```c
// lab3.c
// Andrey Kornilovich
// 10.28.19

//  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>

// 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 0b00000011
// #define ENC_B 0b00001100

#define ENC_A 0b11111100
#define ENC_B 0b11110011

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

//decimal to 7-segment LED display encodings, logic "0" turns on segment
uint8_t dec_to_7seg[12];

//7 seg display counter
int digit = 0;

//num to display on 7 seg
int num_to_display = 0;

//scale to multiply by the encoder output
int count_scale = 1;

uint8_t button_state = 0x00;

// 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_DP); //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);
          // break;
        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
  segment_data[4] = dec_to_7seg[10]; //assign empty value for colon

  //blank out leading zero digits
  int segs;
  for(segs = 0; segs < 3; segs++){
    //if segment is 0, blank it out
    if(segment_data[segs] == dec_to_7seg[0]) {segment_data[segs] = dec_to_7seg[10];}
    else {break;}
  }
}//segment_sum

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
  digit++;
  if(digit > 3) {digit = 0;}
}

int 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:
        button_state ^= 0x01;
        break;
      case 1:
        button_state ^= 0x02;
      default:
        break;
      }
    }
  }

  //set counting scale based on button states
  if(button_state == 0x00) {count_scale = 1;}
  if(button_state == 0x01) {count_scale = 2;}
  if(button_state == 0x02) {count_scale = 4;}
  if(button_state == 0x03) {count_scale = 0;}

}
/*
// other approach with just 2 previous state checking, can get fooled
int process_encoders(){
  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 = 0xFF;          //store the previous SPI packet

  //check current and previous state, return a result if transition back to 0xFF found
  if(prev_spi_a != SPDR){
    //CCW check for encoder A
    if(((SPDR & ENC_A) == 0b00000011) && ((prev_spi_a & ENC_A) == 0b00000010)){
      prev_spi = SPDR;
      return -1;
    }
    //CW check for encoder A
    if(((SPDR & ENC_A) == 0b00000011) && ((prev_spi_a & ENC_A) == 0b00000001)){
      prev_spi = SPDR;
      return 1;
    }
    //CCW check for encoder B
    if(((SPDR & ENC_B) == 0b00001100) && ((prev_spi_a & ENC_B) == 0b00001000)){
      prev_spi = SPDR;
      return -1;
    }
    //CCW check for encoder B
    if(((SPDR & ENC_B) == 0b00001100) && ((prev_spi_a & ENC_B) == 0b00000100)){
      prev_spi = SPDR;
      return 1;
    }
    //set the previous
    prev_spi = SPDR;
  }
  return 0;
}
*/
int process_encoders(){
  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;

  // 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;}
  }
  // 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;
}

void update_bar(void){
  SPDR = button_state;                        //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 = 0xF0;   //set port bits 4-7 B as outputs
  DDRE = 0x40;  // set port E bit 6 as output
}

void tcnt0_init(void){
  TIMSK |= (1<<TOIE0);             //enable interrupts
  TCCR0 |= (1<<CS01) | (1<<CS00);  //normal mode, prescale by 32
  // TCCR0 |= (1<<CS02) | (1<<CS00);  //normal mode, prescale by 128
}

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
}

//this ISR handles main lab functionality and timing
ISR(TIMER0_OVF_vect){
  //process button presses, change modes, and update counts
  process_buttons();
  num_to_display += process_encoders() * count_scale;

  //bound the count to 0 - 1023
  if(num_to_display > 1023) {num_to_display = 0;}
  else if(num_to_display < 0) {num_to_display = 1023;}

  //break up the disp_value to 4, BCD digits in the array: call (segsum)
  segsum(num_to_display);

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

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

  //setup Port I/O, seven seg data, and spi, interrupt, and counter enable
  setup_ports();
  encode_chars();
  tcnt0_init();
  spi_init();
  sei();

  while(1){} // empty while loop

  return 0;
}//main
```