

```

/*****
/* Program          : MAIN.C
/* Function         : XXXX Control Program
/* Author          : John F. Fitter B.E.
/* Language        : HiTech C (ANSI C)
/* Platform        :
/* Target          : MOICON PIC 16C67A @ 16MHz
/* Development     : RICE16 PIC 16C02-ME / 16C77-ME @ 8MHz
/* Target hardware : XXXX Rev A1
/*
/* Version         : 1.3
/* Revisions       :
/*
/*
/* Copyright © 1998 Eagle Air Australia Pty. Ltd. All rights reserved
/*****

// Note: It is the responsibility of each procedure to setup port pin directions prior to use.
//       Procedures are allowed to leave the direction registers in an unknown state.

//       To programmers reading this code. The PIC micros have a hardware stack which is a
//       very limited resource. Much of this code is structured as inline code to limit the
//       use of the stack. Note that interrupts use the stack also, so all code must leave one
//       stack level free for interrupt service procedure use. Code size/stack use trade-offs
//       have been made in this program which inhibits the elegance of some code.

#define _MAIN_C

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <commdefs.h>
#include "main.h"
#include "e2data.h"
#include "serial.h"
#include "spi.h"
#include "delays.h"
#include "lcd44780.h"
#include "25cxxspi.h"
#include "switch.h"
#include "1306spi.h"
#include "motor.h"
#include "buzzer.h"
#include "opticsw.h"

#ifdef _DEVELOPMENT // for software development only
__CONFIG(FOSC0 | 0x40 | 0xfc00); // max. speed of bondout chip is 10MHz
#else
__CONFIG(FOSC1 | WDTE | 0x40); // production devices run at 16MHz
//__ID(_HVER, _HREV, _SVER, _SREV);
#endif // _DEVELOPMENT

/*****
/* Main program.
/*****
main() {

    unsigned char n, m, p;
    signed char prec;

    init_proc();

    // This is the main program endless loop.
    while(true) {
        yield(); // yield control to system
        print_to_lcd(); // default print destination is lcd

        // ***** mainline user code goes here *****
        /* Keep this code inside one bank (2048 bytes) or the compiler will stuff it up */
    }
}

/*****

```

```

/* Selected utility procedures. */
/*****

// Procedure to return the order of magnitude of a double precision floating point number.
// This is essentially the integer part of the logarithm to base 10 with the added advantage
// that it works for negative numbers !!!
signed char int_log10(double dval) {

    signed char n = 0;

    if(dval != 0) { // handle zero
        dval = fabs(dval); // handle negative numbers
        while(dval >= 10) { // brute force - count the divisions
            dval /= 10.; // by 10 to get to less than 10
            n++;
        }
        while(dval < 1) { // brute force - count the multiplies
            dval *= 10.; // by 10 to get to more than 1
            n--;
        }
    }
    return n;
}

// Procedure to return an integer power of 10.
double pow10(signed char power) {

    double n = 1.0;

    while(power > 0) { // brute force - repeat multiply by
        n *= 10.; // 10 and decrement power to zero
        power--;
    }
    while(power < 0) { // brute force - repeat divide by
        n /= 10.; // 10 and increment power to zero
        power++;
    }
    return n;
}

// Procedure to enter/exit from sleep mode. This is not a real sleep mode as implemented in
// hardware on the PIC's. It is only simulated to allow power down of peripherals by program
// control.
void goto_sleep(unsigned char sleep_status) {

    if(c_status.awake = !sleep_status) { // yes, then are we waking up ?
        beep(1); // yes, beep once
        finish_buzzing(); // wait until buzzer has finished
        sleep_elaps = read_eeprom_deselect(SLEEPTIME); // reset sleep timer
    }
    if(!(read_1306(NVFLAGS) & NVFLGSP)) // is sleeping disabled ?
        c_status.awake = true; // then stay awake
    lcd_blank_display(!c_status.awake); // blank or restore display
    enable_lcd_bl(c_status.awake); // set the backlight
}

// Utility procedure to print two bcd digits from one packed bcd byte. The procedure
// assumes that the argument is a properly formed bcd number.
void put_bcd(unsigned char bcd) {
    putchar((bcd >> 4) | 0x30); // print the ms digit
    putchar((bcd & 0x0f) | 0x30); // print the ls digit
}

// Utility procedure to convert a decimal number to bcd and return the result.
unsigned char dec_to_bcd(unsigned char dec) {
    return((dec % 10) + ((dec / 10) << 4));
}

// Utility procedure to convert a bcd number to decimal and return the result.
unsigned char bcd_to_dec(unsigned char bcd) {
    return ((bcd >> 4) * 10 + (bcd & 0x0f));
}

// Utility procedure to reset the processor and optionally restore the factory defaults
// from eeprom and place them into nvram. Stack usage is not important because this procedure
// ends in a processor reset. It can thus be called from any place in the program.

```

```
// The block copy of eeprom to nvram is done here rather than using a general block copy
// procedure because this is the only place a block copy is needed. The argument passing
// overheads for a general block copy cannot be justified.
```

```
// Note : If a block copy is needed in the future for other purposes then this procedure
// should be eliminated. Calls to this procedure should be replaced with a call to the block
// copy followed by a macro setting pclath and pcl.
```

```
void reset_proc(unsigned char restore_def) {
```

```
    int n;
    if(restore_def)
        for(n = 0; n < (UNIQUE-BLONADDR); n++)
            write_1306(n + BLONADDR, read_eeprom_deselect(n + DEFBLDC));
```

```
    // This kills the program stone dead. The program structure is
    // terminated here - "the big BREAK"
```

```
    PCLATH=0;
    PCL=0;
}
```

```
/* *****
/* This is the system yield procedure. It must be called often by mainline code. It checks
/* the yield flag to determine if it should do anything. The yield flag is set by an
/* interrupt every lms and the yield procedure resets it.
/* *****
```

```
void yield() {
```

```
    // If true then this code is executed every 1 ms only if this
    // procedure is called more often than every lms.
```

```
    if(c_status.yield) { // do nothing if lms has not elapsed
        c_status.yield = false; // reset the yield flag
        if(!dbc_elaps) --dbc_elaps; // decrement the debounce timer to zero
        update_buzzer(); // maintain buzzer operation
        process_rx_data(); // check for incoming serial data
    }
```

```
    if(flg_motor_on) { // is the motor running ?
        if(c_status.mtr_slow) { // is the motor in slow speed mode ?
            if(!--slow_timer) slow_timer = 50; // reset slow timer ?
            if(slow_timer > motor_dc) mpwr = false; // is running not allowed ?
            else mpwr = true;
        } else mpwr = true; // set it on or off accordingly
    }
```

```
    // This code is executed every 100ms.
    // Accuracy is +/-100ms
```

```
    if(c_status.exp100ms) { // do nothing if 100ms has not elapsed
        c_status.exp100ms = false; // and reset the flag
        if(!auto_time) { // decrement the auto-timeout timer
            --auto_time; // and flag if it has just timed out
            if(!auto_time) c_status.autotimeout = true;
        }
    }
```

```
    // This code is executed every 1/2 second.
    // Accuracy is +/-100ms
```

```
    if(!--tmr_1s) { // do nothing if 1/2s has not elapsed
        tmr_1s = 5; // reset timer to 5x100ms
        if(!state_time) --state_time; // decrement the state timer to zero
        c_status.new_field = true; // refresh the screen
        if(!motor_time) drive_motor(false); // turn off the motor, else
        else --motor_time; // decrement motor run timer to zero
    }
```

```
    // This code is executed every 6 seconds.
    // Accuracy is +/-100ms
```

```
    if(!--tmr_100mm) { // do nothing if 6s has not elapsed
        tmr_100mm = 60; // if zero, set timer to 60x100ms
        if(!sleep_elaps) goto_sleep(true); // then enter sleep mode (+/-6s) ? or
        else --sleep_elaps; // decrement the sleep timer to zero
    }
```

```
    }
}
/* *****
/* Initialize processor to specified state.
/* The application specific stuff has been cut out of this leaving just the essence of
/* the techniques.
/* *****
```

```

void init_proc() {

    unsigned char n;

    // Initialize global variables (variables not initialized here are
    // initialized to zero by C definition).
    tmr_10ms    = 10;           // set the 10mS timer (counts ms)
    tmr_100ms   = 10;          // set the 100ms timer (counts 10ms)
    tmr_1s      = 10;          // set the 1s timer (counts 100ms)
    tmr_100mm   = 60;          // set the 6s timer (counts 100ms)

    c_status.yield      = false; // yield control to system - not yet
    c_status.exp100ms   = false; // 100ms not expired yet
    c_status.prt_to_lcd = true;  // printf destination is lcd
    c_status.en_timing  = false; // do not allow timing
    c_status.captured   = false; // no timing values have been captured
    c_status.autotimeout = false; // auto timeout has not occurred

    // Initialize peripheral hardware. The order of these procedures is vital - do not
    // change for any reason.
    init_buzzer();           // enable and turn off buzzer
    init_motor();           // enable and turn off motor driver
    init_opticsw();         // enable and turn off optical switches
    init_lcd();             // initialize the lcd
    init_serial();          // initialize serial port and devices
    setup_spi(SPI_MASTER | SPI_H_TO_L | SPI_CLK_DIV_16); // setup spi for 1MHz clock (@16MHz)
    init_eeprom();          // Initialize the eeprom - needs 5ms
                                // from powerup Tpuw-lcd_init does this
    init_1306(0, 0);        // Initialize the clock/calendar.
    init_lcd_bl();          // initialize lcd backlight to default
    enable_peripheral_interrupts(); // setit all in motion !!
    enable_global_interrupts();

    goto_sleep(false);     // ensure controller is awake
    init_switches();        // init keyboard and do initial scan

    // Write the hardware and software version numbers to nvram. This is done here so that
    // if a new controller is being initialized then it at least has a valid version number
    // for the pc to identify.
    write_1306(NVHVER, _HVER<<4|_HREV);
    write_1306(NVSVR, _SVR<<4|_SREV);

    // If the enter key is pressed at powerup then the controller will reset to factory
    // defaults otherwise if the print key is pressed at powerup then the controller waits for
    // a command from the pc. It will execute all pc commands forever, or until the pc issues
    // a reset. This is normally used for maintenance to initialize the eeprom or nvram.
    if(sw_char == KB_PRINT) {
        printf("\fWaiting for PC...");
        beep(3);             // beep 3 times
        finish_buzzing();   // wait until buzzer has finished
        while(true) process_rx_data(); // process all pc commands
    } else if(sw_char == KB_ENTER) {
        print_eestring(S_LDEFS); // else reset and load nvram
        delay_s(3);           // tell the user (eeprom is ok here)
        reset_proc(true);     // let him read the message
    }

    // Write the programmable characters to cgram from eeprom
    n = 64;
    while(n--) lcd_write_cgram(n, read_eeprom_deselect((int)n + PCHAR));

    // Get variables from eeprom/nvram which really must be in ram for performance reasons
    // or because there is not enough stack to get them when they are really needed.
    autorepdly = read_eeprom_deselect(AUTOREPDLY); // keyboard auto-repeat delay (x100ms)
    debounce   = read_eeprom_deselect(DEBOUNCE);  // keyboard debounce delay (x10ms)
    sertimeout = read_eeprom_deselect(SERTIMEOUT); // serial comms timeout (x10ms)
    motor_dc   = read_eeprom_deselect(MOTORDC);   // motor slow speed duty (ms/50ms)
    xl_super2  = read_eeprom_deselect(XLSUPER2);  // translated superscript 2 for printer
    reset_lcd_bl(); // set backlight to proper intensity

    enable_buzzer(!(read_1306(NVFLAGS) & NVFLGNS));
}

// ***** EOF MOI.C *****

```