```
// lab6.c
// Conor Wolfin
// 10.26.2018

//  HARDWARE SETUP:
//  PORTA is connected to the segments of the LED display. and to the pushbutton
s.
//  PORTA.0 corresponds to segment a, PORTA.1 corresponds to segement b, etc.
//  PORTB bits 4-6 go to a,b,c inputs of the 74HC138.
//  PORTB bit 7 goes to the PWM transistor base.

#define ZERO_DIGIT  0b11000000
#define ONE_DIGIT   0b11111001
#define TWO_DIGIT   0b10100100
#define THREE_DIGIT 0b10110000
#define FOUR_DIGIT  0b10011001
#define FIVE_DIGIT  0b10010010
#define SIX_DIGIT   0b10000011
#define SEVEN_DIGIT 0b11111000
#define EIGHT_DIGIT 0b10000000
#define NINE_DIGIT  0b10011000
#define A_DIGIT     0b10001000
#define B_DIGIT     0b10000011
#define C_DIGIT     0b11000110
#define D_DIGIT     0b10100001
#define E_DIGIT     0b10000110
#define F_DIGIT     0b10001110
#define DP_DIGIT    0b01111111
#define COLON_DIGIT 0b00000100
#define VOL_PIN PE5
#define ON 1
#define OFF 0
#define CW 1
#define CCW -1
#define SNOOZEOFF 0
#define SNOOZEON 1
#define SNOOZEALARM 2
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <string.h>
#include <stdlib.h>
#include "hd44780.h"
#include "lm73_functions_skel.h"
#include "twi_master.h"
#include "uart_functions.h"
#include "si4734.h"

// holds data to be sent to the segments. logic zero turns segment on
uint8_t segment_data[5] = {255,255,255,255,255};

// decimal to 7-segment LED display encodings, logic "0" turns on segment
uint8_t dec_to_7seg[12] = {ZERO_DIGIT, ONE_DIGIT, TWO_DIGIT, THREE_DIGIT, FOUR_D
IGIT, FIVE_DIGIT, SIX_DIGIT, SEVEN_DIGIT, EIGHT_DIGIT, NINE_DIGIT, DP_DIGIT, COL
ON_DIGIT};

uint8_t hex_to_7seg[16] = {ZERO_DIGIT, ONE_DIGIT, TWO_DIGIT, THREE_DIGIT, FOUR_D
IGIT, FIVE_DIGIT, SIX_DIGIT, SEVEN_DIGIT, EIGHT_DIGIT, NINE_DIGIT, A_DIGIT, B_DI
GIT, C_DIGIT, D_DIGIT, E_DIGIT, F_DIGIT};

// Flag indicating when interrupt was triggered
volatile uint8_t secondsFlag=1;

// Flag indicating when snooze was triggered
volatile uint8_t snoozeFlag=0;

// Holds value that stores the time of the clock
volatile uint16_t currentTime = 0;
```

```
// Indicates whether Dec or Hex mode
uint8_t DecHex = 10;

//True when data sent from uart is complete
uint8_t data_complete = 0;

//Holds data that is sent to LCD
char LCD_message[42] = {' '};

//Global indicating alarm is set
uint8_t alarmGlobal = OFF;

//Button in position 1
uint8_t buttonPos = 0;

char    lcd_string_array[16];  //holds a string to refresh the LCD
uint8_t i;                     //general purpose index

extern uint8_t lm73_wr_buf[2];//................
extern uint8_t lm73_rd_buf[2];//................

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(1 <<
 PE2);
extern volatile uint16_t current_fm_freq; //0x2706, arg2, arg3; 99.9Mhz, 200khz
steps

/*********************************************************************/
//              SPI_read(uint8_t currentBarGraph)
// Sends the bar graph data and then reads the SPI port.
/*********************************************************************/
uint16_t SPI_read(uint8_t currentBarGraph){
  PORTB &= 0x7F;
  SPDR = currentBarGraph;
  while (bit_is_clear(SPSR,SPIF)){} //wait till 8 bits have been sent
  //PORTD = 0xFF;
  //PORTD = 0x00;
  PORTE = 0x00;         //TODO: May cause problems with RADIO
  PORTE = 0xFF;
  return(SPDR); //return incoming data from SPDR
}
//******************************************************************************
//                            chk_buttons(uint8_t 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.
// Edited to have a state array of size 8 for each button
//******************************************************************************
uint8_t chk_buttons(uint8_t buttons) {
  //Gansels debounce with the state as an array that is used to check against th
e values that buttons is at
  static uint16_t state[8] = {0}; //holds present state
  state[buttons] = (state[buttons] << 1) | (! bit_is_clear(PINA, buttons)) | 0xE
000;
  if (state[buttons] == 0xF000) return 1;
  return 0;
}

//******************************************************************************
*****
//                                   segment_sum

// takes a 16-bit binary input value and places the appropriate equivalent 4 dig
it
```

```
// BCD segment code in the array segment_data for display.

// array is loaded at exit as:  |digit3|digit2|colon|digit1|digit0|
//******************************************************************************
*****
void segsum(uint16_t sum) {

  uint8_t  i = 0;
  uint8_t  digitNum = 1;
  uint16_t sumPlaceHolder = sum;
  while(i < 4 && sumPlaceHolder > (DecHex-1))
  {
    sumPlaceHolder /= DecHex;
    digitNum++;
    i++;
  }
  // Parses 0-4 digits into seperate segment_data[] locations
  switch(digitNum)
  {
    case 1:
      segment_data[4] = hex_to_7seg[sum];
      segment_data[3] = hex_to_7seg[0];// 0xFF;
      segment_data[1] = hex_to_7seg[0];//0xFF;
      segment_data[0] = hex_to_7seg[0];//0xFF;
      break;
    case 2:
      segment_data[4] = hex_to_7seg[(sum % DecHex)];
      segment_data[3] = hex_to_7seg[(sum / DecHex)];
      segment_data[1] = hex_to_7seg[0];//0xFF;
      segment_data[0] = hex_to_7seg[0];// 0xFF;
      break;
    case 3:
      segment_data[4] = hex_to_7seg[sum % DecHex];
      segment_data[3] = hex_to_7seg[(sum/DecHex)%DecHex];
      segment_data[1] = hex_to_7seg[(sum/(DecHex*DecHex))];
      segment_data[0] = hex_to_7seg[0];//0xFF;
      break;
    case 4:
      segment_data[4] = hex_to_7seg[sum % DecHex];
      segment_data[3] = hex_to_7seg[(sum/DecHex)%DecHex];
      segment_data[1] = hex_to_7seg[(sum/(DecHex*DecHex))%DecHex];
      segment_data[0] = hex_to_7seg[sum/(DecHex*DecHex*DecHex)];
    default:
      break;
  }
}

//******************************************************************************
*****
//                                    displaySwitch

// Takes the segment_data[] array that has the #_DIGIT values and displays it to
 the
// current LED digit (displayValue) and returns the next value that will be used
 for displaying
//******************************************************************************
*****
uint8_t displaySwitch(uint8_t displayValue)
{
  switch(displayValue){
    case 0:
      PORTB = 0x07;
      PORTA = segment_data[4];
      break;
    case 1:
      PORTB = 0x17;
      PORTA = segment_data[3];
      break;
    case 2:
```

```
      PORTB = 0x27;
      PORTA = segment_data[2];
      break;
    case 3:
      PORTB = 0x37;
      PORTA = segment_data[1];
      break;
    case 4:
      PORTB = 0x47;
      PORTA = segment_data[0];
      break;
    default:
      break;
  }
  _delay_ms(1);                             //Adds delay for screen congruency
  if(displayValue == 4) return 0;        //Starts display back to 0
  return ++displayValue;
}

//******************************************************************************
*****
//                     ButtonCheck(uint8_t buttonMode)

// Function for when the interrupt was triggered, executing next main loop
// Takes in the current value outputted and returns the adjusted value based on
the number
//******************************************************************************
*****
uint8_t ButtonCheck(uint8_t buttonMode)
{
  //PORTA to input w/ pullups
  DDRA  = 0x00;
  PORTA = 0xFF;
  //enable tristate buffer for pushbutton switches via DEC7 on the encoder
  PORTB = 0x70;
  uint8_t buttonLoop = 0;
  _delay_us(10);                  //BUG"Added delay to get first button to work, n
eed better fix
  while(buttonLoop < 8)
  {
    if(chk_buttons(buttonLoop))
    {
      buttonMode ^= (1<<buttonLoop);
    }
    buttonLoop++;
  }
  DDRA = 0xFF;
  return buttonMode;
}

//******************************************************************************
*****
//                     ClockCounterCorrection(uint16_t displayValue)

// Takes in a value and ensures it is in the format a clock would use
// Returns value in the format HH:MM
//******************************************************************************
*****
uint16_t ClockCounterCorrection(uint16_t displayValue)
{
  static uint8_t displayValueHours;
  static uint8_t displayValueMins;

  displayValueHours = (displayValue / 60);
  displayValueMins  = (displayValue - (60 * displayValueHours));
  displayValue      = ((displayValueHours * 100) + displayValueMins);

  return displayValue;
}
```

```
//******************************************************************************
*****
//     int8_t EncoderValueDirection(uint8_t currentEncoderValue, uint8_t urrentA
djustment)
// Checks direction of encoders turning and returns if the turn was CW or CCW
// Returns positive currentAdjustment value (CW) or negative currentAdjustment v
alue (CCW)
//******************************************************************************
***
int8_t EncoderValueDirection(uint8_t currentEncoderValue)
{
  //Tests current encoder value against previous encoder value
  //Tests if forward by value 0x00 --> 0x01, returns pos adjustment value
  //Tests else if reverse, 0x01 --> 0x00, returns neg adjustment value
  //First If statment checks   0B000000__
  //Second If statment checks  0B0000__00

  static uint8_t previousEncoderValue = 0x0F;

  if((previousEncoderValue & 0x03) == 0x00 && (currentEncoderValue & 0x03) == 0x
01)
  {
    previousEncoderValue = (currentEncoderValue & 0x0F);
    return CW;
  }
  else if((previousEncoderValue & 0x03) == 0x01 && (currentEncoderValue & 0x03)
== 0x00)
  {
    previousEncoderValue = (currentEncoderValue & 0x0F);
    return CCW;
  }

  //Checks the second Encoder
  if((previousEncoderValue & 0x0C) == 0x00 && (currentEncoderValue & 0x0C) == 0x
04)
  {
    previousEncoderValue = (currentEncoderValue & 0x0F);
    return CW;
  }
  else if((previousEncoderValue & 0x0C) == 0x04 && (currentEncoderValue & 0x0C)
== 0x00)
  {
    previousEncoderValue = (currentEncoderValue & 0x0F);
    return CCW;
  }
  previousEncoderValue = currentEncoderValue;
  return 0;
}

//******************************************************************************
//                            External Interrupt 7 ISR
// Handles the interrupts from the radio that tells us when a command is done.
// The interrupt can come from either a "clear to send" (CTS) following most
// commands or a "seek tune complete" interrupt (STC) when a scan or tune comman
d
// like fm_tune_freq is issued. The GPIO2/INT pin on the Si4734 emits a low
// pulse to indicate the interrupt. I have measured but the datasheet does not
// confirm a width of 3uS for CTS and 1.5uS for STC interrupts.
//
// I am presently using the Si4734 so that its only interrupting when the
// scan_tune_complete is pulsing. Seems to work fine. (12.2014)
//
// External interrupt 7 is on Port E bit 7. The interrupt is triggered on the
// rising edge of Port E bit 7.  The i/o clock must be running to detect the
// edge (not asynchronouslly triggered)
//******************************************************************************
ISR(INT7_vect){STC_interrupt = TRUE;}

//******************************************************************************
```

```
*****
//                                     ISR(TIMER0_OVF_vect)

// Triggered when TimerCounter0 overflows (every second)
// Toggles COLON bits
// Counts Seconds, rolls over every 60, increments and rolls clock over
// Counts up too 255 (which inidcates 1 sec with 32Khz clk & 128 prescale)
//******************************************************************************
***
ISR(TIMER0_OVF_vect)
{
  static uint8_t currentSeconds = 0;
  static uint8_t snoozeTimer = 0;
  segment_data[2] ^= 0x03;
  // Second Counter
  if(currentSeconds < 60)
  {
    if(snoozeFlag == SNOOZEON)
    {
      snoozeTimer++;
      segment_data[2] ^= 0x04;
    }

    currentSeconds++;
  }else
  {
    currentTime++;
    currentSeconds = 0;
  }

  if(snoozeTimer == 10)
  {
    snoozeFlag = SNOOZEALARM;
    snoozeTimer = 0;
  }
}


//******************************************************************************
*****
//                                     ISR(USART0_RX_vect)

// Triggers when uart message has sent a single character
// fills buffer with message being received
// Once SPACE char is received message is considered over
// When message is over, reset counter for filling buffer, and indicate message
sent
//******************************************************************************
***
char uart_buff[42] = {' '};
ISR(USART0_RX_vect)
{
  static uint8_t counter = 0;
  uart_buff[counter] = UDR0;
  UDR0 = 0;
  if(uart_buff[counter] == ' ')
  {
    data_complete = 1;
    counter = 0;
  }else
  {
    counter++;
  }
}


//******************************************************************************
*****
//                                     ISR(TIMER1_OVF_vect)
// Triggered when TimerCounter1 overflows
//
//*******************************************************************************
***
ISR(TIMER1_COMPA_vect)
{   PORTD ^= 0b10000000;      //flips the bit, creating a tone
}
//*******************************************************************************
*****
//                                   uint16_t AlarmSet()

// Function entered when the user presses the first button on the button board
// loops until user to inputs time (w/ encoder)
// Once user presses same button, Alarm is set and function is exitted//
//*******************************************************************************
***
uint16_t AlarmSetMode(uint8_t alarmOffset)
{
  static uint8_t  currentEncoderValue  = 0;
  static uint16_t encodersDisplayValue  = 0;
  int8_t currentAdjustmentValue = 0;
  static uint16_t  offsetVal = 1439;
  currentEncoderValue = (SPI_read(currentAdjustmentValue));
  currentAdjustmentValue = EncoderValueDirection(currentEncoderValue);
  encodersDisplayValue += currentAdjustmentValue;
  // Checks if the clock will roll backwards behind 0
  // 1439 is clock time for 23 : 59
  if(!alarmOffset)
  {
    offsetVal = 1439;
  }else{
    offsetVal = 779;
  }

  if((encodersDisplayValue == 0) && (currentAdjustmentValue == CCW))
  {
    encodersDisplayValue = offsetVal;
  }else if(encodersDisplayValue > offsetVal)
  {
    if(!offsetVal){ encodersDisplayValue = 0;}
    else{encodersDisplayValue = 0;}//       encodersDisplayValue = 60;}
  }
  return encodersDisplayValue;
}
//*******************************************************************************
*****
//                                   uint16_t VolumeSetMode()

// Default functiond when in Clock mode
// Adjusts volume OCR3C (w/ encoder)
//*******************************************************************************
***
uint16_t VolumeSetMode()
{
  static uint8_t  currentEncoderValue  = 0;
  static uint16_t encodersVolumeValue  = 0xE0;
  int8_t currentAdjustmentValue = 0;

  currentEncoderValue = (SPI_read(currentAdjustmentValue));
  currentAdjustmentValue = EncoderValueDirection(currentEncoderValue);
  encodersVolumeValue += currentAdjustmentValue;
  if((encodersVolumeValue == 0) && (currentAdjustmentValue == CCW))
  {
    encodersVolumeValue = 0xFF;
  }else if(encodersVolumeValue > 0xFF)
  {
    encodersVolumeValue = 0;
  }
```

```
  return encodersVolumeValue*2;
}

//*******************************************************************************
*****
//                                void LocalTempSensor()
// Checks the temperature of the onboard temperature sensor
// Outputs the temperature to the LCD screen
//*******************************************************************************
*****
void LocalTempSensor(uint16_t lm73_temp)
{
  twi_start_rd(LM73_READ,lm73_rd_buf,2);//................ //read temperature da
ta from LM73 (2 bytes)
  _delay_us(500);    //wait for it to finish
  lm73_temp = lm73_rd_buf[0];//................ //save high temperature byte int
o lm73_temp
  lm73_temp = (lm73_temp<<8);//................ //shift it into upper byte
  lm73_temp |= lm73_rd_buf[1];//................ //"OR" in the low temp byte to
lm73_temp
  itoa(lm73_temp>>7 , lcd_string_array, 10);//................ //convert to stri
ng in array with itoa() from avr-libc

  // Add message to LCD_message buffer
  LCD_message[0] = ’I’;
  LCD_message[1] = ’N’;
  LCD_message[2] = ’T’;
  LCD_message[3] = ’:’;
  LCD_message[4] = lcd_string_array[0];
  LCD_message[5] = lcd_string_array[1];
  LCD_message[6] = ’ ’;
  LCD_message[7] = ’E’;
  LCD_message[8] = ’X’;
  LCD_message[9] = ’T’;
  LCD_message[10] = ’:’;
  LCD_message[11] = uart_buff[1];
  LCD_message[12] = uart_buff[2];
  LCD_message[13] = uart_buff[3];
  LCD_message[14] = ’ ’;
  LCD_message[15] = ’ ’;

  // Print out either empty spaces or ALARM if alarm activated
  uint8_t fill;
  if(alarmGlobal == OFF)
  {
    fill = 16;
  }else{
    LCD_message[16] = ’A’;
    LCD_message[17] = ’L’;
    LCD_message[18] = ’A’;
    LCD_message[19] = ’R’;
    LCD_message[20] = ’M’;
    fill = 21;
  }
  while(fill != 41)
  {
    LCD_message[fill] = ’ ’;
    fill++;
  }

  // Print to LCD if data has completed sending
  // Also needs the encoders to not be in use
  if((buttonPos == 0) && data_complete)
  {
    refresh_lcd(LCD_message);
  }
}

//*******************************************************************************
```

```
*****
//                              void rdaio_init()
// Initialize the radio to work as well as set the frequency and reseting the pi
ns
// used for the radio
// *ROGERS CODE* Thanks Roger!
//******************************************************************************
*****
void radio_init()
{
//Port E inital values and setup.  This may be different from yours for bits 0,1
,6.

//                           DDRE:  0 0 0 0 1 0 1 1
//   (^ edge int from radio) bit 7--| | | | | | | |--bit 0 USART0 RX
//(shift/load_n for 74HC165) bit 6----| | | | | |----bit 1 USART0 TX
//                           bit 5------| | | |------bit 2 (new radio reset, act
ive high)
//                  (unused) bit 4--------| |--------bit 3 (TCNT3 PWM output for
 volume control)

DDRE  |= 0x04; //Port E bit 2 is active high reset for radio
DDRE  |= 0x40; //Port E bit 6 is shift/load_n for encoder 74HC165
DDRE  |= 0x08; //Port E bit 3 is TCNT3 PWM output for volume
DDRE  |= VOL_PIN;
PORTE |= 0x04; //radio reset is on at powerup (active high)
PORTE |= 0x40; //pulse low to load switch values, else its in shift mode


//Given the hardware setup reflected above, here is the radio reset sequence.
//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
DDRE  &= ~(0x80);   //now Port E bit 7 becomes input from the radio interrupt

//Once its setup, you can set the station and get the received signal strength.

current_fm_freq = 8870; //0x2706, arg2, arg3; 99.9Mhz, 200khz steps
fm_pwr_up();            //power up radio
_delay_ms(300);
while(twi_busy()){} //spin while TWI is busy
fm_tune_freq();     //tune to frequency
}

//RADIO ON
void radio_on()
{
fm_pwr_up();            //power up radio
_delay_ms(300);
while(twi_busy()){} //spin while TWI is busy
fm_tune_freq();     //tune to frequency
}
//******************************************************************************
*****
//                              void init()
// Initialize all of the registers at the start of main
//
//******************************************************************************
*****
void init()
{
//  TCNT0 - Norm Mode | Using external 32kHz clock | 128 Prescale       !Count t
o 250 using uint8_t to reach 1 second for clock!
//  TCNT1 - CTC  Mode | Pick freuquency      | Output too PD7           !Outputs
```

```
 to summing amp, which gets outputted to speaker!
//  TCNT2 - Fast PWM  | Output to PB7 (OC2)                             !Control
s brightness of LED Display!
//  TCNT3 - Fast PWM  | Output to PE5 (OC3C)                            !Control
s volume to Audio Amp!
DDRA  = 0xFF;                       //set port A as input

DDRB  = 0xFF;                       //set port B as outputs
DDRD  |= (1 << PD7);                //Sets PortD pin2 to output
DDRE  |= (1 << PE5) | (1 << PE6);   //Sets PortE Pin 6 & 5 to output
DDRE  |= (1 << PE2);                //Sets PortE Pin 2 for Radio_reset to output
PORTD = 0x00;                       //set port D to LOW
PORTB = 0x10;                       //set port B to start with LED1

ASSR   |= (1 << AS0);                   //Use external 32kHz clock
SPCR   |= (1 << SPE)    | (1 << MSTR);   //Enable SPI communication in mastermode
SPSR    = (1 << SPI2X);                 //SPI at 2x speed (8 MHz)
TIMSK  |= (1 << TOIE0) | (1 << OCIE1A); //enable interrupt on compare & overflow
 of TCNT1
TCCR0  |= (1 << CS00)  | (1 << CS02);   //normal mode, prescale by 128
TCCR1A  = 0;
TCCR1B |= (1 << WGM12);                 //CTC mode clear at TOP immediate
TCCR1C  = 0;
TCCR3A |= (1 << COM3C1) | (1 << WGM30); //Set as output compare to OC3C (PE5)
TCCR3A |= (1 << WGM32);
TCCR3B |= (1 << WGM32) | (1 << CS00);
OCR1A  = 0xF0F;
//OCR3C  = 0xFF;                //Volume
TCCR2  |= (1 << WGM21) | (1 << WGM20) | (1 << COM21) | (1 << CS21); // Set TCNT2
 to fast pwm outputting to OC2 (PB7)
ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);               // Set ADC p
rescalar to 128 - 125KHz sample rate @ 16MHz
ADMUX  |= (1 << REFS0); // Set ADC reference to AVCC
ADMUX  |= (1 << ADLAR); // Left adjust ADC result to allow easy 8 bit reading
ADCSRA |= (1 << ADFR);  // Set ADC to Free-Running Mode
ADCSRA |= (1 << ADEN);  // Enable ADC
ADCSRA |= (1 << ADSC);  // Start A2D Conversions
lcd_init();
init_twi();//................ //initalize TWI (twi_master.h)
uart_init();
sei();

radio_init();
// Radio Init;
//set to KRKT radio albany
}
//******************************************************************************
*****
int main()
{
init();
uint16_t lm73_temp;                     //a place to assemble the temperature fr
om the lm73
uint8_t  currentButtonsPressed = 0;     //Stores buttons that are currently pres
sed (holds value when pressed)
uint8_t  currentDisplayDigit = 0;       //Current LED to display on (0 == 1's di
git
uint16_t displayValue = 0;              //Current value to display on LEDs
uint16_t alarmValue = 1;                //Current value held by the alarm
uint8_t  alarmActivated = OFF;          //If the Alarm is ON or OFF, initialize
to OFF
uint8_t  alarmON = OFF;
uint8_t  alarmSET = ON;
uint8_t  alarmOffset = 0;

//set LM73 mode for reading temperature by loading pointer register
lm73_wr_buf[0] = (&lm73_temp);          //load lm73_wr_buf[0] with temperature p
ointer address
twi_start_wr(LM73_WRITE,lm73_wr_buf,2); //start the TWI write process
```

```c
_delay_ms(2);                           //wait for the xfer to finish

clear_display(); //clean up the display

while(1){

  // Button Functionality
  // Pole Buttons
  currentButtonsPressed = ButtonCheck(currentButtonsPressed);

  // Buttons (1):
  // Enter Setting mode (sets time or alarm)
  if(currentButtonsPressed == 0x01)
  {
      buttonPos    = 1;
      alarmValue   = AlarmSetMode(alarmOffset);
      displayValue = alarmValue;
  // Buttons (2):
  // Restart Mode (restarts everything)
  }else if(currentButtonsPressed == 0x02)
  {
      alarmActivated = OFF;
      alarmGlobal    = OFF;
      snoozeFlag = SNOOZEOFF;
      segment_data[2] |= (0xFF);
      currentButtonsPressed = (0x00);
    buttonPos = 0;
      OCR3C = 0;
      currentButtonsPressed = 0;
      currentDisplayDigit = 0;
      displayValue = 0;
      alarmValue = 1;
      //alarmActivated = OFF;
      //alarmGlobal    = OFF;
      alarmON = OFF;
      clear_display();

  // Buttons (1, 2):
  // Sets Alarm
  }else if(currentButtonsPressed == 0x03)
  {
      alarmActivated = ON;
      alarmGlobal = ON;
      segment_data[2] &= 0xFB;
      currentButtonsPressed = (0x00);
    buttonPos = 0;

  // Buttons (3):
  // SNOOZE if Alarm is Set/On
  }else if(currentButtonsPressed == 0x04)
  {
      if(alarmActivated)
      {
        TCCR1B &= (0 << CS11);
        TCCR1B &= (0 << CS12);
        OCR3C   = 0;
        snoozeFlag = SNOOZEON;
        alarmSET = ON;
      }
      currentButtonsPressed = (0x00);
    buttonPos = 0;

  // Buttons (1, 3):
  // Set Time
  }else if(currentButtonsPressed == 0x05)
  {
    currentTime = AlarmSetMode(alarmOffset);
    currentButtonsPressed = (0x00);
    buttonPos = 0;
```

```c

  // Display CurrentTime

  }else if(currentButtonsPressed == 0x08)
  {
    radio_on();
    //alarmOffset ^= 0x01;
    //currentButtonsPressed = (0x00);
  }else{
    displayValue = currentTime;
    currentButtonsPressed = (0x00);
  }

  // Brightness of LED based off Photoresistor
  if(ADCH >= 400)
  {
    OCR2 = 5;
  }else if(ADCH < 20)
  {
    OCR2 = 240;
  }else{
    OCR2 = 255- ADCH;
  }
  //OCR2  = ADCH//395 + (2 * (450 - ADCH));

  // Turn ON alarm if SNOOZE timedout
  if(snoozeFlag == SNOOZEALARM)
  {
    alarmActivated = ON;
    alarmGlobal = ON;
  }

  // Alarm is reached and activated, either by timer or by snooze reached
  // Play alarm
  if(alarmActivated && ((currentTime == alarmValue) || (snoozeFlag == SNOOZEALAR
M)) && (snoozeFlag != SNOOZEON) && (currentButtonsPressed != 0x01))
  {
    //TCCR1B |= (1 << WGM12) | (1<<CS11) | (1<<CS10);           //CTC mode clear
 at TOP immediate
    OCR3C   = VolumeSetMode();
    alarmON = ON;
    radio_init();
    buttonPos = 1;
    OCR3C  = 0x00;              //Volume
  }

  // Display 'ALARM' on LCD
  if(alarmActivated && alarmSET)
  {
    alarmSET = OFF;
  }

  LocalTempSensor(lm73_temp);
  // Turn minute input to HH:MM
  displayValue = ClockCounterCorrection(displayValue);

  // Display to LED screen
  segsum(displayValue);                                         //Divide the dec
imal value to the segment_data[] array
  currentDisplayDigit = displaySwitch(currentDisplayDigit);     //Display the cu
rrent values stored in segment_data[] to current LED

}//while
return 0;
}//main
```