```
// lab3.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 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"

// 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];//.................

void song0(uint16_t note); //Beaver Fight Song
//Mute is on PORTD
//set the hex values to set and unset the mute pin
//I used PORTD-PIN2
#define mute 0x04
#define unmute 0xFB
//Alarm is also on PORTD
//set the hex value for the alarm pin
//I used PORTD-PIN7
#define ALARM_PIN 0x80
#define NUM_SONGS 4
//set this variable to select the song
//(0-3, unless you add more)
volatile uint8_t song;
void play_song(uint8_t song, uint8_t note);
void play_rest(uint8_t duration);
void play_note(char note, uint8_t flat, uint8_t octave, uint8_t duration);
void music_off(void);
void music_on(void);
void music_init(void);
#define C0 0x1DDC
#define Db0 0x1C30
#define D0 0x1A9A
#define Eb0 0x1919
#define E0 0x17B2
#define F0 0x165D
#define Gb0 0x151D
#define G0 0x13ED
#define Ab0 0x12CE
#define A0 0x11C0
#define Bb0 0x10C0
#define B0 0x0FD0
#define C1 0x0EED
#define Db1 0x0E16
#define D1 0x0D4C
#define Eb1 0x0C8D
#define E1 0x0BD8
#define F1 0x0B2E
#define Gb1 0x0A8D
#define G1 0x09F6
#define Ab1 0x0967
#define A1 0x08DF
#define Bb1 0x0860
#define B1 0x07E7
#define C2 0x0776
#define Db2 0x070A
#define D2 0x06A5
#define Eb2 0x0646
#define E2 0x05EB
#define F2 0x0596
```

```
#define Gb2 0x0546
#define G2 0x04FA
#define Ab2 0x04B2
#define A2 0x046F
#define Bb2 0x042F
#define B2 0x03F3
#define C3 0x03BA
#define Db3 0x0384
#define D3 0x0352
#define Eb3 0x0322
#define E3 0x02F5
#define F3 0x02CA
#define Gb3 0x02A2
#define G3 0x027C
#define Ab3 0x0258
#define A3 0x0237
#define Bb3 0x0217
#define B3 0x01F9
#define C4 0x01DC
#define Db4 0x01C1
#define D4 0x01A8
#define Eb4 0x0190
#define E4 0x017A
#define F4 0x0164
#define Gb4 0x0150
#define G4 0x013D
#define Ab4 0x012B
#define A4 0x011B
#define Bb4 0x010B
#define B4 0x00FC
#define C5 0x00ED
#define Db5 0x00E0
#define D5 0x00D3
#define Eb5 0x00C7
#define E5 0x00BC
#define F5 0x00B1
#define Gb5 0x00A7
#define G5 0x009E
#define Ab5 0x0095
#define A5 0x008D
#define Bb5 0x0085
#define B5 0x007D
#define C6 0x0076
#define Db6 0x006F
#define D6 0x0069
#define Eb6 0x0063
#define E6 0x005D
#define F6 0x0058
#define Gb6 0x0053
#define G6 0x004E
#define Ab6 0x004A
#define A6 0x0046
#define Bb6 0x0042
#define B6 0x003E
#define C7 0x003A
#define Db7 0x0037
#define D7 0x0034
#define Eb7 0x0031
#define E7 0x002E
#define F7 0x002B
#define Gb7 0x0029
#define G7 0x0026
#define Ab7 0x0024
#define A7 0x0022
#define Bb7 0x0020
#define B7 0x001E
#define C8 0x001C
#define Db8 0x001B
#define D8 0x0019
```



```
#define Eb8 0x0018
#define E8 0x0015
#define F8 0x0012
#define Gb8 0x0010
#define G8 0x000D
#define Ab8 0x000B
#define A8 0x0009
#define Bb8 0x0007
#define B8 0x0005

volatile uint16_t beat;
volatile uint16_t max_beat;
volatile uint8_t  notes;
void play_note(char note, uint8_t flat, uint8_t octave, uint8_t duration) {
  //pass in the note, it's key, the octave they want, and a duration
  //function sets the value of OCR1A and the timer
  //note must be A-G
  //flat must be 1 (for flat) or 0 (for natural) (N/A on C or F)
  //octave must be 0-8 (0 is the lowest, 8 doesn't sound very good)
  //duration is in 64th notes at 120bpm
  //e.g. play_note('D', 1, 0, 16)
  //this would play a Db, octave 0 for 1 quarter note
  //120 bpm (every 32ms inc beat)
  PORTD &= unmute;       //unmute (just in case)
  beat = 0;              //reset the beat counter
  max_beat = duration;  //set the max beat
  switch (octave) {
    case 0: switch (note) {
      case 'A': if(flat){OCR1A=Ab0;}
        else {OCR1A=A0;}
        break;
      case 'B': if(flat){OCR1A=Bb0;}
        else {OCR1A=B0;}
        break;
      case 'C': OCR1A=C0;
        break;
      case 'D': if(flat){OCR1A=Db0;}
        else {OCR1A=D0;}
        break;
      case 'E': if(flat){OCR1A=Eb0;}
        else {OCR1A=E0;}
        break;
      case 'F': OCR1A=F0;
        break;
      case 'G': if(flat){OCR1A=Gb0;}
        else {OCR1A=G0;}
        break;
      }
      break;
    case 1: switch (note) {
      case 'A': if(flat){OCR1A=Ab1;}
        else {OCR1A=A1;}
        break;
      case 'B': if(flat){OCR1A=Bb1;}
        else {OCR1A=B1;}
        break;
      case 'C': OCR1A=C1;
        break;
      case 'D': if(flat){OCR1A=Db1;}
        else {OCR1A=D1;}
        break;
      case 'E': if(flat){OCR1A=Eb1;}
        else {OCR1A=E1;}
        break;
      case 'F': OCR1A=F1;
        break;
      case 'G': if(flat){OCR1A=Gb1;}
        else {OCR1A=G1;}
        break;
```

```
    }
    break;
  case 2: switch (note) {
    case 'A': if(flat){OCR1A=Ab2;}
      else {OCR1A=A2;}
      break;
    case 'B': if(flat){OCR1A=Bb2;}
      else {OCR1A=B2;}
      break;
    case 'C': OCR1A=C2;
      break;
    case 'D': if(flat){OCR1A=Db2;}
      else {OCR1A=D2;}
      break;
    case 'E': if(flat){OCR1A=Eb2;}
      else {OCR1A=E2;}
      break;
    case 'F': OCR1A=F2;
      break;
    case 'G': if(flat){OCR1A=Gb2;}
      else {OCR1A=G2;}
      break;
    }
    break;
  case 3: switch (note) {
    case 'A': if(flat){OCR1A=Ab3;}
      else {OCR1A=A3;}
      break;
    case 'B': if(flat){OCR1A=Bb3;}
      else {OCR1A=B3;}
      break;
    case 'C': OCR1A=C3;
      break;
    case 'D': if(flat){OCR1A=Db3;}
      else {OCR1A=D3;}
      break;
    case 'E': if(flat){OCR1A=Eb3;}
      else {OCR1A=E3;}
      break;
    case 'F': OCR1A=F3;
      break;
    case 'G': if(flat){OCR1A=Gb3;}
      else {OCR1A=G3;}
      break;
    }
    break;
  case 4: switch (note) {
    case 'A': if(flat){OCR1A=Ab4;}
      else {OCR1A=A4;}
      break;
    case 'B': if(flat){OCR1A=Bb4;}
      else {OCR1A=B4;}
      break;
    case 'C': OCR1A=C4;
      break;
    case 'D': if(flat){OCR1A=Db4;}
      else {OCR1A=D4;}
      break;
    case 'E': if(flat){OCR1A=Eb4;}
      else {OCR1A=E4;}
      break;
    case 'F': OCR1A=F4;
      break;
    case 'G': if(flat){OCR1A=Gb4;}
      else {OCR1A=G4;}
      break;
    }
    break;
  case 5: switch (note) {
```



```
    case 'A': if(flat){OCR1A=Ab5;}
      else {OCR1A=A5;}
      break;
    case 'B': if(flat){OCR1A=Bb5;}
      else {OCR1A=B5;}
      break;
    case 'C': OCR1A=C5;
      break;
    case 'D': if(flat){OCR1A=Db5;}
      else {OCR1A=D5;}
      break;
    case 'E': if(flat){OCR1A=Eb5;}
      else {OCR1A=E5;}
      break;
    case 'F': OCR1A=F5;
      break;
    case 'G': if(flat){OCR1A=Gb5;}
      else {OCR1A=G5;}
      break;
    }
    break;
  case 6: switch (note) {
    case 'A': if(flat){OCR1A=Ab6;}
      else {OCR1A=A6;}
      break;
    case 'B': if(flat){OCR1A=Bb6;}
      else {OCR1A=B6;}
      break;
    case 'C': OCR1A=C6;
      break;
    case 'D': if(flat){OCR1A=Db6;}
      else {OCR1A=D6;}
      break;
    case 'E': if(flat){OCR1A=Eb6;}
      else {OCR1A=E6;}
      break;
    case 'F': OCR1A=F6;
      break;
    case 'G': if(flat){OCR1A=Gb6;}
      else {OCR1A=G6;}
      break;
    }
    break;
  case 7: switch (note) {
    case 'A': if(flat){OCR1A=Ab7;}
      else {OCR1A=A7;}
      break;
    case 'B': if(flat){OCR1A=Bb7;}
      else {OCR1A=B7;}
      break;
    case 'C': OCR1A=C7;
      break;
    case 'D': if(flat){OCR1A=Db7;}
      else {OCR1A=D7;}
      break;
    case 'E': if(flat){OCR1A=Eb7;}
      else {OCR1A=E7;}
      break;
    case 'F': OCR1A=F7;
      break;
    case 'G': if(flat){OCR1A=Gb7;}
      else {OCR1A=G7;}
      break;
    }
    break;
  case 8: switch (note) {
    case 'A': if(flat){OCR1A=Ab8;}
      else {OCR1A=A8;}
      break;
```

```
      case 'B': if(flat){OCR1A=Bb8;}
        else {OCR1A=B8;}
        break;
      case 'C': OCR1A=C8;
        break;
      case 'D': if(flat){OCR1A=Db8;}
        else {OCR1A=D8;}
        break;
      case 'E': if(flat){OCR1A=Eb8;}
        else {OCR1A=E8;}
        break;
      case 'F': OCR1A=F8;
        break;
      case 'G': if(flat){OCR1A=Gb8;}
        else {OCR1A=G8;}
        break;
      }
      break;
    default: OCR1A=0x0000;
  }
}
void play_song(uint8_t song, uint8_t note) {
  //if you add a song, you'll have to add it to this
  //switch statement.
  switch (song) {
  case 0: song0(note); //beaver fight song
    break;
  default: song0(note); //defaults to beaver fight song
  }
}
void play_rest(uint8_t duration) {
  //mute for duration
  //duration is in 64th notes at 120bpm
  PORTD |= mute;
  beat=0;
  max_beat = duration;
}

void music_off(void) {
  //this turns the alarm timer off
  notes=0;
  TCCR1B &= ~((1<<CS11)|(1<<CS10));
  //and mutes the output
  PORTD |= mute;
}

void music_on(void) {
  //this starts the alarm timer running
  notes=0;
  TCCR1B |= (1<<CS11)|(1<<CS10);
  //unmutes the output
  PORTD &= unmute;
  //and starts the selected song
  play_song(song, notes);
}

void music_init(void) {
  //initially turned off (use music_on() to turn on)
  TIMSK |= (1<<OCIE1A);  //enable timer interrupt 1 on compare
  TCCR1A = 0x00;         //TCNT1, normal port operation
  TCCR1B |= (1<<WGM12);  //CTC, OCR1A = top, clk/64 (250kHz)
  TCCR1C = 0x00;         //no forced compare
  OCR1A = 0x0031;        //(use to vary alarm frequency)
  music_off();
  beat = 0;
  max_beat = 0;
  notes = 0;
  song = 0;              //beaver fight song
}
```



```
void song0(uint16_t note) { //beaver fight song (Max and Kellen)
  switch (note) {
    case 0: play_note('F', 0, 4, 8);
       break;
    case 1: play_note('E', 0, 4, 8);
       break;
    case 2: play_note('D', 0, 4, 8);
       break;
    case 3: play_note('C', 0, 4, 8);
       break;
    case 4: play_note('A', 0, 4, 6);
       break;
    case 5: play_note('A', 1, 4, 2);
       break;
    case 6: play_note('A', 0, 4, 6);
       break;
    case 7: play_note('A', 1, 4, 2);
       break;
    case 8: play_note('A', 0, 4, 16);
       break;
    case 9: play_note('F', 0, 4, 8);
       break;
    case 10: play_note('E', 0, 4, 8);
       break;
    case 11: play_note('D', 0, 4, 8);
       break;
    case 12: play_note('C', 0, 4, 8);
       break;
    case 13: play_note('B', 1, 4, 6);
       break;
    case 14: play_note('A', 0, 4, 2);
       break;
    case 15: play_note('B', 1, 4, 6);
       break;
    case 16: play_note('A', 0, 4, 2);
       break;
    case 17: play_note('B', 1, 4, 16);
       break;
    case 18: play_note('G', 0, 4, 3);
       break;
    case 19: play_rest(1); //rest
       break;
    case 20: play_note('G', 0, 4, 7);
       break;
    case 21: play_rest(1); //rest
       break;
    case 22: play_note('G', 1, 4, 4);
       break;
    case 23: play_note('G', 0, 4, 6);
       break;
    case 24: play_note('A', 0, 4, 2);
       break;
    case 25: play_note('B', 1, 4, 8);
       break;
    case 26: play_note('A', 0, 4, 2);
       break;
    case 27: play_rest(2);
       break;
    case 28: play_note('A', 0, 4, 8);
       break;
    case 29: play_note('A', 1, 4, 4);
       break;
    case 30: play_note('A', 0, 4, 6);
       break;
    case 31: play_note('B', 1, 4, 2);
       break;
    case 32: play_note('C', 0, 5, 4);
       break;
    case 33: play_note('D', 1, 5, 4);
```

```
      break;
    case 34: play_note('D', 0, 5, 4);
      break;
    case 35: play_note('B', 0, 4, 8);
      break;
    case 36: play_note('A', 0, 4, 4);
      break;
    case 37: play_note('G', 0, 4, 8);
      break;
    case 38: play_note('A', 0, 4, 8);
      break;
    case 39: play_note('G', 0, 4, 24);
      break;
    case 40: play_rest(8);
      break;
    case 41: play_note('F', 0, 4, 8);
      break;
    case 42: play_note('E', 0, 4, 8);
      break;
    case 43: play_note('D', 0, 4, 8);
      break;
    case 44: play_note('C', 0, 4, 8);
      break;
    case 45: play_note('A', 0, 4, 6);
      break;
    case 46: play_note('A', 1, 4, 2);
      break;
    case 47: play_note('A', 0, 4, 6);
      break;
    case 48: play_note('A', 1, 4, 2);
      break;
    case 49: play_note('A', 0, 4, 16);
      break;
    case 50: play_note('F', 0, 4, 8);
      break;
    case 51: play_note('G', 1, 4, 8);
      break;
    case 52: play_note('G', 0, 4, 8);
      break;
    case 53: play_note('D', 0, 4, 8);
      break;
    case 54: play_note('B', 1, 4, 6);
      break;
    case 55: play_note('A', 0, 4, 2);
      break;
    case 56: play_note('B', 1, 4, 6);
      break;
    case 57: play_note('A', 0, 4, 2);
      break;
    case 58: play_note('B', 1, 4, 16);
      break;//phrase
    case 59: play_note('D', 0, 4, 16);
      break;
    case 60: play_note('D', 0, 5, 16);
      break;
    case 61: play_note('A', 0, 4, 16);
      break;
    case 62: play_note('C', 0, 5, 16);
      break;
    case 63: play_note('B', 1, 4, 8);
      break;
    case 64: play_note('C', 0, 5, 4);
      break;
    case 65: play_note('D', 0, 5, 4);
      break;
    case 66: play_note('A', 0, 4, 8);
      break;
    case 67: play_note('G', 0, 4, 8);
      break;
```



```
    case 68: play_note('F', 0, 4, 24);
      break;
    case 69: play_rest(8);
      break;
    default: notes=-1;
  }
}//song0
/*********************************************************************/
//              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;
  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;
}

//*************************************************************************
*****
//                                  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)
{
  uint16_t ms = 0;
  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;
  }
  ms++;
  if(ms % 8 == 0) {
    //for note duration (64th notes)
    beat++;
  }
```



```
}


//*************************************************************************
*****
//                                  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
  if(beat >= max_beat) {   //if we've played the note long enough
    notes++;               //move on to the next note
    play_song(song, notes);//and play it
  }
}
  //PORTD ^= 0b10000000;

//*************************************************************************
*****
//                                  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)
{
  static char previous_LCD_message[42];
  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 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 Port pin2 D to output
DDRE  |= (1 << PE5) | (1 << PE6);   //Sets Port pin6 E 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  = 0x00;
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();
music_init();
init_twi();//.............. //initalize TWI (twi_master.h)
uart_init();
sei();

}

//*************************************************************************
*****
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
_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;

// Display CurrentTime

}else if(currentButtonsPressed == 0x08)
{
  //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;
    buttonPos = 1;
  }

  // Display 'ALARM' on LCD
  if(alarmActivated && alarmSET)
  {
    music_on();
    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
```