```
//  File:     lab3.c
//  Author:   Bradley Anderson
//  Created:  Oct-23, 2017
//
//  Collaboration:
//  Worked with Kyle O'Brien and Makenzie Brian

//  -- Power Pins --
//  BLACK  := VCC
//  WHTITE := GND


//  -- Button Board --
//  J1  :=  Brown
//  J2  :=  Red
//  J3  :=  Orange
//  J4  :=  Yellow
//  J5  :=  Green
//  J6  :=  Blue
//  J7  :=  Purple
//  J9  :=  Gray
//
//  GND      :=  White
//  VCC      :=  Black
//  SW_COM   :=  *nc*
//  COM_LVL  :=  Brown    (GND)
//  COM_EN   :=  Orange   (7seg DEC7)


//  -- 4 Digit Display --
//  SEL0  :=  Green      (PORTB 4)
//  SEL1  :=  Blue       (PORTB 5)
//  SEL2  :=  Purple     (PORTB 6)
//  EN    :=  White      (VCC)
//  EN_N  :=  Black      (GND)
//  PWM   :=  Gray       (PORTF 3)
//  DEC5  :=  *nc*
//  DEC6  :=  *nc*
//  DEC7  :=  Orange     (buttonBoard COM_EN)
//  NC    :=  NC
//
//  A    :=  Brown    (buttonBoard J1)
//  B    :=  Red      (buttonBoard J2)
//  C    :=  Orange   (buttonBoard J3)
//  D    :=  Yellow   (buttonBoard J4)
//  E    :=  Green    (buttonBoard J5)
//  F    :=  Blue     (buttonBoard J6)
//  G    :=  Purple   (buttonBoard J7)
//  DP   :=  Gray     (buttonBoard J8)
//
//  A_2    :=  Rib0-Red   (PORTA 0)
//  B_2    :=  Rib1       (PORTA 1)
//  C_2    :=  Rib2       (PORTA 2)
//  D_2    :=  Rib3       (PORTA 3)
//  E_2    :=  Rib4       (PORTA 4)
//  F_2    :=  Rib5       (PORTA 5)
//  G_2    :=  Rib6       (PORTA 6)
//  DP_2   :=  Rib7       (PORTA 7)
//  GND    :=  Rib8       (PORTA 8)
//  VCC    :=  Rib9       (PORTA 9)
```



```
//  -- Encoders --
//  NC     :=  *nc*
//  NC     :=  Brown    *nc*
//  NC     :=  Red      *nc*
//  NC     :=  Orange   *nc*
//  SOUT   :=  Yellow   (PORTB 3)
//  SIN    :=  Green    *nc*
//  CKINH  :=  Blue     (GND, Clock Inhibiter)
//  SCK    :=  Purple   (PORTB 1, Clock)
//  SH/LD  :=  Gray     (PORTE 6, Shift/Load)
//  GND    :=  White
//  VCC    :=  Black


//  -- Bar Graph --
//  SD_OUT  :=  Brown    *nc*
//  SRCLK   :=  Red      (PB1 SCLK)
//  REGCLK  :=  Orange   (PB0)
//  OE_N    :=  Yellow   (PB7)
//  SDIN    :=  Green    (PB2 MOSI)
//  VDD     :=  Black
//  GND     :=  White


//  -- PORTA -> 4 Digit Display --
//  PA0 :=  Rib0 (Red)
//  PA1 :=  Rib1
//  PA2 :=  Rib2
//  PA3 :=  Rib3
//  PA4 :=  Rib4
//  PA5 :=  Rib5
//  PA6 :=  Rib6
//  PA7 :=  Rib7
//  PA8 :=  Rib8
//  PA9 :=  Rib9


//  -- PORTB --
//  PB0 :=  Orange   (bargraph REGCLK)
//  PB1 :=  Red      (bargraph&encoder SCLK)
//  PB2 :=  Green    (bargraph MOSI)
//  PB3 :=  YELLOW   (encoder SIN)
//  PB4 :=  Green    (7seg SEL0)
//  PB5 :=  Blue     (7seg SEL1)
//  PB6 :=  Purple   (7seg SEL2)
//  PB7 :=  Yellow   (bargraph OE_N)
//  PB8 :=  White    (GND)
//  PB9 :=  Black    (VCC)
#define RCLK PB0
#define SCLK PB1
#define MOSI PB2
#define MISO PB3
#define SEL0 PB4
#define SEL1 PB5
#define SEL2 PB6
#define OE_N PB7


//  -- PORTE --
//  PE0 :=  *nc*
//  PE1 :=  *nc*
//  PE2 :=  *nc*
```

```
//  PE3 :=  *nc*
//  PE4 :=  *nc*
//  PE5 :=  *nc*
//  PE6 :=  Gray  (Shift/Load encoder)
//  PE7 :=  *nc*
//  PE8 :=  White (GND)
//  PE9 :=  Black (VCC)
#define SHLD PE6

//  -- PORTF --
//  PF0 :=  *nc*
//  PF1 :=  *nc*
//  PF2 :=  *nc*
//  PF3 :=  Gray (7seg PWM)
//  PF4 :=  *nc*
//  PF5 :=  *nc*
//  PF6 :=  *nc*
//  PF7 :=  *nc*
//  PF8 :=  *nc*
//  PF9 :=  *nc*
#define PWM PF3


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

// bits used for digit selection

// DEMUX to LED wiring
#define SELD1 (0x0 << SEL0)
#define SELD2 (0x1 << SEL0)
#define SELD3 (0x3 << SEL0)
#define SELD4 (0x4 << SEL0)
#define SELDD (0x2 << SEL0)
#define SELBN (0x7 << SEL0)
#define SELCL !SELBN

// Blank 7segment
#define BLNK 0xFF

#define TRUE 1
#define FALSE 0
typedef unsigned char bool;
bool a = TRUE;
bool b = FALSE;

// 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];

// Select digit array
uint8_t digitSelect[8];
```



```

// Holds value of buttons from last check
volatile uint8_t buttonState;

// Holds state of encoders
volatile uint8_t encoderState;

// Number displayed to 7seg
volatile uint16_t segNum = 0;

// Number displayed to bargraph
volatile uint8_t barNum = 0;

// Function prototypes
uint8_t chk_button(uint8_t);
void toggle_button_bus();
void spiTxRx();
void interpret_encoders();
void outputToBargraph(uint8_t);
uint8_t checkDirection(uint8_t, uint8_t);
void spi_init();
void timer_init();
void digit_init();
void increment();
void decrement();

//*****************************************************************************
//    -- 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_button(uint8_t button) {
  static uint16_t State[8] = {0};     // Static array is initialied once at comp
ile time
  State[button] = (State[button]<<1) | !bit_is_clear(PINA, button) | 0xE000;
  if (State[button] == 0xFF00) return TRUE;
  return FALSE;
} //chk_button



//*****************************************************************************
*****
//    -- segment_sum --
//  takes a 16-bit binary input value and places the appropriate equivalent 4 di
git
//  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)
{
```

Printed by


```c
  //determine how many digits there are
  //break up decimal sum into 4 digit-segments
  //blank out leading zero digits
  //now move data to right place for misplaced colon position

  uint8_t i=0;     // for counter
  uint8_t ldZero = TRUE;

  segment_data[0] = sum % 10;
  segment_data[1] = sum/10 % 10;
  segment_data[2] = 10;          // keep colon off; dig10 is mapped to BLNK

  segment_data[3] = sum/100 % 10;
  segment_data[4] = sum/1000 % 10;

  // Covert dec to BCD, ignoring colon and blanking leading zeros
  //ldZero=TRUE -> index has not yet found a non-zero digit
  for (i=4; i > 0; --i)
  {
    if (ldZero && (segment_data[i]==0))
      segment_data[i] = BLNK;
    else
    {
      if (i!=2) ldZero = FALSE;
      segment_data[i] = dec_to_7seg[segment_data[i]];
    }//if
  }//for

  segment_data[0] = dec_to_7seg[segment_data[i]];

  return;
}//segment_sum



//****************************************************************************
*****
//    -- Checks State of Buttons on 7seg Bus --
//****************************************************************************
*****
void toggle_button_bus() {

  //make PORTA an input port with pullups
  DDRA = 0x00;     // 0 is input, 1 is output
  PORTA = 0xFF;    // 0 is float, 1 is pull-up

  //enable tristate buffer for pushbutton switches
  PORTB &= SELCL;
  PORTB |= SELBN;

  //buttonState=0;
  int i;
  //now check each button and increment the count as needed
  for (i=0; i<8; i++)
  {
    if (chk_button(i))
      buttonState ^= 1<<i;
  }//for


  //disable tristate buffer for pushbutton switches
  PORTB &= SELCL;
```



```c
  // Reset A as output
  DDRA = 0xFF;
}



//****************************************************************************
*****
//    -- Transmits and Receives to/from SPI --
//****************************************************************************
*****
void spiTxRx() {

  // Toggle Encoder Shift/Load
  PORTE &= ~(1<<SHLD);
  PORTE |= (1<<SHLD);

  // SPI write from global variable
  SPDR = barNum;

  // Wait for 8 clock cycles
  while(bit_is_clear(SPSR, SPIF)) {}

  // Save the most recent serial reading into global variable
  encoderState = SPDR;

  // Toggle Bargraph Register Clock
  PORTB |= (1<<RCLK);
  PORTB &= ~(1<<RCLK);
}


#define RWFN 4
#define LWFN 5
//#define LWFN 6
//#define RWFN 7
#define RMSK 0b0011
#define LMSK 0b1100
//****************************************************************************
*****
//    -- Performs Logic to Test Direction of Encoder Movement --
//****************************************************************************
*****
void interpret_encoders(){
  uint8_t curr=0;
  uint8_t prev=0;
  volatile static uint8_t encR_cwse = 0;
  volatile static uint8_t encR_ccws = 0;
  volatile static uint8_t encL_cwse = 0;
  volatile static uint8_t encL_ccws = 0;
  volatile static uint8_t encStatusReg=0;

  // encStatusReg variable decoding
  // bit7     bit6     bit5     bit4     bit3     bit2     bit1     bit0
  //                   LWFN     RWFN     LPrv     LPrv     RPrv     RPrv
  // WFN = Wait for Next

  // Encoder states
  curr = (encoderState & 0x0F);
  prev = (encStatusReg & 0x0F);
```

```
  // Right Encoder Changed State
  if ((curr & RMSK) != (prev & RMSK)) {

    // Shift registers to keep track of turning speed
    switch(checkDirection((curr & RMSK),(prev & RMSK))) {
      case 0b01:
        encR_cwse = (encR_cwse<<1)|1;
        encR_ccws =  encR_ccws>>1;
        break;
      case 0b10:
        encR_cwse =  encR_cwse>>1;
        encR_ccws = (encR_ccws<<1)|1;
        break;
      default:
        encR_cwse = encR_cwse>>1;
        encR_ccws = encR_ccws>>1;
        break;
    }

    // When at notch, reset turning speed
    if ((curr & RMSK) == RMSK) {
      encR_cwse = 0;
      encR_ccws = 0;
    }

    // Check right encoder
    if (encStatusReg & (1<<RWFN)) {

      if (encR_cwse >= 0b11) {

        // Extra increments to compensate for missed bits
        if (encR_cwse >= 0b11111) {
          if (encR_cwse >= 0b111111) {
            increment();
          }
          increment();
        }

        increment();
        encStatusReg &= ~(1<<RWFN);
        encR_cwse = 0;
        encR_ccws = 0;

      } else if (encR_ccws >= 0b11) {

        // Extra decrements to compensate for missed bits
        if (encR_ccws >= 0b11111) {
          if (encR_ccws >= 0b111111) {
            decrement();
          }
          decrement();
        }

        decrement();
        encStatusReg &= ~(1<<RWFN);
        encR_cwse = 0;
        encR_ccws = 0;
      }
    }

    // When at halfway point, enable state change
    //  This prevents a floating state next to notch triggering an event
```



```
    if ((curr & RMSK) == 0x00) {
      encStatusReg |= (1<<RWFN);
    }

    encStatusReg &= ~RMSK;
    encStatusReg |= (encoderState & RMSK);
  }


  // Left Encoder Changed State
  if ((curr & LMSK) != (prev & LMSK)) {

    // Shift registers to keep track of turning speed
    switch(checkDirection(((curr & LMSK)>>2),((prev & LMSK)>>2))) {
      case 0b01:
        encL_cwse = (encL_cwse<<1)|1;
        encL_ccws =  encL_ccws>>1;
        break;
      case 0b10:
        encL_cwse =  encL_cwse>>1;
        encL_ccws = (encL_ccws<<1)|1;
        break;
      default:
        encL_cwse = encL_cwse>>1;
        encL_ccws = encL_ccws>>1;
        break;
    }

    // When at notch, reset turning speed
    if ((curr & LMSK) == LMSK) {
      encL_cwse = 0;
      encL_ccws = 0;
    }

    // Check right encoder
    if (encStatusReg & (1<<LWFN)) {

      if (encL_cwse >= 0b11) {

        // Extra increments to compensate for missed bits
        if (encL_cwse >= 0b11111) {
          if (encL_cwse >= 0b111111) {
            increment();
          }
          increment();
        }

        increment();
        encStatusReg &= ~(1<<LWFN);
        encL_cwse = 0;
        encL_ccws = 0;

      } else if (encL_ccws >= 0b11) {

        // Extra decrements to compensate for missed bits
        if (encL_ccws >= 0b11111) {
          if (encL_ccws >= 0b111111) {
            decrement();
          }
          decrement();
        }
```

```
        decrement();
        encStatusReg &= ~(1<<LWFN);
        encL_cwse = 0;
        encL_ccws = 0;
      }
    }

    // When at halfway point, enable state change
    //  This prevents a floating state next to notch triggering an event
    if ((curr & LMSK) == 0x00) {
      encStatusReg |= (1<<LWFN);
    }

    encStatusReg &= ~LMSK;
    encStatusReg |= (encoderState & LMSK);
  }
}


#define CWSE 0b01
#define CCWS 0b10
//*****************************************************************************
*****
//    -- Encoder Checker
//    Return Value
//    bit1    bit0
//    0       1   Clockwise
//    1       0   Counter Clockwise
//*****************************************************************************
*****
uint8_t checkDirection(uint8_t curr, uint8_t prev) {

  curr &= 0b11;
  prev &= 0b11;
  switch (curr) {
    case 0b01:
      switch (prev){
        case 0b11:
          return CWSE;
        case 0b00:
          return CCWS;
      }
      break;
    case 0b00:
      switch (prev){
        case 0b01:
          return CWSE;
        case 0b10:
          return CCWS;
      }
      break;
    case 0b10:
      switch (prev){
        case 0b00:
          return CWSE;
        case 0b11:
          return CCWS;
      }
      break;
    case 0b11:
      switch (prev){
        case 0b10:
```



```
          return CWSE;
        case 0b01:
          return CCWS;
      }
      break;
  }//switch

  return 0;
}

//*****************************************************************************
*****
//    -- Timer 0 Compare Interrupt --
//*****************************************************************************
*****
ISR(TIMER0_COMP_vect){

  spiTxRx();

  interpret_encoders();

  toggle_button_bus();

}//ISR TIM0_COMP_vect
//*****************************************************************************
*****



//*****************************************************************************
*****
//    -- Serial Peripheral Interface Initialization --
//
//*****************************************************************************
*****
void spi_init() {

  // Direction Registers
  DDRB |= (1<<RCLK) | (1<<SCLK) | (1<<MOSI) | (1<<OE_N);
  DDRE |= (1<<SHLD);

  // SPI Control Register
  SPCR |= (1<<SPE) | (1<<MSTR) | (0<<SPR1)|(1<<SPR0);

  // SPI Status Register
  SPSR |= (1<<SPI2X);

  // SPI Data Register
  PORTB &= ~(1<<OE_N);
}



//*****************************************************************************
*****
//    -- TIMER Initialization --
//*****************************************************************************
*****
void timer_init(){

  // Timer counter 0 setup, running off i/o clock
```

```
  // Asynchronous Status Register, pg107
  //    Run off of external clock
  ASSR  |= (1<<AS0);

  // Timer/Counter Interrupt Mask, pg109
  //    enable compare interrupt
  TIMSK |= (1<<OCIE0);

  // Timer/Counter Control Register, pg104
  //    CTC mode, no prescale
  TCCR0 = ((1<<WGM01)|(0<<WGM00) | (0<<COM01)|(0<<COM00) | (0<<CS02)|(0<<CS01)|(
1<<CS00));

  // Output Compare Register
  //    Set button&encoder check time with this
  OCR0 = 0x1F;
}



//*****************************************************************************
*****
//  -- Digit Initialization
//*****************************************************************************
*****
void digit_init(){

  // select pins for DEMUX in array form
  digitSelect[0] = SELD1;
  digitSelect[1] = SELD2;
  digitSelect[2] = SELBN;
  digitSelect[3] = SELD3;
  digitSelect[4] = SELD4;

  // BCD mapping
  dec_to_7seg[0] = (uint8_t) 0b11000000;
  dec_to_7seg[1] = (uint8_t) 0b11111001;
  dec_to_7seg[2] = (uint8_t) 0b10100100;
  dec_to_7seg[3] = (uint8_t) 0b10110000;
  dec_to_7seg[4] = (uint8_t) 0b10011001;
  dec_to_7seg[5] = (uint8_t) 0b10010010;
  dec_to_7seg[6] = (uint8_t) 0b10000010;
  dec_to_7seg[7] = (uint8_t) 0b11111000;
  dec_to_7seg[8] = (uint8_t) 0b10000000;
  dec_to_7seg[9] = (uint8_t) 0b10010000;
  dec_to_7seg[10] = (uint8_t) 0xFF;

  // 0 is input, 1 is output
  DDRB = (1<<SEL0)|(1<<SEL1)|(1<<SEL2);
  //
  DDRF = (1<<PWM);
  PORTF &= ~(1<<PWM);
}


#define bState0  0b110
#define bState1  0b000
#define bState2  0b010
#define bState4  0b100
#define bState5  0b101
#define bState10 0b1010
//*****************************************************************************
```



```
*****
//    -- Conditionally Increment Based on State
//*****************************************************************************
*****
void increment() {

  switch (buttonState) {
    default:
      segNum++;
    case bState0:
      break;
    case bState1:
      segNum++;
      break;
    case bState2:
      segNum += 2;
      break;
    case bState4:
      segNum += 4;
      break;
    case bState5:
      segNum += 5;
      break;
    case bState10:
      segNum += 10;
      break;
  }
  if (segNum >= 1024) {
    segNum = 0;
  }
}



//*****************************************************************************
*****
//    -- Conditionally Decrement Based on State
//*****************************************************************************
*****
void decrement() {
  switch (buttonState) {
    default:
      segNum--;
    case bState0:
      break;
    case bState1:
      segNum--;
      break;
    case bState2:
      segNum -= 2;
      break;
    case bState4:
      segNum -= 4;
      break;
    case bState5:
      segNum -= 5;
      break;
    case bState10:
      segNum -= 10;
      break;
  }
  if (segNum >= 1024) {
```

```c
    segNum = 1023;
  }
}



//************************************************************************************
*****
//                                  main
//  Does main stuff
//************************************************************************************
*****
int main(void) {

  /*
   * #define RCLK Register Clock
   * #define SCLK Serial Clock
   * #define MOSI Serial to bargraph
   * #define MISO Serial from encoders
   * #define SEL0 Digit select to 7seg
   * #define SEL1 Digit select to 7seg
   * #define SEL2 Digit select to 7seg
   * #define OE_N Enable to bargraph
   * #define SHLD Shift/Load to encoder
   */

  uint8_t i = 0;

  digit_init();
  timer_init();
  spi_init();

  sei();

  while(1)
  {

    //interpret_encoders();

    //barNum++;
    //segNum--;
    //segNum = buttonState;
    // Update number to digitSelect[i]
    segsum(segNum);
    barNum = buttonState;

    //make PORTA an output
    DDRA = 0xFF;

    //bound a counter (0-4) to keep track of digit to display
    for (i=0; i<5; i++)
    {
      // Clear digit select
      PORTB &= SELCL;

      //update digit to display
      PORTB |= digitSelect[i];

      //send 7 segment code to LED segments
      PORTA = segment_data[i];

      //dimming/flicker correction
```



```c
      //_delay_ms(10);
      _delay_us(200);
    }//for

    //_delay_us(100);

    //outputToBargraph(0b10101010);

  }//while

  return 0;

}//main
```