```
/* RGB RingCoder Example Code
  by: Jim Lindblom
  SparkFun Electronics
  date: February 14, 2012
  license: Beerware - Consider this code open domain, please use it
  however you please. If you find it useful, and we meet someday,
  you can buy me a beer (or a Root Beer).
  environment: Arduino 1.0 (older versions should work)

  This code uses interrupts to read the encoder state. If you don't
  like this version of the encoder reading, I encourage you to check
  out this link:
  http://arduino.cc/playground/Main/RotaryEncoders

  Some more great encoder code:
  http://www.circuitsathome.com/mcu/reading-rotary-encoder-on-arduino

  This example code is designed for SparkFun's RGB Ringcoder
  Breakout board. The Ringcoder board has an illuminated rotary
  encoder, and a 16-LED circular bar graph. Two shift registers
  are used to illuminate the bar graph LEDs.

  This sketch allows you to set the color of the rotary encoder's
  RGB LED. At the beginning of the sketch, turn the encoder to
  select the red intensity. Pressing the switch will store that
  color and move on to blue. Pressing the switch again will go to
  selecting green, again will go back to red.As you're selecting
  each color, the circular bar graph will fill up to indicate
  the intensity of the color you're selecting.

  Hardware connection is as follows:

  LED Ringcoder Board          Arduino
  -------------------          -------
   GND (ground)                  GND
    B (Encoder B)                2 - External interrupt 0
    A (Encoder A)                3 - External interrupt 1
RED (Encoder Red LED)            5 - PWM enabled
BLU (Encoder Blue LED)           6 - PWM enabled
GRN (Encoder Green LED)          9 - PWM endabled
SW (Encoder push button)         7
   DAT (shift data)              8
   CLR (shift clear)             10
   CLK (shift clock)             11
  LATCH (shift latch)            12
   EN (shift enable)             13
     VCC (power)                 5V
*/

// The CONTINUOUS define will adjust how the encoderPosition is
// incremented and decremented. If CONTINUOUS is defined, the
// encoderPosition will be able to go from 255 to 0, and vice-versa.
// If CONTINUOUS is disabled, encoderPosition will max out at 255
// and will not go from 0 to 255 if decremented.
//#define CONTINUOUS

// These three defines (ROTATION_SPEED, ENCODER_POSITION_MAX, and
// ENCODER_POSITION_MIN) control how fast the circular bar graph
// will fill up as you rotate the encoder.These depend on the
// encoderPosition variable being an unsigned 8-bit type.
// These will only be used if CONTINUOUS is NOT defined.
```

```
#define ROTATION_SPEED 3  // MIN: 0, MAX: 5, 3 is a good value
#define ENCODER_POSITION_MAX  (256 >> (ROTATION_SPEED - 1)) - 1
#define ENCODER_POSITION_MIN  0  // Don't go below 0

// Pin definitions - Encoder:
int aPin = 3;  // Encoder A pin, D3 is external interrupt 1
int bPin = 2;  // Encoder B pin, D2 is external interrupt 0
int redPin = 5;  // Encoder's red LED - D5 is PWM enabled
int bluPin = 6;  // Encoder's blue LED- D6 is PWM enabled
int grnPin = 9;  // Encoder's green LED - D9 is PWM enabled
int swhPin = 7;  // Encoder's switch pin

// Pin definitions - Shift registers:
int enPin = 13;  // Shift registers' Output Enable pin
int latchPin = 12;  // Shift registers' rclk pin
int clkPin = 11;  // Shift registers' srclk pin
int clrPin = 10;  // shift registers' srclr pin
int datPin = 8;  // shift registers' SER pin

// The encoderPosition variable stores the position of the encoder.
// It's either incremented or decremented in the encoder's
// interrupt handler (readEncoder()). It's widely used in both the
// loop() and ledRingFiller() and ledRingFollower() functions to
// control the LEDs.
signed int encoderPosition;  // Store the encoder's rotation counts

// These three variables (ledCount, ledValue[], and ledPins[]
// are used to store the intensity of the RG and B leds in the
// rotary encoder.
//   ledCount is used to keep track of which LED is being
// configured (increased by pressing the encoder's switch).
//   ledValue[] stores the 8-bit analog value for the LED. They're
// common-anode, so 255 is OFF and 0 is ON.
//   ledPins[] just stores each of the pins of the LED in an array
// the pins should be defined above.
//byte ledCount = 0;
enum ledCounter {RED = 0, BLUE = 1, GREEN = 2, NONE = 3};
byte ledCount = RED;
byte ledValue[3] = {255, 255, 255};
byte ledPins[3] = {redPin, bluPin, grnPin};

void setup()
{
  // Setup encoder pins, they should both be set as inputs
  // and internally pulled-up
  pinMode(aPin, INPUT);
  digitalWrite(aPin, HIGH);
  pinMode(bPin, INPUT);
  digitalWrite(bPin, HIGH);

  // just to be safe, let's not interrupt until everything's setup
  noInterrupts();
  // Attach interrupts to encoder pins. Whenever one of the encoder
  // pins changes (rise or fall), we'll go to readEncoder()
  attachInterrupt(0, readEncoder, CHANGE);
  attachInterrupt(1, readEncoder, CHANGE);

  // setup switch pins, set as an input, no pulled up
  pinMode(swhPin, INPUT);
  digitalWrite(swhPin, LOW);  // Disable internal pull-up
```

```
  // Setup led pins as outputs, and write their intial value.
  // initial value is defined by the ledValue global variable
  pinMode(redPin, OUTPUT);
  analogWrite(redPin, ledValue[RED]);  // Red off
  pinMode(grnPin, OUTPUT);
  analogWrite(grnPin, ledValue[GREEN]);  // Green off
  pinMode(bluPin, OUTPUT);
  analogWrite(bluPin, ledValue[BLUE]);  // Blue off

  // Setup shift register pins
  pinMode(enPin, OUTPUT);  // Enable, active low, this'll always be
LOW
  digitalWrite(enPin, LOW);  // Turn all outputs on
  pinMode(latchPin, OUTPUT);  // this must be set before calling
shiftOut16()
  digitalWrite(latchPin, LOW);  // start latch low
  pinMode(clkPin, OUTPUT);  // we'll control this in shiftOut16()
  digitalWrite(clkPin, LOW);  // start sck low
  pinMode(clrPin, OUTPUT);  // master clear, this'll always be HIGH
  digitalWrite(clrPin, HIGH);  // disable master clear
  pinMode(datPin, OUTPUT);  // we'll control this in shiftOut16()
  digitalWrite(datPin, LOW);  // start ser low

  // To begin, we'll turn all LEDs on the circular bar-graph OFF
  digitalWrite(latchPin, LOW);  // first send latch low
  shiftOut16(0x0000);
  digitalWrite(latchPin, HIGH);  // send latch high to indicate data
is done sending

  // Now we can enable interrupts and start the code.
  interrupts();
}

void loop()
{
  // If the switch is pressed, we'll increment ledCount
  if (digitalRead(swhPin) == HIGH)
  {
    // In here we'll increment ledCount, but we have to make sure
    // it stays within our bounds (0-3).
    if (ledCount == NONE)  // If we're at NONE we need to go back to
RED
      ledCount = RED;
    else
      ledCount++;  // Otherwise, just increment ledCount

    // ledCount should change just once, per click. So, we'll
    // do nothing while waiting for the switch to go back to LOW
    while(digitalRead(swhPin) == HIGH)
      ;  // do nothing
  }

  // Every time through the loop, the LED bar graph should be
  // updated. There are two options here, you can use ledRingFiller()
  // which will fill up the bar graph. Or you can use
ledRingFollower()
  // which will illuminate only one bar graph LED at a time.
  ledRingFiller(ROTATION_SPEED);  // Update the Bar graph LED
  // Uncomment the below line, and comment out the one above
  // if you want to activate the ledRingFollower();
  //ledRingFollower(ROTATION_SPEED);  // Update the bar graph LED
```

```
  // Every time through loop, the current active LED will be updated
  // If the current LED is NONE, then all LEDs will remain the same
  if (ledCount != NONE)  // Only update the LED if it's RED, GREEN or
BLUE
  {
    // Calculate the ledValue of the active LED. We use
    // encoderPosition, and compensate with the ROTATION_SPEED
    // vaule to get something between 0 and 255. The '255 - ' part
    // is required because the LEDs are pulled high.
    ledValue[ledCount] = 255 - (encoderPosition * ROTATION_SPEED);
    // Now we analogWrite the calculated value to the active LED
    analogWrite(ledPins[ledCount], ledValue[ledCount]);
  }
}

// void ledRingFiller(byte rotationSpeed) - This is one of two
// functions that can be used to update the led ring bar graph thing.
//   This will illuminate all LEDs up to the equivalent of
// encoderPosition.
//   The input variable, rotationSpeed, should be some value between
// 1 and 5.
//   This function uses encoderPosition, updated by the readEncoder()
// interrupt handler, to decide what LEDs to illuminate.
void ledRingFiller(byte rotationSpeed)
{
  // ledShift stores the bit position of the upper-most LED
  // this value should be between 0 and 15 (shifting a 16-bit vaule)
  unsigned int ledShift = 0;
  // each bit of ledOutput represents a single LED on the ring
  // this should be a value between 0 and 0xFFFF (16 bits for 16 LEDs)
  unsigned int ledOutput = 0;

  // Only do this if encoderPosition = 0, if it is 0, we don't
  // want any LEDs lit up
  if (encoderPosition != 0)
  {
    // First set ledShift equal to encoderPosition, but we need
    // to compensate for rotationSpeed.
    ledShift = encoderPosition & (0xFF >> (rotationSpeed-1));
    // Now divide ledShift by 16, also compensate for rotationSpeed
    ledShift /= 0x10>>(rotationSpeed-1);
    // This for loop sets each bet that is less signfigant than
    // ledShift. This is what sets ledBarFiller apart from
    // ledBarFollower()
    for (int i=ledShift; i>=0; i--)
      ledOutput |= 1<<i;
  }

  // Now we just need to write to the shift registers. We have to
  // control latch manually, but shiftOut16 will take care of
  // everything else.
  digitalWrite(latchPin, LOW);  // first send latch low
  shiftOut16(ledOutput);  // send the ledOutput value to shiftOut16
  digitalWrite(latchPin, HIGH);  // send latch high to indicate data
is done sending
}

// void ledRingFollower(byte rotationSpeed) - This is one of two
// functions that can be used to update the led ring bar graph thing.
//   This will illuminate a single LED equivalent to the value
```

```
// of encoderPosition.
//   The input variable, rotationSpeed, should be some value between
// 1 and 5.
//   This function uses encoderPosition, updated by the readEncoder()
// interrupt handler, to decide what LEDs to illuminate.
void ledRingFollower(byte rotationSpeed)
{
  // ledShift stores the bit position of the upper-most LED
  // this value should be between 0 and 15 (shifting a 16-bit vaule)
  unsigned int ledShift = 0;
  // each bit of ledOutput represents a single LED on the ring
  // this should be a value between 0 and 0xFFFF (16 bits for 16 LEDs)
  unsigned int ledOutput = 0;

  // Only do this if encoderPosition = 0, if it is 0, we don't
  // want any LEDs lit up
  if (encoderPosition != 0)
  {
    // First set ledShift equal to encoderPosition, but we need
    // to compensate for rotationSpeed.
    ledShift = encoderPosition & (0xFF >> (rotationSpeed-1));
    // Now divide ledShift by 16, also compensate for rotationSpeed
    ledShift /= 0x10>>(rotationSpeed-1);
    // Now just use ledShift to calculate ledOutput.
    // ledOutput will only have 1 bit set
    ledOutput = 1 << ledShift;
  }

  // Now we just need to write to the shift registers. We have to
  // control latch manually, but shiftOut16 will take care of
  // everything else.
  digitalWrite(latchPin, LOW);  // first send latch low
  shiftOut16(ledOutput);  // send the ledOutput value to shiftOut16
  digitalWrite(latchPin, HIGH);  // send latch high to indicate data
is done sending
}

// This function'll call shiftOut (a pre-defined Arduino function)
// twice to shift 16 bits out. Latch is not controlled here, so you
// must do it before this function is called.
//   data is sent 8 bits at a time, MSB first.
void shiftOut16(uint16_t data)
{
  byte datamsb;
  byte datalsb;

  // Isolate the MSB and LSB
  datamsb = (data&0xFF00)>>8;  // mask out the MSB and shift it right
8 bits
  datalsb = data & 0xFF;  // Mask out the LSB

  // First shift out the MSB, MSB first.
  shiftOut(datPin, clkPin, MSBFIRST, datamsb);
  // Then shift out the LSB
  shiftOut(datPin, clkPin, MSBFIRST, datalsb);
}

// readEncoder() is our interrupt handler for the rotary encoder.
// This function is called every time either of the two encoder
// pins (A and B) either rise or fall.
//   This code will determine the directino of rotation, and
```

```
// update the global encoderPostion variable accordingly.
//   This code is adapted from Rotary Encoder code by Oleg.
void readEncoder()
{
  noInterrupts();  // don't want our interrupt to be interrupted
  // First, we'll do some software debouncing. Optimally there'd
  // be some form of hardware debounce (RC filter). If there is
  // feel free to get rid of the delay. If your encoder is acting
  // 'wacky' try increasing or decreasing the value of this delay.
  delayMicroseconds(5000);  // 'debounce'

  // enc_states[] is a fancy way to keep track of which direction
  // the encoder is turning. 2-bits of oldEncoderState are paired
  // with 2-bits of newEncoderState to create 16 possible values.
  // Each of the 16 values will produce either a CW turn (1),
  // CCW turn (-1) or no movement (0).
  int8_t enc_states[] = {0,-1,1,0,1,0,0,-1,-1,0,0,1,0,1,-1,0};
  static uint8_t oldEncoderState = 0;
  static uint8_t newEncoderState = 0;

  // First, find the newEncoderState. This'll be a 2-bit value
  // the msb is the state of the B pin. The lsb is the state
  // of the A pin on the encoder.
  newEncoderState = (digitalRead(bPin)<<1) | (digitalRead(aPin));

  // Now we pair oldEncoderState with new encoder state
  // First we need to shift oldEncoder state left two bits.
  // This'll put the last state in bits 2 and 3.
  oldEncoderState <<= 2;
  // Mask out everything in oldEncoderState except for the previous
state
  oldEncoderState &= 0xC0;
  // Now add the newEncoderState. oldEncoderState will now be of
  // the form: 0b0000(old B)(old A)(new B)(new A)
  oldEncoderState |= newEncoderState; // add filteredport value

  // Now we can update encoderPosition with the updated position
  // movement. We'll either add 1, -1 or 0 here.
  encoderPosition += enc_states[oldEncoderState];

  // This next bit will only happen if CONTINUOUS is not defined.
  // If CONTINUOUS is defined, encoderPosition will roll over from
  // -32768 (assuming it's a signed int) to to 32767 if decremented,
  // or 32767 to -32768 if incremented.
  //   That can be useful for some applications. In this code, we
  // want the encoder value to stop at 255 and 0 (makes analog writing
easier)
  #ifndef CONTINUOUS
    // If encoderPosition is greater than the MAX, just set it
    // equal to the MAX
    if (encoderPosition > ENCODER_POSITION_MAX)
      encoderPosition = ENCODER_POSITION_MAX;
    // otherwise, if encoderPosition is less than the MIN, set it
    // equal to the MIN.
    else if (encoderPosition < ENCODER_POSITION_MIN)
      encoderPosition = ENCODER_POSITION_MIN;
  #endif

  interrupts();  // re-enable interrupts before we leave
}
```