AnimatedLEDStrip Library

A Library for Easy LED Strip Animations

Created by Max Narvaez, Will Roberts, Ziqi Wei and Chan Jin Park

Overview

This Arduino library was designed to simplify the usage of animations on LED strips so that an animation can be run with only one function call rather than spending time writing and debugging the logic behind an animation. Though this library only supplies 12 animations, the range of uses of them are endless. The user can also create their own animations, which we would love to hear about and potentially include in future versions of the library.

The base of this library is the FastLED library, which gives the user control over an LED strip. The aspect in which FastLED was lacking was predefined animations. Though they gave examples of potential animations (some of which we modified and used here), the user would still be required to copy the animation code into their program's code and then modify it to work with their setup.

Because of time limitations, this library is only configured to work with NeoPixel LEDs, though a user can modify the constructors of the LEDStrip and AnimatedLEDStrip classes to use another LED chipset.

The arduino can be controlled via serial communication when the included `AnimatedLEDStrip_Serial_Arduino.ino` file is loaded on the Arduino. Details on how to use this are located in the Controlling Your LED Strip with Arduino Serial Communication section.

This library was also created with FIRST Robotics Competition teams in mind. During the stressful 6 ½ week build season, only the largest and most experienced teams have the time to think about adding LEDs to their robot before competition. This library hopes to help the smaller teams and teams with less coding experience by simplifying the coding aspect of adding LEDs to a robot. If the RoboRIO is connected to an Arduino via USB, the RoboRIO can send serial communications to the Arduino using its SerialPort class.

Using this Manual

Classes and Structs

Classes and Structs will show how to construct an instance, any information about compatibility with other classes or structs, and a list of default values (if applicable).

Methods (and Animations)

Methods (and Animations) have a:

Name

`method prototype`

and then a description.

The method that actually performs the animation is called the main *animation* method. Methods with the same name but parameters other than the main method are called overload *animation* methods. These overload methods change the parameters so that they are compatible with the main *animation* method, such as changing three int parameters into a ColorContainer before sending the ColorContainer to the main *animation* method.

The main animation method prototype is listed first, any overloads are listed after in a smaller font.

Controlling Your LED Strip

AnimatedLEDStrip Class

AnimatedLEDStrip contains the animation methods in the library. It is a child class of the LEDStrip class[1].

Construction

To construct an AnimatedLEDStrip, call the `AnimatedLEDStrip(int numLEDs, int pin)` constructor with:

- The number of LEDs in the strip, and
- The pin the strip is connected to

When constructed, the AnimatedLEDStrip constructs a LEDStrip instance and creates a new shuffleArray.

Variables

`shuffleArray` - A pointer to an array of ints. `shuffleArray` holds a list of indices of pixels, up to the size of the LED strip. This is used in conjunction with the `shuffle()` method to give the AnimatedLEDStrip a way to select pixels in a random order without selecting the same pixel more than once.

Animations

There are 12 animations that can be used with an AnimatedLEDStrip:

- Alternate
- Fade Pixel Red
- Fade Pixel Green
- Fade Pixel Blue
- Fade Pixel All
- Multi-Pixel Run
- Pixel Run
- Pixel Run with Trail
- Smooth Chase
- Sparkle
- Sparkle to Color
- Wipe

Helper Methods

AnimatedLEDStrip has 1 helper method:

- Shuffle

[1] See LEDStrip Class

LEDStrip Class

LEDStrip contains the methods used to set the colors of pixels in the LED strip. As a child class of the CFastLED class and a parent class to the AnimatedLEDStrip[2] class, LEDStrip is the class connecting the AnimatedLEDStrip library to the FastLED library.

Construction

LEDStrip can be constructed independently from the AnimatedLEDStrip, leaving you with a LED strip that can only be set to static colors.

To construct a LEDStrip, call the `LEDStrip( int numLEDs, int pinIn )` constructor with:

- The number of LEDs in the strip, and
- The pin the strip is connected to

When constructed, the LEDStrip adds a new controller for a set of NeoPixel LEDs[3]. Because FastLED wants the pin number to be defined prior to compile time, the file Pin_Defs.h defines pin numbers that are then used for the pin parameter. The LEDStrip constructor uses if-else-if statements to determine which pre-defined pin variable to use based on the pin number sent to the constructor.

`numLEDs` and `pinIn` are stored in `pixelCount` and `pin`, respectively. Memory is allocated for `ledArray`.

LEDStrips can also be constructed by copying or assignment.

Variables

`pixelCount` - The number of LEDs in the strip represented by the LEDStrip instance.

`ledArray` - An array of CRGB[4] structs. This array holds the color of each pixel, with index `0` representing the first LED in the strip (closest to source of signal).

`pin` - The pin the LED strip is connected to.

Methods

LEDStrip has 10 set methods, 6 get methods and an index operator method (giving direct access to the `ledArray` array)

- Set Methods
    - Set Pixel Color
    - Set Pixel Red
    - Set Pixel Green

[2] See AnimatedLEDStrip Class
[3] https://www.adafruit.com/category/168
[4] See CRGB Struct

- Set Methods, cont.
    - Set Pixel Blue
    - Set Strip Color
    - Set Strip Red
    - Set Strip Green
    - Set Strip Blue
    - Fill LEDs from Palette
    - Fill LEDs with Gradient
- Get Methods
    - Get `pixelCount`
    - Get Pixel Color
    - Get Pixel Red
    - Get Pixel Green
    - Get Pixel Blue
    - Get `ledArray`

Set Pixel Color Method

```
void LEDStrip::setPixelColor( int pixel, ColorContainer colorValues )
```

```
void LEDStrip::setPixelColor( int pixel, int rIn, int gIn, int bIn )
```

The set pixel color method is used to set the color of a single pixel in the LED strip (specified by `pixel`) to `colorValues`. The main set pixel color method uses a ColorContainer to set the intensities of the red, green and blue in the pixel, while the overload set pixel color method takes in three separate int values and then sends those to the main set pixel color method in a ColorContainer.

Set Pixel Red, Set Pixel Green, Set Pixel Blue Methods

```
void LEDStrip::setPixelRed( int pixel, int rIn )
```

```
void LEDStrip::setPixelGreen( int pixel, int gIn )
```

```
void LEDStrip::setPixelBlue( int pixel, int bIn )
```

The set pixel red, set pixel green and set pixel blue methods set the red, green and blue intensities, respectively, of a single pixel in the LED strip (specified by pixel). rIn, gIn and bIn can be any value from 0 through 255.

Set Strip Color Method

```
void LEDStrip::setStripColor( ColorContainer colorValues )

void LEDStrip::setStripColor( int rIn, int gIn, int bIn )
```

The set strip color method is used to set the color of all pixels in the LED strip to `colorValues`. The main set strip color method uses a ColorContainer to set the intensities of the red, green and blue in the pixels, while the overload set strip color method takes in three separate int values and then sends those to the main set strip color method in a ColorContainer.

Fill LEDs from Palette Method

```
template<class paletteType> void LEDStrip::fillLEDsFromPalette(
      const paletteType & palette, uint8_t startIndex, TBlendType blend,
      uint8_t brightness )
```

The fill LEDs from palette methods are based on the `FillLEDsFromPaletteColors()` method in the ColorPalette example for the FastLED library. By using the palette[5] feature of the FastLED library, we can fill the LEDs with colors that fade between each other over multiple pixels[6]. There are 7 overload fill LEDs from palette methods, one for each type of palette.

`paletteType` can be any of the following[7]:

- `CRGBPalette16`
- `CRGBPalette32`
- `CRGBPalette256`
- `CHSVPalette16`
- `CHSVPalette32`
- `CHSVPalette256`
- `TProgmemRGBPalette16`[8]
- `TProgmemRGBPalette32`
- `TProgmemHSVPalette16`
- `TProgmemHSVPalette32`

`startIndex` is used to specify where in the palette the color for index `0` is located.

`blend` is used to specify if the palette should be blended[9]. The default is LINEARBLEND.

`brightness` is an optional parameter and will be set to 255 (full) if excluded.

---

[5] See Color Palettes
[6] See Color Palettes - Using Colors Stored in a Palette
[7] See Color Palettes
[8] Because `TProgmem` palettes are just `uint32_t` arrays, only one function is needed to handle all of them
[9] See Color Palettes - Using Colors Stored in a Palette

Fill LEDs with Gradient Method

```
void LEDStrip::fillLEDsWithGradient( ColorContainer colorValues1,
      ColorContainer colorValues2 )

void LEDStrip::fillLEDsWithGradient( ColorContainer colorValues1,
      ColorContainer colorValues2, ColorContainer colorValues3 )

void LEDStrip::fillLEDsWithGradient( ColorContainer colorValues1,
      ColorContainer colorValues2, ColorContainer colorValues3,
      ColorContainer colorValues4 )
```

The fill LEDs with gradient method utilizes the FastLED function `fill_gradient_RGB()`. This method can take two, three or four colors. With two parameters, it will fade from the first color to the second color over the length of the strip. With three parameters, it will fade from the first color to the second over the first half of the strip and from the second color to the third over the second half of the strip. With four parameters, it will fade from the first color to the second over the first third of the strip, from the second color to the third over the second third of the strip and from the third color to the fourth over the last third of the strip.

Get `pixelCount` Method

`int LEDStrip::getPixelCount()`

The get `pixelCount` method returns the number of LEDs in the strip, which was stored in `pixelCount` at construction.

Get Pixel Color Method

`ColorContainer LEDStrip::getPixelColor( int pixelIn )`

The get pixel color method returns a ColorContainer holding the color of one of the pixels in the LED strip (specified by `pixelIn`).

Get Pixel Red, Get Pixel Green, Get Pixel Blue Methods

`int LEDStrip::getPixelRed( int pixelIn )`

`int LEDStrip::getPixelGreen( int pixelIn )`

`int LEDStrip::getPixelBlue( int pixelIn )`

The get pixel red, get pixel green and get pixel blue methods return the red, green, blue intensity (respectively) of one of the pixels in the LED strip (specified by `pixelIn`) as an integer.

Get `ledArray` Method

`CRGB * LEDStrip::getLEDArray()`

The get `ledArray` method returns a pointer to the CRGB[10] array that holds the colors for each pixel.

[10] See CRGB Struct

Alternate Animation

```
void AnimatedLEDStrip::alternate( ColorContainer colorValues1, ColorContainer
      colorValues2, int delayTime )

void AnimatedLEDStrip::alternate( int r1In, int g1In, int b1In, int r2In, int g2In,
      int b2In, int delayTime )
```

The `alternate()` animation switches between two specified colors at the specified rate.

- The whole AnimatedLEDStrip is set to `colorValues1`
- The function delays `delayTime` milliseconds
- The whole AnimatedLEDStrip is set to `colorValues2`
- The function delays `delayTime` milliseconds
- The function returns

Note that the strip will remain set to `colorValues2` after the animation is complete.

To create the illusion that the strip is alternating back and forth between the two colors, call this function repeatedly.

Fade Pixel Red, Fade Pixel Green and Fade Pixel Blue Animations

```
fadePixelRed( int pixel, int startIntensity, int endIntensity,
      bool revertAtCompletion = false )

fadePixelGreen( int pixel, int startIntensity, int endIntensity,
      bool revertAtCompletion = false )

fadePixelBlue( int pixel, int startIntensity, int endIntensity,
      bool revertAtCompletion = false )
```

The `fadePixelRed()`, `fadePixelGreen()` and `fadePixelBlue()` functions fade the LED's red/green/blue value from `startIntensity` to `endIntensity`. They run as fast as possible, only limited by the speed of the processor, but longer fades (e.g. 0 to 255) will take longer to run. As the function runs, the intensity is increased or decreased in each run of a for loop until the intensity reaches `endIntensity`. If `startIntensity` and `endIntensity` are equal, the function will return without any change to the LEDs.

If `revertAtCompletion` is set to true, then the pixel will revert back to the color it held before the fade animation was called.

Fade Pixel All Animation

```
void fadePixelAll( int pixel, int startRedIntensity, int startGreenIntensity,
      int startBlueIntensity, int endRedIntensity, int endGreenIntensity,
      int endBlueIntensity, bool revertAtCompletion = false )
```

The `fadePixelAll()` method is similar to the `fadePixelRed()`, `fadePixelGreen()` and `fadePixelBlue()` methods, but fades all three values simultaneously in a while loop. The three values increment/decrement one intensity at a time; once they reach their end intensity, they stop fading. Thus, it is possible, for example, for the red value to reach `endRedIntensity` while green and blue have not reached their respective end intensities. Green and blue will continue fading while red stays at `endRedIntensity`. Once all three values have reached their respective end intensities, the while loop will end.

If `revertAtCompletion` is set to true, then the pixel will revert back to the color it held before the fade animation was called.

Multi-Pixel Run Animation

```
void AnimatedLEDStrip::multiPixelRun( int spacing, direction chaseDirection,
      ColorContainer colorValues, ColorContainer altColorValues = CRGB::Black )

void AnimatedLEDStrip::multiPixelRun( int spacing, direction chaseDirection,
      int rIn1, int gIn1, int bIn1, int rIn2 = 0, int gIn2 = 0, int bIn2 = 0 )
```

The main `multiPixelRun` function runs an animation where:

- The LEDs at locations `n, spacing + n, (spacing * 2) + n`, etc. are set to colorValues
- The LEDs are updated
- The controller pauses for 1/20th of a second
- The LEDs at the locations specified above are set to altColorValues
- n is incremented/decremented

If `chaseDirection` is set to forward, this runs from `n = 0` while `n < spacing`.
If `chaseDirection` is set to backward, this runs from `n = spacing - 1` while `n >= 0`.

To create the illusion that the LEDs are following each other through the full length of the strip, this function should be called repeatedly.

The overload `multiPixelRun` function creates a ColorContainer out of `rIn1`, `gIn1` and `bIn1`, another ColorContainer out of `rIn2`, `gIn2` and `bIn2` (defaults to black), then calls the main `multiPixelRun` method with `spacing`, `chaseDirection` and the new ColorContainers as parameters.

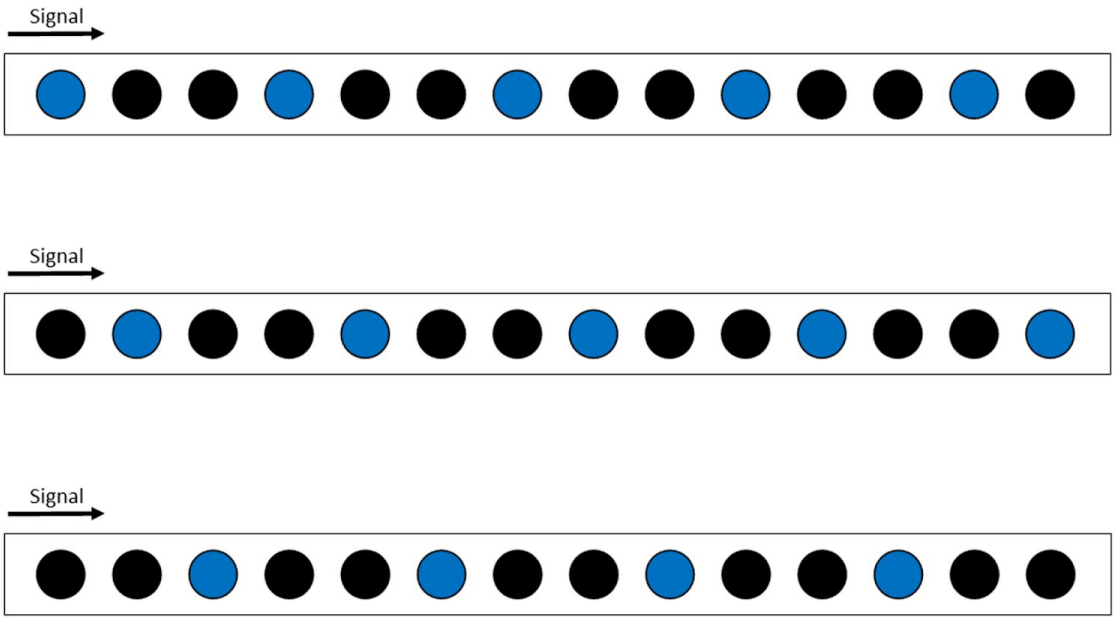

Diagram showing a call to `multiPixelRun(3, forward, CRGB::Blue)`

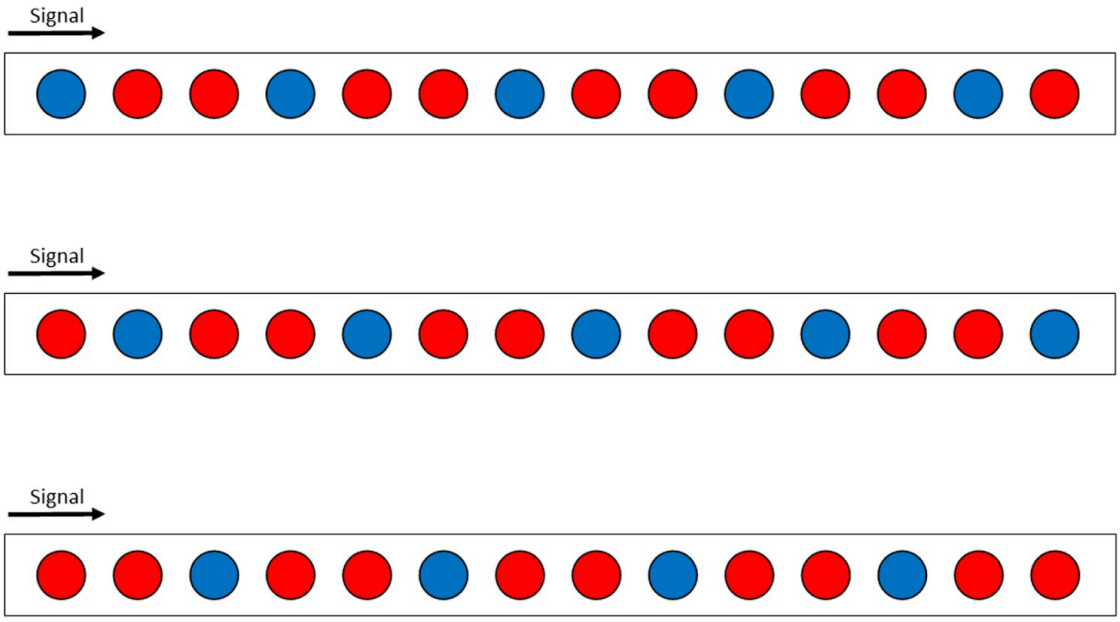

Diagram showing an example call to `multiPixelRun(3, forward, CRGB::Blue, CRGB::Red)`

Pixel Run Animation

```
void AnimatedLEDStrip::pixelRun( direction movementDirection,
      ColorContainer colorValues, ColorContainer altColorValues = CRGB::Black )

void AnimatedLEDStrip::pixelRun( direction movementDirection, int r1In, int g1In,
      int b1In, int r2In = 0, int g2In = 0,  int b2In = 0 )
```

The pixel run animation shows a pixel 'running' from one end of the strip to another. A pixel is set to colorValues while the rest of the pixels are set to altColorValues, then the next pixel is set to colorValues (and the first pixel is changed back to altColorValues), etc.

`movementDirection` controls which 'direction' the animation appears to be moving in[11].

`colorValues` is the color of the 'running' pixel.

`altColorValues` is the color of the non-'running' pixels. This defaults to black.

Pixel Run with Trail Animation

```
void AnimatedLEDStrip::pixelRunWithTrail( direction movementDirection,
      ColorContainer colorValues, ColorContainer altColorValues = CRGB::Black )

void AnimatedLEDStrip::pixelRunWithTrail( direction movementDirection, int r1In,
      int g1In, int b1In, int r2In = 0, int g2In = 0,  int b2In = 0 )
```

The pixel run with trail animation is similar to the pixel run animation but the pixel leaves a 'trail' behind it. By using the `blend()` function in the FastLED library, the pixels are returned back to `altColorValues` over approximately 20 cycles rather than returning to `altColorValues` immediately.

`movementDirection` controls which 'direction' the animation appears to be moving in[12].

`colorValues` is the color of the 'running' pixel.

`altColorValues` is the color of the non-'running' pixels. This defaults to black. Pixels that recently were the 'running' pixel will be somewhere between `colorValues` and `altColorValues`.

[11] See Direction Enum
[12] See Direction Enum

Smooth Chase Animation

```
template<class paletteType> void AnimatedLEDStrip::smoothChase(
      const paletteType & palette, direction movementDirection,
      uint8_t brightness = 255)
```

The `smoothChase()` function has 7 overloaded functions (same name and parameters as the template version) that pass along their parameters to the template version of the function.

`paletteType` can be any of the following[13]:

- `CRGBPalette16`
- `CRGBPalette32`
- `CRGBPalette256`
- `CHSVPalette16`
- `CHSVPalette32`
- `CHSVPalette256`
- `TProgmemRGBPalette16`[14]
- `TProgmemRGBPalette32`
- `TProgmemHSVPalette16`
- `TProgmemHSVPalette32`

`movementDirection` controls which 'direction' the animation appears to be moving in[15].

`brightness` is an optional parameter and will be set to 255 (full) if excluded.

To create the illusion that the LEDs are following each other through the full length of the strip, this function should be called repeatedly.

[13] See Color Palettes
[14] Because `TProgmem` palettes are just `uint32_t` arrays, only one function is needed to handle all of them
[15] See Direction Enum

Sparkle Animation

void AnimatedLEDStrip::sparkle(ColorContainer sparkleColor)

void AnimatedLEDStrip::sparkle(int rIn, int gIn, int bIn)

The sparkle animation changes the color of one pixel to the specified color before immediately returning that pixel to its original color. This repeats until all pixels have been ‘sparkled’. By utilizing the shuffleArray and the shuffle method, we can ensure that every pixel is sparkled only once.

Sparkle to Color Animation

void AnimatedLEDStrip::sparkleToColor(ColorContainer destinationColor)

void AnimatedLEDStrip::sparkleToColor(int rIn, int gIn, int bIn)

The sparkle to color animation is similar to the sparkle animation, but the ‘sparkled’ pixels retain the `destinationColor` throughout and after the animation. It is similar to how a wipe animation works in relation to a pixel run animation. It is essentially a cross between `setPixelColor()` and `sparkle()`. Like with `wipe()`, this function should not be called repeatedly with the same color without changing pixel colors between calls, as only the first call will cause a visual change in the LEDs.

Wipe Animation

```
void AnimatedLEDStrip::wipe( ColorContainer colorValues,
      direction wipeDirection)
```

```
void AnimatedLEDStrip::wipe( int rIn, int gIn, int bIn, direction wipeDirection)
```

The wipe animation changes the strip from one color to another like `setStripColor()`, but updates the LEDs after every change. This creates an animation that is a cross between `setStripColor()` and `pixelRun()`. All pixels in the strip will be set to colorValues by the completion of the function. This function should not be called repeatedly with the same color without changing pixel colors between calls, as only the first call will cause a visual change in the LEDs.

Direction Enumeration

```
enum direction { forward, backward };
```

The direction enum is used when an animation appears to ‘move’ in a certain direction.

`forward`	‘Moves’ with the flow of the signal (away from the end connected to the controller)
`backward`	‘Moves’ against the flow of the signal (towards the end connected to the controller)

Fade Direction Enumeration

```
enum fadeDirection { fadeUp, fadeDown };
```

The fade direction enum is an internally used enum for use in animations that fade from one color to another. It is set based on if the intensity needs to increase (`fadeUp`) to reach the destination color, or if the intensity needs to decrease (`fadeDown`) to reach the destination color.

Controlling Your LED Strip with Arduino Serial Communication

Using Serial Communication to Control Your LED Strip

The AnimatedLEDStrip library has code for your Arduino that allows you to control your LED strip with commands sent over a serial protocol. This is in the `AnimatedLEDStrip_Serial_Arduino.ino` file.

Overview

The Arduino gets commands from the serial bus using the Messenger library[16]. The code has variables for every parameter in every animation. If a parameter is not used in the current animation, it is set to `NULL`.

Whenever a serial communication is received, the function `serialEvent()` will be called once `loop()` returns to its beginning. This will then call `messageReady()` which will then call the appropriate method to save the values from the message to their respective variables. Once that is complete, `loop()` will resume.

In `loop()`, `switch` statements are used to determine which animation method needs to be called. This allows us to repeat animations endlessly until another communication is received.

NOTE: Currently the Arduino code only works reliably with LED strips of 50 pixels or less. Any more and some animations may not run or may run incorrectly. We guess that this may be an issue with memory on the Arduino. Hopefully a future version will work with more pixels.

Setup

First change `NUMLEDS` and `PIN` to the proper values for your project. Load the code onto the Arduino and connect the Arduino to your computer (or other device). If using the Arduino Serial Monitor, change the line ending type to “Carriage Return” or “Both NL & CR”.

Running an Animation

```
(S | A) <Animation> (Parameters...)
```

Animations can be run Static or Animated. This is determined by the `S` or `A` at the beginning of the message. Note that some animations (such as `COL1`, `COL2`, `COL3`, `COL4`, `STC`, `WIP`) will look the same when run Static or Animated. When an animation is run Animated, the animation method (determined by `currentAnimation`, among other variables) is called repeatedly with relevant variables as parameters. When an animation is run as Static, all variables are cleared at completion, so the strip shows the last values it received.

Next in the message is the animation’s abbreviation. The abbreviations are listed under Animation Enumeration[17]. Note that all the Fade Pixel animations[18] use the same abbreviation

[16] Yes, we know it is deprecated, but it fulfilled our needs
[17] See Animation Enumeration

(the type of fade is determined next) and that the Pixel Run[19] and Pixel Run with Trail[20] animations use the same abbreviation (the existence of a trail is determined later in the message).

Next come parameters for the animation, such as colors, direction, etc.

Last comes a carriage return (`0x0D` or `\r`). This is required so the Messenger library knows when the message is complete.

Animation Message Parameters[21]

COL1 - Set Strip Color

`(S | A) COL1 <Color>\r`

`COL1` takes one color parameter in the format of a hexadecimal WITHOUT the 0x at the beginning. For example:

```
S COL1 FF0000\r
```

will run

```
setStripColor(ColorContainer(FF0000));
```

COL2 - Fill LEDs with Gradient

`(S | A) COL2 <Color> <Color>\r`

`COL2` takes two color parameters in the format of hexadecimals WITHOUT the 0x at the beginning. For example:

```
S COL2 FF0000 00FF00\r
```

will run

```
fillLEDsWithGradient(ColorContainer(FF0000), \
ColorContainer(00FF00));
```

[18] See Fade Pixel Red, Fade Pixel Green and Fade Pixel Blue Animations and Fade Pixel All Animation
[19] See Pixel Run Animation
[20] See Pixel Run with Trail Animation
[21] See Animation Enumeration for list of abbreviations with their corresponding functions

COL3 - Fill LEDs with Gradient

(S | A) COL1 <Color> <Color> <Color>\r

COL3 takes three color parameters in the format of hexadecimals WITHOUT the 0x at the beginning. For example:

```
S COL3 FF0000 00FF00 0000FF\r
```

will run

```
fillLEDsWithGradient(ColorContainer(FF0000), \
ColorContainer(00FF00), ColorContainer(0000FF));
```

COL4 - Fill LEDs with Gradient

(S | A) COL1 <Color> <Color> <Color> <Color>\r

COL4 takes four color parameters in the format of hexadecimals WITHOUT the 0x at the beginning. For example:

```
S COL4 FF0000 00FF00 0000FF FFFFFF\r
```

will run

```
fillLEDsWithGradient(ColorContainer(FF0000), \
ColorContainer(00FF00), ColorContainer(0000FF), \
ColorContainer(FFFFFF));
```

ALT - Alternate

(S | A) ALT <Color> <Color> <Duration>\r

ALT takes two color parameters in the format of a hexadecimal WITHOUT the 0x at the beginning and a duration in the form of an int. For example:

```
A ALT FF0000 00FF00 500\r
```

will run

```
alternate(ColorContainer(FF0000), ColorContainer(00FF00), 500);
```

FDP - Fade Pixel

(S | A) FDP (RED | GRN | BLU) <Pixel> <Startintensity> <Endintensity> (R | N)\r

(S | A) FDP ALL <Pixel> <Startrintensity> <Startgintensity> <Startbintensity> <Endrintensity> <Endgintensity> <Endbintensity> (R | N)

`FDP RED`, `FDP GRN` and `FDP BLU` take three ints (`Pixel`, `Startintensity` and `Endintensity`) and a R or N. R and N are used to denote if the animation should revert at completion (R = Revert; N = No Revert).

```
A FDP RED 5 50 255 R\r
```

will run

```
fadePixelRed(5, 50, 255, true);
```

`FDP All` takes 7 ints (`Pixel`, `Startrintensity`, `Startgintensity`, `Startbintensity`, `Endrintensity`, `Endgintensity`, `Endbintensity`) and a R or N. R and N are used to denote if the animation should revert at completion (R = Revert; N = No Revert). For example:

```
A FDP ALL 5 50 255 120 80 102 50 N\r
```

will run

```
fadePixelAll(5, 50, 255, 120, 80, 102, 50, false);
```

MPR - Multi-Pixel Run

(S | A) MPR <Spacing> <Color> <Color> (F | B)\r

`MPR` takes an int (`Spacing`), two color parameters in the format of a hexadecimal WITHOUT the 0x at the beginning, and a F or B. F and B are used to denote if the animation should run ‘forwards’ (F) or ‘backwards’ (B). For example:

```
A MPR 5 FF0000 00FF00 B\r
```

will run

```
multiPixelRun(5, backward, ColorContainer(FF0000), \
ColorContainer(00FF00));
```

PXR - Pixel Run & Pixel Run with Trail

(S | A) PXR <Color> <Color> (F | B) (T | N)\r

PXR takes two color parameters in the format of a hexadecimal WITHOUT the 0x at the beginning, a F or B and a T or N. F and B are used to denote if the animation should run 'forwards' (F) or 'backwards' (B). T and N are used to denote if the animation should have a 'trail' or not (T = `pixelRunWithTrail()`; N = `pixelRun()`) For example:

```
A PXR FF0000 00FF00 B N\r
```

will run

```
pixelRun(backward, ColorContainer(FF0000), \
ColorContainer(00FF00));
```

and

```
A PXR FF0000 00FF00 F T\r
```

will run

```
pixelRunWithTrail(forward, ColorContainer(FF0000), \
ColorContainer(00FF00));
```

SCH - Smooth Chase

(S | A) SCH (CLC | FTC | HTC | LVC | OCC | PTC | RBC | RSC) (F | B) <Brightness>\r

SCH takes a palette abbreviation[22], a F or B and an int between `0` and `255` (`Brightness`). F and B are used to denote if the animation should run 'forwards' (F) or 'backwards' (B). For example:

```
A SCH RBC B 255\r
```

will run

```
smoothChase(RainbowColors_p, backward, 255);
```

[22] See Smooth Chase Palette Enumeration

SPK - Sparkle

(S | A) SPK <Color>\r

SPK takes a color parameter in the format of a hexadecimal WITHOUT the 0x at the beginning. For example:

```
A SPK FF0000\r
```

will run

```
sparkle(ColorContainer(FF0000));
```

STC - Sparkle to Color

(S | A) STC <Color>\r

STC takes a color parameter in the format of a hexadecimal WITHOUT the 0x at the beginning. For example:

```
A STC FF0000\r
```

will run

```
sparkleToColor(ColorContainer(FF0000));
```

WIP - Sparkle to Color

(S | A) WIP <Color> (F | B)\r

WIP takes a color parameter in the format of a hexadecimal WITHOUT the 0x at the beginning and a F or B. F and B are used to denote if the animation should run ‘forwards’ (F) or ‘backwards’ (B). For example:

```
A SPK FF0000\r
```

will run

```
sparkle(ColorContainer(FF0000));
```

Animation Enumeration

```
enum animation { COL1, COL2, COL3, COL4, ALT, FDP, PXR, SCH, SPK, STC, WIP }
      currentAnimation;
```

The animation enumeration is used in the `AnimatedLEDStrip_Serial_Arduino.ino` file. It is a list of all animations and is used by `currentAnimation` to keep track of what animation is currently running on the LED strip.

`COL1`	The LED strip shows a solid color (`setStripColor()`[23])
`COL2`	The LED strip shows a solid gradient between two colors (`fillLEDsWithGradient()`[24])
`COL3`	The LED strip shows a solid gradient between three colors (`fillLEDsWithGradient()`)
`COL4`	The LED strip shows a solid gradient between four colors (`fillLEDsWithGradient()`)
`ALT`	Alternate animation (`alternate()`[25])
`FDP`	Fade Pixel animation (`fadePixel*()`[26])
`PXR`	Pixel Run animation (`pixelRun()`[27] and `pixelRunWithTrail()`[28])
`SCH`	Smooth Chase animation (`smoothChase()`[29])
`SPK`	Sparkle animation (`sparkle()`[30])
`STC`	Sparkle to Color animation (`sparkleToColor()`[31])
`WIP`	Wipe animation (`wipe()`[32])

Fade Type Enumeration

```
enum fadeType { RED, GRN, BLU, ALL } currentFadeType;
```

The fade type enumeration is used in the `AnimatedLEDStrip_Serial_Arduino.ino` file. It is used by `currentFadeType` to keep track of which type of fade pixel animation is in use.

`RED`	Fade Pixel Red animation (`fadePixelRed()`[33])
`GRN`	Fade Pixel Green Animation (`fadePixelGreen()`[34])
`BLU`	Fade Pixel Blue Animation (`fadePixelBlue()`[35])
`ALL`	Fade Pixel All Animation (`fadePixelAll()`[36])

---

[23] See Set Strip Color Method
[24] See Fill LEDs with Gradient Method
[25] See Alternate Animation
[26] See Fade Pixel Red, Fade Pixel Green and Fade Pixel Blue Animations and Fade Pixel All Animation
[27] See Pixel Run Animation
[28] See Pixel Run with Trail Animation
[29] See Smooth Chase Animation
[30] See Sparkle Animation
[31] See Sparkle to Color Animation
[32] See Wipe Animation
[33] See Fade Pixel Red, Fade Pixel Green and Fade Pixel Blue Animations
[34] See Fade Pixel Red, Fade Pixel Green and Fade Pixel Blue Animations
[35] See Fade Pixel Red, Fade Pixel Green and Fade Pixel Blue Animations
[36] See Fade Pixel All Animation

Animation Direction Enumeration

```
enum animationDirection { FWD, BKW } currentAnimationDirection;
```

The animation direction enumeration is used in the `AnimatedLEDStrip_Serial_Arduino.ino` file. It is used by `currentAnimationDirection` to keep track of which direction the current animation should be 'running' in. It is very similar to the Direction Enumeration[37] in the library.

`FWD`	Forward
`BKW`	Backward

Animation Trail Enumeration

```
enum animationTrail { TRL, NTRL } currentAnimationTrail;
```

The animation trail enumeration is used in the `AnimatedLEDStrip_Serial_Arduino.ino` file. It is used by `currentAnimationTrail` to keep track of if a Pixel Run animation[38] has a trail.

`TRL`	Include a trail (`pixelRunWithTrail()`[39])
`NTRL`	Do not include a trail (`pixelRun()`[40])

Animation Revert Enumeration

```
enum animationRevert { REV, NREV } currentAnimationRevert;
```

The animation revert enumeration is used in the `AnimatedLEDStrip_Serial_Arduino.ino` file. It is used by `currentAnimationRevert` to keep track of if a Fade Pixel animation[41] will revert at completion.

`REV`	Revert
`NREV`	Do not revert

[37] See Direction Enumeration
[38] See Pixel Run Animation and Pixel Run with Trail Animation
[39] See Pixel Run with Trail Animation
[40] See Pixel Run Animation
[41] See Fade Pixel Red, Fade Pixel Green and Fade Pixel Blue Animations and Fade Pixel All Animation

Smooth Chase Palette Enumeration

```
enum smoothChasePalette { CLC, FTC, HTC, LVC, OCC, PTC, RBC, RSC }
      currentSmoothChasePalette;
```

The smooth chase palette enumeration is used in the `AnimatedLEDStrip_Serial_Arduino.ino` file. It is used by currentSmoothChasePalette to keep track of which default smooth chase palette[42] is in use.

CLC	CloudColors_p
FTC	ForestColors_p
HTC	HeatColors_p
LVC	LavaColors_p
OCC	OceanColors_p
PTC	PartyColors_p
RBC	RainbowColors_p
RSC	RainbowStripeColors_p (RainbowStripesColors_p)

[42] See Smooth Chase Animation and Color Palettes

Using Colors with the AnimatedLEDStrip Library

ColorContainer Class

The AnimatedLEDStrip library uses its own class for storing colors, called ColorContainer. A ColorContainer contains a color split into its red, green and blue intensities, stored in `r`, g and b, respectively. `r`, g and b are limited to 0 through 255.

Construction

To construct a ColorContainer, call the ColorContainer constructor with:

- Three int values (for `r`, g and b), or
- A 6-digit hexadecimal value, or
- A CRGB default[43]

If sent three int values, `r`, g and b will be set directly from those parameters. If sent a hexadecimal representation of a color, the constructor will parse out `r`, g and b from the hexadecimal value. If sent a CRGB default, the constructor will copy the `r`, g and b values from the CRGB to the `r`, g and b values in the ColorContainer.

Compatibility

If a method expecting a ColorContainer receives a CRGB, the CRGB will automatically be converted to a ColorContainer.

Methods

The ColorContainer class has 7 set methods and 4 get methods.

- Set Methods
    - Set r
    - Set g
    - Set b
    - Set rgb (from 3 int values)
    - Set rgb (from 1 6-digit hex value)
    - Set rgb (from CRGB reference)
    - Blackout
- Get Methods
    - Get r
    - Get g
    - Get b
    - Get Color Hex

[43] See CRGB Struct and Appendix A

Set r, Set g and Set b Methods

```
void ColorContainer::setr( int intensity )
```

```
void ColorContainer::setg( int intensity )
```

```
void ColorContainer::setb( int intensity )
```

The set `r`, set g and set b methods set the `r`, g and b variables, respectively, to the specified intensity.

Set rgb Method

```
void ColorContainer::setrgb( int rIn, int gIn, int bIn )
```

```
void ColorContainer::setrgb( long hexIn )
```

```
void ColorContainer::setrgb( const CRGB & CRGBIn )
```

There are three variations of the set rgb method. One takes in three integer parameters and saves them in `r`, g and b. Another takes in a hexadecimal representation of a color and parses out `r`, g and b. The last copies a CRGB[44] instance's `r`, g and b values to `r`, g and b.

Blackout Method

```
void ColorContainer::blackout()
```

The blackout method is a special set method that sets `r`, g and b to `0`.

[44] See CRGB Struct

Get r, Get g and Get b Methods

```
int ColorContainer::getr()
```

```
int ColorContainer::getg()
```

```
int ColorContainer::getb()
```

The get `r`, get g and get b methods return the `r`, g and b values, respectively, stored in the ColorContainer.

Get Color Hex Method

```
long ColorContainer::getColorHex()
```

The get color hex method returns a hexadecimal representation of the color stored in the ColorContainer.

Explanations of Relevant Aspects of the FastLED Library

CRGB Struct

CRGB is a struct created by the FastLED library to hold the red, green and blue values for a color. These values can be accessed by accessing `r` or `red` for the red value, g or `green` for the g value, b or `blue` for the blue value, or `raw` for an array of all three `uint8_t` values. Using an index operator with index `0`, 1 or 2 on a CRGB will return the CRGB's red, green or blue value, respectively.

Construction

A CRGB can be created by calling its constructor with:

- Three 8-bit integers, or
- One 24-bit integer, or
- An LEDColorCorrection enum value, or
- A ColorTemperature enum value, or
- Another CRGB variable, or
- A CHSV[45] variable

Compatibility

If a CRGB is sent to a method expecting a ColorContainer[46], the CRGB will be automatically converted to a ColorContainer.

Defaults

There are 150 default colors, listed in Appendix A. These can be used by typing `CRGB::defaultName`.

[45] See CHSV Struct
[46] See ColorContainer Class

CHSV Struct

CHSV is another struct created by the FastLED library to hold the hue, saturation and value values for a color. These values can be accessed by accessing `hue` or `h` for the hue value, `saturation`, `sat` or `s` for the saturation value, `value`, `val` or `v` for the value value, or `raw` for an array of all three `uint8_t` values. Using an index operator with index 0, 1 or 2 on a CHSV will return the CHSV's hue, saturation or value value, respectively.

Construction

A CHSV can be created by calling its constructor with:

- Three 8-bit integers, or
- Another CHSV variable

Compatibility

If a function that expects a CRGB[47] variable is sent a CHSV variable, it will automatically convert it to a CRGB value.

Defaults

There are eight default values for the hue value of a CHSV variable:

```
HSVHue.HUE_RED        = 0
HSVHue.HUE_ORANGE     = 32
HSVHue.HUE_YELLOW     = 64
HSVHue.HUE_GREEN      = 96
HSVHue.HUE_AQUA       = 128
HSVHue.HUE_BLUE       = 160
HSVHue.HUE_PURPLE     = 192
HSVHue.HUE_PINK       = 224
```

[47] See CRGB Struct

Color Palettes

CRGBPalette16, CRGBPalette32, CRGBPalette256,
CHSVPalette16, CHSVPalette32, CHSVPalette256,
TProgmemRGBPalette16, TProgmemRGBPalette32,
TProgmemHSVPalette16, TProgmemHSVPalette32

Palettes are a set of classes in the FastLED library that hold arrays of colors. There are three main palette types for each color system (RGB and HSV) that can hold 16, 32 or 256 CRGB[48] or CHSV[49] values. There are also two more types for each color system (see the section on `TProgmem` palettes). If a 16- or 32-size palette is used, the FastLED library automatically scales it up to a 256-size palette when it uses them. This allows the user to save memory on the Arduino.

Constructing a Palette

A CRGB palette can be created by calling its constructor with:

- 16 CRGB (or CHSV) variables, or
- An array of CRGB variables of the same size, or
- An array of CHSV variables of the same size, or
- Another palette variable of the same or smaller size, or
- One CRGB or CHSV variable (will fill the full palette with the one color), or
- Two CRGB variables or two CHSV variables (will fill the palette with a series of colors that 'fade' from the first color to the second), or
- Three CRGB variables or three CHSV variables (will fill the palette with a series of colors that 'fade' from the first color to the second in the first half, then the second color to the third in the last half), or
- Four CRGB variables or four CHSV variables (will fill the palette with a series of colors that 'fade' from the first color to the second in the first third, then the second color to the third in the middle third and the third color to the fourth in the last third)

A CHSV palette can be created by calling its constructor with:

- 16 CHSV variables, or
- An array of CHSV variables of the same size, or
- Another CHSV palette variable of the same or smaller size, or
- One CHSV variable (will fill the full palette with the one color), or
- Two CHSV variables (will fill the palette with a series of colors that 'fade' from the first color to the second), or
- Three CHSV variables (will fill the palette with a series of colors that 'fade' from the first color to the second in the first half, then the second color to the third in the last half), or

[48] See CRGB Struct
[49] See CHSV Struct

- Four CHSV variables (will fill the palette with a series of colors that 'fade' from the first color to the second in the first third, then the second color to the third in the middle third and the third color to the fourth in the last third)

Using Colors Stored in a Palette

Colors can be extracted from palettes using the `ColorFromPalette(const paletteType pal, uint8_t index, uint8_t brightness = 255, TBlendType blendType = LINEARBLEND)` function. The function will return a CRGB or CHSV variable (depending on the `paletteType`) scaled to the specified brightness (full if not specified). `TBlendType` has two settings: `NOBLEND` and `LINEARBLEND`[50].

When calling `ColorFromPalette()`, a 16- or 32-size palette is essentially 'scaled up' to a virtual 256-size palette. This means that each color is stored to the index 16 times its original index in the 16-size palette (or 8 times its original index in the 32-size palette) (e.g. with a 16-size palette, the value in index `0` is stored in index `0`, 1 in 16, 2 in 32, etc.). The function must then determine what the color should be for the indices between those it just set. `blendType` controls how this step works. If `NOBLEND` is selected, then the function will return the same color for each successive index until the next pre-stored color is reached. For example, calling the function with a 16-size palette and with `NOBLEND` specified, indices 3 and 14 will return the same color. If `LINEARBLEND` is selected, the function will return a color that is a mix between the two nearest specified colors. If the whole 'palette' of 256 colors is viewed (i.e. when a loop is used to set all the pixels in a strip to successive indices in the 'palette') the colors will 'fade' from one color to the next. For example, calling the function with a 16-size palette with `LINEARBLEND` specified, index 3 will return a color close to the first color in the palette, but with a hint of the second color, and index 14 will return a color close to the second color but with a hint of the first color.

Because palettes are only 'scaled up' when `ColorFromPalette()` is called, the user can save space on the arduino by using a 16- or 32-size palette. (If you want to scale up a palette and save the larger palette, use `UpscalePalette(sourcePalette, destinationPalette)`[51])

`TProgmem` (Default) Palettes

`TProgmem` palettes are special in that they are only saved on the Arduino if they are mentioned in the program. The FastLED library has 8 built-in palettes (`CloudColors_p`, `LavaColors_p`, `OceanColors_p`, `ForestColors_p`, `RainbowColors_p`, `RainbowStripeColors_p` (a.k.a. `RainbowStripesColors_p`), `PartyColors_p`, and `HeatColors_p` - all are `TProgmemRGB16` palettes) that will only be included if they are used in the program. `TProgmem` palettes are arrays of `uint32_t` values (which can be specified with default CRGB colors or standard 24-bit integers).

[50] Note that 256-size palettes are unaffected by this setting
[51] Note that scaling up to a 256-size palette using `UpscalePalette()` will use the `ColorFromPalette()` function with `blendType` set to `LINEARBLEND`, while scaling up to a 32-size palette using `UpscalePalette()` will only double the size of the palette and store each color twice

Appendix A - CRGB Defaults

There are 150 default colors, which include 146 of the 148 default HTML color names (excluding `LightGray` (though `LightGrey` is included) and `RebeccaPurple` (`0x663399`)), plus 4 exclusive to the FastLED library.

AliceBlue	0xF0F8FF	Fuchsia	0xFF00FF	MistyRose	0xFFE4E1
Amethyst*	0x9966CC	Gainsboro	0xDCDCDC	Moccasin	0xFFE4B5
AntiqueWhite	0xFAEBD7	GhostWhite	0xF8F8FF	NavajoWhite	0xFFDEAD
Aqua	0x00FFFF	Gold	0xFFD700	Navy	0x000080
Aquamarine	0x7FFFD4	Goldenrod†	0xDAA520	OldLace	0xFDF5E6
Azure	0xF0FFFF	Gray	0x808080	Olive	0x808000
Beige	0xF5F5DC	Grey	0x808080	OliveDrab	0x6B8E23
Bisque	0xFFE4C4	Green	0x008000	Orange	0xFFA500
Black	0x000000	GreenYellow	0xADFF2F	OrangeRed	0xFF4500
BlanchedAlmond	0xFFEBCD	Honeydew†	0xF0FFF0	Orchid	0xDA70D6
Blue	0x0000FF	HotPink	0xFF69B4	PaleGoldenrod†	0xEEE8AA
BlueViolet	0x8A2BE2	IndianRed	0xCD5C5C	PaleGreen	0x98FB98
Brown	0xA52A2A	Indigo	0x4B0082	PaleTurquoise	0xAFEEEE
BurlyWood	0xDEB887	Ivory	0xFFFFF0	PaleVioletRed	0xDB7093
CadetBlue	0x5F9EA0	Khaki	0xF0E68C	PapayaWhip	0xFFEFD5
Chartreuse	0x7FFF00	Lavender	0xE6E6FA	PeachPuff	0xFFDAB9
Chocolate	0xD2691E	LavenderBlush	0xFFF0F5	Peru	0xCD853F
Coral	0xFF7F50	LawnGreen	0x7CFC00	Pink	0xFFC0CB
CornflowerBlue	0x6495ED	LemonChiffon	0xFFFACD	Plaid*	0xCC5533
Cornsilk	0xFFF8DC	LightBlue	0xADD8E6	Plum	0xDDA0DD
Crimson	0xDC143C	LightCoral	0xF08080	PowderBlue	0xB0E0E6
Cyan	0x00FFFF	LightCyan	0xE0FFFF	Purple	0x800080
DarkBlue	0x00008B	LightGoldenrodYellow†		Red	0xFF0000
DarkCyan	0x008B8B		0xFAFAD2	RosyBrown	0xBC8F8F
DarkGoldenrod†	0xB8860B	LightGreen	0x90EE90	RoyalBlue	0x4169E1
DarkGray	0xA9A9A9	LightGrey	0xD3D3D3	SaddleBrown	0x8B4513
DarkGrey	0xA9A9A9	LightPink	0xFFB6C1	Salmon	0xFA8072
DarkGreen	0x006400	LightSalmon	0xFFA07A	SandyBrown	0xF4A460
DarkKhaki	0xBDB76B	LightSeaGreen	0x20B2AA	SeaGreen	0x2E8B57
DarkMagenta	0x8B008B	LightSkyBlue	0x87CEFA	Seashell†	0xFFF5EE
DarkOliveGreen	0x556B2F	LightSlateGray	0x778899	Sienna	0xA0522D
DarkOrange	0xFF8C00	LightSlateGrey	0x778899	Silver	0xC0C0C0
DarkOrchid	0x9932CC	LightSteelBlue	0xB0C4DE	SkyBlue	0x87CEEB
DarkRed	0x8B0000	LightYellow	0xFFFFE0	SlateBlue	0x6A5ACD
DarkSalmon	0xE9967A	Lime	0x00FF00	SlateGray	0x708090
DarkSeaGreen	0x8FBC8F	LimeGreen	0x32CD32	SlateGrey	0x708090
DarkSlateBlue	0x483D8B	Linen	0xFAF0E6	Snow	0xFFFAFA
DarkSlateGray	0x2F4F4F	Magenta	0xFF00FF	SpringGreen	0x00FF7F
DarkSlateGrey	0x2F4F4F	Maroon	0x800000	SteelBlue	0x4682B4
DarkTurquoise	0x00CED1	MediumAquamarine†	0x66CDAA	Tan	0xD2B48C
DarkViolet	0x9400D3	MediumBlue	0x0000CD	Teal	0x008080
DeepPink	0xFF1493	MediumOrchid	0xBA55D3	Thistle	0xD8BFD8
DeepSkyBlue	0x00BFFF	MediumPurple	0x9370DB	Tomato	0xFF6347
DimGray	0x696969	MediumSeaGreen	0x3CB371	Turquoise	0x40E0D0
DimGrey	0x696969	MediumSlateBlue	0x7B68EE	Violet	0xEE82EE
DodgerBlue	0x1E90FF	MediumSpringGreen		Wheat	0xF5DEB3
FairyLight*	0xFFE42D		0x00FA9A	White	0xFFFFFF
FairyLightNCC*	0xFF9D2A	MediumTurquoise	0x48D1CC	WhiteSmoke	0xF5F5F5
FireBrick	0xB22222	MediumVioletRed	0xC71585	Yellow	0xFFFF00
FloralWhite	0xFFFAF0	MidnightBlue	0x191970	YellowGreen	0x9ACD32
ForestGreen	0x228B22	MintCream	0xF5FFFA		

* Exclusive to the FastLED library

† Note difference in capitalization