RGB PWM Lab

Goal: to use Arduino PWM output to produce a meaningful (and pretty) result, and to compare what our eyes see to what scientific equipment sees, specifically a spectrometer. For this lab you will need an Arduino, power source or USB-B cable, 600-ohm resistor, RGB LED, and a breadboard with some jumpers.

The Arduino's PWM output is 8-bit, which means it takes an *argument* of a number between 0b00000000 and 0b11111111, or in decimal 0 to 255. This is known as *8-bit resolution*, since the microcontroller is capable of resolving 256 discrete steps of output. We're going to apply this principle to learn where the concept of 24-bit color comes from.

The earliest computer monitors had 1-bit color, where each pixel could either be black or white. With the advent of sophisticated graphics hardware and color pixels, the *bit depth* of displays grew higher and colors got more realistic. Monitors today are capable of 24-bit color, which divides the output into primary colors red, green, and blue and allows 8 bits for each color. Furthermore, by including 8 bits for alpha (transparency), the RGBA system delivers 32-bit color. A fancy digital camera may record 10 or 12 bits per color in RAW format, while some high-end displays use 16 bits per color and 16 bits for the alpha channel for 64 bits total.

We're going to exploit the similarities in the Arduino's PWM output and the 24-bit color model to create our own colors. Remember that in 24-bit color, each color (often referred to as a channel) gets 8 bits. Our Arduino's output has 8-bit resolution. To begin, let's look at how colors are usually coded in computer programs.

On computers, colors can be represented by a 6-digit hexadecimal (base 16) number. In base 10, we count from 0 to 9 before we need to add another placeholder, and then we count from 10-99 before doing so again. In hexadecimal, we count zero to fifteen before we need a new placeholder! So, in addition to the arabic numerals 0-9, we need six more symbols. The convention is to use A, B, C, D, E, and F. So, counting in base 16 looks like this:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

...and then we add a placeholder and keep going:

10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 2A, 2B, 2C, 2D, 2E, 2F, 30...FA, FB, FC, FD, FE, FF, 100...

...and so on! There are logical and aesthetic reasons for working in base 16. Computers think strictly in binary. If you need to mess with individual bits (such as we'll do when programming colors), then binary or hexadecimal are the natural choices. 10 is not a power of 2, and so when

we're writing code that explicitly tells a computer what to do, it's often not a good choice, although you still could achieve identical results! It just wouldn't make sense to anyone else who reads your code why you did it in base 10, and it would require extra time on your part to do the conversions.

So, back to how colors are represented on a computer. When we write numbers in hexadecimal, it's necessary to denote them as such. There are two ways to do this. One is with a prefix "0x" and the other is with a suffix "h". For example. The hexadecimal numbers 0x36 and 36h are equivalent. In a computer program, a number with neither prefix nor suffix is assumed to be in decimal (binary uses "0b" and "b" prefixes and suffixes, respectively). The numbers 0x36, 110110b and 54 are equivalent.

24-bit colors are encoded in a **6 digit hexadecimal number.** Remember, we get 8 bits per color which is 2 digits in hexadecimal, so every two digits corresponds to either red, green, or blue. in fact, the standard is

0xRRGGBB

So, if we wanted to program pure colors, we would maximize one while minimizing the others. For example, if we wanted pure red, we would set the red bits equal to their maximum value (FF) while setting green and blue to their minimum (00). It would look like 0xFF0000. Similarly, pure green and blue would be 0x00FF00 and 0x0000FF, respectively. Mixing colors is achieved by setting the red, green, and blue bits to the "amount" of each color we want on a scale of 0x00 to 0xFF.

Activity: Go to http://www.colorpicker.com/ and experiment with entering different values in the hexadecimal code box to see what colors you get. Or, click in the colored areas to see which hexadecimal code is generated. Keep this site open, since we'll use it to compare our Arduino colors.

At this point, we'll discuss the circuit in class. Draw the circuit here for future reference. Don't forget to neatly label all components.

Now, we'll program our Arduinos to emit color. This lab makes use of the analogWrite() function that's built in to the Arduino framework. This function takes two arguments, *pin* and *value. Pin* is the physical pin you want to write to, and *value* is a number from 0 to 255 (or 0x00 to 0xFF, or 0b00000000 to 0b11111111). What the function does is create a PWM square wave at approximately 490 Hz and at a *duty cycle* specified by *value*. The duty cycle of a PWM wave is the amount of time the signal is high during a single period. Duty cycle is expressed in percent, so a 75% duty cycle wave would be high 75% of the period and low 25% of the period. Therefore, calling analogWrite(9, 127) would output a 50% duty cycle square wave on pin 9. Why 50%? Because 127 is extremely close to half of 255. If we were to use 255 as our *value,* the wave would have 100% duty cycle, that is, always on. This is an extremely advantageous feature since many devices are sensitive to the average voltage of an input. If we were to use a 50% duty cycle square wave that goes from 0 to 5 volts as the driver for an LED, it would be the equivalent of providing a constant 2.5 volts! This, of course, assumes the frequency is fast enough. A 50% duty cycle of a 1 Hz wave is equivalent to providing 5 volts for half a second and 0 volts for half a second. Fortunately, the 490 Hz from the Arduino is sufficient for exploiting the "averaging" aspect of PWM. You can also use hexadecimal numbers as your *value*, so analogWrite(9,127) is equivalent to analogWrite(9,0x7F).

Now let's actually program the Arduino. We will discuss the program in class as we write it, as it contains many important techniques for writing cohesive, easily-editable code.

```
#define RED 0x7F
#define GREEN 0x7F
#define BLUE 0x7F

int redPin = 9;
int greenPin = 10;
int bluePin = 11;

void setup()
{
    analogWrite(redPin,RED);
    analogWrite(greenPin,GREEN);
    analogWrite(bluePin,BLUE);
}

void loop()
{
    // do nothing here, since we only want the microcontroller to set color once.
}
```

Type the code in exactly as above. Don't worry about how your programming environment will color the code, since the above code was written in a different editor. Upload your code to your Arduino.

Question: What color does your LED produce? Go to the colorpicker website and enter 7F7F7F in the hex input box. What color does it return?

Now edit your code. On the colorpicker website, pick your favorite color and note its hex code. program that into your Arduino, remembering the 0xRRGGBB format. Upload your new code.

Question: how closely does the output of the LED match the color you chose?

Now edit your code again, using values of 0xFF for red and blue, and 0x00 for green. Upload your code and note the color the LED produces. Turn off the lights in the room and insert your LED into the integrating sphere of the lab spectrometer.

Question: pure purple light has a wavelength around 400 to 430 nm. What do you see in the spectrometer? Was it what you expected?

Now edit your code one last time. This code will also be discussed in class.

```
int redPin = 9;
int greenPin = 10;
int bluePin = 11;

void setup()
{
    // empty function since there's nothing to set up
}

void loop()
{
    analogWrite(redPin,random(0x00, 0xFF));
    analogWrite(greenPin,random(0x00, 0xFF));
    analogWrite(bluePin,random(0x00, 0xFF));
    delay(1000);
}
```

Enjoy the light show! You can also increase the argument in delay() to prolong the colors and study the changing output with the spectrometer. Alternatively, decrease the delay to produce rapidly changing colors.