```
// Andrey Kornilovich
// lab6_128.c

//  HARDWARE SETUP:

#define F_CPU 16000000 // cpu speed in hertz
#define TRUE 1
#define FALSE 0
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include "hd44780.h"
#include <string.h>
#include <stdlib.h>
#include "uart_functions.h"
#include "lm73_functions_skel.h"
#include "twi_master.h"
#include "si4734.h"

// definitions for segment pins and port B control pins
#define SEG_A 0x01
#define SEG_B 0x02
#define SEG_C 0x04
#define SEG_D 0x08
#define SEG_E 0x10
#define SEG_F 0x20
#define SEG_G 0x40
#define SEG_DP 0x80

#define DEC_1 0x10
#define DEC_2 0x20
#define DEC_3 0x40
#define PWM 0x80

#define ENC_A 0b11111100
#define ENC_B 0b11110011

volatile int segment_data[5];              //holds data to be sent to the segments

volatile int dec_to_7seg[13];              //decimal to 7-segment LED display encodings

volatile int digit = 0;                    // 7 seg display counter

volatile int time = 0;              // time value to display to 7 seg

volatile int alarm = 0;             // alarm value to display to 7 seg

volatile uint8_t set_time = 0;      // UI state for setting the time

volatile uint8_t set_alarm = 0;     // UI state for setting the alarm

volatile uint8_t set_radio = 0;     // UI state for setting the alarm

volatile uint8_t alarm_is_set = 0;  // flag for if the alarm is set

volatile uint8_t snooze = 0;        // snooze active flag

volatile uint8_t radio_or_alarm = 0;        // snooze active flag

volatile int radio_state = 0;       // state of the radio volume

volatile uint8_t play_alarm = 0;    // flag to play the alarm tone

volatile uint8_t button_state = 0x00;      // current UI state controlled by buttons

volatile uint8_t rcv_rdy;           // flag for uart status

volatile char uart_str[16];         // string for storing uart data

char lcd_str_top[16] = " ";      // holds alarm info to send to lcd

char lcd_str_bottom[16] = " ";      // holds temp info to send to lcd

volatile uint16_t lm73_temp;         // 16 bit value from lm73

uint8_t lm73_wr_buf[2];             // buffer to write to the lm73

uint8_t lm73_rd_buf[2];             // buffer to read from the lm73


// include necessary variables for running the radio
enum radio_band{FM, AM, SW};
volatile enum radio_band current_radio_band;

volatile uint16_t  current_fm_freq =  9910; //0x2706, arg2, arg3; 99.9Mhz, 200khz steps
// volatile uint16_t  current_fm_freq =  9780; //0x2706, arg2, arg3; 99.9Mhz, 200khz steps
extern uint8_t  si4734_wr_buf[9];
extern uint8_t  si4734_rd_buf[9];
extern uint8_t  si4734_tune_status_buf[8];
extern volatile uint8_t STC_interrupt;     //indicates tune or seek is done

uint16_t eeprom_fm_freq;
uint16_t eeprom_am_freq;
uint16_t eeprom_sw_freq;
uint8_t  eeprom_volume;

uint16_t current_am_freq;
uint16_t current_sw_freq;
uint8_t  current_volume;
```

```c
//Used in debug mode for UART1
char uart1_tx_buf[40];      //holds string to send to crt
char uart1_rx_buf[40];      //holds string that recieves data from uart


// write to dec_to_7seg all the pins to display 0-9, blank, and the decimal point
void encode_chars(void){
  dec_to_7seg[0] = ~(SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F); //0
  dec_to_7seg[1] = ~(SEG_B | SEG_C); //1
  dec_to_7seg[2] = ~(SEG_A | SEG_B | SEG_G | SEG_E | SEG_D); //2
  dec_to_7seg[3] = ~(SEG_A | SEG_B | SEG_C | SEG_G | SEG_D); //3
  dec_to_7seg[4] = ~(SEG_F | SEG_G | SEG_B | SEG_C); //4
  dec_to_7seg[5] = ~(SEG_A | SEG_F | SEG_G | SEG_C | SEG_D); //5
  dec_to_7seg[6] = ~(SEG_A | SEG_F | SEG_G | SEG_C | SEG_D | SEG_E); //6
  dec_to_7seg[7] = ~(SEG_A | SEG_B | SEG_C); //7
  dec_to_7seg[8] = ~(SEG_A | SEG_B | SEG_C | SEG_D | SEG_E | SEG_F | SEG_G); //8
  dec_to_7seg[9] = ~(SEG_A | SEG_B | SEG_F | SEG_G | SEG_C | SEG_D); //9
  dec_to_7seg[10] = 0xFF; //display nothing
  dec_to_7seg[11] = ~(SEG_A | SEG_B); //Colon
  dec_to_7seg[12] = ~(SEG_DP);
}

// calling this sets PORT B to output to a specific digit
void pick_digit(int digit){
  // set the correct port B output without clobbering the rest of the register
  switch (digit){
        // first (msb) digit, Y4 on decoder
        case 0:
          PORTB &= ~(DEC_1 | DEC_2); // &= to clear decoder control pins
          PORTB |= DEC_3;            // |= set decoder control pin
          break;
        // second digit, Y3 on decoder
        case 1:
          PORTB &= ~(DEC_3);
          PORTB |= (DEC_1 | DEC_2);
          break;
        // third digit, Y1 on decoder
        case 2:
          PORTB &= ~(DEC_2 | DEC_3);
          PORTB |= DEC_1;
          break;
        // fourth (lsb) digit, Y0 on decoder
        case 3:
          PORTB &= ~(DEC_1 | DEC_2 | DEC_3);
          break;
        // colon, Y2 on decoder
        case 4:
          PORTB &= ~(DEC_1 | DEC_3);
          PORTB |= DEC_2;
          break;
        // enable button board, Y7 on decoder
        case 5:
          PORTB |= (DEC_1 | DEC_2 | DEC_3);
          break;
        // no digit or button board (off), Y6 on decoder
        case 6:
          PORTB &= ~(DEC_1);
          PORTB |= (DEC_2 | DEC_3);
        default:
          break;
  }
}

//******************************************************************************
//                            chk_buttons
//Checks the state of the button number passed to it. It shifts in ones till
//the button is pushed. Function returns a 1 only once per debounced button
//push so a debounce and toggle function can be implemented at the same time.
//Adapted to check all buttons from Ganssel's "Guide to Debouncing"
//Expects active low pushbuttons on PINA port.  Debounce time is determined by
//external loop delay times 12.
//

uint8_t chk_buttons(uint8_t button) {
  static uint16_t state[8] = {0}; // holds present state
  state[button] = (state[button] << 1) | (! bit_is_clear(PINA, button)) | 0xE000;
  if (state[button] == 0xF000) return 1;
  return 0;
}

//***********************************************************************************
//                                   segment_sum
//takes a 16-bit binary input value and places the appropriate equivalent 4 digit
//BCD segment code in the array segment_data for display.
//array is loaded at exit as:  |digit3|digit2|digit1|digit0|colon
void segsum(uint16_t sum) {
  // break up decimal sum into 4 digit-segments
  segment_data[0] = dec_to_7seg[sum/1000]; //msb
  segment_data[1] = dec_to_7seg[(sum/100) % 10];
  segment_data[2] = dec_to_7seg[(sum/10) % 10];
  segment_data[3] = dec_to_7seg[sum % 10]; //lsb
}

void radio_segsum(uint16_t sum){
  if(sum/1000 > 0){segment_data[0] = dec_to_7seg[sum/1000];}
  else{segment_data[0] = dec_to_7seg[10];}
  segment_data[1] = dec_to_7seg[(sum/100) % 10];
  segment_data[2] = dec_to_7seg[(sum/10) % 10] & dec_to_7seg[12];
  segment_data[3] = dec_to_7seg[sum % 10]; //lsb
}

void update_7seg(void){
  // make PORTA an output
  DDRA = 0xFF;

  // assign port A and display to a digit
  PORTA = segment_data[digit];
  pick_digit(digit);

  // increment the digit and reset
  if(digit > 4) {digit = 0;}
  else{digit++;}
}

void process_buttons(void){
  // make PORTA an input port with pullups
  DDRA = 0x00;
  PORTA = 0xFF;

  // enable tristate buffer for pushbutton switches
  pick_digit(5);

 // now check each button and set the state as needed
  int button;
  for(button = 0; button < 8; button++){
    if(chk_buttons(button)){
      switch (button){
        case 0: // set time case
          if(button_state == 0x01){
            button_state = 0x00;
            set_time = 0;
          }
          else{button_state = 0x01;}
          break;
        case 1: // set alarm case
          if(button_state == 0x02){
            button_state = 0x00;
          }
          else{button_state = 0x02;}
          break;
        case 2: // snooze button case
          snooze = 1;
          break;
        case 3: // clear alarm
          set_alarm = 0;
          button_state = 0x00;
          set_time = 0;
          alarm = 0;
          alarm_is_set = 0;
          play_alarm = 0;
          radio_state = 0;
          set_property(0x4000, 0x0000); // mute

          strcpy(lcd_str_top, "Alarm clr!\0");
          break;

        case 4: // tune and play radio
          if(button_state == 0x10){
            button_state = 0x00;
          }
          else{
            button_state = 0x10;

            set_property(0x4000, 0x003F); // unmute
            fm_tune_freq();     //tune to frequency
            segment_data[4] = dec_to_7seg[10]; // turn off colon
          }
          break;

        case 7: // use radio or tone for alarm
          if(radio_or_alarm == 0){
            radio_or_alarm = 1;
            lcd_str_bottom[10] = ’r’;
            lcd_str_bottom[11] = ’a’;
            lcd_str_bottom[12] = ’d’;
            lcd_str_bottom[13] = ’i’;
            lcd_str_bottom[14] = ’o’;
          }
          else{
            radio_or_alarm = 0;
            lcd_str_bottom[10] = ’t’;
            lcd_str_bottom[11] = ’o’;
            lcd_str_bottom[12] = ’n’;
            lcd_str_bottom[13] = ’e’;
            lcd_str_bottom[14] = ’ ’;
          }
          break;
        default:
          break;
      }
    }
  }

  // exiting alarm setting mode, and set global alarm state
  if(set_alarm == 1 && button_state != 0x02){
    alarm_is_set = 1;
    set_alarm = 0;
    strcpy(lcd_str_top, "Alarm set!\0");
  }
  // exiting radio mode, mute and set flags
  if(set_radio == 1 && button_state != 0x10){
    set_radio = 0;
    set_property(0x4000, 0x0000); // mute
  }
  // set flags for setting the time
  if(button_state == 0x01){
    set_time = 1;
    set_alarm = 0;
    set_radio = 0;
  }
  // set flags for setting the alarm
  if(button_state == 0x02){
    set_alarm = 1;
    set_time = 0;
    set_radio = 0;
  }
  // set flags for setting the radio
  if(button_state == 0x10){
    set_radio = 1;
    set_alarm = 0;
    set_time = 0;
  }
}

int process_encoders(int enc){
  PORTE &= (0 << PE6);              // flip the load bit on the shift reg
  PORTE |= (1 << PE6);
  SPDR = 0x00;                      // dummy SPI data
  while(bit_is_clear(SPSR, SPIF)) {}  // wait till data sent out (while loop)

  // SPDR now stores encoder information
  static uint8_t prev_spi_a = 0xFF;          // store the previous SPI packet
  static uint8_t prev_spi_b = 0xFF;          // store the previous SPI packet

  // flags for return outputs
  static int direction_a = 0;
  static int output_a = 0;

  static int direction_b = 0;
  static int output_b = 0;

  int return_val = 0;

  if(enc == 0){
  // update on new SPDR
  if((prev_spi_a | ENC_A) != (SPDR | ENC_A)){
    // sets initial direction (based on encoder A masks, output starts at 0)
    if((prev_spi_a | ENC_A) == 0xFF && (SPDR | ENC_A) == 0b11111110) {prev_spi_a = SPDR; direction_a = 1; output_a = 0;}
    if((prev_spi_a | ENC_A) == 0xFF && (SPDR | ENC_A) == 0b11111101) {prev_spi_a = SPDR; direction_a = -1; output_a = 0;}

    // checks 3/4 rotation state, signals output is ready for home position
    if((direction_a == -1) && (SPDR | ENC_A) == 0b11111110) {output_a = 1; prev_spi_a = SPDR;}
    if((direction_a == 1) && (SPDR | ENC_A) == 0b11111101) {output_a = 1; prev_spi_a = SPDR;}

    // disable output when going back
    if((direction_a == -1) && (SPDR | ENC_A) == 0b11111101) {output_a = 0; prev_spi_a = SPDR;}
    if((direction_a == 1) && (SPDR | ENC_A) == 0b11111110) {output_a = 0; prev_spi_a = SPDR;}

    // if ready for a return value, set it to the intended direction
    if((SPDR | ENC_A) == 0xFF && output_a == 1) {output_a = 0; prev_spi_a = SPDR; return_val += direction_a;}

    // if back at home position and output is 0, reset states
    else if((SPDR | ENC_A) == 0xFF && direction_a != 0) {output_a = 0; direction_a = 0; prev_spi_a = SPDR;}
  }
  }

  if(enc == 1){
  // same as above
  if((prev_spi_b | ENC_B) != (SPDR | ENC_B)){
    if((prev_spi_b | ENC_B) == 0xFF && (SPDR | ENC_B) == 0b11111011) {prev_spi_b = SPDR; direction_b = 1; output_b = 0;}
    if((prev_spi_b | ENC_B) == 0xFF && (SPDR | ENC_B) == 0b11110111) {prev_spi_b = SPDR; direction_b = -1; output_b = 0;}

    if((direction_b == -1) && (SPDR | ENC_B) == 0b11111011) {output_b = 1; prev_spi_b = SPDR;}
    if((direction_b == 1) && (SPDR | ENC_B) == 0b11110111) {output_b = 1; prev_spi_b = SPDR;}

    if((direction_b == -1) && (SPDR | ENC_B) == 0b11110111) {output_b = 0; prev_spi_b = SPDR;}
    if((direction_b == 1) && (SPDR | ENC_B) == 0b11111011) {output_b = 0; prev_spi_b = SPDR;}

    if((SPDR | ENC_B) == 0xFF && output_b == 1) {output_b = 0; prev_spi_b = SPDR; return_val += direction_b;}
    else if((SPDR | ENC_B) == 0xFF && direction_b != 0) {output_b = 0; direction_b = 0; prev_spi_b = SPDR;}
  }
  }

  return return_val;
}

// grab button states, and display to bar graph
void update_bar(void){
  static uint8_t temp;
  temp = button_state;
  if(alarm_is_set == 1){temp += 0x80;}
  if(radio_or_alarm == 1){temp += 0x20;}

  SPDR = temp;                       //load SPDR to send to bar graph
  while(bit_is_clear(SPSR, SPIF)) {}  //wait till data sent out (while loop)
  PORTB |= (1 << PB0);          //HC595 output reg - rising edge...
  PORTB &= (0 << PB0);          //and falling edge
}

void setup_ports(void){
  // 1 for output, 0 for input
        DDRB |= (1 << PB4) | (1 << PB5) | (1 << PB6) | (1 << PB7);   //set port bits 4-7 B as outputs
  DDRE = 0x40;  // set port E bit 6 as output
  DDRC |= (1 << PC0) | (1 << PC1) | (1 << PC7);  // set PC0/1 as alarm output, PC7 for volume
}

void spi_init(void){
  DDRB |= (1 << PB0) | (1 << PB1) | (1 << PB2) | (1 << PB3); //Turn on SS, MOSI, SCLK, MISO
  SPCR |= (1 << SPE) | (1 << MSTR); //enable SPI, master mode
  SPSR |= (1 << SPI2X); // double speed operation
}

void tcnt0_init(void){
  ASSR |= (1 << AS0);             // enable 32kHz external clock
  TIMSK |= (1<<TOIE0);             //enable interrupts
  TCCR0 |= (0 << CS02) | (0 << CS01) | (1 << CS00);  //normal mode, no prescale
}

void init_alarm(void){
  TIMSK |= (1 << OCIE1A);       // enable interrupt
  TCCR1A = 0x00;                // set WGM to 0, disable output compare
  TCCR1B |= (1 << WGM12);       // 256 prescale
  TCCR1C = 0x00;                // no force compare
  OCR1A = 0x0000;               // set PWM duty cycle for interrupt
}

void init_volume(void){
  DDRE |= (1 << PE3);

  TCCR3A |= (1 << COM3A1) | (1 << WGM31);
  TCCR3B |= (1 << WGM32) | (1 << CS30);
  TCCR3C = 0x00;

  OCR3A = 300;
}

void init_brightness(void){
  //Initalize ADC and its ports
  DDRF  &= ~(_BV(DDF7)); //make port F bit 7 is ADC input
  PORTF &= ~(_BV(PF7));  //port F bit 7 pullups must be off

  // ADC enable
  ADMUX |= (1 << REFS0) | (1 << MUX0) | (1 << MUX1) | (1 << MUX2); //single-ended, input PORTF bit 7, right adjusted, 10 bits
  ADCSRA |= (1 << ADEN) | (1 << ADPS2) | (1 << ADPS0); //ADC enabled, don’t start yet, single shot mode

  //TCNT2 setup for providing the brightness control
  //fast PWM mode, TOP=0xFF, clear on match, clk/128
  //output is on PORTB bit 7
  // TCCR2 = (1<<WGM21) | (1<<WGM20) | (1<<COM21) | (1<<COM20) | (1<<CS20) | (1<<CS21);
  TCCR2 = (1<<WGM21) | (1<<WGM20) | (1<<COM21) | (1<<CS20) | (1<<CS21);
}

void update_brightness(void){
  static uint16_t adc_result;     //holds adc result

  ADCSRA |= (1 << ADSC);                 //poke ADSC and start conversion
  while(bit_is_clear(ADCSRA, ADIF)){}    //spin while interrupt flag not set
  ADCSRA |= (1 << ADIF);                //its done, clear flag by writing a one
  adc_result = ADC;


  // adc output is expected to be ~900-1000 for dark, <50 when bright
  if(adc_result <= 100){OCR2 = 0x00;}
  else if(adc_result <= 200){OCR2 = 0x30;}
  else if(adc_result <= 300){OCR2 = 0x50;}
  else if(adc_result <= 400){OCR2 = 0x70;}
  else if(adc_result <= 500){OCR2 = 0x80;}
  else if(adc_result <= 600){OCR2 = 0xA0;}
  else if(adc_result <= 700){OCR2 = 0xB0;}
  else if(adc_result <= 800){OCR2 = 0xC0;}
  else if(adc_result <= 850){OCR2 = 0xC7;}
  else if(adc_result <= 900){OCR2 = 0xD0;}
  else if(adc_result <= 910){OCR2 = 0xD7;}
  // else if(adc_result <= 920){OCR2 = 0xE0;}
  // else if(adc_result <= 930){OCR2 = 0xE7;}
  // else{OCR2 = 0xE7;}
  else{OCR2 = 0xE0;}

  /*
  if(adc_result <= 100){OCR2 = 0x00;}
  else if(adc_result <= 150){OCR2 = 0x30;}
  else if(adc_result <= 200){OCR2 = 0x50;}
  else if(adc_result <= 250){OCR2 = 0x70;}
  else if(adc_result <= 300){OCR2 = 0x80;}
  else if(adc_result <= 375){OCR2 = 0xA0;}
  else if(adc_result <= 450){OCR2 = 0xB0;}
  else if(adc_result <= 525){OCR2 = 0xC0;}
  else if(adc_result <= 600){OCR2 = 0xC7;}
  else if(adc_result <= 675){OCR2 = 0xD0;}
  else if(adc_result <= 750){OCR2 = 0xD7;}
  else if(adc_result <= 920){OCR2 = 0xE0;}
  else if(adc_result <= 930){OCR2 = 0xE7;}
  // else{OCR2 = 0xE7;}
  else{OCR2 = 0xF0;}
  */
  // OCR2 = 0xD0;
  // OCR2 = 0xF0;

  // bound ADC values, darker room
  // if (adc_result < 380){OCR2 = 255;}
  // else{OCR2 = (adc_result * -0.35) + 310;}

  // if (adc_result < 220){OCR2 = 220;}
  // else{OCR2 = (adc_result * -0.3) + 210;}
}

void init_strings(){
  lcd_str_bottom[10] = 't';
  lcd_str_bottom[11] = 'o';
  lcd_str_bottom[12] = 'n';
  lcd_str_bottom[13] = 'e';
  lcd_str_bottom[14] = ' ';
}

// refresh lcd with our 2 strings, called every 0.5 sec
void update_lcd(){
  line1_col1();
  string2lcd(" ");
  string2lcd(lcd_str_top);
  line2_col1();
  string2lcd(" ");
  string2lcd(lcd_str_bottom);
}

ISR(TIMER1_COMPA_vect){
  // play square wave if alarm is active
  if(play_alarm == 1 && snooze == 0){
    PORTC ^= 0x03;
    // increment and reset value for annoying tone
    OCR1A += 10;
    if(OCR1A >= 0x09FF) {OCR1A = 0;}
  }
  // otherwise disable output
  else{
    TCCR1B &= (0 << CS11) | (0 << CS10);
  }
}

//1 second counter
ISR(TIMER0_OVF_vect){
  static int timer0_count = 0;
  static int seconds = 0;
  static int colon_state = 0;
  static uint8_t snooze_cnt = 0;

  // tcnt0 overflows every 8 ms, 125 * 8ms = 1sec
  // if(timer0_count == 125){
  if(timer0_count == 128){
    // flip colon every second
    if(colon_state == 0 && set_radio != 1){
      segment_data[4] = dec_to_7seg[11];
      colon_state = 1;
    }
    else{
      segment_data[4] = dec_to_7seg[10];
      colon_state = 0;
    }

    seconds++;

    // cycle through snooze
    if(snooze){snooze_cnt++;}
    if(snooze_cnt == 10){
    // if(snooze_cnt == 600){
      snooze = 0;
      snooze_cnt = 0;
    }

    //increment the time
    if(seconds == 60){
    // if(seconds == 1){
      time++;
      seconds = 0;
    }

    // reset loop
    timer0_count = 0;

    // update displays
    // update_lcd();
  }

  // update LCD every 0.5 sec
  if(timer0_count == 64){update_lcd();}

  timer0_count++;
}

void init_radio(){
  DDRE  |= 0x04; //Port E bit 2 is active high reset for radio
  PORTE |= 0x04; //radio reset is on at powerup (active high)
  //hardware reset of Si4734
  PORTE &= ~(1<<PE7); //int2 initially low to sense TWI mode
  DDRE  |= 0x80;      //turn on Port E bit 7 to drive it low
  PORTE |=  (1<<PE2); //hardware reset Si4734
  _delay_us(200);     //hold for 200us, 100us by spec
  PORTE &= ~(1<<PE2); //release reset
  _delay_us(30);      //5us required because of my slow I2C translators I suspect
                      //Si code in "low" has 30us delay...no explaination given
  DDRE  &= ~(0x80);   //now Port E bit 7 becomes input from the radio interrupt
}

void set_volume(){
  // use encoders to cycle between volume levels, bounded range
  static int volume_state = 3;
  volume_state += process_encoders(0);
  if(volume_state > 5) {volume_state = 5;}
  else if(volume_state < 0) {volume_state = 0;}

  // assign OCR3A to set volume level
  switch (volume_state)
  {
  case 0:
    OCR3A = 0;
    break;
  case 1:
    OCR3A = 200;
    break;
  case 2:
    OCR3A = 250;
    break;
  case 3:
    OCR3A = 300;
    break;
  case 4:
    OCR3A = 350;
    break;
  case 5:
    OCR3A = 400;
  default:
    break;
  }
}

// routine for UART RX from mega168
ISR(USART0_RX_vect){
  static char rx_char;
  static uint8_t  i;

  // start rcv portion
  rx_char = UDR0;

  uart_str[i++] = rx_char;  //store in array

  // end of message
  if(rx_char == ’\0’){
    rcv_rdy = 1;

    uart_str[--i]  = (’ ’);     //clear the count field
    uart_str[i+1]  = (’ ’);
    uart_str[i+2]  = (’ ’);
    i = 0;
  }
}

//***********************************************************************************
uint8_t main(){

  //setup Port I/O, init functions, and interrupt enable
  setup_ports();
  encode_chars();
  tcnt0_init();
  init_brightness();
  init_volume();
  spi_init();
  init_alarm();
  lcd_init();
  init_strings();
  uart_init();
  init_twi();
  init_radio();
  set_volume();
  sei();

  // write start command to lm73
  lm73_wr_buf[0] = 0x00;
  twi_start_wr(LM73_ADDRESS, lm73_wr_buf, 1);

  _delay_ms(2);

  // power up routine for si4734 radio board
  fm_pwr_up();        //power up radio
  while(twi_busy()){} //spin while TWI is busy
  _delay_ms(150);
  fm_tune_freq();     //tune to frequency

  set_property(0x4000, 0x0000); // mute

  while(1){
    // process buttons to set UI states
    process_buttons();

    if(set_time == 1){
      // increment minutes
      time += process_encoders(0);
      //increment hours
      time += process_encoders(1) * 100;
    }

    else if (set_alarm == 1){
      // increment minutes
      alarm += process_encoders(0);
      // increment hours
      alarm += process_encoders(1) * 100;
    }

    else if (set_radio == 1){
      set_volume();
      int enc_temp = process_encoders(1);
      if(enc_temp != 0){
        // tune radio, set FM range
        current_fm_freq += 20 * enc_temp;
        if(current_fm_freq > 10800){current_fm_freq = 8810;}
        if(current_fm_freq < 8800){current_fm_freq = 10790;}
        fm_tune_freq();     //tune to frequency
      }
    }

    // if back at "home" in the UI, encoders set volume
    else {set_volume();}

    // bound time and alarm counts to 24 hr clock
    if(time % 100 == 60){time += 40;}
    if(time % 100 == 99){time -= 40;}
    if(time >= 2400){time = 0;}
    if(time < 0){time = 2359;}

    if(alarm % 100 == 60){alarm += 40;}
    if(alarm % 100 == 99){alarm -= 40;}
    if(alarm >= 2400){alarm = 0;}
    if(alarm < 0){alarm = 2359;}

    // break up the disp_value to 4, BCD digits in the array
    if(set_alarm == 1){segsum(alarm);}
    else if(set_radio == 1){
      int temp = current_fm_freq;
      radio_segsum(temp / 10);
    }
    else{segsum(time);}

    // update displays
    update_brightness();
    update_bar();
    update_7seg();

    if(alarm_is_set == 1){
      // enable here depending on snooze
      if(time == alarm){
        play_alarm = 1;
      }
      // play either the radio, or the tone
      if(play_alarm && snooze == 0){
        // play sound, enable interrupt
        if(radio_or_alarm == 0){
          TCCR1B |= (1 << WGM12) | (1 << CS11) | (1 << CS10);
          if(radio_state == 1){
            set_property(0x4000, 0x0000); // mute radio
            radio_state = 0;
          }
        }
        else{
          if(radio_state == 0){
            set_property(0x4000, 0x003F); // unmute
            fm_tune_freq();
            radio_state = 1;
          }
          TCCR1B &= (0 << CS11) | (0 << CS10);
        }
      }
      // mute the radio when snoozed
      else{
        if(radio_state == 1){
          set_property(0x4000, 0x0000); // mute
          radio_state = 0;
        }
      }
    }

    // check new UART data
    if(rcv_rdy == 1){
      rcv_rdy = 0;
      // place new data from 168 board into lcd strings
      lcd_str_bottom[3] = ’ ’;
      lcd_str_bottom[4] = uart_str[1];
      lcd_str_bottom[5] = uart_str[2];
      // lcd_str_bottom[6] = ’C’;
      lcd_str_bottom[6] = uart_str[3];
      lcd_str_bottom[7] = uart_str[4];
      lcd_str_bottom[8] = ’C’;
      lcd_str_bottom[9] = ’ ’;
    }

    // read temp data
    twi_start_rd(LM73_ADDRESS, lm73_rd_buf, 2);

    // main loop delay
    // _delay_ms(2);
    _delay_ms(1);

    // shift new temp data in, and store result
    lm73_temp = lm73_rd_buf[0];
    lm73_temp = (lm73_temp << 8);
    lm73_temp |= lm73_rd_buf[1];
    lm73_temp = (lm73_temp >> 7);

    static char temp[16];
    itoa(lm73_temp, temp, 10);

    // place temp data into lcd string
    lcd_str_bottom[0] = temp[0];
    lcd_str_bottom[1] = temp[1];
    lcd_str_bottom[2] = ’C’;
  }

  return 0;
}//main
```