2 = 0; // prescale is 1:4, 1.024 ms rollover
    ps1 = 0;
    ps0 = 1;
}

void calibrate(void)
{
    #asm
    CALL 0x03ff
    MOVWF OSCCAL
    #endasm
}

#int_rtcc tmr0_int_handler(void)
{
    gp1 = !gp1;
    if(gp1)
    {
        TMR0 = on_time;
    }
    else
    {
        TMR0 = off_time;
    }
}

#int_default default_int_handler(void)
{
}

#include <delay.c>

```

Program INT_EE_1.C (PIC12CE672).

This program illustrates how to write to and read from the 16 byte internal EEPROM in the PIC12CE673/CE674 devices. It is similar to that discussed for the PIC12CE518 and CE519.

In fact, these programs were debugged using the Advanced Transdata RICE17A equipped with a PB67X probe which does support the internal I2C EEPROM and the general ideas were then mapped over to the PIC12CE518/CE519.

Note that the SCL and SDA bits on bits 7 and 6 of the GPIO are always handled as a pair in a byte type write to the GPIO. All attempts at using a bit type approach proved unsuccessful.

Note that variable `high_two_bits` is defined globally and the high two bits are used to define the states of SCL and SDA.

Thus, making SDA high or low is a matter of setting or clearing bit 6 in `high_two_bits` and then oring this into the bit 7 and 6 positions of GPIO.

```
void i2c_internal_high_sda(void)
{
    high_two_bits = high_two_bits | 0x40; // X1
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}

void i2c_internal_low_sda(void)
{
    high_two_bits = high_two_bits & 0x80; // X0
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}

// Program INT_EE_1.C, (12CE674)
//
// Illustrates how to write to and read from internal EEPROM.
//
// Note that I was unable to successfully implement this using;
// scl = 0; sda = 1; etc. Rather, a global variable "high_two_bits" was
// defined and bits 7 and 6 were set and cleared as appropriate and then
// output to GPIO as GPIO = GPIO & 0x3f | high_two_bits.
//
// My guess as to why the scl=0; sda=1; approach does not work is that
// I assume there are no TRIS bits associated with bits 7 and 6 of GPIO.
// Thus, in implementing scl=0; the PIC is actually reading SCL and SDA
// from the internal EEPROM and making SCL a zero. Unfortunately, this
// may affect the state of SDA. For example, assume SDA was a zero and
// the statement SCL=1 is implemented. But, in doing so, the PIC may
// read a one from the internal EEPROM on SDA and the result is that not
// only is SCL a one, but SDA is also a one.
//
// Serial LCD is connected to GP0. Serial data is 9600 baud, inverted.
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

#case

#device PIC12CE674

#include <defs_672.h>
#include <string.h> // for strcpy

#include <delay.h>
#include <ser_672.h>

#define TxData 0 // use GP0
#define INV // send inverted RS232
```

```

#define TRUE !0
#define FALSE 0

byte i2c_internal_eeprom_random_read(byte adr);
void i2c_internal_eeprom_random_write(byte adr, byte dat);

// standard I2C routines for internal EEPROM
byte i2c_internal_in_byte(byte ack);
void i2c_internal_out_byte(byte o_byte);

void i2c_internal_start(void);
void i2c_internal_stop(void);
void i2c_internal_high_sda(void);
void i2c_internal_low_sda(void);
void i2c_internal_high_scl(void);
void i2c_internal_low_scl(void);

byte high_two_bits; // bits 7 and 6 of GPIO

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

    //OSCCAL=_READ_OSCCAL_DATA();

    pcfg2 = 1;; // GP0, GP1, GP2, GP4 configured as general purpose IOs
    pcfg1 = 1;
    pcfg0 = 1;

    high_two_bits = 0xc0; // bits 7 and 6 at one
    GPIO = GPIO & 0x3f | high_two_bits;

    ser_init();

    while(1)
    {
        mem_adr=0x00;
        for(n=0; n<4; n++) // write four bytes to EEPROM
        {
            dat = n+10;
            i2c_internal_eeprom_random_write(mem_adr, dat);
            ++mem_adr;
        }
        // now, read the data back and display

        mem_adr=0x00;
        for(n=0; n<4; n++)
        {
            dat = i2c_internal_eeprom_random_read(mem_adr);
            ser_hex_byte(dat);
            ser_char(' ');
            ++mem_adr;
        }
        ser_new_line();
        delay_ms(1000);
    } // end of while
}

byte i2c_internal_eeprom_random_read(byte adr)
{
    byte d;
    i2c_internal_start();
    i2c_internal_out_byte(0xa0);
    i2c_internal_out_byte(adr);

    i2c_internal_start();
    i2c_internal_out_byte(0xa1);
    d = i2c_internal_in_byte(0); // no ack prior to stop
    i2c_internal_stop();
    return(d);
}

```

```

}

void i2c_internal_eeprom_random_write(byte adr, byte dat)
{
    i2c_internal_start();
    i2c_internal_out_byte(0xa0);
    i2c_internal_out_byte(adr);
    i2c_internal_out_byte(dat);
    i2c_internal_stop();
    delay_ms(25); // wait for byte to burn
}

byte i2c_internal_in_byte(byte ack)
{
    byte i_byte, n;
    i2c_internal_high_sda();
    for (n=0; n<8; n++)
    {
        i2c_internal_high_scl();
        if (sda_in)
        {
            i_byte = (i_byte << 1) | 0x01; // msbit first
        }
        else
        {
            i_byte = i_byte << 1;
        }
        i2c_internal_low_scl();
    }
    if (ack)
    {
        i2c_internal_low_sda();
    }
    else
    {
        i2c_internal_high_sda();
    }
    i2c_internal_high_scl();
    i2c_internal_low_scl();

    i2c_internal_high_sda();
    return(i_byte);
}

void i2c_internal_out_byte(byte o_byte)
{
    byte n;
    for(n=0; n<8; n++)
    {
        if(o_byte&0x80)
        {
            i2c_internal_high_sda();
            //ser_char('1'); // used for debugging
        }
        else
        {
            i2c_internal_low_sda();
            //ser_char('0'); // used for debugging
        }
        i2c_internal_high_scl();
        i2c_internal_low_scl();
        o_byte = o_byte << 1;
    }
    i2c_internal_high_sda();

    i2c_internal_high_scl(); // provide opportunity for slave to ack
    i2c_internal_low_scl();
    //ser_new_line(); // for debugging
}

void i2c_internal_start(void)

```

```

{
    i2c_internal_low_scl();
    i2c_internal_high_sda();
    i2c_internal_high_scl(); // bring SDA low while SCL is high
    i2c_internal_low_sda();
    i2c_internal_low_scl();
}

void i2c_internal_stop(void)
{
    i2c_internal_low_scl();
    i2c_internal_low_sda();
    i2c_internal_high_scl();
    i2c_internal_high_sda(); // bring SDA high while SCL is high
    // idle is SDA high and SCL high
}

void i2c_internal_high_sda(void)
{
    high_two_bits = high_two_bits | 0x40; // X1
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}

void i2c_internal_low_sda(void)
{
    high_two_bits = high_two_bits & 0x80; // X0
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}

void i2c_internal_high_scl(void)
{
    high_two_bits = high_two_bits | 0x80; // 1X
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}

void i2c_internal_low_scl(void)
{
    high_two_bits = high_two_bits & 0x40; // 0X
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}

#include <delay.c>
#include <ser_672.c>

```

Program FIRST_TM.C. (PIC12CE672).

This program counts the number of times the PIC has been booted. Location 0x0 in the on-board I2C EEPROM is used to store the count and locations 0xc – 0xf are used to establish if this is the first time the processor has been booted.

Function `first_time()` reads locations 0xc – 0xf and compares with patterns 0x78, 0x87, 0xa5 and 0x5a. The idea is that is very unlikely that EEPROM would be shipped from the factory having this very same pattern. If, this pattern is not found, it is written to the four locations and the `first_time()` function returns TRUE. However, if the patterns are found, the function returns FALSE. That is, this is not the first time the PIC has been booted.

If it is the first time, a counter in EEPROM memory location 0x00 is set to 0x01. If, it not the first time, the counter is fetched from EEPROM, incremented and saved back to EEPROM.

Note that the idea of a `first_time` function is very important in using this I2C EEPROM and there is no way to initialize values when “burning” these PICs. Rather, it must be done in the program.

```

// Program FIRST_TM.C, (12CE674)
//
// Illustrates the use of internal EEPROM.
//
// Four locations 0x0c - 0x0f in the internal EEPROM are used to determine if this
// is the first time the processor has been booted by checking for the pattern
// 0x78, 0x87, 0xa5, 0x5a. If the pattern is not present, the routine writes this
// pattern to locations 0x0c - 0x0f.
//
// Location 0x00 in EEPROM is used to store the number of times the processor has been booted.
//
// Serial LCD is connected to GP0. Serial data is 9600 baud, inverted.
//
// copyright, Peter H. Anderson, Elmore, VT, July, '01

```

```
#case
```

```
#device PIC12CE674
```

```
#include <defs_672.h>
```

```
#include <string.h> // for strcpy
```

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

```
#include <ser_672.h>
```

```
#define TxData 0 // use GP0
```

```
#define INV // send inverted RS232
```

```
#define TRUE !0
```

```
#define FALSE 0
```

```
byte i2c_internal_eeprom_random_read(byte adr);
```

```
void i2c_internal_eeprom_random_write(byte adr, byte dat);
```

```
// standard I2C routines for internal EEPROM
```

```
byte i2c_internal_in_byte(byte ack);
```

```
void i2c_internal_out_byte(byte o_byte);
```

```
void i2c_internal_start(void);
```

```
void i2c_internal_stop(void);
```

```
void i2c_internal_high_sda(void);
```

```
void i2c_internal_low_sda(void);
```

```
void i2c_internal_high_scl(void);
```

```
void i2c_internal_low_scl(void);
```

```
byte first_time(void);
```

```
void calibrate(void);
```

```
byte high_two_bits; // bits 7 and 6 of GPIO
```

```
void main(void)
```

```
{
```

```
    byte count;
```

```
    pcfg2 = 1; // configure A/D for AN0 (GP0) - Not used in this example
```

```
    pcfg1 = 1; // others as IO
```

```
    pcfg0 = 0;
```

```
//    calibrate(); // do not use this function during emulation
```

```
    high_two_bits = 0xc0; // bits 7 and 6 at one
```

```
    GPIO = GPIO & 0x3f | high_two_bits;
```

```
    ser_init();
```

```
    if (first_time())
```

```
    {
```

```
        count = 1;
```

```
    }
```

```

else
{
count = i2c_internal_eeprom_random_read(0);
++count;
}

i2c_internal_eeprom_random_write(0, count);
ser_dec_byte(count, 3);

while(1)
{
}
}

byte first_time(void)
{
byte i, j;
byte const patts[4] = {0x78, 0x87, 0xa5, 0x5a};

for (i = 0; i < 4; i++)
{
if (i2c_internal_eeprom_random_read(i+0x0c) != patts[i])
// if locations 0x0c - 0x0f different from patts
{
for (j = 0; j < 4; j++)
{
i2c_internal_eeprom_random_write(j+0x0c, patts[j]);
// program the patts at 0x0c - 0x0f
}
return(TRUE);
}
}
return(FALSE);
}

// Note that the implementations of i2c_internal_eeprom_random_read(),
// i2c_internal_eeprom_random_write() and the low level i2c routines have been
// deleted in this discussion.

void calibrate(void)
{
#asm
CALL 0x03ff
MOVWF OSCCAL
#endasm
}

#include <delay.c>
#include <ser_672.c>

```

Program NV_TOTAL.C (PIC12CE674).

This program uses the internal EEPROM to implement a 32-bit totalizer. That is, it continually counts the number of events appearing on input T0CKI/GP2 and periodically saves this to four bytes in EEPROM. Thus, if power is momentarily lost, the total count is preserved, at least most of it.

A means of clearing the total count in EEPROM is implemented by the processor checking the state of GP3 on boot and if it is at ground, the four bytes are set to zero.

TMR0 is configured for input on T0CKI / GP2 and the prescaler is assigned to the watchdog timer. Thus, TMR0 increments with each event appearing on the T0CKI input. An interrupt occurs when TMR0 rolls over which sets the global variable tmr0_int_occ. When main() sees that this variable is TRUE, it adds 256 to the current count in EEPROM.

The program also uses the interrupt on change (termed #int_rb in the CCS compiler) to interface with a master processor. The master changes the state of GP1 which interrupts the processor and on noting the interrupt, the PIC fetches the

residual count off TMR0, adds this to the count in EEPROM and sends the 32 bit totalized count in hexadecimal format to the master using 9600 baud serial. Note that the response to the master changing the state on GP1 is not instantaneous.

Note that the 32-bit quantity is implemented using a structure consisting of two longs. Structures must be passed by reference. That is, the address of the beginning of the structure is passed.

For example, the call;

```
add_long_to_count32(&count32, 256);
```

adds 256 to the current value of count32.

Function read_count32() reads the current value of the four bytes in EEPROM and write_count32() writes the count to EEPROM. Thus, adding a quantity to the 32 bit quantity in EEPROM involves reading the count, adding to the count and then writing it back to EEPROM.

This routine was verified and in fact, a program quite like this was done for a programmer friend in Finland who is using it to monitor the counts of a tipping bucket to measure rainfall. Actually, a bit of an overkill. However, this is a rather complex routine and before you put it in millions of vehicles, it probably is a good idea to look at my implementation with a 100X magnifying glass.

Note that in this program I opted to update the EEPROM only after 256 events on T0CKI and whenever the master requested the current count. There is a tradeoff here in just how often to update the EEPROM and the limited endurance of the EEPROM. For example, if nominally 10 events are expected per second, a rollover and a write to EEPROM occurs about every 25 secs. Assuming an EEPROM endurance of 2 million, a write every 25 secs translates into a life of 578 days.

But, for rainfall, it probably doesn't make sense to wait for 256 tips of the bucket to write to EEPROM. Perhaps, every ten tips, which corresponds to 0.1 inches of rainfall. Note that this may easily be implemented by preloading TMR0 with 246 such that rollover occurs ten tips later.

```
// Program NV_TOTAL.C (12CE674), CCS PCB
//
// An implementation of a non-volatile totalizer.
//
// On boot checks GP3 input. If at ground, sets four bytes in EEPROM to zero.
//
// Uses TMR0 to count external events. On rollover, an interrupt occurs and 256
// is added to the four byte quantity stored in EEPROM.
//
// A change on input GP1, such as a PC sending a character, causes the program to
// read the 4-byte quantity from EEPROM, add whatever residual count is in the
// TMR0 counter and send the 32 bit quantity to the PC or similar in hex format.
//
// An application might be an outboard processor to monitor the counts of a tipping
// bucket in a rain fall measurement device.
//
// Note that on power failure, only residual content of TMR0 is lost. In this
// example, the full 256 count of TMR0 is used. This might be reduced as discussed
// in the text.
//
//                                     PIC12C672
//
// Count Source ----- GP2 (T0CKI) Use external pullup if necessary.
//
// ----- GP3 (/CLEAR)
// PC Com Port ---- 22K ----- GP1 (Send)
// GP0 -----> To PC Com Port
//
```

```

// copyright, Peter H. Anderson, Elmore, VT, July, '01

#case

#device PIC12CE674

#include <defs_672.h>
#include <string.h> // for strcpy

#include <delay.h>
#include <ser_672.h>

#define TxData 0 // use GP0
#define INV // send inverted RS232

#define TRUE !0
#define FALSE 0

struct long32
{
    unsigned long h;
    unsigned long l;
};

void add_long_to_count32(struct long32 *p_q, long v);
void write_count32(struct long32 *p_q);
void read_count32(struct long32 *p_q);

byte i2c_internal_eeprom_random_read(byte adr);
void i2c_internal_eeprom_random_write(byte adr, byte dat);

// standard I2C routines for internal EEPROM
byte i2c_internal_in_byte(byte ack);
void i2c_internal_out_byte(byte o_byte);

void i2c_internal_start(void);
void i2c_internal_stop(void);
void i2c_internal_high_sda(void);
void i2c_internal_low_sda(void);
void i2c_internal_high_scl(void);
void i2c_internal_low_scl(void);

byte high_two_bits; // bits 7 and 6 of GPIO
byte tmr0_int_occ, gpio_change_int_occ;

void main(void)
{
    byte x, current_count;
    struct long32 count32;

//    calibrate(); // do not use this function during emulation

    pcf2 = 1; // GP0, GP1, GP2, GP4 configured as general purpose IOs
    pcf1 = 1;
    pcf0 = 1;

    not_gppu = 0; // enable weak pullups

    ser_init();

    high_two_bits = 0xc0; // bits 7 and 6 at one
    GPIO = GPIO & 0x3f | high_two_bits;

    if (!gp3) // if on boot, the clear input is at zero
    {
        count32.h = 0x0000;
        count32.l = 0x0000;
        write_count32(&count32);
    }

    gp3 = 0; // make it an output 0 to avoid any int on change

```

```

tris3 = 0;

// configure TMR0
t0cs = 1; // input on T0CKI (GP2)
t0se = 1;
ps2 = 0;
ps1 = 0;
ps0 = 0;
psa = 1; // prescaler assigned to watch dog timer
TMR0 = 0;
t0if = 0;
t0ie = 1;

// config for int on change
gpif = 0;
gpie = 1;
x = GPIO;

tmr0_int_occ = FALSE;
gpio_change_int_occ = FALSE;

while(1)
{
    gie = 1; // enable interrupts
    if (tmr0_int_occ) // there was a roll over
    {
        read_count32(&count32);
        add_long_to_count32(&count32, 256);
        write_count32(&count32);
        tmr0_int_occ = FALSE;
    }

    if (gpio_change_int_occ) // it must be the GO lead
    {
        while(gie)
        {
            gie = 0;
        }
        current_count = TMR0; // fetch the residual count in TMR0
        TMR0 = 0x00;
        read_count32(&count32);
        add_long_to_count32(&count32, (long) current_count); // add any residual
        write_count32(&count32);

        x = (byte)(count32.h >> 8);
        ser_hex_byte(x);
        x = (byte)(count32.h);
        ser_hex_byte(x);

        x = (byte)(count32.l >> 8);
        ser_hex_byte(x);
        x = (byte)(count32.l);
        ser_hex_byte(x);

        ser_new_line();
        gpio_change_int_occ = FALSE;

        gpif = 0;
        gie = 1;
    }
}
}

void add_long_to_count32(struct long32 *p_q, long v) // adds v to the 32 bit structure
{
    unsigned long old;
    old = p_q->l;
    p_q->l = p_q->l + v;
    if (p_q->l < old) // there was an overflow
    {
        ++(p_q->h);
    }
}

```

```

}
}

void write_count32(struct long32 *p_q) // save the 32-bit quantity to EEPROM
{
    byte adr, *p_byte;
    p_byte = (byte *) p_q; // p_byte now points to beginning of structure
    for (adr = 0; adr < 4; adr++)
    {
        i2c_internal_eeprom_random_write(adr, *(p_byte + adr));
    }
}

void read_count32(struct long32 *p_q) // read the 32-bit quantity
{
    byte adr, *p_byte;
    p_byte = (byte *) p_q; // p_byte now points to beginning of structure
    for (adr = 0; adr < 4; adr++)
    {
        *(p_byte + adr) = i2c_internal_eeprom_random_read(adr);
    }
}

byte i2c_internal_eeprom_random_read(byte adr)
{
    byte d;
    i2c_internal_start();
    i2c_internal_out_byte(0xa0);
    i2c_internal_out_byte(adr);

    i2c_internal_start();
    i2c_internal_out_byte(0xa1);
    d = i2c_internal_in_byte(0); // no ack prior to stop
    i2c_internal_stop();
    return(d);
}

void i2c_internal_eeprom_random_write(byte adr, byte dat)
{
    i2c_internal_start();
    i2c_internal_out_byte(0xa0);
    i2c_internal_out_byte(adr);
    i2c_internal_out_byte(dat);
    i2c_internal_stop();
    delay_ms(25); // wait for byte to burn
}

byte i2c_internal_in_byte(byte ack)
{
    byte i_byte, n;
    i2c_internal_high_sda();
    for (n=0; n<8; n++)
    {
        i2c_internal_high_scl();
        if (sda_in)
        {
            i_byte = (i_byte << 1) | 0x01; // msbit first
        }
        else
        {
            i_byte = i_byte << 1;
        }
        i2c_internal_low_scl();
    }
    if (ack)
    {
        i2c_internal_low_sda();
    }
    else
    {

```

```

    i2c_internal_high_sda();
}
i2c_internal_high_scl();
i2c_internal_low_scl();

i2c_internal_high_sda();
return(i_byte);
}

void i2c_internal_out_byte(byte o_byte)
{
    byte n;
    for(n=0; n<8; n++)
    {
        if(o_byte&0x80)
        {
            i2c_internal_high_sda();
            //ser_char('1'); // used for debugging
        }
        else
        {
            i2c_internal_low_sda();
            //ser_char('0'); // used for debugging
        }
        i2c_internal_high_scl();
        i2c_internal_low_scl();
        o_byte = o_byte << 1;
    }
    i2c_internal_high_sda();

    i2c_internal_high_scl(); // provide opportunity for slave to ack
    i2c_internal_low_scl();
    //ser_new_line(); // for debugging
}

void i2c_internal_start(void)
{
    i2c_internal_low_scl();
    i2c_internal_high_sda();
    i2c_internal_high_scl(); // bring SDA low while SCL is high
    i2c_internal_low_sda();
    i2c_internal_low_scl();
}

void i2c_internal_stop(void)
{
    i2c_internal_low_scl();
    i2c_internal_low_sda();
    i2c_internal_high_scl();
    i2c_internal_high_sda(); // bring SDA high while SCL is high
    // idle is SDA high and SCL high
}

void i2c_internal_high_sda(void)
{
    high_two_bits = high_two_bits | 0x40; // X1
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}

void i2c_internal_low_sda(void)
{
    high_two_bits = high_two_bits & 0x80; // X0
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}

void i2c_internal_high_scl(void)
{
    high_two_bits = high_two_bits | 0x80; // 1X
    GPIO = (GPIO & 0x3f) | high_two_bits;
}

```

```

    delay_10us(5);
}

void i2c_internal_low_scl(void)
{
    high_two_bits = high_two_bits & 0x40; // 0X
    GPIO = (GPIO & 0x3f) | high_two_bits;
    delay_10us(5);
}

void calibrate(void)
{
    #asm
        CALL 0x03ff
        MOVWF OSCCAL
    #endasm
}

#int_rb gpio_change_int_handler(void)
{
    byte x;
    x = GPIO;
    if (x & 0x02) // if GP1 is at a logic one
    {
        gpio_change_int_occ = TRUE;
    }
}

#int_rtcc tmr0_int_handler(void)
{
    tmr0_int_occ = TRUE;
}

#int_default default_handler(void)
{
}

#include <delay.c>
#include <ser_672.c>

```

Program 32KHZ.C (PIC12C672).

The idea of this routine was to illustrate how the 12C672 might be operated in a low power mode using a 32.768 kHz watch type crystal as a clock source.

The context of the design is to monitor that a ventilation fan in a barn is on. This might be done by measuring the output of a Motorola MPX5010 differential pressure sensor using an A/D converter. If the reading is above some threshold value, all is well. However, to avoid false alarms, the program was written such that the pressure must be below the threshold for a period of time, in this example, 30 seconds. Thus, we have the “no alarm” state and the “timing to alarm” state. If at any time while “timing to alarm”, the pressure rises above the threshold, the program returns to the “no alarm” state.

However, if the pressure is low for the full 30 seconds, the program enters the alarm state where a soanalert alarm is pulsed on output GP1. The program remains in this state as long as the low pressure condition exists.

If the pressure returns, the program exits the alarm state and enters the “timing from alarm” state. If the pressure remains above the threshold for a full 10 secs, the program loops back to the “idle – no alarm” state. However, if at any time during this 10 secs, the pressure again goes low, the program returns immediately to the alarm state.

If, at any time, input GP3 is grounded, the program simply loops. This might be used during installation.

The practicality of this approach is certainly subject to debate. Sometimes, its fun to define a problem just to see how difficult it is. Actually, it's a breeze with the use of the dreaded "goto". Otherwise, it seems to become a bit more puzzling a problem. My point is that the "goto" does have a place. And, of course, there is no good reason for using anything as precise as the 32.768 kHz crystal to achieve a very accurate 1.00000 second time base in this kind of application.

An LED on output GP2 was used while debugging to more easily determine that the program was working. The number of blips identifies what state the PIC is executing;

```

Idle (GP3 at ground)    1-blip
Idle - No Alarm         2
Timing to Alarm        3
Alarm                  4
Timing from Alarm      5

```

In using a 32.768 kHz watch crystal, $f_{osc} / 4$ is 8192 Hz. Using a prescale value of 32, the input to TMR0 is 256 Hz. Thus, the periodicity of TMR0 is one second.

Thus, in achieving the "timing to alarm" and "timing from alarm", TMR0 is configured for the $f_{osc} / 4$ as the clock source. The prescaler is assigned to TMR0 and it is configured for a prescale of 1:32. TMR0 interrupts are enabled. Thus, each second, a rollover occurs and the elapsed time (global variable `et_secs`) is incremented in the interrupt service routine. The program continually loops until the elapsed time is greater than the timeout value.

Note that in using the 32.768 crystal, the DELAY.C routines developed for a 4.0 MHz clock are useless. A very primitive `delay_32kHz_ms()` was implemented using a simple loop. I am uncertain this is any too accurate.

Note that in performing the A/D conversion, the SLEEP and A/D interrupt were not used as SLEEP would cause TMR0 to stop. Rather, the A/D conversion is initiated by bringing bit `adgo` to a one and waiting until this bit goes to a zero.

```

// 32KHZ.C 12C672, (CCS PCM)
//
// Pressure Alarm. Intended to monitor that a venilation fan is continually on.
//
// Monitors pressure using an Motorola MPX5010 by measuring the A/D value on GP0. If the
// pressure drops below ALARM_THRESH for TIME_OUT seconds, an alarm is sounded. If at
// any time during the timing the pressure returns to normal, the program returns to the idle -
// no alarm state.
//
// The program remains in the alarm state if the pressure remains low. If the pressure returns
// for 10 seconds, the program returns to the idle - no alarm state.
//
// If at any time, a pushbutton on input GP3 is depressed, the program returns to the HOME state and
// simply loops.
//
// A LED on GP2 is used for debugging to determine which state the program is in;
//
//     Idle (GP3 at ground)    1-blip
//     Idle - No Alarm         2
//     Timing to Alarm        3
//     Alarm                  4
//     Timing from Alarm      5
//
// Uses 32.768 kHz external clock.
//
// MPX5010 ----- GP0/AN0 (term 7)
//
// GRD ----- GP3 (term 4) (internal weak pullup)
//           GP1 (term 6) ----- ALARM ----- GRD
//
//           GP2 (term 5) ----- 330 ----->|----- GRD
//

```

```
// copyright, Peter H. Anderson, Baltimore, MD, Aug, '01
```

```
#case
```

```
#device PIC12C672
```

```
#include <defs_672.h>
```

```
#define TRUE !0
```

```
#define FALSE 0
```

```
#define TEST
```

```
#ifndef TEST
```

```
void test_blip(byte num_blips); // used for debugging
```

```
#endif
```

```
byte ad_meas(void);
```

```
void alarm_burst(byte num_bursts);
```

```
void calibrate(void);
```

```
void delay_32kHz_ms(byte ms);
```

```
byte et_secs;
```

```
#define TIME_OUT 30 // number of secs for alarm
```

```
#define ALARM_THRESH 0x50
```

```
void main(void)
```

```
{
```

```
    byte ad_val;
```

```
    not_gppu = 0;
```

```
HOME:
```

```
    while(!gp3) // idle if gp3 is at ground
```

```
    {
```

```
#ifndef TEST
```

```
    test_blip(1);
```

```
#endif
```

```
#asm
```

```
    CLRWDT
```

```
#endasm
```

```
    t0ie = 0;
```

```
    adie = 0;
```

```
    while(gie)
```

```
    {
```

```
        gie = 0;
```

```
    }
```

```
    gp1 = 0;
```

```
    }
```

```
IDLE:
```

```
    while(1) // operational state - no alarm
```

```
    {
```

```
#ifndef TEST
```

```
    test_blip(2);
```

```
#endif
```

```
#asm
```

```
    CLRWDT
```

```
#endasm
```

```
    if (!gp3)
```

```
    {
```

```
        goto HOME;
```

```
    }
```

```
    gp1 = 0; // be sure alarm is off
```

```
    tris1 = 0;
```

```
    ad_val = ad_meas();
```

```
    if (ad_val < ALARM_THRESH)
```

```
    {
```

```

        break;
    }
}

TIMING_TO_ALARM:
// set up TMR0
t0cs = 0; // fosc / 4 is clock source
psa = 0; // prescale assigned to TMR0
ps2 = 1; // 1:32 prescale
ps1 = 0;
ps0 = 0;

et_secs = 0;
TMR0 = 0;
t0if = 0;
t0ie = 1;
gie = 1;

while(1) // operational state - alarm, but not timed out
{
#ifdef TEST
    test_blip(3);
#endif
#asm
    CLRWDT
#endasm
    if (!gp3)
    {
        goto HOME;
    }
    gp1 = 0; // be sure alarm is off
    trisl = 0;
    ad_val = ad_meas();
    if (ad_val >= ALARM_THRESH)
    {
        goto HOME;
    }

    if (et_secs >= TIME_OUT)
    {
        while(gie)
        {
            gie = 0;
        }
        break; // to ALARM
    }
}

ALARM_STATE: // alarm state
while(1)
{
#ifdef TEST
    test_blip(4);
#endif
#asm
    CLRWDT
#endasm
    if (!gp3)
    {
        goto HOME;
    }
    ad_val = ad_meas();
    if (ad_val >= ALARM_THRESH) // alarm condition appears to be resolved
    {
        break; // to timing from alarm
    }
    alarm_burst(10);
}

TIMING_FROM_ALARM:
// set up TMR0

```

```

t0cs = 0; // fosc / 4 is clock source
psa = 0; // prescale assigned to TMR0
ps2 = 1; // 1:32 prescale
ps1 = 0;
ps0 = 0;

et_secs = 0;
TMR0 = 0;
t0if = 0;
t0ie = 1;
gie = 1;

while(1) // alarm cleared
{
#ifdef TEST
    test_blip(5);
#endif

#asm
    CLRWDT
#endasm
    if (!gp3)
    {
        goto HOME;
    }
    ad_val = ad_meas();
    if (ad_val < ALARM_THRESH) // alarm condition again present
    {
        goto ALARM_STATE;
    }
    if (et_secs >= 10)
    {
        goto HOME; // alarm is cleared
    }
}

#ifdef TEST
void test_blip(void num_blips)
{
    byte n;

    gp2 = 0;
    tris2 = 0; // make it an output

    for (n=0; n<num_blips; n++)
    {
        gp2 = 1;
        delay_32kHz_ms(100);
        gp2 = 0;
        delay_32kHz_ms(100);
    }

    delay_32kHz_ms(250);
    delay_32kHz_ms(250);
}
#endif

void alarm_burst(byte num_bursts)
{
    byte n;
    gp1 = 0;
    tris1 = 0;
    for(n = 0; n < num_bursts; n++)
    {
        gp1 = 1;
        delay_32kHz_ms(100);
        gp1 = 0;
        delay_32kHz_ms(100);
    }
}

```

```

void delay_32kHz_ms(byte ms)
{
    while(ms--)
    {
#asm
        CLRWDT
        NOP
        NOP
        NOP
        NOP
        NOP
#endasm
    }
}

byte ad_meas(void)
{
    pcf2=1; // config for 1 analog channel on GP0
    pcf1=1;
    pcf0=0;

    adcs1=1;
    adcs0=1; // internal RC

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

    chs1 = 0; // channel 0
    chs0 = 0;

    delay_32kHz_ms(1);

    go_done = 1;
    while(go_done)
    {
    }
    return(ADRES);
}

#int_rtcc tmr0_int_handler(void)
{
    ++et_secs;
}

#int_default default_int_handler(void)
{
}

```