Accurate Low-Cost Expanded-Scale Analog Voltmeter with Novel Charge/Discharge Indication for Monitoring Single Lithium-Ion Cells

Jonathan Scott
School of Engineering
The University of Waikato
Hamilton, New Zealand
Email: jonathanscott@ieee.org

***Abstract*—This manuscript presents a very low-cost circuit using a PIC12F683 microcontroller and an LM285 bandgap voltage reference that provides an expanded-scale to an analog panel meter suitable for monitoring a single Lithium-ion battery. Charge or discharge of the cell is indicated by periodic twitches of the needle to the left or the right. The design is especially suitable for small devices subject to intermittant use or that recharge by power-harvesting. The design is open-source, with circuit, board layout, and source code provided.**

I. Introduction

When a rechargeable battery in a small appliance is used regularly and delivers power such as to be exhausted in a few hours, the user is generally satisfied to be given some warning of low remaining capacity, and to have equipment shut down when the battery is judged to be "flat". Where the equipment runs for longer periods, and especially where it is subject to uncertain current drain or periodic partial recharging from the likes of small solar cells, the user desires some indication of the state of the power reserve.

Much research appears in the literature on methods for determining state-of-charge (SoC) in general by means of Coulomb counting, terminal voltage, and battery impedance, or a combination, for example see [1] and [2]. Nevertheless, in the case of low-duty batteries, the EMF is the most reliable approach. If you want to know what your battery is doing, you are best to look at the terminal voltage, assuming you are not using large currents. The two cited references assume this as their way of finding the "true" state-of-charge against which to measure various alternative methods.

For low-duty applications, terminal voltage is close to the open-circuit EMF. In the case of lead-acid batteries, open-circuit EMF varies roughly linearly with SoC, in the region where it is best to operate the battery. For Lithium-ion cells, the function is not so simple. It is roughly linear with small slope from 20% to 50%, but below 20% the voltage falls increasingly rapidly, and above 50% the voltage rises rather more steeply with capacity. When this characteristic is viewed in real time on a meter, it is reasonably easy to develop a feel for the cell SoC. In "first-world temperatures", meaning twenty-something degrees Celcius, starting at about 4.2V for "fully charged" the battery falls down towards about 3.7–3.8V around half charge, and then falls slowly towards 3.6V at something like 20%. Cell voltage falls quite quickly below 20%, and healthy usage suggests you stay above 3.4 or 3.5V, although manufacturers keen to maximise their vital statistics will cite "flat" voltages as low as 2.75V, or even lower.

Again for low-duty application, it is reassuring to know if you are gaining or losing SoC. Are the solar cells delivering more power than you are using? In other words, something like a center-zero meter provides useful information. Older readers will recall the days when cars, whose electrics were not as reliable as today, featured a center-zero ammeter on the dashboard or in the instrument cluster, occasionally with non-linear scales, so a driver would know if the battery was gaining or losing, and could resolve larger and smaller rates. [3]

The design in this manuscript achieves an indication of current flow by periodically deflecting the meter needle to the left for discharge or right for charge. These small deflections or "twitches" vary in intensity and give a quick visual hint of gains or losses, larger or smaller.

II. Hardware

A low-cost 8-bit microcontroller is teamed with a low-cost bandgap reference. The circuit runs on the output voltage from a single lithium cell. This prototype was tested with a variety of 18650-type cells. The microcontroller drives a panel meter using its pulse-width modulation (PWM) output. It also measures its 2.5V reference voltage, and the voltage across a resistor in series with the panel meter, so the micro can determine the current in the meter in spite of varying supply voltage. All battery current is drawn through a small resistor in parallel with two back-to-back Schottky diodes, and the meter measures the voltage dropped across this combination too. It is provided with an LED that indicates when the circuit boots correctly, and confirms when the button is pressed. There is a momentary-contact press button to allow the current measurement to calibrate zero. This last is only needed if you want milliamp precision in the charge indication, and the circuit will function well even without a zero calibration of the current measurement.

The power consumption of prototypes was about 750μA with a 100μA meter movement. Versions with a 1mA meter have larger current draw, dominated by the meter movement itself.

Figure 1 shows the reprinted scale of the meter with suitable colour codes for use with a single cell.

Figure 2 depicts the circuit of the meter.

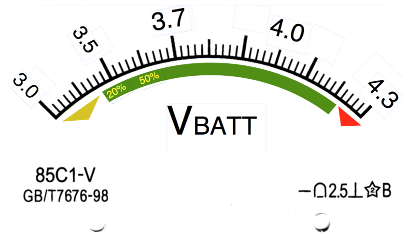

Fig. 1. Reprinted face of the panel meter to provide the expanded scale. Colour regions on the scale indicate typical operating regions for a cell. The scale is “pessimistic” to the extent that some capacity remains at the lower end of the green bar.

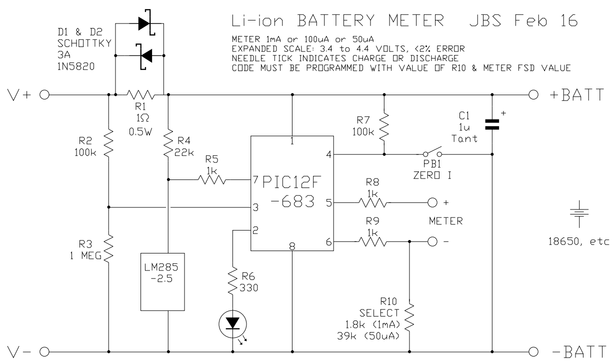

Fig. 2. Circuit diagram of the expanded-scale meter.

Figure 3 provides the layout of components on a prototype circuit board. The board is designed to be bolted directly to the back of an analog meter housing. To that end it has pads spaced 600mil and 1000mil. It is equally possible if less elegant to solder the meter to the board with flying leads. Figure 4 shows a meter installed between a battery and a small commercial charger circuit.

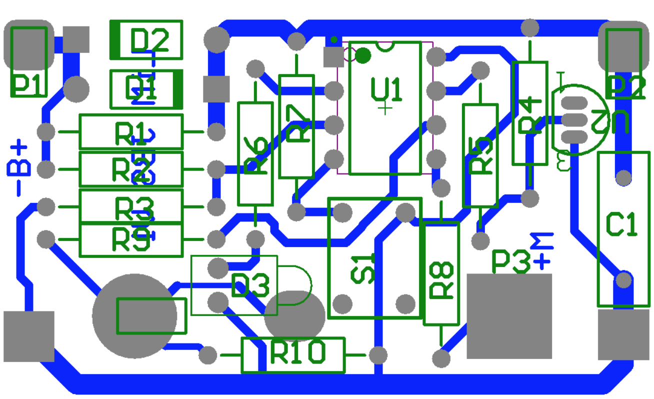

Fig. 3. Single-sided PCB layout for the circuit. Note that pads are provided for meters with 0.6-inch and 1.0-inch spacing of the terminal posts. Bottom-side copper is in dark blue, screen-print overlay in green, pads in grey.

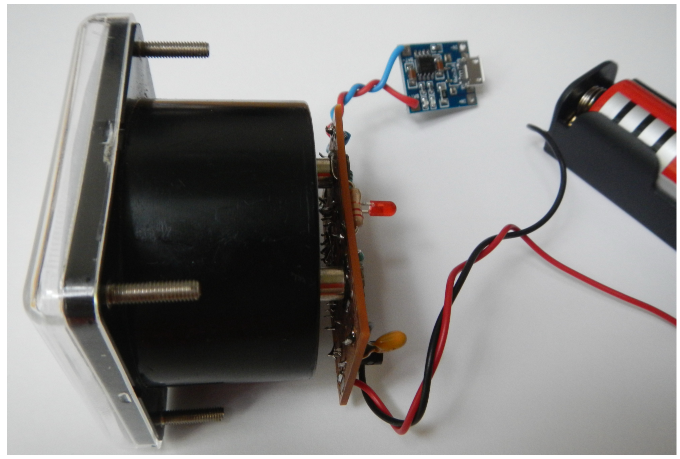

Fig. 4. Photograph of the PCB attached to the terminals of an analog moving-coil panel meter.

III. Firmware

The PIC12F683 carries out the following functions:

- On cold boot it flashes the LED and slowly swipes the meter movement from 0 to 100% PWM output and back down, before starting to read voltage;
- Locks out operation if compiled with unacceptable meter FSD and meter current sense resistor value combination (flashes forever);
- Filters the readings so that needle movements are slow and smooth;
- Measures the 2.50V bandgap voltage as a fraction of supply voltage;
- Calculates the supply voltage from the bandgap reading;
- Measures the voltage on the other side of the diode-resistor shunt;
- Estimates the battery current from the shunt voltage reading;
- Stores a corrected current zero point if the button is pressed (LED acknowledges);
- Drives the panel meter with a PWM signal;
- Measures the peak current flowing in the meter in the PWM mark intervals;
- Calculates the PWM duty cycle to obtain the required reading on the panel meter despite variable supply;
- Keeps time and periodically changes the PWM drive for 65ms to momentarily deflect the meter needle right or left to reflect charge or discharge respectively;
- Turns on the LED in a fast series of flashes in case of overvoltage.

The needle “ticks” serve to suggest charge or discharge by moving the needle with varying intensity in small motions to the left (discharge) or right (charge). These twitches occur about every 2 seconds and last a couple of hundred milliseconds including the meter needle settling back. The intensity of the twitch reflects the current in a roughly logarithmic scale, with a few mA producing a small tick, and currents upward of a few hundred mA providing quite a sharp tick.

Around 65% of the program and 55% of the data memory is used. The project was compiled with the HiTech Standard commercial compiler, version 9.3.

IV. Precision

The LM285-2.5 is specified with an accuracy of $\pm 1.5\%$ over temperature and current. In practice, in this application, in domestic temperature setting, it will have error closer to $\pm 0.5\%$ error. Prototypes had a reference voltage between 2.489V and 2.507, or less than four-tenths of one percent error.

The selection of the meter-current sense resistor, R10, and its precision, will both affect reading accuracy. The resistor R10 was measured with an Agilent U1242B DMM and the code loaded with the exact value, refer to the "#define RM" line near the top of the code, reproduced below. The message is that the meter error can be dominated by this source of error if the resistor is not a precision type or the value individually customised. The meter FSD current will also affect accuracy. This may be a risk with cheaper meters. The code also requires a nominal or individually-measured number for the FSD value, in microamps.

Although the PIC supports 10-bit PWM, only 8 bits were used, and this leads to a potential reading error of around 0.5% of scale or a bit more than 0.1% of voltage owing to the expanded scale. This error is additional to that from the correct selection and coding of R10 value, and the meter movement Full-Scale Deflection (FSD) value. This error is less than the error inherent in many analog panel meters, and in most cases a good eye is required to read off a value with 1% precision anyway. In summary, the user could implement 10-bit PWM and 10-bit arithmetic calculation, but this is likely only be perceptible if the meter is a high-quality "four inch" meter rather than the "two-inch" type normally encountered.

V. Measured Results

A prototype was tested with "nominal" resistor and FSD current values in the code. The meter display was compared to a precision bench DMM. The meter was mechanically zeroed before the application of power, and its orientation with respect to gravity held vertical and constant.[1] It read about 3.495V with 3.501V applied (-0.2%), 3.78V with 3.801V applied (-0.4%), 4.08V with 4.100V applied (-0.5%), and 4.28V with 4.305V applied (-0.6%). This error cannot be fully explained by the rounding error anticipated from the arithmetic and the voltage reference precision, and was attributed to the meter movement and resistor tolerance. Adjustment of the meter FSD and RM values in code corrects out enough of this error for manual reading precision to be the limit.

Acknowledgement

The author wishes to thank Benson Chang for help with CAD, component selection, and PCB fabrication.

[1] A 100μA Shinohara Japanese movement with a 1% rated linearity was used in the first prototype. Movement from vertical to horizontal position caused a 2% shift in the zero set, and might affect meter linearity as well.

References

[1] Martin Coleman, Chi Kwan Lee, Chunbo Zhu, and William Gerard Hurley, "State-of-Charge Determination From EMF Voltage Estimation: Using Impedance, Terminal Voltage, and Current for Lead-Acid and Lithium-Ion Batteries", IEEE Transactions on industrial electronics, VOL. 54, NO. 5, October 2007.

[2] Hung-Cheng Chen, Shuo-Rong Chou, Hong-Chou Chen, Shing-Lih Wu, and Liang-Rui Chen, "Fast Estimation of State of Charge for Lithium-ion Battery", *IEEE International Symposium on Computer, Consumer and Control*, 2014, pp284–287.

[3] Jonathan Scott, "Expanded scale vehicle ammeter", Electronics Today International, May 1981, pp33–36.

[4] *PICC STD ANSI C Compiler*, Hi-Tech Software (now Microchip Technology), Australia, 2009.

```c
/*
;Filename:        Main.c
;Author:        JBS
;Date:          Feb 2016
;Description:    C program for a Monitoring meter for an Li-ion battery using a PIC12F683
;                Modified for nonlinear scale
;
;*****************************************************************************
; Channels are: GP0/AN0 pin7 - Vref 2.500V
;                GP1/AN1 pin6 - meter current analog input
;                GP2/PWM pin5 - meter PWM drive
;                GP3 pin4 - PButton
;                GP4/AN3 pin3 - I sense input
;                GP5 pin2 - LED drive
;*****************************************************************************
*/

#include <pic.h>
#include <stdlib.h>

__CONFIG(INTIO & WDTDIS & MCLRDIS & BOREN & UNPROTECT & PWRTEN);

#define    LED                GPIO5        // Output with LED to Gnd
//                        GPIO4        // AN3 - I sense AN3=(10/11)*1024 + Ibat*10/Vlsb
#define    PB                GPIO3        // PushButton (active low)
//        PWM                GPIO2        // PWM
//        Isense            GPIO1        // AN1 - meter current sense
//        Imeter            GPIO0        // AN0 - 2.500 Vref

#define SELECTMETER        CHS0=1;CHS1=0
#define SELECTVREF        CHS0=0;CHS1=0
#define SELECTISENSE    CHS0=1;CHS1=1

// *** THESE VALUES MUST BE SET FOR THE HARDWARE ***
// 100uA Shinohara meter with 22k sense resistor
//#define METERSCALEUA    100                // current in microamps for FSD on panel meter
//#define RM                22115            // value of resistor sensing meter current

// $2 "1mA" meter from aliexpress with "2k2"
//#define METERSCALEUA    1050            // current in microamps for FSD on panel meter
//#define RM                2157            // value of resistor sensing meter current
#define METERSCALEUA    939            // current in microamps for FSD on panel meter
#define RM                2163            // value of resistor sensing meter current
// 100uA Shinohara meter with 18k sense resistor
//#define METERSCALEUA    100                // current in microamps for FSD on panel meter
//#define RM            17690            // value of resistor sensing meter current

__EEPROM_DATA(0xA3,0x03,0x00,0x00,0x00,0x00,0x00,0x00);      // Visens0 (default=931 = 0x03A3)
__EEPROM_DATA('J','B','S','-','2','0','1','6');              // copyright sig
__EEPROM_DATA(0x00,0x00,0x00,'V','1','.','0','2');              // version
#define VISENS0 0x00                                        // where divider reading for zero load current lives in EE

//Function Prototypes
unsigned char NVReadByte(unsigned char);
void NVWriteByte(unsigned char, unsigned char);
void interrupt Isr(void);
void NVReadShort(unsigned char, volatile unsigned short int *);
void NVWriteShort(unsigned char, volatile unsigned short int *);

//Global Variable Declarations
unsigned short int pTMR1    @ 0x0E;    // to address TMR1 as an integer
unsigned char delay65ms;            // delay counter
short unsigned int Vref=628;        // ADC reading of 2.500V ref input (default is for ideal r values)
short unsigned int Vrm=809;            // ADC reading of voltage across resistor in series with meter (FSD, 4V)
short unsigned int Visens;            // ADC reading from sense of voltage on other side of diodes
short unsigned int Visens0;            // Visens for zero current flowing
signed long int Vs_mV;                // supply voltage (battery voltage) in millivolts
unsigned short int Im_max_uA;        // meter current when PWM ON, in uA
bit storeRequest;                    // request to store new zero-current value in EEPROM
bit bootDone;                        // running flag
bit lockout;                        // flag is impossible values present at compile
bit loop;                            // reboot flag, clear to stop infinite loop

void interrupt Isr(void)
```

```c
{
    signed short int display;        // temp var
    static unsigned char clicks;     // count 65ms ISRs for timekeeping
    static unsigned char pbcnt=0;     // button debounce
    unsigned short int fsd;            // the PWM value for FSD on meter at this supply voltage & meter series resista
    unsigned char nudge;             // temp var
    signed long int ltmp;             // temp var

    if(TMR2IF){
        GODONE=1;                                         // trigger ADC to read meter sense voltage
        TMR2IF=0;                                         // clear flag
        while(GODONE){;}                                 // wait until ADC complete
        if(CCPR1L>8){                                     // if PWM is driving high for a decent period
            if(ADRESL+256*ADRESH>Vrm){Vrm+=1;}             // allow Vrm to slew towards measured value...
            if(ADRESL+256*ADRESH<Vrm){Vrm-=1;}             // to slow response to changes which will be gradual
        }
        clicks++;                                         // count 65ms periods
        if(delay65ms)delay65ms-=1;                         // delay counter
        SELECTISENSE;                                     // point ADC at current sense input
        for(display=0;display<20;display+=1){             // allow long time for multiplexer to settle
            asm("nop;");}                                 // ...we are not in a hurry
        GODONE=1;                                         // trigger ADC
        while(GODONE){;}                                 // wait until ADC complete
        Visens = ADRESL+256*ADRESH;                         // get value
        SELECTVREF;                                         // point ADC at 2.500 V ref
        for(display=0;display<20;display+=1){             // multiplexer settle time
            asm("nop;");}                                 // again wait as we are not in a hurry
        GODONE=1;                                         // trigger ADC
        while(GODONE){;}                                 // wait until ADC complete
        if(ADRESL+256*ADRESH>Vref){Vref+=1;}             // filter by slewing slowly
        if(ADRESL+256*ADRESH<Vref){Vref-=1;}             // positive and negative
        SELECTMETER;                                     // point ADC to meter for next time

        Vs_mV = (2500L*1024L)/Vref;                               // Vref=2500mV, so now we have supply voltage in mV
        Im_max_uA = (125L*Vrm*Vs_mV)/128L/RM;                   // Current in meter at 100% PWM in uA = 1e6*((Vrm/1024)*(
        fsd = 256L*METERSCALEUA/Im_max_uA;                       // PWM value for exact FSD (needle at end of scale)

        if(!bootDone){return;}                                   // no write to meter if still booting

        if(Vs_mV>3500){                                           // 0.1V/div, 2nd to 10th divisions, 3.5 to 4.3V
            display = 200 + (Vs_mV-3500);                       // display ranges 200 upwards at 100mV/div, fsd=1000=4300
        }else{
            display = 10*(Vs_mV-3000)/25;                       // display ranges 200 downwards at 250mV/div
        }

        display = (long)display*(fsd)/1000L;                   // convert 0-1000 range to value required in PWM, 0-fsd
        if(display<0) {CCPR1L=0;}                               // keep value in range 0 to...
        else if(display>255) {CCPR1L=255;}                       // 255
        else CCPR1L=display;                                   // write to meter the value clipped to 255

        if(clicks>38){                                         // 2.5 sec period expired, apply a "tick"
            clicks=0;                                         // reset timing counter
            display=0;                                         // clear signal, default value
            nudge=2;                                         // amount to make visible twitch in needle (can change with me
            if(Visens>Visens0+1)display+=nudge;                 // if above this threshold, increase size of "twitch" or "n
            if(Visens+1<Visens0)display-=nudge;                 // or if the opposite polarity
            if(Visens>Visens0+10)display+=nudge;             // and so on...
            if(Visens+10<Visens0)display-=nudge;
            if(Visens>Visens0+20)display+=nudge;
            if(Visens+20<Visens0)display-=nudge;
            if(Visens>Visens0+40)display+=nudge;
            if(Visens+40<Visens0)display-=nudge;
            if(Visens>Visens0+55)display+=nudge;
            if(Visens+55<Visens0)display-=nudge;
            if(Visens>Visens0+70)display+=nudge;
            if(Visens+70<Visens0)display-=nudge;
            if(Visens>Visens0+85)display+=nudge;
            if(Visens+85<Visens0)display-=nudge;
            if(Visens>Visens0+100)display+=nudge;
            if(Visens+100<Visens0)display-=nudge;
            if(Visens>Visens0+115)display+=nudge;
            if(Visens+115<Visens0)display-=nudge;
            if(Visens>Visens0+130)display+=nudge;
            if(Visens+130<Visens0)display-=nudge;
```

```
            display=CCPR1L+display;                              // nudge needle to indicate charge/discharge
            if(display<0) {display=0;}                          // keep value in range 0 to...
            if(display>255) {display=255;}                      // 255
            CCPR1L = display;                                   // write back summed value
        }

        // check for button pushes
        if(PB==0){                                              // button pressed (if fitted)
            if(++pbcnt>10){                                      // pressed for 650ms, debounced and not a casual bump
                LED=1;                                          // show push registered
                if(pbcnt>154){                                  // 10 secpnd push
                    loop=0;                                      // reboot
                }
            }
        }else{
            if(pbcnt>=10) storeRequest=1;                       // request update as PB released from a long push
            pbcnt=0;                                            // clear counter
        }

        if(Vs_mV>4300) {                                        // overvoltage!
            if(clicks&0x01==1){LED=1;}else{LED=0;}              // flash LED rapidly (plus draw some current!)
        }

    return;
    }
    // should never get here
    while(1){LED=1;asm("nop;");asm("nop;");LED=0;asm("nop;");}    // lock up with fash flashes, for debugging
    return;                                                        //    GIE = 1 - enable global ints - done in RETFIE
}

//****************************************************************************
//Main() - Main Routine
//****************************************************************************
void main()
{
    short unsigned int VforFSD;

    OSCCON = 0b01100001;            //Internal osc, 4MHz
    VRCON = 0;                        //Turn Off Voltage Reference Peripheral
    CMCON1 = 0x07;                  //Turn Off Comparator Peripheral
    TRISIO = 0b00010011;            //Set GP0, GP1 and GP4 as input
    ANSEL = 0b01011011;                //16 prescale (4us TAD), AN0, AN1 & AN3 enabled
    ADCON0 = 0b10001101;            //Right-justified(10b),Vddref,ch3,enabled
    OPTION = 0b00000100;            //T0CS=0, PSA=0, PS0=PS1=0, PS2=1 (prescale 1us/32 on T0)
    T1CON = 0x00;                    //disable TMR1, int osc, 1:1 prescale
    PIR1 = 0x00;                    //clear all peripheral int flags
    PIE1 = 0x02;                    //only T2 interrupt enabled
    INTCON = 0b01000000;            //GIE=0, PEIE=1, T0IE=INTE=GPIE=T0IF=INTF=GPIF=0
    PR2 = 0xFF;                        //8-bit PWM
    T2CON = 0b01111111;                //T2 on, prescale 16, postscale 16 (62.5kHz, 4ms period, INTs every 65ms)
    CCP1CON = 0b00001100;            //PWM on, active HI
    CCPR1L=0;                        // start at zero

    NVReadShort(VISENS0, &Visens0);                                   // read zero-current sense value from EEPROM
    VforFSD=(((long unsigned int)(METERSCALEUA))*(RM+1000)/1000L);    // voltage in mV needed to get METERSCALEUA acro
    lockout=0;                                                        // clear flag
    if(VforFSD>4000){                                                // impossible RM and METERSCALEUA values, can't g
        lockout=1;                                                    // prevent full boot
    }
    if(VforFSD<1400){                                                // impossible RM and METERSCALEUA values, can't g
        lockout=1;                                                    // prevent full boot
    }
    GIE=1;                                                            // enable ISR (probably done when EEPROM read an
    delay65ms=48;                                                    // 3 secs
    while(delay65ms){                                                // flash
        if(lockout && delay65ms<2){delay65ms+=16;}                    // never complete boot
        if(delay65ms&0x01){LED=0;}else{LED=1;}                        // flash LED to show cold boot
    }LED=1;                                                            // Leave LED on
    delay65ms=64;                                                    // 4 sec
    while(delay65ms){CCPR1L=256-delay65ms*4;}                        // sweep meter up
    delay65ms=64;                                                    // 4 sec
    while(delay65ms){CCPR1L=delay65ms*4-1;}                            // sweep meter down
    LED=0;                                                            // LED off, boot done
```

```c
    loop=1;
    while(loop){                                // keep looping unless flag clears to reboot
        bootDone=1;                                // set flag to say boot done, OK to update meter
        if(storeRequest){                        // request to save Iscale to EEPROM
            storeRequest=0;                        // clear flag
            Visens0=Visens;                        // update zero value
            NVWriteShort(VISENS0,&Visens0);        // store in EEPROM
            LED=0;                                // clear LED showing it done
        }
    // end of perpetual loop
    }
}

//***************************************************************************
//Functions
//***************************************************************************

//***************************************************************************
//NonVolatile memory reads and writes
//***************************************************************************
unsigned char NVReadByte (unsigned char nvrbadr)
{
        EEADR = nvrbadr;
        RD = 1;                  /* Part of EECON1. */
        return (EEDATA);
}
void NVWriteByte (unsigned char nvwbadr, unsigned char nvwbdat)
{
    GIE = 0;                            // disable ints globally
    do {
        EEADR = nvwbadr;                // load EEPROM desired write address
        EEDATA = nvwbdat;                // load EEPROM data byte to be written
        WREN = 1;                        // enable writes
        WRERR = 0;                        // clear error bit
        EECON2 = 0x55;                    // start HW sequence
        EECON2 = 0xaa;                    // middle HW step
        WR = 1;                            // initiate HW write
        while (WR)                        // until WRite completed...
            ;                            // do nothing
        WREN = 0;                        // disable further writes
        EEIF = 0;                        // clear error int flag (not used)
    } while (WRERR);                    // confirm no errors and data good
    GIE = 1;                            // re-enable ints globally
    return;
}

void NVReadShort(unsigned char adr, volatile unsigned short int *i)
{
        EEADR = adr;
        RD = 1;
        *((unsigned char*)i)=EEDATA;
        EEADR = adr+1;
        RD = 1;
        *((unsigned char*)i+1)=EEDATA;
        return;
}
void NVWriteShort(unsigned char adr, volatile unsigned short int *i)
{
        NVWriteByte(adr, *((unsigned char*)i) );
        NVWriteByte(adr+1, *((unsigned char*)i+1) );
        return;
}
```