```
// lab3.c
// Victor Garcia Flores
// 11.01.2019

//  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 bit 1 goes RCLK and CLK on the 74HC595 and 74HC165 respectively.
//  PORTC bits 0-2 go to a,b,c inputs of the 74HC138.
//  PORTC.3 goes to the PWM transistor base.
//      PORTE.6 goes to SH/LD for the 74HC165
//      PORTE.7 goes to CLK_INH for the 74HC165

#define F_CPU 16000000 // cpu speed in hertz
#define TRUE 1
#define FALSE 0
#include <avr/io.h>
#include <math.h>
#include <util/delay.h>
#include <avr/interrupt.h>

//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
//Note: They are arranged so that the value of a possible integer matched with t
he position
uint8_t dec_to_7seg[12] = {0b11000000, 0b11111001,0b10100100,0b10110000,0b100110
01,0b10010010,0b10000010,0b11111000,0b10000000,0b10011000,0b01111111,0b11111111}
;

//startup flag
uint8_t start_flag=0;

//variable for current value
uint16_t CurrCountVal = 0;

//Global Button Variables
uint8_t ButtonState = 1; //increment and decrement value
uint8_t buttons[2] = {0}; // used to see which button was pressed

//Encoder Globals
uint8_t raw_encoder = 0; //raw data from break out board
uint8_t stateJumps = 0; //Count state jumps
//Encoder #1
uint8_t prevL_Encoder=0;
uint8_t currL_Encoder=0;
//Encoder #2
uint8_t prevR_Encoder=0;
uint8_t currR_Encoder=0;

//volatile raw segment data
uint16_t volatile encoder_test;

/*********************************************************************/
//                          spi_init
//Initalizes the SPI port on the mega128. Does not do any further
//external device specific initalizations.  Sets up SPI to be:
//master mode, clock=clk/2, cycle half phase, low polarity, MSB first
//interrupts disabled, poll SPIF bit in SPSR to check xmit completion
/*********************************************************************/
void spi_init(void){
  DDRD = (1<<PD2); //regclk
  DDRB   = ((1<<PB0)|(1<<PB1)|(1<<PB2)| (0<<PB3)); //output mode for MOSI, SCLK
  SPCR   = (1<<SPE) | (1<<MSTR); //master mode, clk low on idle, leading edge sa
mple (p. 167)
  SPSR   = (1<<SPI2X); //choose double speed operation // double speed operation
 }//spi_init
```



```
 /*********************************************************************/
 //                          tcnt0_init
 //Initalizes timer/counter0 (TCNT0). TCNT0 is running in async mode
 //with external 32khz crystal.  Runs in normal mode with no prescaling.
 //Interrupt occurs at overflow 0xFF.
//*************************************************************************
 void tcnt0_init(void){
   //ASSR   |= (1<<AS0); //ext osc TOSC
   TIMSK  |=  (1<<TOIE0); //enable TCNT0 overflow interrupt
   TCCR0  |=  (1<<CS00); //normal mode, no prescale
 }

//*************************************************************************
//                           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}; //We do what we did in lab 1, but this t
ime as an array so we can address the other buttons
        state[button] = ((state[button]<<1) | (!bit_is_clear(PINA,button)) | 0xE
000);
        if(state[button] == 0xFF00) return 1;
        return 0;
}

/*********************************************************************/
//                          bargraph_updater
//Used to update bargraph values with inc/decrement value
//The scalar inc/dec value will be displayed in binary
/*********************************************************************/
void bargraph_updater(uint8_t state){
  uint8_t output = 0; //what the bargraph will display
  if(state == 0){ //when both buttons are pressed do nothing
    output = 0b00000000;
  }
  else if(state == 1){ //increment/decrement by 1
    output = 0b00000001;
  }
  else if(state == 2){//increment/decrement by 2
    output = 0b00000010;
  }
  else if(state == 4){//increment/decrement by 4
    output = 0b00000100;
  }

  /* Start transmission */
  SPDR = output;
  while (bit_is_clear(SPSR,SPIF)){} //spin till SPI data has been sent

  PORTD |= (1<<PD2);        //send rising edge to regclk on HC595
  PORTD &= ~(1<<PD2);            //send falling edge to regclk on HC595
}
/*********************************************************************/
//                          section_tester
//This is is used to test to see if we get to certain places in the code
//Whatever value is passed into this function will be presented onto the graph
/*********************************************************************/
void section_tester(uint8_t state){
  /* Start transmission */
  SPDR = state;
  while (bit_is_clear(SPSR,SPIF)){} //spin till SPI data has been sent

  PORTD |= (1<<PD2);        //send rising edge to regclk on HC595
```

```
  PORTD &= ~(1<<PD2);            //send falling edge to regclk on HC595
}
//*************************************************************************

//*************************************************************************
*****
//                                segment_sum
//takes a 16-bit binary input value and places the appropriate equivalent 4 digi
t
//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) {
        //Variables for values digit  positions
        uint8_t OnesVal;
        uint8_t TensVal;
        uint8_t HundredsVal;
        uint8_t ThousandsVal;

        //Decoder "Sel#" Positions for Digits. Note: The lower bytes are 0 becau
se we aren't using them in PORTB. This also keeps in mind the value we desire in
 PWN
        //determine how many digits there are
        int NumDigits = 0;
        int tempSum = sum;
        while(sum){
                tempSum /= 10;
                NumDigits++;
        }

        //break up decimal sum into 4 digit-segments
        //---ONES---
        OnesVal = sum % 10;
        segment_data[0] = OnesVal;

        //--- Tens ---
        TensVal = (sum/10) % 10;
        segment_data[1] = TensVal;

        //--- HUNDREDS ---
        HundredsVal = (sum/100) % 10;
        segment_data[3] = HundredsVal;

        //--- THOUSANDS ---
        ThousandsVal = (sum/1000) % 10;
        segment_data[4] = ThousandsVal;

        //DDRA = 0xFF; //Make PORT A an OUTPUT
        if(sum<10){ //if there is only one digit
                //1st Set
                PORTC = 0x00;
                PORTA = dec_to_7seg[OnesVal];
                _delay_ms(2);
                PORTC = 0x04;
                PORTA = 0b11111111;
    _delay_ms(2);
        }
        else if((sum >= 10) && (sum < 100)){ //if there are two digits
                //1st Set
                PORTC = 0x00;
                PORTA = dec_to_7seg[OnesVal];
                _delay_ms(2);

                //2nd Set
                PORTC = 0x01;
                PORTA = dec_to_7seg[TensVal];
    _delay_ms(2);

        }
```



```
        else if((sum>=100)&&(sum<1000)){ //if there are three digits
                //1st Set
                PORTC = 0x00;
                PORTA = dec_to_7seg[OnesVal];
                _delay_ms(2);

                //2nd Set
                PORTC = 0x01;
                PORTA = dec_to_7seg[TensVal];
                _delay_ms(2);

                //3rd Set
                PORTC = 0x03;
                PORTA = dec_to_7seg[HundredsVal];
    _delay_ms(2);
        }
        else if(sum>= 1000){ //if there are four digits
                //1st Set
                PORTC = 0x00;
                PORTA = dec_to_7seg[OnesVal];
                _delay_ms(2);

                //2nd Set
                PORTC = 0x01;
                PORTA = dec_to_7seg[TensVal];
                _delay_ms(2);

                //3rd Set
                PORTC = 0x03;
                PORTA = dec_to_7seg[HundredsVal];
                _delay_ms(2);

                //4th Set. Note: No segments need clearing.
                PORTC = 0x04;
                PORTA = dec_to_7seg[ThousandsVal];
    _delay_ms(2);
        }

}//segment_sum
//*************************************************************************
****


//*************************************************************************
*****
// Function Name:void AllSegments_BitClearer
// This function is put to clear previous digit values on the seven segment disp
lay.
// Goal: The goal is to avoid ghosting and help set un-used segments to zero.
//*************************************************************************
void AllSegments_BitClearer(){
        DDRA = 0xFF;
  asm volatile("nop");
  asm volatile("nop");
        //Ones
        PORTC = 0x00;
        PORTA = 0b11111111;
        _delay_ms(2);

        //Tens
        PORTC = 0x01;
        PORTA = 0b11111111;
        _delay_ms(2);

        //Hundreds
        PORTC = 0x04;
        PORTA = 0b11111111;
        _delay_ms(2);
```

```
        //Thousands
        PORTC = 0x04;
        PORTA = 0b11111111;
        _delay_ms(2);
}

//***********************************************************************************
*****
//
                        Read_Buttons
//***********************************************************************************
void Read_Buttons(){
  //Let's read button data
  DDRA = 0x00; //sets as input
  PORTA = 0xFF; //pulls up the resistors
  PORTC |= ((1<<PC0)|(1<<PC1)|(1<<PC2)); //Select bits for the buttons
        for(int BttnNum = 0;BttnNum <= 2;BttnNum++){
                uint8_t count = 0; //counter for how many buttons pressed
                //If a certain button at position x is pressed
    if(chk_buttons(BttnNum)){
      // Fing out which button was pressed and increment accordingly
      if(BttnNum == 0){
        //CurrCountVal += 1;
                                ButtonState = 2;
                                count++;
      }
      else if(BttnNum == 1){
        //CurrCountVal += 2;
                                ButtonState = 4;
                                count++;
      }
                        if(count == 2){
                                ButtonState = 0;
                        }
    }
  }
}

void handle_BttnData(){
  //uint8_t temp[2] = {0};
  if(buttons[0] == 1 && buttons[1] == 1){ //if both buttons
    ButtonState = 0; //value we inc/dec by
  }
  else if(buttons[0] == 1 && buttons[1] == 0){//if first button
    ButtonState = 2; //value we inc/dec by
  }
  else if(buttons[0] == 0 && buttons[1] == 1){//if second button
    ButtonState = 4; //value we inc/dec by
  }
}

void Read_ButtonsV2(){
        int BttnNum = 0;
        //Let's read button data
  DDRA = 0x00; //sets as input
  PORTA = 0xFF; //pulls up the resistors
  PORTC = ((1<<PC0)|(1<<PC1)|(1<<PC2)); //Select bits for the buttons

  for(BttnNum = 0; BttnNum <= 7; BttnNum++){
    if(chk_buttons(BttnNum)){ //If we read button input
      if(BttnNum == 0){ //first button is pressed
        //ButtonState = 2; //value we inc/dec by
        buttons[0] = 1; //button array
      }
      else if(BttnNum == 1){ //second button is pressed
        //ButtonState = 4; //value we inc/dec by
        buttons[1] = 1; //button array
      }
                        else if(BttnNum = 7){
```



```
                                CurrCountVal -= ButtonState;
                        }
      handle_BttnData();
    }
  }
  //reset button state
  buttons[0] = 0;
  buttons[1] = 0;
}

void Display_Seg(uint16_t value){
  //Makre PORTA an output
  DDRA = 0xFF;
  asm volatile("nop");
  asm volatile("nop");
  //disable tristate buffer for pushbutton switches
  PORTC = 0x00;

  //Parse Values and display them
  segsum(value);
}

/*********************************************************************/
//
                                spi_read
//Reads the SPI port.
/*********************************************************************/
uint8_t spi_read(void){
        SPDR = 0x00; //"dummy" write to SPDR
        while (bit_is_clear(SPSR,SPIF)){} //wait till 8 clock cycles are done
        return(SPDR); //return incoming data from SPDR
}

/*********************************************************************/
//
                                Encoder_Data
//Toggles SHIFT_LN_N on parallel shift register to get data into the flip flops
//Sets CLK_INH to low so we can read from QH.
//Remember: Most significant bit is at position H
/*********************************************************************/
void Encoder_Data(){
        int i;
        //Remember: PE6-> SHIFT_LN_N and PE7-> CLK_INH
        //Toggle SH_LD to get their values into the flip flops
        PORTE ^= (1<<PE6);
        PORTE ^= (1<<PE6);

        //Output to through QH by changing CLK_INH
        PORTE ^= (1<<PE7);//CLK_INH
        raw_encoder = spi_read();

        //Stop the output
        PORTE ^= (1<<PE7);//CLK_INH

        //TESTER to see raw value on bargraph (will show both inputs on graph)
        /*if(raw_encoder == 0){
                section_tester(raw_encoder);
        }
        else{
                section_tester(raw_encoder);
        }*/

        //left Encoder
        currL_Encoder = raw_encoder;
        // get rid of LHS bits
        // what we want: 0bxx
  for (i=7; i>1; i--) {
    currL_Encoder &= ~(1<<i);
  }
```

```
	//TESTER to see if leading values cleared
	/*if(raw_encoder == 0){
		section_tester(currL_Encoder);
	}
	else{
		section_tester(currL_Encoder);
	}*/

	//Right encoder
	currR_Encoder = (raw_encoder>>2);
	//get rid of LHS bits
	// what we want (0bxx)
	for (i=7; i>1;i--) {
		currR_Encoder &= ~(1<<i);
	}
	//TESTER to see if value was shifted and leading values cleared
	/*if(raw_encoder == 0){
		section_tester(currL_Encoder);
	}
	else{
		section_tester(currL_Encoder);
	}*/

	//If it's a first time start-up
	if(start_flag == 0){
		prevL_Encoder = currL_Encoder;//set them equal
		prevR_Encoder = currR_Encoder;//set them equal
		start_flag = 1;
	}

	// --------- LEFT ENCODER ----------//
	if(currL_Encoder){
		if(currL_Encoder == 0b11){
			if(prevL_Encoder == 0b01){
				CurrCountVal += ButtonState;
				prevL_Encoder = currL_Encoder;
				_delay_ms(1);
			}
			else if(prevL_Encoder == 11){
				if(prevL_Encoder == 10){
					CurrCountVal -= ButtonState;
					prevL_Encoder = currL_Encoder;
					_delay_ms(1);
				}
			}
		}
		else{
			prevL_Encoder = currL_Encoder;
			_delay_ms(1);
		}
	}

	// --------- RIGHT ENCODER ----------//
	if(currR_Encoder){
		if(currR_Encoder == 0b11){
			if(prevR_Encoder == 0b01){
				CurrCountVal += ButtonState;
				prevR_Encoder = currR_Encoder;
				_delay_ms(1);
			}
		}
		else if(currR_Encoder == 11){
			if(prevR_Encoder == 10){
				CurrCountVal -= ButtonState;
				prevR_Encoder = currR_Encoder;
				_delay_ms(1);
			}
		}
		else{
```



```
				prevR_Encoder = currR_Encoder;
				_delay_ms(1);
			}
		}
	}

	/*********************************************************************/
	//                          timer/counter0 ISR
	//Updates bargraph and 7-segment display while it reads encoder and button data.
	//The order was set to prioritize states and values.
	//*********************************************************************/
	ISR(TIMER0_OVF_vect){
		//Show the current inc/dec status
		bargraph_updater(ButtonState);

		//Read Button input
		Read_ButtonsV2();

		//Interpret encoder data (also make sure we don't go past bounded values
	)
		Encoder_Data();

		if(CurrCountVal < 0){
			CurrCountVal += 1023;
		}
		else if(CurrCountVal > 1023){
			CurrCountVal -= 1023;
		}

		//Display on 7 segment
		Display_Seg(CurrCountVal);
	}

	//*********************************************************************/
	//
	                        Main
	//*********************************************************************/
	int main()
	{
		tcnt0_init();  //initalize counter timer zero
		spi_init();    //initalize SPI port
		DDRC = 0x0F; //set port bits 0-3 C as outputs
		asm volatile("nop");
		asm volatile("nop");
		DDRE = ((1<<PE7) | (1<<PE6)); //Outputs for CLK_INH and SHIFT_LN_N
		PORTE = ((1<<PE7) | (1<<PE6)); //By default, disable CLK_INH (don't want
	 an output to QH yet) and SH/LD (active low)
		sei();         //enable interrupts before entering loop

		while(1){
			//Clear 4 segments to prevent ghosting
			AllSegments_BitClearer();
	   _delay_ms(2);

		}//while
	}
```