```
/*****************************************
Project : Guitar Tuner
Version :
Date    : 1/11/2003
Author  : Dean Carpenter
Company : Areyes, Inc.
Comments:
Guitar Tuner
PD0 - out - LED pulse indicator
PD1 - out - OpAmp power active LO
PD2 - in  - button
PD6 - in  - electret mic input
PortB - LCD
PB3 - LCD power active HI


Chip type           : AT90S2313
Clock frequency     : 4.000000 MHz
Memory model        : Tiny
Internal SRAM size  : 128
External SRAM size  : 0
Data Stack size     : 23
Bit variable size   : 0
(s)print features   : int
Char is unsigned
8-bit enums
Automatic Register Allocation
*****************************************/

#include <90s2313.h>;
#include <stdlib.h>
#include <sleep.h>
#include <string.h>

// Alphanumeric LCD Module functions
#asm
   .equ __lcd_port=0x18
#endasm
#include <lcd.h>

#define WDTOE           (1 << 4)                    // watchdog timer bits
#define WDE             (1 << 3)

#define ACCURACY        4                           // +/- counts to center count
#define AVG_BITS        5
#define AVG_CNT         (1 << AVG_BITS)             // number freq cnts to average up
#define TOLERANCE_BITS  4                           // number of bits of tolerance/error for In-Tune

// Note - #defines below are just a fancy way to come up with appropriate counts for
// the various _CTR constants.  These are used for timeouts in the code, and can be replaced
// with simple constants.  Be careful, playing with these or their calculation #defines can
// sometimes have unexpected impact on the code.
#define F_CPU           (_MCU_CLOCK_FREQUENCY_)     // CPU clock frequency
#define T0_PRESCALER    64                          // CPU prescaler value for timer0
#define ICP_PRESCALER   64                          // CPU prescaler value for ICP
#define BASE_FREQUENCY  (F_CPU/ICP_PRESCALER)       // counter frequency for Hz calculations

#define IDLE_MINUTES    1                           // minutes for timeout
#define IDLE_CTR        (F_CPU / T0_PRESCALER / 256 * IDLE_MINUTES * 60) // actual counter value
#define LCD_REFRESH     100                         // refresh lcd every n ms
#define LCD_CTR         (F_CPU / T0_PRESCALER / 256 * LCD_REFRESH / 1000) // actual counter value
#define EXPIRE          122                         // must get valid count within this many ms
#define EXPIRE_CTR      (F_CPU / T0_PRESCALER / 256 * EXPIRE / 1000)
```

```
#define LONG_PUSH       1                               // # seconds to count as a long button push
#define LONG_PUSH_CTR   (F_CPU / T0_PRESCALER / 256 * LONG_PUSH)
#define DEBOUNCE        20                              // # ms to count as a valid button push 20ms
#define DEBOUNCE_CTR    (F_CPU / T0_PRESCALER / 256 * DEBOUNCE / 1000)

#define XY_FREQ         5                               // char position of frequency display
#define XY_STRING       2                               // char position of string name
#define XY_RESULTS      2                               // char position of Hi/Lo/OK

// button status
#define NONE            'N'                             // waiting for button push
#define LONG            'L'                             // Long push
#define SHORT           'S'                             // Short push
#define PUSHED          'p'                             // was pushed, now need to check time

#define BURN            60                              // how many edges to skip to allow
                                                        // signal to stabilize
// Bass guitar string frequencies
#define BASS_B          30.87
#define BASS_E          41.2
#define BASS_A          55
#define BASS_D          73.42
#define BASS_G          97.99
#define BASS_C          130.81

// Acoustic guitar string frequencies
#define ACOUSTIC_E      82.4                            // top string (1)
#define ACOUSTIC_A      110
#define ACOUSTIC_D      146.72
#define ACOUSTIC_G      195.92
#define ACOUSTIC_B      246.92
#define ACOUSTIC_EH     329.8                           // bottom string (6)

// The guitar note span
//  # # #  # #  # # #  # #
//EF G A BC D EF G A BC D E
//1    2    3    4 * 5    6

enum { GUITAR_ACOUSTIC, GUITAR_BASS };                  // valid guitar types
enum { LOW, HIGH, TUNED };                              // notes heard can be in one of these states
enum { AUTOMODE, MANUAL };                              // Tuner mode states

unsigned char   line1[9];                               // for lcd writing
unsigned char   line2[9];                               // for lcd writing
unsigned char   blanks[] = "       ";                   // 8 bytes

eeprom unsigned char    tuning_modes[2][9] = {          // 18 bytes
        { "Automode" },
        { "Manual  " }
};

eeprom unsigned char    tuning_results[3][3] = {        // 9 bytes
        { "Lo" },
        { "Hi" },
        { "OK" }
};

typedef eeprom struct  {
        unsigned char   guitarname[9];
        unsigned char   stringname[6];                  // guitar string mnemonics
        unsigned int    frequencies[6];
        unsigned int    centers[6];
        unsigned int    transitions[5];
```

```
}  Guitar;                                          // total 49 bytes

eeprom Guitar guitars[2] = {                       // total 98 bytes
    {
        { "Acoustic"},
        { 'e', 'B', 'G', 'D', 'A', 'E' },
        { ACOUSTIC_EH, ACOUSTIC_B, ACOUSTIC_G, ACOUSTIC_D, ACOUSTIC_A, ACOUSTIC_E },
        {
          BASE_FREQUENCY/ACOUSTIC_EH,              // High E
          BASE_FREQUENCY/ACOUSTIC_B,               // B
          BASE_FREQUENCY/ACOUSTIC_G,               // G
          BASE_FREQUENCY/ACOUSTIC_D,               // D
          BASE_FREQUENCY/ACOUSTIC_A,               // A
          BASE_FREQUENCY/ACOUSTIC_E                // Low E
        },
        {
          BASE_FREQUENCY/(ACOUSTIC_B+((ACOUSTIC_EH-ACOUSTIC_B)/2)),     // E to B
          BASE_FREQUENCY/(ACOUSTIC_G+((ACOUSTIC_B-ACOUSTIC_G)/2)),      // B to G
          BASE_FREQUENCY/(ACOUSTIC_D+((ACOUSTIC_G-ACOUSTIC_D)/2)),      // G to D
          BASE_FREQUENCY/(ACOUSTIC_A+((ACOUSTIC_D-ACOUSTIC_A)/2)),      // D to A
          BASE_FREQUENCY/(ACOUSTIC_E+((ACOUSTIC_A-ACOUSTIC_E)/2))       // A to E
        }
    },

    {
        { "Bass    " },
        { 'C', 'G', 'D', 'A', 'E', 'B' },
        { BASS_C, BASS_G, BASS_D, BASS_A, BASS_E, BASS_B },
        {
          BASE_FREQUENCY/BASS_C,
          BASE_FREQUENCY/BASS_G,
          BASE_FREQUENCY/BASS_D,
          BASE_FREQUENCY/BASS_A,
          BASE_FREQUENCY/BASS_E,
          BASE_FREQUENCY/BASS_B
        },
        {
          BASE_FREQUENCY/(BASS_G+((BASS_C-BASS_G)/2)),
          BASE_FREQUENCY/(BASS_D+((BASS_G-BASS_D)/2)),
          BASE_FREQUENCY/(BASS_A+((BASS_D-BASS_A)/2)),
          BASE_FREQUENCY/(BASS_E+((BASS_A-BASS_E)/2)),
          BASE_FREQUENCY/(BASS_B+((BASS_E-BASS_B)/2))
        }
    }
};

unsigned char   burn_cnt;                          // flag for first rising edge indicator
unsigned int    icp_total;                         // rollup of ICP counter
unsigned int    icp_cnt;                           // ICP counter

unsigned int    cycle_cnt;
unsigned int    icp_last;                          // for icp int count comparison
unsigned int    hertz;                             // actual freq
unsigned char   button_flag = NONE;                // button status LONG SHORT NONE
unsigned char   tune_mode = AUTOMODE;              // auto or manual tuning mode
unsigned char   guitar_type = GUITAR_ACOUSTIC;     // Tuning an ACOUSTIC or a BASS guitar
unsigned char   current_freq;                      // currently selected manual freq

// INT0 timers and counters
unsigned int    button_timer;                      // pushbutton timer
unsigned int    shutdown_timer;                    // idle timeout to shutdown
unsigned char   expire;                            // minimum time to receive next icp edge
                                                   // note - change to INT if using faster crystal
```

```
// External Interrupt 0 service routine
// Pushing the mode button comes here
// We only want to restart the timer if we have already handled the last push
interrupt [EXT_INT0] void ext_int0_isr(void)
{
  if (button_flag == NONE) {
        button_timer = 0;
        button_flag = PUSHED;
  }
}


// Timer 0 overflow interrupt service routine
// At 8MHz, with prescaler = 256, int is 31.250kHz or every 0.032ms, overflow at 122Hz  or 8.192ms
// At 4MHz, with prescaler = 256, int is 15.625kHz or every 0.064ms, overflow at 61Hz   or 16.384ms
// At 4MHz, with prescaler = 64,  int is 62.500kHz or every 0.016ms, overflow at 244Hz  or 4.096ms
// At 4MHz, with prescaler = 8,   int is 500.00kHz or every 0.002ms, overflow at 1953Hz or 0.512ms
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
static unsigned char    lcd_refresh;            // lcd refresh timer count

        expire++;                               // expiration cntr for cycle count timeout

// debouncing the button, looking for SHORT or LONG push
        if (!PIND.2) {                          // check for button status
          button_timer++;
          if (button_timer == LONG_PUSH_CTR) { // long enough for a LONG push
            button_flag = LONG;                 // say so
          }
        } else {                                // button is back up
          if (button_flag == PUSHED) {          // not a LONG push ?
            if (button_timer > DEBOUNCE_CTR) {  // nope - is it at least a debounce
              button_flag = SHORT;              // yup - say so
            }
          }
        }

        if (PIND.6) {                           // flash whenever we have a pulse
                PORTD.0 = 0;
        } else {
                PORTD.0 = 1;
        }

        shutdown_timer++;

        if (lcd_refresh++ == LCD_CTR) {         // time to refresh ?
                lcd_refresh = 0;
                lcd_clear();
                lcd_puts(line1);
                lcd_gotoxy(0,1);
                lcd_puts(line2);
        }
//      #asm("wdr")
}


// Timer 1 input capture interrupt service routine
// Incoming pulses from the opamps trigger the input capture int here
// We ignore the first BURN pulses to let the signal stabilize a bit, so count those
interrupt [TIM1_CAPT] void timer1_capt_isr(void)
{
        TCNT1 = 0;                              // reset timer 1
```

```
        burn_cnt += 1;                          // count transitions
        icp_cnt = ICR1;                         // grab counter
}


// Copy string from eeprom to sram
void strcpye( char * sram_str, char eeprom * eep_str )
{
char * sram_dest;
char eeprom * eep_src;

        sram_dest = sram_str;
        eep_src = eep_str;

        while (*eep_src != 0) {
                *sram_dest++ = *eep_src++;
        }
}


void Tuner_init(void)
{
        // Input/Output Ports initialization
        // Port B initialization
        // Func0=In Func1=In Func2=In Func3=In Func4=In Func5=In Func6=In Func7=In
        // State0=T State1=T State2=T State3=T State4=T State5=T State6=T State7=T
        // PB0 - out - LCD RS
        // PB1 - out - LCD R/W
        // PB2 - out - LCD EN
        // PB3 - out - LCD power source
        // PB4 - out - LCD D4
        // PB5 - out - LCD D5
        // PB6 - out - LCD D6
        // PB7 - out - LCD D7
        DDRB=0xff;
        PORTB=0x0ff;

        // Port D initialization
        // PD0 - out - LED pulse indicator
        // PD1 - out - Opamp power control active LO
        // PD2 - in  - button - pullup on
        // PD6 - in  - electret mic input
        DDRD=0x0b;
        PORTD=0x0d;

        // Timer/Counter 0 initialization
        // Clock source: System Clock
        // At 8MHz :
        // Clock value: 7.813 kHz  ovf = 30Hz  prescaler 1024 is 0x05
        // At 4MHz
        //              31.250 kHz ovf = 122Hz prescaler 256  is 0x04
        //              62.500 kHz ovf = 244Hz prescaler 64   is 0x03
        //              500kHz         = 1953Hz prescaler 8   is 0x02
        TCCR0=0x03;
        TCNT0=0x00;

        // Timer/Counter 1 initialization
        // Clock source: System Clock
        // Clock value: 125.000 kHz
        // Mode: Normal top=FFFFh
        // OC1 output: Discon.
        // Noise Canceler: On   tccr1b bit 7  0xc3 for on /64
        // Input Capture on Rising Edge
```

```
        TCCR1A=0x00;
        TCCR1B=0xc3;
        TCNT1H=0x00;
        TCNT1L=0x00;

        OCR1H=0x00;             // 2313
        OCR1L=0x00;

        // External Interrupt(s) initialization
        // INT1: On
        // INT1 Mode: Falling Edge
        // INT0: Off
        GIMSK=0x40;             // 0x40 for int0 only
        MCUCR=0x02;             // 0x02 for int0 falling edge only
        GIFR=0x40;              // 0x40 for int0 enable only

        // Timer(s)/Counter(s) Interrupt(s) initialization

        TIMSK=0x0A;             // 2313

        // Analog Comparator initialization
        // Analog Comparator: Off
        // Analog Comparator Input Capture by Timer/Counter 1: Off
        // Analog Comparator Output: Off
        ACSR=0x80;

        shutdown_timer = 0;             // make sure we don't suddenly shut down

        // LCD module initialization
        lcd_init(8);

        // Setup the Splash screen
        strcpye(line1, guitars[GUITAR_ACOUSTIC].guitarname);
        strcpye(line2, tuning_modes[AUTOMODE]);

        // Watchdog Timer initialization
        // Watchdog Timer Prescaler: OSC/2048
//      WDTCR=0x0F;
}


// Handle button pushes
// In auto mode    short push -> go to manual mode
//                 long push  -> switch guitars to tune, acoustic or bass
// In manual mode  short push -> change tuning freq to next
//                 long push  -> go to auto mode
void change_mode(void)
{
  shutdown_timer = 0;                   // reset shutdown timer every push
  switch (tune_mode) {
    case AUTOMODE :
      if (button_flag == SHORT) {       // automode -> manualmode
        tune_mode = MANUAL;
        current_freq = sizeof(guitars[guitar_type].stringname) - 1;
      } else {                          // long push
        if (guitar_type == GUITAR_ACOUSTIC) {
          guitar_type = GUITAR_BASS;
        } else {
          guitar_type = GUITAR_ACOUSTIC;
        }
      }
      strcpye(line1, guitars[guitar_type].guitarname);
      strcpye(line2, tuning_modes[tune_mode]);
```

```
      break;

    case MANUAL :
      if (button_flag == SHORT) {       // manualmode -> change frequency
        current_freq = (current_freq + 1) % (sizeof (guitars[guitar_type].stringname));
        strcpy(&line1[1], blanks);      // clear line
        line1[XY_STRING] = guitars[guitar_type].stringname[current_freq]; // show which string we're tuning to
        itoa(guitars[guitar_type].frequencies[current_freq], &line1[XY_FREQ]);  // target freq
      } else {                          // long push
        tune_mode = AUTOMODE;           // manualmode -> automode
        strcpye(line1, guitars[guitar_type].guitarname);
        strcpye(line2, tuning_modes[AUTOMODE]);
      }
      break;
  }

  button_flag = NONE;                   // indicate we've handled the button push
}


void main(void)
{
unsigned int    cnt_diff, tolerance;

        Tuner_init();

        // Global enable interrupts
        #asm("sei")

        while (1) {
                cycle_cnt = 0;
                burn_cnt = 0;
                icp_total = 0;

                // Sit in an endless loop, waiting for at least BURN pulses/cycles to go by
                // Once that's happened, we can go on
                // We also watch for button pushes here, and handle them if necessary
                // If things time out, we put ourselves to sleep
                //    Upon wakeup, jump into reset to make sure we start fresh
                while (burn_cnt < BURN) {                       // burn first few cycles to allow wave stabilization
                        if ((button_flag == SHORT) || (button_flag == LONG)) {
                                change_mode();                  // short/long button push
                        }
                        if (shutdown_timer == IDLE_CTR) {       // time to sleep
                                MCUCR = 0x00;                   // level int0
                                sleep_enable();
                                DDRD = 0;                       // show we're asleep - set everything to tri-state inputs
                                PORTD = 0;                      // that shuts off power to LCD (and opamps hopefully)
                                PORTB = 0;
//                              #asm("cli")
//                              #asm("LDI  R31,0x18")
//                              #asm("OUT  WDTCR,R31")
//                              #asm("LDI  R31,0x10")
//                              #asm("OUT  WDTCR,R31")
//                              #asm("sei")
                                powerdown();
                                #asm ("rjmp 0x00")              // wakeup come here - reset completely
                        }


                }

                // OK, pulses coming in, and we've burned off the first junky ones
```

```
// Now we need to grab the next AVG_CNT cycles - if we don't get them all within
// the EXPIRE timeout, we break out
icp_last = icp_cnt;                             // Save last cycle timing counter
expire = 0;                                     // reset edge expiration cntr

while (cycle_cnt < AVG_CNT) {
        burn_cnt = 0;
        while (burn_cnt == 0) {                 // wait for new capture (icp_last == icp_cnt)
                if (expire > EXPIRE_CTR) {      // no edges for too long ?
                        cycle_cnt = AVG_CNT;
                        break;                  // this makes sure that the check below doesn't trigger
                }                               //   if expire incs *just* after we get a good capture
        }

        if (expire > EXPIRE_CTR) break;         // did it take too long to get the cycles ?

        // Is this cycle count close enough to the last one, or do we skip it ?
        // The difference between last count and current count must be
        // less than the last count / 16 (arbitrary tolerance value :)

        if (icp_last > icp_cnt) {
                cnt_diff = icp_last - icp_cnt;
        } else {
                cnt_diff = icp_cnt - icp_last;
        }
        tolerance = icp_last >> TOLERANCE_BITS;

        if (cnt_diff <= tolerance) {            // within limits
                icp_total += icp_cnt;           // snag current icp counter
                cycle_cnt++;                    // note got another cycle
                icp_last = icp_cnt;             // save count for next comparison
                expire = 0;                     // reset edge cntr
        }
}

// OK, got a good set of cycles within time limits
// Now have to see what freq it was, and which string freq it's closest to
if (expire <= EXPIRE_CTR) {
        shutdown_timer = 0;                     // reset shutdown timer

        // OK - got 2^AVG_BITS cycles
        icp_total = icp_total >> AVG_BITS;      // get avg count

        hertz = BASE_FREQUENCY / icp_total;     // get actual hertz value
        if (hertz > 999) hertz = 999;           // max value we can display

        // If we're in AUTO mode, have to find the closest string freq
        if (tune_mode == AUTOMODE) {
                // go through transition frequencies
                for (cnt_diff=0;
                     cnt_diff<sizeof(guitars[guitar_type].transitions)/sizeof(guitars[guitar_type].transitions[0]);
                     cnt_diff++) {
                        if (icp_total < guitars[guitar_type].transitions[cnt_diff]) // stop if lower than this transition count
                                break;
                }
        } else {                                                        // manual tuning
                cnt_diff = current_freq;                                // targeting THIS string freq
        }

        // show which string we're tuning to
        strcpy(&line1[1], blanks);                                      // blank all but first char
        line1[XY_STRING] = guitars[guitar_type].stringname[cnt_diff];   // show which note
        itoa(guitars[guitar_type].frequencies[cnt_diff], &line1[XY_FREQ]);      // show target freq
```

```
                    // cnt_diff now holds the string index for target string
                    strcpy(&line2[1], blanks);                                          // clear junk
                    if (icp_total-ACCURACY < guitars[guitar_type].centers[cnt_diff]) {      // too high
                      strcpye(&line2[XY_RESULTS], tuning_results[HIGH]);
                    } else {
                      if (icp_total+ACCURACY > guitars[guitar_type].centers[cnt_diff]) {    // too low ?
                              strcpye(&line2[XY_RESULTS], tuning_results[LOW]);
                        } else {
                              strcpye(&line2[XY_RESULTS], tuning_results[TUNED]);
                        }
                    }
                    itoa(hertz, &line2[XY_FREQ]);                                       // current freq
                }
        }
}
```