Multi-tasking the Arduino - Part 3

Created by Bill Earl

https://learn.adafruit.com/multi-tasking-the-arduino-part-3

Last updated on 2023-08-29 02:44:37 PM EDT

Table of Contents

Overview

Illustration by David Earl

Digital RGB LEDs like the Neopixel are awesome for creating stunning displays and lighting effects. But integrating them into an interactive project can be a challenge. The Arduino is a single-minded little processor that only likes to do one thing at a time. So how do you get it to pay attention to external inputs while generating all those mezmerizing pixel patterns?

Some of the most common Neopixel questions in the Adafruit forums are:

- How can I make my Neopixel project respond reliably to button presses?
- How can I run two (or more) different Neopixel patterns at the same time?
- How can I make my Arduino do other things while my Neopixel pattern is running?

In this guide, we'll look at some ways to structure your Neopixel code to keep it responsive and make it more amenable to multitasking.

The Problem? Loops and Delays

Virtually all the example code consists of loops that step through the various phases of the pixel animation. The code is so busy updating the pixels that the main loop never has a chance to check the switches.

But is it really busy? In reality, the code spends most of it's time doing absolutely nothing! This is because the timing is all done with calls to delay(). As we saw in part one of this series (), the delay is quite literally a total waste of time. We definitely want to ditch the delay.

But that still leaves us with the loops. If you try to write one loop that performs two animation patterns at the same time, the code starts to get real ugly real fast. We need to lose the loop too.

Deconstructing the Loop

Don't be a Quitter

One commonly suggested solution to the responsiveness problem is to check your switches inside your loops and quit the loop if a switch is pressed. The switch check can be neatly combined with the delay. And the loop can be re-written to exit early.

This does make your program more resonsive, but there are a couple of fundamental problems with this approach:

- You are still using delay() (), so you still can't do two (or more) patterns at once.
- And what if you don't really want to quit? We want a way to remain responsive - without necessarily disturbing the execution of the pattern.

Will interrupts help?

We learned how to use Interrupts in Part 2 of this series (). But they are not the total answer to the these problems either.

With interrupts, you can eliminate the need to poll the switches. That makes the code a little neater, but it doesn't really address the rest of the problems listed above

Thinking outside the loop:

It is possible to solve all these problems. But we will need to both ditch the delay and lose the loop. The good news is, the resulting code is surprisingly simple.

The Neopixel loop & delay problem is very similar to the servo sweep example we re-engineered in Part 1 (). We can apply the same state machine approach to Neopixel patterns.

The basic idea is to encapsulate the patterns in a C++ class. Capture all the state variables as member variables and move the core of the loop logic into an Update() function that uses millis() to manage the timing.

The Update function can be called from your loop() or a timer interrupt and you can update many patterns simultaneously on each pass, while monitoring user interactions at the same time.

Common Code

In this tutorial we will re-engineer some of the more popular pixel patterns, including:

- Rainbow Cycle
- Color Wipe
- Theater Chase
- Scanner
- Fader

We'll encapsulate these patterns into a single "NeoPattern" class derived from the Ad afruit_NeoPixel class so you can freely change between patterns on the same strip.

And, since NeoPattern is derived from Adafruit_NeoPixel, you can use all the normal funcitons from the NeoPixel library on it too!

Before we get into the pattern code imlementation. Let's take a look at what these patterns have in common, what we need to know to manage them. From that we can create a class structure that will handle them all:

Member Variables:

The class definition below contains member variables to contain the pattern in use, the directon it is running in and the current state of the pattern state machine.

Pro-Tip: The Direction option is handy when dealing with neopixel rings that have counter-clockwise vs clockwise pixel ordering.

These variables represent a tradeoff between speed and size. The per-pixel overhead is the same as for the Adafruit_Neopixel base class. Most of the pattern state variables are common to most of the patterns

Given the limited SRAM available on the Arduino, some calculations will be done 'on-the-fly' in the Update() functions and save the extra space for more pixels!

Callback on Completion:

If initialized, the "OnComplete()" callback function is invoked on completion of each iteration of the pattern. You can define the callback to perform any action - including making changes to the pattern and/or operating state. We'll show some examples of how this is used later.

```
#include &lt;Adafruit_NeoPixel.h&gt;

// Pattern types supported:
enum  pattern { NONE, RAINBOW_CYCLE, THEATER_CHASE, COLOR_WIPE, SCANNER, FADE };
// Patern directions supported:
enum  direction { FORWARD, REVERSE };

// NeoPattern Class - derived from the Adafruit_NeoPixel class
class NeoPatterns : public Adafruit_NeoPixel
{
    public:

    // Member Variables:
    pattern  ActivePattern;  // which pattern is running
    direction Direction;     // direction to run the pattern

    unsigned long Interval;   // milliseconds between updates
    unsigned long lastUpdate; // last update of position

    uint32_t Color1, Color2;  // What colors are in use
    uint16_t TotalSteps;  // total number of steps in the pattern
    uint16_t Index;  // current step within the pattern

    void (*OnComplete)();  // Callback on completion of pattern
```

Constructor:

The constructor initializes the base-class, as well as an (optional) pointer to a callback function.

```
    // Constructor - calls base-class constructor to initialize strip
    NeoPatterns(uint16_t pixels, uint8_t pin, uint8_t type, void (*callback)())
    :Adafruit_NeoPixel(pixels, pin, type)
    {
```

```
        OnComplete = callback;
    }
```

The Updater:

The Update() function works much like the Update() functions from part 1 () of the series. It checks the elapsed time since the last update and returns immediately if it is not time to do anything yet.

If it determines that it is time do actualy update the pattern, it will look at the ActivePa ttern member variable to decide which pattern-specific update function to call.

For each of the listed patterns, we will need to write an initialization function and an Update() function. We'll show how to do this in the next few pages:

```
    // Update the pattern
    void Update()
    {
        if((millis() - lastUpdate) &gt; Interval) // time to update
        {
            lastUpdate = millis();
            switch(ActivePattern)
            {
                case RAINBOW_CYCLE:
                    RainbowCycleUpdate();
                    break;
                case THEATER_CHASE:
                    TheaterChaseUpdate();
                    break;
                case COLOR_WIPE:
                    ColorWipeUpdate();
                    break;
                case SCANNER:
                    ScannerUpdate();
                    break;
                case FADE:
                    FadeUpdate();
                    break;
                default:
                    break;
            }
        }
    }
```

Incrementing the Pattern

The Increment() function looks at the current state of Direction and increments or decrements Index accordingly. When Index reaches the end of the pattern, it is wrapped around to restart the pattern. If the OnComplete() callback function is non-null then it is called.

```
    // Increment the Index and reset at the end
    void Increment()
    {
```

```
        if (Direction == FORWARD)
        {
           Index++;
           if (Index &gt;= TotalSteps)
            {
                Index = 0;
                if (OnComplete != NULL)
                {
                    OnComplete(); // call the comlpetion callback
                }
            }
        }
        else // Direction == REVERSE
        {
            --Index;
            if (Index &lt;= 0)
            {
                Index = TotalSteps-1;
                if (OnComplete != NULL)
                {
                    OnComplete(); // call the comlpetion callback
                }
            }
        }
    }
```

RainbowCycle

The Rainbow Cycle uses the color wheel to create a rainbow effect that cycles over the length of the strip. This is a straightforward re-structuring of the RainbowCycle pattern in the StrandTest example sketch from the library.

Initialization:

The RainbowCycle() function initializes the NeoPatterns class for running the Rainbow Cycle pattern.

ActivePattern is set to RAINBOW_CYCLE.

The "interval" parameter specifies the number of milliseconds between updates, which determines the speed of the pattern.

The "dir" parameter is optional. It defaults to FORWARD operation, but you can specify REVERSE.

TotalSteps is set to 255, which is the number of colors in the color wheel.

Index is set to 0, so we start with the first color in the wheel.

```
    // Initialize for a RainbowCycle
    void RainbowCycle(uint8_t interval, direction dir = FORWARD)
```

```
    {
        ActivePattern = RAINBOW_CYCLE;
        Interval = interval;
        TotalSteps = 255;
        Index = 0;
        Direction = dir;
    }
```

Update:

The RainbowCycleUpdate() function sets each pixel on the strip to a color from the wheel. The starting point on the wheel is offset by Index.

After writing to the strip with a call to show(), Increment() is called to update the Index.

```
    // Update the Rainbow Cycle Pattern
    void RainbowCycleUpdate()
    {
        for(int i=0; i&lt; numPixels(); i++)
        {
            setPixelColor(i, Wheel(((i * 256 / numPixels()) + Index) &amp;amp; 255));
        }
        show();
        Increment();
    }
```

ColorWipe

The ColorWipe pattern paints a color, one pixel at a time, over the length of the strip.

Initialization:

The ColorWipe() function initializes the NeoPattern object to execute the ColorWipe pattern.

The color parameter specifies the color to 'wipe' across the strip.

The interval parameter specifies the number of milliseconds between successive pixels in the wipe.

The direction parameter is optional and specifies the direction of the wipe. The default direction, if not specified is FORWARD.

TotalSteps is set to the number of pixels in the strip.

```
// Initialize for a ColorWipe
    void ColorWipe(uint32_t color, uint8_t interval, direction dir = FORWARD)
    {
        ActivePattern = COLOR_WIPE;
        Interval = interval;
        TotalSteps = numPixels();
        Color1 = color;
        Index = 0;
        Direction = dir;
    }
```

Update:

The ColorWipeUpdate() function sets the next pixel in the strip, then calls show() to write to the strip and Increment() to update the Index.

```
// Update the Color Wipe Pattern
    void ColorWipeUpdate()
    {
        setPixelColor(Index, Color1);
        show();
        Increment();
    }
```

TheaterChase

The TheaterChase pattern emulates the classic chase pattern from '50s era theater marquees. In this variation, you can specify the foreground and background colors for the chasing lights.

Initialization:

The TheaterChase() function initializes the NeoPattern object for executing the TheaterChase pattern.

The color1 and color2 parameters specify the foreground and background colors of the pattern.

The increment parameter specifies the number of milliseconds between updates of the pattern and controls the speed of the chase effect.

The direction parameter is optional and allows you to run the pattern in FORWARD (the default) or REVERSE.

TotalSteps is set to the number of pixels in the strip.

```
// Initialize for a Theater Chase
    void TheaterChase(uint32_t color1, uint32_t color2, uint8_t interval, direction
dir = FORWARD)
    {
        ActivePattern = THEATER_CHASE;
        Interval = interval;
        TotalSteps = numPixels();
        Color1 = color1;
        Color2 = color2;
        Index = 0;
        Direction = dir;
   }
```

Update:

TheaterChaseUpdate() advances the chase pattern by one pixel. Every 3rd pixel is drawn in the foreground color (color1). All the rest are drawn in color2.

After writing the pixel colors to the strip via show(), Increment is called to update Inde x.

```
// Update the Theater Chase Pattern
    void TheaterChaseUpdate()
    {
        for(int i=0; i&lt; numPixels(); i++)
        {
            if ((i + Index) % 3 == 0)
            {
                setPixelColor(i, Color1);
            }
            else
            {
                setPixelColor(i, Color2);
            }
        }
        show();
        Increment();
    }
```

Scanner

The Scanner pattern consists of a single bright led scanning back and forth, leaving a trail of fading leds behind as it goes.

Initialization:

Scanner() initializes the NeoPattern object for executing the Scanner pattern.

The color parameter is the color of the scanning pixel.

The interval parameter specifies the number of milliseconds between updates and controls the speed of the scanning.

A scan is considered to be a complete round-trip from one end of the strip to the other and back with no additional dwell-time on the end pixels. So the TotalSteps is set to twice the number of pixels minus two.

```
// Initialize for a SCANNNER
void Scanner(uint32_t color1, uint8_t interval)
{
    ActivePattern = SCANNER;
    Interval = interval;
    TotalSteps = (Strip-&gt;numPixels() - 1) * 2;
    Color1 = color1;
    Index = 0;
}
```

Update:

ScannerUpdate() iterates over the pixels in the strip. If the pixel corresponds to the current value of Index or TotalSteps - Index, we set the pixel to color1.

Otherwise, we 'dim' the pixel to create a fading trail effect.

As with all the other patterns, the ScannerUpdate calls show() to write to the strip and Increment() to advance the state machine.

```
// Update the Scanner Pattern
void ScannerUpdate()
{
    for (int i = 0; i &lt; numPixels(); i++)
    {
        if (i == Index) // first half of the scan
        {
            Serial.print(i);
            setPixelColor(i, Color1);
        }
        else if (i == TotalSteps - Index) // The return trip.
        {
            Serial.print(i);
            setPixelColor(i, Color1);
        }
        else  // fade to black
        {
            setPixelColor(i, DimColor(getPixelColor(i)));
        }
    }
    show();
    Increment();
}
```

Fader

The Fader pattern is one of the most commonly requested Neopixel effects on the forums. This pattern produces a smooth linear fade from one color to another.

Initialization:

Fade() initializes the NeoPattern object to execute the Fade pattern.

The color1 parameter specifies the starting color.

color2 specifies the ending color.

steps specifies how many steps it should take to get from color1 to color2.

interval specifies the number of milliseconds between steps and controls the speed of the fade.

direction allows to to reverse the fade so it goes from color2 to color1.

```
    // Initialize for a Fade
    void Fade(uint32_t color1, uint32_t color2, uint16_t steps, uint8_t interval,
direction dir = FORWARD)
    {
        ActivePattern = FADE;
        Interval = interval;
        TotalSteps = steps;
        Color1 = color1;
        Color2 = color2;
        Index = 0;
        Direction = dir;
    }
```

Update:

FadeUpdate() sets all the pixels on the strip to the color corresponding to the next step.

The color is calculated as a linear interpolation between the red, green and blue components of color1 and color2. To keep it fast, we use all integer math. To minimize truncation errors, the division is performed last.

As with all the other patterns, the ScannerUpdate calls show() to write to the strip and Increment() to advance the state machine.

```
// Update the Fade Pattern
    void FadeUpdate()
    {
        uint8_t red = ((Red(Color1) * (TotalSteps - Index)) + (Red(Color2) *
Index)) / TotalSteps;
        uint8_t green = ((Green(Color1) * (TotalSteps - Index)) + (Green(Color2) *
Index)) / TotalSteps;
        uint8_t blue = ((Blue(Color1) * (TotalSteps - Index)) + (Blue(Color2) *
Index)) / TotalSteps;
        ColorSet(Color(red, green, blue));
        show();
        Increment();
    }
```

Common Utility Functions

Now we'll add some common utility functions that can be used by any of the patterns:

First we have the Red(), Blue() and Green() functions. These are the inverse of the Neopixel Color() function. They allow us to extract the Red, Blue and Green components of a pixel color.

```
    // Returns the Red component of a 32-bit color
    uint8_t Red(uint32_t color)
    {
        return (color &gt;&gt; 16) &amp; 0xFF;
    }

    // Returns the Green component of a 32-bit color
    uint8_t Green(uint32_t color)
    {
        return (color &gt;&gt; 8) &amp; 0xFF;
    }

    // Returns the Blue component of a 32-bit color
    uint8_t Blue(uint32_t color)
    {
        return color &amp; 0xFF;
    }
```

The DimColor() funtion uses Red(), Green() and Blue() to pull apart a color and re-construct a dimmed version of it. It shifts the red, green and blue components of the pixel color one bit to the right - effectively dividing by 2. Since red, green ane blue are 8-bit values, they will fade to black after the 8th call at the latest.

This function is used to create the fading tail effect in the Scanner pattern.

```
// Return color, dimmed by 75% (used by scanner)
    uint32_t DimColor(uint32_t color)
    {
        uint32_t dimColor = Color(Red(color) &gt;&gt; 1, Green(color) &gt;&gt; 1,
Blue(color) &gt;&gt; 1);
        return dimColor;
    }
```

Next we have the Wheel() function as used by the RainbowCycle pattern. This is borrowed pretty much as-is from the StrandTest example.

```
// Input a value 0 to 255 to get a color value.
    // The colours are a transition r - g - b - back to r.
    uint32_t Wheel(byte WheelPos)
    {
        WheelPos = 255 - WheelPos;
        if(WheelPos &lt; 85)
        {
            return Color(255 - WheelPos * 3, 0, WheelPos * 3);
        }
        else if(WheelPos &lt; 170)
        {
            WheelPos -= 85;
            return Color(0, WheelPos * 3, 255 - WheelPos * 3);
        }
        else
        {
            WheelPos -= 170;
            return Color(WheelPos * 3, 255 - WheelPos * 3, 0);
        }
    }
```

The Reverse() function will reverse the direction of pattern execution.

```
    // Reverse direction of the pattern
    void Reverse()
    {
        if (Direction == FORWARD)
        {
            Direction = REVERSE;
            Index = TotalSteps-1;
        }
        else
        {
            Direction = FORWARD;
            Index = 0;
        }
    }
```

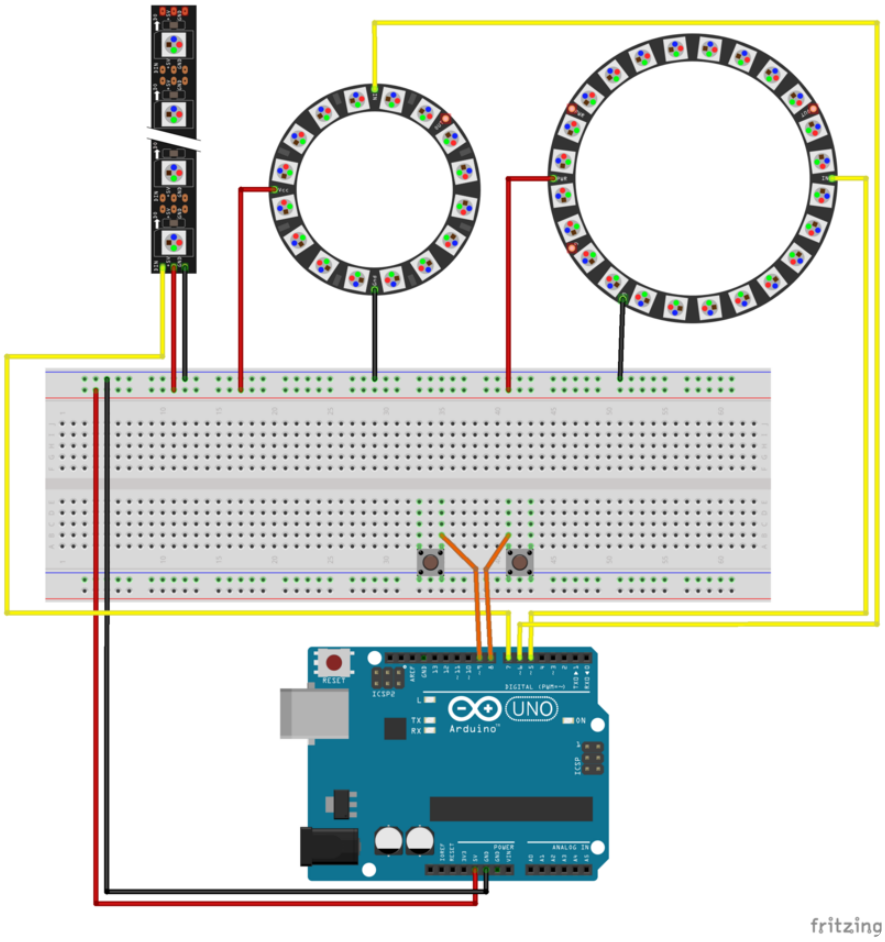
And finally, we have the ColorSet() function to set all pixels to the same color synchronously. This is used by the Fader pattern.

```
    // Set all pixels to a color (synchronously)
    void ColorSet(uint32_t color)
    {
        for (int i = 0; i &lt; numPixels(); i++)
        {
            setPixelColor(i, color);
        }
        show();
    }
```

Wiring

The example code in this guide was developed using 16 and 24 pixel rings and a 16 pixel strip (actually a pair of Neopixel Sticks) wired to 3 separate Arduino pins as shown in the diagram below.

Pushbuttons are connected to pins 8 and 9 to demonstrate responsiveness to user inputs.



Using NeoPatterns

Declare some NeoPatterns

Before we get started, we need to declare some NeoPattern objects to control our pixels.

Using the wiring diagram from the previous page, we'll initialize a 16 and 24 pixel ring, as well as a 16 pixel strip constructed from 2 8-pixel sticks.

```
// Define some NeoPatterns for the two rings and the stick
//  and pass the adderess of the associated completion routines
NeoPatterns Ring1(24, 5, NEO_GRB + NEO_KHZ800, &amp;Ring1Complete);
NeoPatterns Ring2(16, 6, NEO_GRB + NEO_KHZ800, &amp;Ring2Complete);
NeoPatterns Stick(16, 7, NEO_GRB + NEO_KHZ800, &amp;StickComplete);
```

Getting The Action Started

And, as with regular NeoPixel strips, we need to call begin() to get everything initialized prior to using the strips. We'll also set up some input pins for our pushbuttons and kick off an initial pattern for each strip.

```
// Initialize everything and prepare to start
void setup()
{
    // Initialize all the pixelStrips
    Ring1.begin();
    Ring2.begin();
    Stick.begin();

    // Enable internal pullups on the switch inputs
    pinMode(8, INPUT_PULLUP);
    pinMode(9, INPUT_PULLUP);

    // Kick off a pattern
    Ring1.TheaterChase(Ring1.Color(255,255,0), Ring1.Color(0,0,50), 100);
    Ring2.RainbowCycle(3);
    Ring2.Color1 = Ring1.Color1;
    Stick.Scanner(Ring1.Color(255,0,0), 55);
}
```

Updating the Patterns

To keep things running smoothly, you just need to call the Update() function on each NeoPattern on a regular basis. Since we have eliminated the inner loops and delays from the patterns, this is simple to do in either your loop() function as we show here, or in a timer interrupt handler as shown in part 2 of this series ().

The simple case would be to just call the update functions for each of the NeoPatterns you have defined.

```
// Main loop
void loop()
{
    // Update the rings.
    Ring1.Update();
    Ring2.Update();
    Stick.Update();
}
```

User Interactions

With no inner-loops, your code can remain responsive to any user interactions or external events. And you can instantly change patterns or colors based on these events.

To make things more interesting, we will take advantage of this capability and expand our loop to react to button presses on the two pushbuttons:

Pressing Button #1 (pin 8) will:

- Change the pattern on Ring1 from THEATER_CHASE to FADE
- Change the speed of the RainbowCycle on Ring2
- Turn the stick to solid RED

Pressing button #2 (pin 9) will:

- Change the pattern on Ring1 from THEATER_CHASE to COLOR_WIPE
- Change the pattern on Ring2 from THEATER_CHASE to COLOR_WIPE

When no buttons are pressed, all patterns resume normal operation.

```
// Main loop
void loop()
{
    // Update the rings.
    Ring1.Update();
    Ring2.Update();

    // Switch patterns on a button press:
    if (digitalRead(8) == LOW) // Button #1 pressed
    {
        // Switch Ring1 to FASE pattern
        Ring1.ActivePattern = FADE;
        Ring1.Interval = 20;
        // Speed up the rainbow on Ring2
        Ring2.Interval = 0;
        // Set stick to all red
        Stick.ColorSet(Stick.Color(255, 0, 0));
    }
    else if (digitalRead(9) == LOW) // Button #2 pressed
    {
        // Switch to alternating color wipes on Rings1 and 2
        Ring1.ActivePattern = COLOR_WIPE;
        Ring2.ActivePattern = COLOR_WIPE;
        Ring2.TotalSteps = Ring2.numPixels();
        // And update tbe stick
        Stick.Update();
    }
    else // Back to normal operation
    {
        // Restore all pattern parameters to normal values
```

```
        Ring1.ActivePattern = THEATER_CHASE;
        Ring1.Interval = 100;
        Ring2.ActivePattern = RAINBOW_CYCLE;
        Ring2.TotalSteps = 255;
        Ring2.Interval = min(10, Ring2.Interval);
        // And update tbe stick
        Stick.Update();
    }
}
```

Completion Callbacks

Left alone, each pattern will repeat continuously. The optional completion callback gives you the ability to perform some action on completion of each pattern cycle.

The action performed in the completion callback can be to change some aspect of the pattern or trigger some other external event.

Ring1 Completion Callback

The Ring1 completion callback affects the operation of both Ring1 and Ring2. It looks at the state of Button #2. If it is pressed, it will:

- Speed up Ring2 by decreasing its Interval. This effectively 'wakes up' Ring2.
- Slow down Ring1 by increasing its Interval. This effectively puts Ring1 to sleep.
- Selects a radom color from the wheel for the next iteration of Ring1's pattern.

If no button is pressed, it simply reverses the direction of the normal chase pattern.

```
// Ring1 Completion Callback
void Ring1Complete()
{
    if (digitalRead(9) == LOW)  // Button #2 pressed
    {
        // Alternate color-wipe patterns with Ring2
        Ring2.Interval = 40;
        Ring1.Color1 = Ring1.Wheel(random(255));
        Ring1.Interval = 20000;
    }
    else  // Retrn to normal
    {
      Ring1.Reverse();
    }
}
```

Ring2 Completion Callback

The Ring2 completion callback is the complement to the Ring1 completion callback.

If Button #2 is pressed, it will:

- Speed up Ring1 by decreasing its Interval. This effectively 'wakes up' Ring1.
- Slow down Ring2 by increasing its Interval. This effectively puts Ring2 to sleep.
- Selects a random color from the wheel for the next iteration of Ring2's pattern.

If no button is pressed, it simply picks a random speed for the next iteration of the normal Ring2 Rainbow Cycle pattern.

```
// Ring 2 Completion Callback
void Ring2Complete()
{
    if (digitalRead(9) == LOW)  // Button #2 pressed
    {
        // Alternate color-wipe patterns with Ring1
        Ring1.Interval = 20;
        Ring2.Color1 = Ring2.Wheel(random(255));
        Ring2.Interval = 20000;
    }
    else  // Retrn to normal
    {
        Ring2.RainbowCycle(random(0,10));
    }
}
```

Stick Completion Callback

The Stick completon callback just slects a random color from the wheel for the next pass of the scanner pattern.

```
// Stick Completion Callback
void StickComplete()
{
    // Random color change for next scan
    Stick.Color1 = Stick.Wheel(random(255));
}
```

Put it all together...

The code below includes the complete NeoPattern class, along with some 'test-drive' code in the form of an Arduino Sketch. Copy and paste this code into the IDE and wire your pixels as shown in the wiring diagram.

```
#include &lt;Adafruit_NeoPixel.h&gt;

// Pattern types supported:
enum  pattern { NONE, RAINBOW_CYCLE, THEATER_CHASE, COLOR_WIPE, SCANNER, FADE };
// Patern directions supported:
enum  direction { FORWARD, REVERSE };
```

```
// NeoPattern Class - derived from the Adafruit_NeoPixel class
class NeoPatterns : public Adafruit_NeoPixel
{
    public:

    // Member Variables:
    pattern  ActivePattern;  // which pattern is running
    direction Direction;     // direction to run the pattern

    unsigned long Interval;   // milliseconds between updates
    unsigned long lastUpdate; // last update of position

    uint32_t Color1, Color2;  // What colors are in use
    uint16_t TotalSteps;  // total number of steps in the pattern
    uint16_t Index;  // current step within the pattern

    void (*OnComplete)();  // Callback on completion of pattern

    // Constructor - calls base-class constructor to initialize strip
    NeoPatterns(uint16_t pixels, uint8_t pin, uint8_t type, void (*callback)())
    :Adafruit_NeoPixel(pixels, pin, type)
    {
        OnComplete = callback;
    }

    // Update the pattern
    void Update()
    {
        if((millis() - lastUpdate) &gt; Interval) // time to update
        {
            lastUpdate = millis();
            switch(ActivePattern)
            {
                case RAINBOW_CYCLE:
                    RainbowCycleUpdate();
                    break;
                case THEATER_CHASE:
                    TheaterChaseUpdate();
                    break;
                case COLOR_WIPE:
                    ColorWipeUpdate();
                    break;
                case SCANNER:
                    ScannerUpdate();
                    break;
                case FADE:
                    FadeUpdate();
                    break;
                default:
                    break;
            }
        }
    }

    // Increment the Index and reset at the end
    void Increment()
    {
        if (Direction == FORWARD)
        {
           Index++;
           if (Index &gt;= TotalSteps)
            {
                Index = 0;
                if (OnComplete != NULL)
                {
                    OnComplete(); // call the comlpetion callback
                }
            }
        }
```

```
        else // Direction == REVERSE
        {
            --Index;
            if (Index &lt;= 0)
            {
                Index = TotalSteps-1;
                if (OnComplete != NULL)
                {
                    OnComplete(); // call the comlpetion callback
                }
            }
        }
    }

    // Reverse pattern direction
    void Reverse()
    {
        if (Direction == FORWARD)
        {
            Direction = REVERSE;
            Index = TotalSteps-1;
        }
        else
        {
            Direction = FORWARD;
            Index = 0;
        }
    }

    // Initialize for a RainbowCycle
    void RainbowCycle(uint8_t interval, direction dir = FORWARD)
    {
        ActivePattern = RAINBOW_CYCLE;
        Interval = interval;
        TotalSteps = 255;
        Index = 0;
        Direction = dir;
    }

    // Update the Rainbow Cycle Pattern
    void RainbowCycleUpdate()
    {
        for(int i=0; i&lt; numPixels(); i++)
        {
            setPixelColor(i, Wheel(((i * 256 / numPixels()) + Index) &amp;amp; 255));
        }
        show();
        Increment();
    }

    // Initialize for a Theater Chase
    void TheaterChase(uint32_t color1, uint32_t color2, uint8_t interval, direction
dir = FORWARD)
    {
        ActivePattern = THEATER_CHASE;
        Interval = interval;
        TotalSteps = numPixels();
        Color1 = color1;
        Color2 = color2;
        Index = 0;
        Direction = dir;
   }

    // Update the Theater Chase Pattern
    void TheaterChaseUpdate()
    {
        for(int i=0; i&lt; numPixels(); i++)
        {
            if ((i + Index) % 3 == 0)
```

```
            {
                setPixelColor(i, Color1);
            }
            else
            {
                setPixelColor(i, Color2);
            }
        }
        show();
        Increment();
    }

    // Initialize for a ColorWipe
    void ColorWipe(uint32_t color, uint8_t interval, direction dir = FORWARD)
    {
        ActivePattern = COLOR_WIPE;
        Interval = interval;
        TotalSteps = numPixels();
        Color1 = color;
        Index = 0;
        Direction = dir;
    }

    // Update the Color Wipe Pattern
    void ColorWipeUpdate()
    {
        setPixelColor(Index, Color1);
        show();
        Increment();
    }

    // Initialize for a SCANNNER
    void Scanner(uint32_t color1, uint8_t interval)
    {
        ActivePattern = SCANNER;
        Interval = interval;
        TotalSteps = (numPixels() - 1) * 2;
        Color1 = color1;
        Index = 0;
    }

    // Update the Scanner Pattern
    void ScannerUpdate()
    {
        for (int i = 0; i &lt; numPixels(); i++)
        {
            if (i == Index)  // Scan Pixel to the right
            {
                 setPixelColor(i, Color1);
            }
            else if (i == TotalSteps - Index) // Scan Pixel to the left
            {
                 setPixelColor(i, Color1);
            }
            else // Fading tail
            {
                 setPixelColor(i, DimColor(getPixelColor(i)));
            }
        }
        show();
        Increment();
    }

    // Initialize for a Fade
    void Fade(uint32_t color1, uint32_t color2, uint16_t steps, uint8_t interval,
direction dir = FORWARD)
    {
        ActivePattern = FADE;
        Interval = interval;
```

```
        TotalSteps = steps;
        Color1 = color1;
        Color2 = color2;
        Index = 0;
        Direction = dir;
    }

    // Update the Fade Pattern
    void FadeUpdate()
    {
        // Calculate linear interpolation between Color1 and Color2
        // Optimise order of operations to minimize truncation error
        uint8_t red = ((Red(Color1) * (TotalSteps - Index)) + (Red(Color2) *
Index)) / TotalSteps;
        uint8_t green = ((Green(Color1) * (TotalSteps - Index)) + (Green(Color2) *
Index)) / TotalSteps;
        uint8_t blue = ((Blue(Color1) * (TotalSteps - Index)) + (Blue(Color2) *
Index)) / TotalSteps;

        ColorSet(Color(red, green, blue));
        show();
        Increment();
    }

    // Calculate 50% dimmed version of a color (used by ScannerUpdate)
    uint32_t DimColor(uint32_t color)
    {
        // Shift R, G and B components one bit to the right
        uint32_t dimColor = Color(Red(color) &gt;&gt; 1, Green(color) &gt;&gt; 1,
Blue(color) &gt;&gt; 1);
        return dimColor;
    }

    // Set all pixels to a color (synchronously)
    void ColorSet(uint32_t color)
    {
        for (int i = 0; i &lt; numPixels(); i++)
        {
            setPixelColor(i, color);
        }
        show();
    }

    // Returns the Red component of a 32-bit color
    uint8_t Red(uint32_t color)
    {
        return (color &gt;&gt; 16) &amp; 0xFF;
    }

    // Returns the Green component of a 32-bit color
    uint8_t Green(uint32_t color)
    {
        return (color &gt;&gt; 8) &amp; 0xFF;
    }

    // Returns the Blue component of a 32-bit color
    uint8_t Blue(uint32_t color)
    {
        return color &amp; 0xFF;
    }

    // Input a value 0 to 255 to get a color value.
    // The colours are a transition r - g - b - back to r.
    uint32_t Wheel(byte WheelPos)
    {
        WheelPos = 255 - WheelPos;
        if(WheelPos &lt; 85)
        {
            return Color(255 - WheelPos * 3, 0, WheelPos * 3);
```

```
        }
        else if(WheelPos &lt; 170)
        {
            WheelPos -= 85;
            return Color(0, WheelPos * 3, 255 - WheelPos * 3);
        }
        else
        {
            WheelPos -= 170;
            return Color(WheelPos * 3, 255 - WheelPos * 3, 0);
        }
    }
};

void Ring1Complete();
void Ring2Complete();
void StickComplete();

// Define some NeoPatterns for the two rings and the stick
//  as well as some completion routines
NeoPatterns Ring1(24, 5, NEO_GRB + NEO_KHZ800, &amp;Ring1Complete);
NeoPatterns Ring2(16, 6, NEO_GRB + NEO_KHZ800, &amp;Ring2Complete);
NeoPatterns Stick(16, 7, NEO_GRB + NEO_KHZ800, &amp;StickComplete);

// Initialize everything and prepare to start
void setup()
{
  Serial.begin(115200);

   pinMode(8, INPUT_PULLUP);
   pinMode(9, INPUT_PULLUP);

    // Initialize all the pixelStrips
    Ring1.begin();
    Ring2.begin();
    Stick.begin();

    // Kick off a pattern
    Ring1.TheaterChase(Ring1.Color(255,255,0), Ring1.Color(0,0,50), 100);
    Ring2.RainbowCycle(3);
    Ring2.Color1 = Ring1.Color1;
    Stick.Scanner(Ring1.Color(255,0,0), 55);
}

// Main loop
void loop()
{
    // Update the rings.
    Ring1.Update();
    Ring2.Update();

    // Switch patterns on a button press:
    if (digitalRead(8) == LOW) // Button #1 pressed
    {
        // Switch Ring1 to FADE pattern
        Ring1.ActivePattern = FADE;
        Ring1.Interval = 20;
        // Speed up the rainbow on Ring2
        Ring2.Interval = 0;
        // Set stick to all red
        Stick.ColorSet(Stick.Color(255, 0, 0));
    }
    else if (digitalRead(9) == LOW) // Button #2 pressed
    {
        // Switch to alternating color wipes on Rings1 and 2
        Ring1.ActivePattern = COLOR_WIPE;
        Ring2.ActivePattern = COLOR_WIPE;
        Ring2.TotalSteps = Ring2.numPixels();
        // And update tbe stick
```

```
        Stick.Update();
    }
    else // Back to normal operation
    {
        // Restore all pattern parameters to normal values
        Ring1.ActivePattern = THEATER_CHASE;
        Ring1.Interval = 100;
        Ring2.ActivePattern = RAINBOW_CYCLE;
        Ring2.TotalSteps = 255;
        Ring2.Interval = min(10, Ring2.Interval);
        // And update tbe stick
        Stick.Update();
    }
}

//------------------------------------------------------------
//Completion Routines - get called on completion of a pattern
//------------------------------------------------------------

// Ring1 Completion Callback
void Ring1Complete()
{
    if (digitalRead(9) == LOW)  // Button #2 pressed
    {
        // Alternate color-wipe patterns with Ring2
        Ring2.Interval = 40;
        Ring1.Color1 = Ring1.Wheel(random(255));
        Ring1.Interval = 20000;
    }
    else  // Retrn to normal
    {
      Ring1.Reverse();
    }
}

// Ring 2 Completion Callback
void Ring2Complete()
{
    if (digitalRead(9) == LOW)  // Button #2 pressed
    {
        // Alternate color-wipe patterns with Ring1
        Ring1.Interval = 20;
        Ring2.Color1 = Ring2.Wheel(random(255));
        Ring2.Interval = 20000;
    }
    else  // Retrn to normal
    {
        Ring2.RainbowCycle(random(0,10));
    }
}

// Stick Completion Callback
void StickComplete()
{
    // Random color change for next scan
    Stick.Color1 = Stick.Wheel(random(255));
}
```

Take it for a spin!