Neopixel Effects

Neopixel Effects

Some Classes for

Neopixel Effects

from

A. Groß

—

23. April 2018

last Change: July 10, 2018

Contents

1 Manual

This documentation contains some C++ classes that perform effects on a Neopixel device. The classes are developed for an ESP8266 with the goal to keep usage as simple as possible. The code is based on the `Adafruit_NeoPixel` library and the web site https://www.tweaking4all.com/hardware/arduino/adruino-led-strip-effects/ .

1.1 Intent

Most of the code, that is found, creates the effects within the `loop()` function using a delay to justify the speed of the effect. This is whwn you want to have an effect only, but when you want to perform other stuff within the loop you get delays, caused by the Neopixel effect.

The idea was to allow other tasks while the effects are running, including more than one effect at a time sharing the same or multiple Neopixel devices. Therefore the classes do not draw the whole effect at once, but require frequent update calls that can be placed within the `loop()` function. Most classes contain a flag that indicates that the effect is done after frequent calls to the effect's `update()` method. Timing within the effect is based on the ESP's `millis()` function so it plays no role what tasks are done besindes the effext, as long as they do not take longer than the delay you defined for the effect.

1.2 Development

Because compiling the code and flashing the result to the ESP takes relatively long, I decided to create a simple simulation of a Neopixel device using Qt on a Linux system. The `Adafruit_NeoPixel` library was ported to the Linux system by removing all the hardware related stuff and implementing a Qt signal that calls the main system drawing method when the `show()` function is called.

1.3 Current

Using a lot of LEDs can consume high current. Because this may damage your USB Port or other parts of your hardware, a new class, `npNeoPixel`, was derived from the `Adafruit_NeoPixel` class to add a method `npAmps()` to estimate the current that is needed for a defined pattern of the LEDs. Also, the `npShow()` method is added that justifies the current if it is too high. To use that, the class `npNeoPixel` has an additional parameter in its constructor, called `maxAmps`.

```
npNeoPixel::npNeoPixel(uint16_t n, uint8_t p, neoPixelType t, float maxAmps)
                       : Adafruit_NeoPixel(n, p, t)
```

The default of `maxAmps` is 0.5 Amperes. The calculation of the needed current is roughly estimated by taking the sum of the values for each LED, multiplied by 0.02 Amperes and devided by 255. Attention: The white pixel is not used for this calculation. This method is NOT absolutly safe and gives just a hint of the current needed for a specific pattern.

1.4 Concept

The classes are based on a virtual Neopixel device that is mapped to a `npNeoPixel` (the derived class). The advantage is that it is possible to split up one real device into several virtal devices to run different effects on the virtal devices at the same time. If - for example - you installed a Neopixel device in several lines every line can have it's own effect. The following pictures shows ways to install:

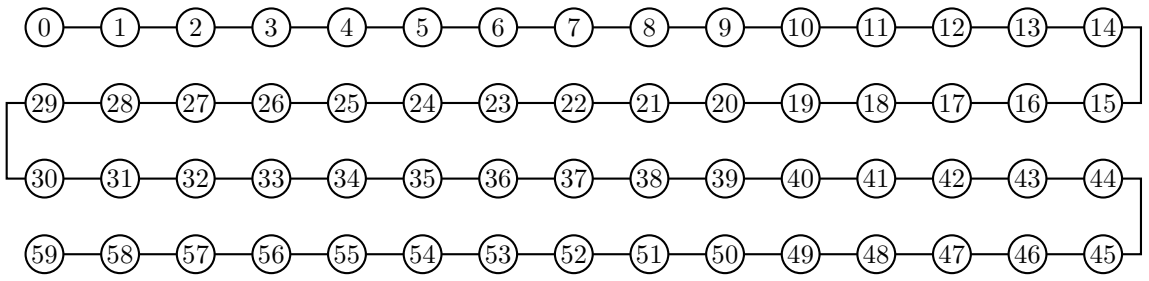

Figure 1.1: Example of connecting Neopixel strips

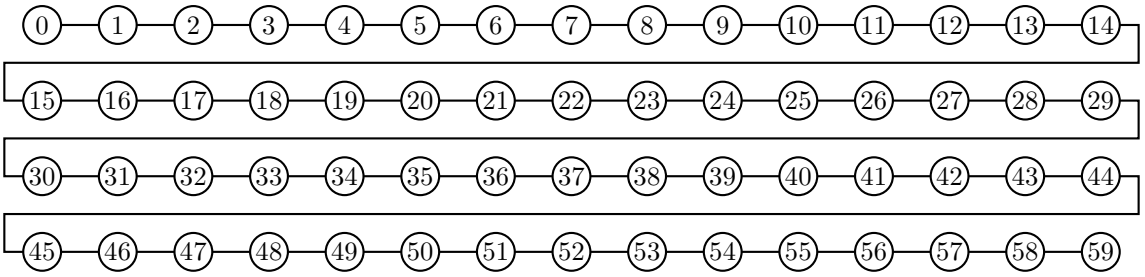

Figure 1.2: 2. Example of connecting Neopixel strips

You can see that numbering is affected when we decide to install the strips the way shown in figure 1.1 or 1.2. The effects introduced in this document can be configured for both kinds of installation.

Both installations can be devided into any number of virtual strips containing 1 to 60 pixels, running horizontaly or verticaly through the installation at the same time.

To define a virtual strip, the class `npVirtualNeo` is used.

1.5 Classes

All effects are provided in C++ classes that are derived from a base class with common properties and methods. They all use the class to create a virtual instance of a Neopixel device.

1.5.1 npVirtualNeo Class

To create a virtal Neopixel strip we just need a real Neopixel device and a definition of an area on this device.

The area that will be mapped to a virtual strip is defined by a start pixel and an end pixel. If you have a physical strip containing 10 pixels and you want to define the first 5 pixels as one virtual strip and the second 5 pixels as the other, the start and end pixels for the two strips are:

```
start1 = 0;
end1   = strip->numPixels()/2-1 = 4;

start2 = strip->numPixels()/2.  = 5;
end2   = strip->numPixels()-1.  = 9;
```

In the next picture the physical strip with 10 pixels shown. The first virtual strip (yellow) is at (0,4), the second (green) at (5,9). Because counting begins at 0, there is no pixel no 10!

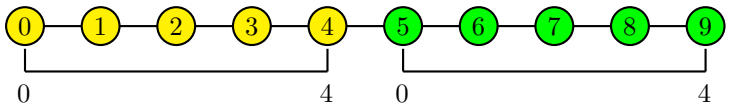

Figure 1.3: Defining two virtual Strips with Length of 5

To define these two strips, use the following commands:

```
npVirtualNeo Strip01(npNeoPixelInstance, 0, 4);
npVirtualNeo Strip02(npNeoPixelInstance, 5, 9);
```

The variable `npNeoPixelInstance` represents the address of a npNeoPixel instance. Then the number of the start- and the end pixel is passed to the constructor. All other parameters are skipped because their default values fit a physical strip installed as one line (one row, numPixel columns).

`Strip01` and `Strip02` can now be used to draw to the physical strip. If you want to set pixel 1 of Strip2 to be red, the physicalk pixel 6 will be affected (remember that counting starts at pixel 0).

```
Strip02.setPixelColor(1, 255, 0, 0);
```

Note that `start` and `end` can be swapped to change the direction of an effect. It is also possible to overlap virtual strips, but if the result is usable depends on the used effect. Defining Strip02 like this

```
npVirtualNeo Strip02(realNeo, 9, 5);
```

will map pixel 0 of the virtual strip to pixel 9 of the physical strip. Virtual pixel 1 will be physical pixel 8 and so on.

When only every second or third pixel should be part of the strip, add the stepsize as 4th parameter:

```
npVirtualNeo Strip02(realNeo, 5, 9, 2);
```

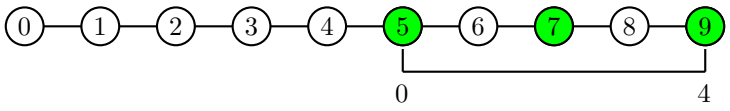

Figure 1.4: Defining a virtual Strips having Gaps

1.5.1.1 Virtual Strips in Multiple Lines, horizontal Orientation

Defining virtual strips on a physical strip with all pixels in one line is easy. When the physical strip is not installed as one line, the definitions depend in the way it is installed and how you want to arrange your effects.

When the strip is installed like in figure 1.5, a virtual strip defined in the range from pysical pixel 10 to physical pixel 19 will bend over the firt two lines. If the pysical strip is installed as shown in figure 1.6 the same definition will cause the virtual strip to be broken at the end of the first line and continue at the beginning of the second one.

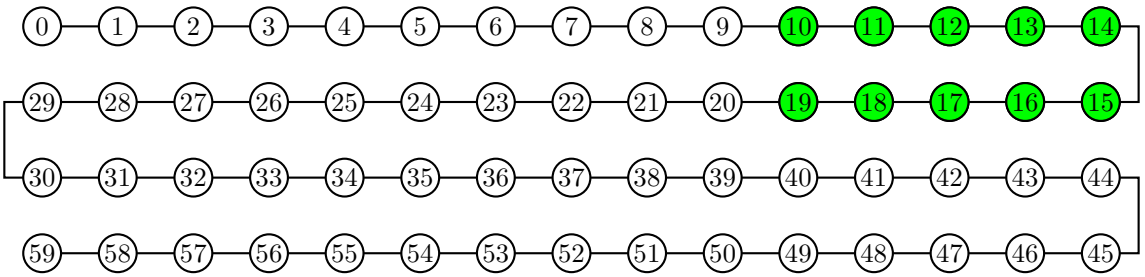

Figure 1.5: Defining a virtual Neostrip 1

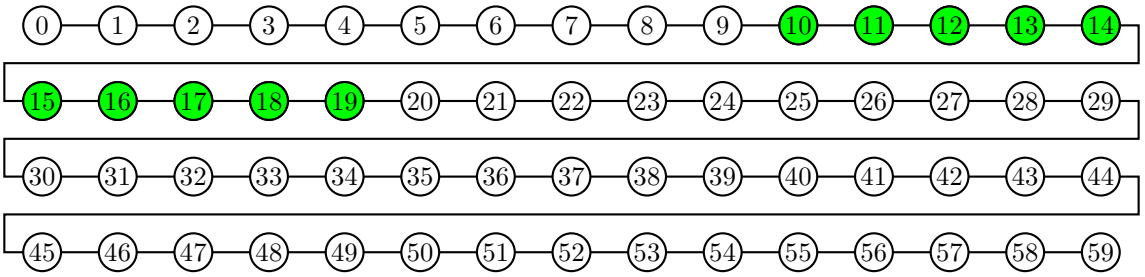

Figure 1.6: Defining a virtual Neostrip 2

In either cases, it is possible to switch between these effects using the `dir` parameter when defining the virtual strip. The default value for this parameter is `npVirtualNeo::ROWHORI` telling the constructor that we want the virtual strip to follow the connections of the physical strip. So the virtual strip is shiftet along the physical one by the start value (and may be change the direction if start and end is swapped).

If you have an installation like shown in figure 1.5 but want your virtual strip run row by row, everytime in the same direction, use the parameter `npVirtualNeo::ZIGHORI` indicating the need of turning the direction at every 2nd row.

Here is how the virtual strip is defined when the direction of every 2nd row has to be changed, no difference if the physical strip is installed like figure 1.5 or 1.6.

```
npVirtualNeo(neo, 20, 39, 1, npVirtualNeo::ZIGHORI, 10)
```

1.5.1.2 Virtual Strips in Multiple Lines, vertical Orientation

If the virtual strip has to be vertical, but the installation has physically horizontal lines, use the parameters `npVirtualNeo::ROWVERT` and `npVirtualNeo::ZIGVERT` depending in the installation of the physical strip. Numbering will start at pixel 0 top left and will continue at the first pixel at row 2, then at the first pixel at row 3 and so on.

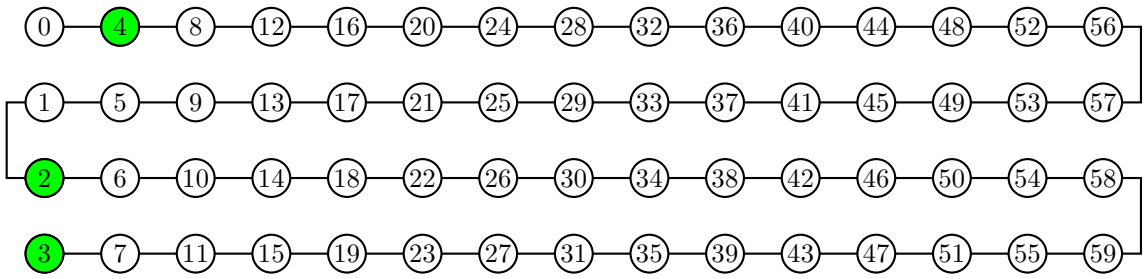

Figure 1.7: Defining a vertical virtual Neostrip

A virtual strip with the definition

```
npVirtualNeo(neo, 2, 4, 1, npVirtualNeo::ZIGVERT, 10)
```

will be positiond like shown in figure 1.7, at pixels 2, 3 and 4.

1.5.2 npBase class

Different effects usually contain same common properties like color, target-strip, delay-values, internal variables and so on. To keep development of additional effects as easy as possible, there is one base class from which all effect classes are derived. Two virtual functions, `doInit()` and `doUpdate()` are defined. These two base classes must be developed within a new effect, where `doInit()` is called when the class is created and restarted, and `doUpdate()` is called at every `update()` call to the developed effect class.

In your main loop, do NOT call `doUpdate()` directly. This will mess up timing and cause other issues. Your `doUpdate()` method is called when the `update()` method is called and the delay time has passed. In addition, time stamps are kept for the effects to allow time dependent actions and the physical strip is updated. See more information in the reference guide.

1.5.3 npBouncingBall

The Bouncing Ball effect simulates a ball thrown in the hight and than falling down to the start of the virtal strip, bouncing a few times up and down. Like most of the effects, it has two different constructors.

The first constructor takes a color, defined by red, green and blue (0-255), a delay to adjust the speed of the effect, and an instance of virtual strip.

The second constructor just takes a delay and a virtual strip. The color is calculated from the pixel position of the ball and will show it in rainbow colors.

Figure 1.8: npBouncingBall

```
npBouncingBall(unsigned char red, unsigned char green, unsigned char blue,
               unsigned int SpeedDelay, npVirtualNeo Target);
npBouncingBall(unsigned int SpeedDelay, npVirtualNeo Target);
```

1.5.4 npColorWipe

The Color Wipe effect fills the virtual strip with the defined color or rainbow colors, depending which of the constructors are used.

```
npColorWipe(unsigned char red, unsigned char green, unsigned char blue,
            unsigned int SpeedDelay, npVirtualNeo Target);
npColorWipe(unsigned int SpeedDelay, npVirtualNeo Target);
```

Figure 1.9: npColorWipe

1.5.5 npCylon

The Cylon effect shows a small line of the `Size` passed to the constructor. The color can be defined and like all effect classes, delay and instance of a virtual strip are needed.

```
npCylonBounce(unsigned char red, unsigned char green, unsigned char blue,
              unsigned int Size, unsigned int SpeedDelay, npVirtualNeo Target);
npCylonBounce(unsigned int Size, unsigned int SpeedDelay, npVirtualNeo Target);
```

Figure 1.10: npCylon

1.5.6 npFade

The Fade effect fades all pixels of the virtual Strip from off to the defined color or a rainbow color, depending on the used constructor. When the color is reached, it is faded back to black.

```
npFade(unsigned char red, unsigned char green, unsigned char blue,
               unsigned int SpeedDelay, npVirtualNeo Target);
npFade(unsigned int SpeedDelay, npVirtualNeo Target);
```

1.5.7 npFadeToColor

The Fade To Color effect fades all pixels of the virtual Strip to a predefined color or to a rainbow color, depending what constructor is used. A third constructor is available that copies the contents of the physical strip when the constuctor is called to fade every pixel individually to the these colors when the effect is used.

The parameter `FadeValue` defines the size of the steps that is used for fading.

```
npFadeToColor(unsigned char red, unsigned char green, unsigned char blue,
              unsigned int SpeedDelay, unsigned int FadeValue, npVirtualNeo Target);
npFadeToColor(unsigned int SpeedDelay, unsigned int FadeValue, npVirtualNeo Target);
npFadeToColor(Adafruit_NeoPixel *Strip, unsigned int SpeedDelay,
              unsigned int FadeValue, npVirtualNeo Target);
```

Figure 1.11: npFadeToColor

1.5.8 npFire

The Fire effect simulates a burning fire within the area of the virtual strip.

```
npFire(int Cooling, int Sparking, unsigned int SpeedDelay, npVirtualNeo Target);
```

Figure 1.12: npFire

1.5.9 npFlag

The Flag effect takes three colors and runs a little line composed of these colors througt the virtual strip. `Size` pixels are used for every color. So, if `Size` is 3, the whole line will have e length of 9 pixels.

If `KeepBackground` is set `true`, every former pixel is restored when the effect leaves a pixel.

```
npFlag(unsigned char red1, unsigned char green1, unsigned char blue1,
       unsigned char red2, unsigned char green2, unsigned char blue2,
       unsigned char red3, unsigned char green3, unsigned char blue3,
       unsigned int SpeedDelay, unsigned int Size,
       npVirtualNeo Target, bool KeepBackground);
```

Figure 1.13: npFlag

1.5.10 npFuse

The Fuse effect simulates a burning fuse, sparcling a little bit, moving along the virtual strip.

```
npFuse(unsigned char red, unsigned char green, unsigned char blue,
       unsigned int SpeedDelay, npVirtualNeo Target, bool KeepBackground);
npFuse(unsigned int SpeedDelay, npVirtualNeo Target, bool KeepBackground);
```

Figure 1.14: npFuse

1.5.11 npMeteor

The Meteor effect simulates a meteor that is going down. The pixels are faded to black after the meteor passes.

```
npMeteor(unsigned char red, unsigned char green, unsigned char blue,
         int meteorSize, int meteorTrailDecay, bool meteorRandomDecay,
         unsigned int SpeedDelay, npVirtualNeo Target, bool KeepBackground);
npMeteor(int meteorSize, int meteorTrailDecay, bool meteorRandomDecay,
         unsigned int SpeedDelay, npVirtualNeo Target, bool KeepBackground);
```

Figure 1.15: npMeteor

1.5.12 npRainbow

The Rainbow effect simulates a rainbow that fills the whole virtual strip, changing colors.

```
npRainbow(unsigned int SpeedDelay, npVirtualNeo Target);
```

Figure 1.16: npRainbow

1.5.13 npRunningDot

The Running Dot effect runs a colored pixel through the virtual strip.

```
npRunningDot(unsigned char red, unsigned char green, unsigned char blue,
             unsigned int SpeedDelay, npVirtualNeo Target, bool KeepBackground);
npRunningDot(unsigned int SpeedDelay, npVirtualNeo Target, bool KeepBackground);
```

Figure 1.17: npRunningDot

1.5.14 npRunningLight

The Running Light effect shows colored lines, faded in and out, moving through the virtual strip.

```
npRunningLight(unsigned char red, unsigned char green, unsigned char blue,
               unsigned int SpeedDelay, int Size, npVirtualNeo Target);
npRunningLight(unsigned int SpeedDelay, int Size, npVirtualNeo Target);
```

Figure 1.18: npRunningLight

1.5.15 npSparkle

The Sparcle effect shows random sparcles along the virtual strip.

```
npSparkle(unsigned char red, unsigned char green, unsigned char blue,
          unsigned int SpeedDelay, npVirtualNeo Target, bool KeepBackground);
npSparkle(unsigned int SpeedDelay, npVirtualNeo Target, bool KeepBackground);
```

Figure 1.19: npSparkle

1.5.16 npStrobe

The Strobe effect simulates a stroboscope over the whole length of the virtual strip.

```
npStrobe(unsigned char red, unsigned char green, unsigned char blue,
         unsigned int SpeedDelay, npVirtualNeo Target, bool KeepBackground);
npStrobe(unsigned int SpeedDelay, npVirtualNeo Target, bool KeepBackground);
```

Figure 1.20: npStrobe

1.5.17 npTheaterChase

The Theater Chase effect shows dots with a distance of `Distance` pixels running through the virtual strip.

```
npTheaterChase(unsigned char red, unsigned char green, unsigned char blue,
               unsigned int Distance, unsigned int SpeedDelay,
               npVirtualNeo Target, bool KeepBackground);
npTheaterChase(unsigned int Distance, unsigned int SpeedDelay,
               npVirtualNeo bound, bool KeepBackground);
```

Figure 1.21: npTheaterChase

1.6 Using the Effects

First of all you have to include the header file of the effect you want to use. In the following example, we use the running dot effect, called `npRunningDot` .

```
#include "npRunningDot.h"
```

The next task is to initialize your Neopixel strip. Use the parameters that fits your NeoPixel strip.

```
#define PIN 13  // use the pin where you connected the NeoPixels strip
#define MAXPIXELS 60
npNeoPixel neo = npNeoPixel(MAXPIXELS, PIN, NEO_GRB + NEO_KHZ800);
```

The next task is the creation of a virtual strip. In this example we will use the whole physical strip:

```
npVirtualNeo vNeo(&neo, 0, neo.numPixels()-1);
```

Then, create your instance of the `npRunningDot` class:

```
npRunningDot RunningDot1(255, 0, 0, 50, vNeo);
```

The last two steps can also be combined:

```
npRunningDot RunningDot1(255, 0, 0, 50,
                         npVirtualNeo(&neo, 0, neo.numPixels()-1));
```

The first three parameters define the color of the running dot (red). The delay (in milliseconds) tells the instance when its time to move the dot and the virtual strip parameter defines the area on the physical strip, where the running dot moves.

The main loop of our ESP program then looks like this

```
loop {
  RunningDot1.update();
}
```

when the effect should run once, or

```
loop {
  RunningDot1.update();

  if (RunningDot1.hasFinished() && RunningDot1.getCounter() <= 2) {
    RunningDot1.restart();
  }
}
```

when the effect should run multiple times.

The call to `update()` must be done to proceed with the effect. In most cases it will return immediately. Most effects have a finished state when they passed the whole strip. When this state is reached, `update()` returns immediately doing nothing. To restart the effect, call the `restart()` method that re-initializes the class so `update()` proceeds or restarts, depending in the effect. In the example above, a counter is queried to allow the instance just two times to run. This is useful to restart an effect a few times before showing something else.

Note that different effects can run at the same time. How it looks like, depends on the type of effects. The main reason to develop this type of effects is to be able to process other things in parallel.

2 Reference

2.1 npNeoPixel Class

The `npNeoPixel` class is derived from the original `Adafruit_NeoPixel` class to add a method to estimate the current of the LED pattern. Use this class to create effects based on the `npBase` class.

```
npNeoPixel(uint16_t n,
           uint8_t p=6,
   neoPixelType t=NEO_GRB + NEO_KHZ800,
 float maxAmps = 0.5);
```

- `n`: The number of pixels of your strip.
- `p`: The pin your strip is connected to.
- `neoPixelType`: The type of your strip.
- `maxAmps`: The maximum current your power supply can deliver. The default is 0.5A. The npShow() method of `npNeoPixel` will scale down any pattern to this value, if needed - but the calculation is just estimated based on linear assumptions of the current drawn by one pixel at a specified brightness. You are responsible to use a proper power supply.

2.2 npVirtualNeo Class

To create a virtual Neopixel strip we just need a real Neopixel device and a definition of an area on this device. There are two constructors. The first constructor allows the definition of a virtual strip by defining a range of physical pixels to be part of the virtual strip. The second constructor allows to add any pixels in any order.

2.2.1 First Constructor

The parameters of the first constructor are:

```
npVirtualNeo(npNeoPixel *Strip,
             unsigned int start,
             unsigned int end,
             unsigned int step,
             direction dir,
             unsigned int rows)
```

- `Strip`: The address of an instance of a `npNeoPixel` strip.
- `start`: The index (beginning at 0) of the real pixel that will be the first pixel of the virtual strip.
- `end`: The index (beginning at 0) of the real pixel that will be the last pixel of the virtual strip. `start` and `end` are inclusive. That means that a virtual strip with the `start = 10` and `end = 20` contains 11 pixels. To devide a real strip with an even number of pixels into two virtual strips of the same size, the first virtual strip has the range of:

  ```
  start = 0
  end   = strip->numColors()/2-1,
  ```

  the second one will range from

  ```
  start = strip->numColors()/2
  end   = strip->numColors()-1
  ```

- `step`: The step size within a virtual strip. This can be used if only every second or third pixel is wanted to be part of the virtual strip. The default is 1.
- `dir`: The `dir` parameter is needed when the strip is installed in more than 1 line, like in figure 1.1 and 1.2 above. Valid values are:

  `npVirtualNeo::ROWHORI`: This is the default. The virtual strip will be horizontal with pixel 0 at the top left position and numbering is row by row. Use this when the strip is installed as shown in figure 1.2.

  `npVirtualNeo::ZIGHORI`: The virtual strip will be horizontal with pixel 0 at the top left position and numbering is from left to right on the first row, right to left at the second row, left to right at the thirt row, and so on (see figure 1.1).

  `npVirtualNeo::ROWVERT`: This will define vertical virtual strips when the installation is done like shown in 1.2. The first pixel is the one top left.

`npVirtualNeo::ZIGVERT`: The virtual strip will be vertical, beginning at the top left pixel as pixel 0. The second pixel is the left most pixel in row 2. Use ZIGVERT when the strip is installed like shown in figure 1.1.

- `rows`: When the strip is installed in more than one row and you want to define a vertical virtual strip, you must tell the constructor how many rows you have. Each row must have the same size and the number of pixels in your physical strip must be divisible by the number of rows. If not, vertical strips will not work as expected.

2.2.1.1 Functionality of npVirtualNeo

The following methods can be used when working with npVirtualNeo:

`unsigned int getStart()`
Use `getStart()` to query the physical pixel number of the virtual strip's pixel number 0.

`unsigned int getEnd()`
`getEnd()` returns the physical pixel number of the last pixel of the virtal strip.

`bool isReverse()`
`isReverse()` returns `true`, if the direction of the virtual strip is different from that in the physical strip. This happens when a virtual strip is defined from a start pixel index that is larger than the end pixel index. In this case all effects will run in the other direction.

`unsigned int numPixels()`
`numPixels()` returns the number of pixels contained in the virtual strip.

`unsigned int getPosition(unsigned int value)`
`getPosition()` takes a pixel index and calculates the corresponding pixel index on the physical strip.

`npNeoPixel *getStrip()`
`*getStrip()` returns the address of the pysical device.

`bool checkNeoIndex(unsigned int index)`
`checkNeoIndex(unsigned int index)` returns false, when the passed index does not fit onto the physical strip the virtual strip is defined for.

`bool checkLocalIndex(unsigned int index)`
`bool checkLocalIndex(unsigned int index)` returns false if the passed index does not match the virtual strips pixel range.

`void setPixelColor(unsigned int ledNo, unsigned char red,`
`unsigned char green, unsigned char blue)`
`void setPixelColor()` sets the virtual strip's pixel `ledNo` to the passed color. If `ledNo` is outside the range of pixels, nothing is done.

`void setPixelColor(unsigned int ledNo, uint32_t color)`
`void setPixelColor(unsigned int ledNo, uint32\_t color)` sets the virtual strip's pixel `ledNo` to the color defined in a unsigned variable of the form `0xRRGGBB`.

`uint32_t getPixelColor(unsigned int ledNo)`
`uint32\_t getPixelColor(unsigned int ledNo)` returns the color of the virtual strip's pixel `ledNo`.

2.2.2 Second Constructor

The second constructor has only one parameter:

`npVirtualNeo(npNeoPixel *Strip)`

- `Strip`: The address of an instance of a `npNeoPixel` strip.

Using this constructor allows to stay away from any physical arrangement of the pixels. Any pixel can be added to the virtual strip to define an individual sequence of pixels.

In addition to the methods above, the following methods are available to define single pixels of the virtual strip.

`bool add(unsigned int ledNo)`
`bool add(unsigned int ledNo)` adds the pysical pixel `ledNo` to the virtual strip. The maximum number of pixels is limited to the length of the physical strip. The method returnes `false` if the pixel could not be added (invalid index or maximum pixels reached). This method is only working when the strip is defined using the constructor `npVirtualNeo(npNeoPixel *Strip)`

`void clear()`
`void clear()` removes all pixel definitions from the virtual strip.

2.2.3 Example

In this example, we have 10 lines of 10 pixels, connected alternating. We want to run 10 dots vertical at every column of the installation.

```
1 #include <npRunningDot.h>
2
3 #define ESP8266
4
5 #define NEO 13
6 #define MAXPIXELS 60
```

```
7  npNeoPixel neo = npNeoPixel(MAXPIXELS,
8                              NEO,
9                              NEO_GRB + NEO_KHZ800);
10 unsigned int Delay = 50;
11
12 npRunningDot RunningDot0(255, 255, 0, Delay,
13                          npVirtualNeo(&neo,  0,  9, 1,
14                          npVirtualNeo::ZIGVERT, 10));
15 npRunningDot RunningDot1(255,   0, 0, Delay,
16                          npVirtualNeo(&neo, 19, 10, 1,
17                          npVirtualNeo::ZIGVERT, 10));
18 npRunningDot RunningDot2(255, 255, 0, Delay,
19                          npVirtualNeo(&neo, 20, 29, 1,
20                          npVirtualNeo::ZIGVERT, 10));
21 npRunningDot RunningDot3(255,   0, 0, Delay,
22                          npVirtualNeo(&neo, 39, 30, 1,
23                          npVirtualNeo::ZIGVERT, 10));
24 npRunningDot RunningDot4(255, 255, 0, Delay,
25                          npVirtualNeo(&neo, 40, 49, 1,
26                          npVirtualNeo::ZIGVERT, 10));
27 npRunningDot RunningDot5(255,   0, 0, Delay,
28                          npVirtualNeo(&neo, 59, 50, 1,
29                          npVirtualNeo::ZIGVERT, 10));
30 npRunningDot RunningDot6(255, 255, 0, Delay,
31                          npVirtualNeo(&neo, 60, 69, 1,
32                          npVirtualNeo::ZIGVERT, 10));
33 npRunningDot RunningDot7(255,   0, 0, Delay,
34                          npVirtualNeo(&neo, 79, 70, 1,
35                          npVirtualNeo::ZIGVERT, 10));
36 npRunningDot RunningDot8(255, 255, 0, Delay,
37                          npVirtualNeo(&neo, 80, 89, 1,
38                          npVirtualNeo::ZIGVERT, 10));
39 npRunningDot RunningDot9(255,   0, 0, Delay,
40                          npVirtualNeo(&neo, 99, 90, 1,
41                          npVirtualNeo::ZIGVERT, 10));
42
43 void setup() {
44   // put your setup code here, to run once:
45   neo.begin();
46   neo.clear();
```

```
47    neo.show();
48  }
49
50  void loop() {
51    RunningDot0.update();
52    RunningDot1.update();
53    RunningDot2.update();
54    RunningDot3.update();
55    RunningDot4.update();
56    RunningDot5.update();
57    RunningDot6.update();
58    RunningDot7.update();
59    RunningDot8.update();
60    RunningDot9.update();
61  }
```

Listing 2.1: Example

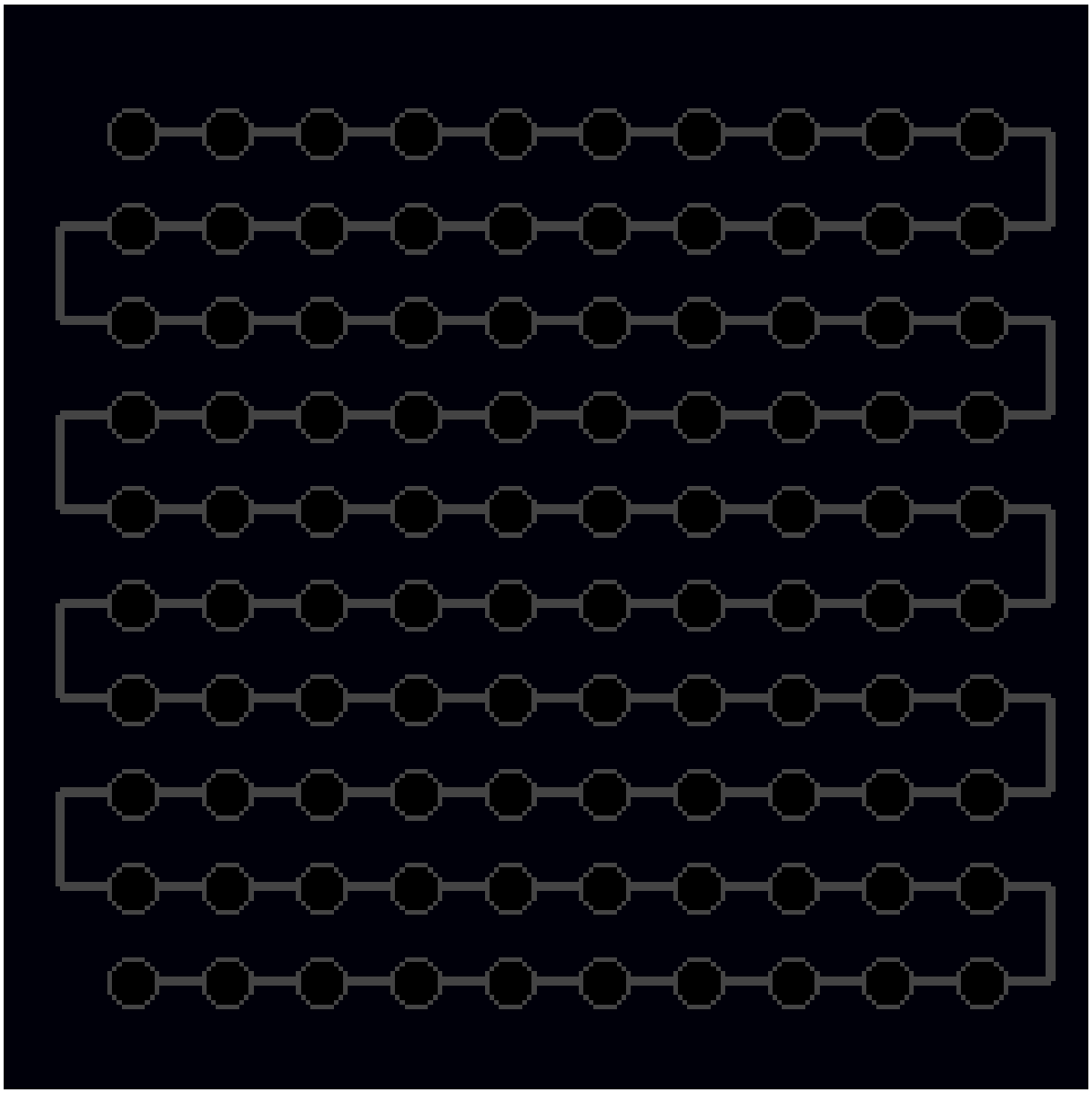

Figure 2.1: Multiple vertical virtual strips

2.3 npBase Class

This class contains many important functions that are used by the different effects.

```
npBase(npVirtualNeo Target, unsigned int Delay,
```

```
       unsigned char red, unsigned char green, unsigned char blue,
       bool KeepBackground = false, bool AutoUpdate = true)
npBase(npVirtualNeo Target, unsigned int Delay,
       bool KeepBackground = false, bool AutoUpdate = true)
```

The first constructor allows to define a specific color for your effect. Use the second constructor if you want to get some rainbow colors.

Your effect class must be derived from this base class:

```
#include "npBase.h"
class myEffect : public npBase {
public:
  myEffect(unsigned char red, unsigned char green, unsigned char blue,
           unsigned int SpeedDelay, npVirtualNeo Target,
           bool KeepBackground = false, bool AutoUpdate = true);
  myEffect(unsigned int SpeedDelay, npVirtualNeo Target,
           bool KeepBackground = false, bool AutoUpdate = true);
  ~myEffect();
private:
  void doInit();
  bool doUpdate();
  // whatever you need
  // ....
};
```

When your class is constructed, pass the needed arguments to the base class like this:

```
//------------------------------------------------------------------------
myEffect::myEffect(unsigned char red, unsigned char green, unsigned char blue,
                   unsigned int SpeedDelay, npVirtualNeo Target,
                   bool KeepBackground) :
                   npBase(Target, SpeedDelay, red, green, blue, KeepBackground)
{
  init();
  // .....
}
//------------------------------------------------------------------------
myEffect::myEffect(unsigned int SpeedDelay, npVirtualNeo Target,
                   bool KeepBackground) :
                   npBase(Target, SpeedDelay, KeepBackground)
```

```
{
  init();
  // ....
}
```

2.3.1 Within the Constructor of the Base Class

Depending on the constructor you call, the variables `r`, `g` and `b` are initialized with the color you selected and `doRainbow` is set to `false`, or - if the second constructor is used, `doRainbow` is set to `true`.

The variable `target` is set to the address of your virtual strip.

The variable `loopDelay` is set to your `Delay` value.

The variable `keepBackground` is set to your `KeepBackground` value.

The variable `doDelay` is set to `true`. If you set this variable to `false` within your `doUpdate()` method, the delay is ignored the next time you call `update()` in the main loop.

The variable `autoUpdate` contains the passed value (default = `true`). If set to `true`, the strip is updated automatically when a pixel changed. If set to `false` you have to update the strip in your main loop (`cpShow()` method of the npNeopixel class).

2.3.2 Initializing the Base Class

In the constructor of your effect class, call `init()` to initialize important things. Within the `init()` method, your `doInit()` method is called and the `Position` variable is set to `0`. This variable is used to run your effect through the virtual strip. The variable `done` is set to `false`. `done` can be set to `true` when your effect should end at a specific position. It can be reset when you call the `restart()` method. The `restart()` method calls the `init()` method to reset the effect.

2.3.3 The Update Method

When you call the `update()` method from the main loop, the variable `done` is checked. If it is set to `false`, the method checks if the delay time has passed (only when `doDelay` is `true`) and your `doUpdate()` method is called. If your `doUpdate()` method returns true, the target strip is repainted, when `autoUÃ¼date` is `true`.

2.3.4 Other Methods

There are methods defined in the npBase class that can be used for rainbow colors and fade effects.

2.3.4.1 Rainbow colors

The method

```
Wheel(unsigned int WheelPos, unsigned char *r, unsigned char *g, unsigned char *b
```

returns the red, green and blue value of the color wheel defined by `Wheelpos` (between 0 and 255). Use this function to get a color that depends on the position of the pixel within your virtual strip, for example calling

```
Wheel(((Position * 256 / target->numPixels()) + Position) & 255, &r, &g, &b);
```

2.3.4.2 Fading

The method

```
bool fadeToColor(unsigned int ledNo,
     unsigned char red, unsigned char green, unsigned char blue,
     int fadeValue, bool *allOff).
```

fades the current value of the defined pixel (`ledNo`) one step (`fadeValue`) to finally get the color you passed. `fadeToColor()` must be called multiple times to finish the fade effect. If the `fadeToColor()` method sets `allOff` to `true`, the color is reached and fading is finished.

When the return value is `true`, the pixel color changed and the virtual strip should be repainted. This is done automatically by the `update()` method.

Two other fading methods that internally call `fadeToColor()` are defined:

```
bool npBase::fadeToBlack(unsigned int ledNo,
                         int fadeValue, bool *allOff)
```

simply fades the pixel to black (off).

```
bool npBase::fadeToColor(unsigned int ledNo, unsigned int color,
                         int fadeValue, bool *allOff)
```

allows to define the color as one unsigned integer instead of single red, green and blue variables. The value can be defined as a hex value `0xRRGGBB`.

2.4 Building doInit()

Because the `doInit()` method of `npBase` is virtual, you have to define your own one, even if you do not need an init method:

```
void npRunningDot::doInit()
{
}
```

But in many cases, depending on your effect, you will need your own variables to be initialized when the effect is created or restarted.

2.5 Building doUpdate()

The `doUpdate()` method is the most important one. Here is the place where your effect is defined. `doUpdate()` will be called frequently within the `update()` method of the main loop and the base class keeps track that it is not called more often than defined (`delay`).

The color of your effect is stored in the variables `r`, `g` and `b`. If no color is defined, the variable `doRainbow` is set to true and you have to calculate the color within the loop:

```
bool npRunningDot::doUpdate()
{
  int rc = false;
  if (doRainbow) {
    Wheel(((Position * 256 / target->numPixels()) + Position) & 255, &r, &g, &b);
  }
  .
  .
```

The next thing to take care of is the `keepBackground` variable. It is used to indicate that the effect should restore the former contents of a pixel when it is passed. This will not make sense in every effect but it is useful in many cases. Let us assume our effect sets one pixel at every update and the contents of this pixel should be restored. You have to define a variable for the original pixal value in your class definition, lets say `uint32_t oldColor;`. This value should be set to `0` in your `doInit()` method. Also, define a variable for the saved pixel's position, lets call it `lastPosition`. This must also be set to `0` within the `doInit()` method of your constructor, depending if you want the effect to continue or start from beginning when you restart it.

```
void npRunningDot::doInit()
{
  lastPosition = 0;
  oldColor = 0;
}
```

Then, in the `doUpdate()` method, set the pixel of the last run to the stored value und save the pixel value of the next pixel:

```
  .
  .
  if (keepBackground) {
    setPixelColor(lastPosition, oldColor);
    oldColor = getPixelColor(Position);
  }
  else setPixelColor(lastPosition, 0, 0, 0);
  .
  .
```

Now we set the new pixel using the call

```
  .
  .
  rc = setPixelColor(Position, r, g, b);
  .
  .
```

The remaining lines of the `doUpdate()` method is housekeeping. The `lastPosition` is updated, `Position` is increased by 1 (or the step size you want) and a check is done to find out if the effect is done (when it passed the whole virtual strip).

```
  .
  .
  lastPosition = Position;
  Position++;
  if (Position >= target->numPixels()) {
    done = true;
    Position = 0;
  }
  return rc;
}
```

The whole definition of the effect class is defined like this:

myEffect.h:

```
#ifndef NPMYEFFECT_H
#define NPMYEFFECT_H

#include "npBase.h"
class myEffect : public npBase {
public:
  myEffect(unsigned char red, unsigned char green, unsigned char blue,
           unsigned int SpeedDelay, npVirtualNeo Target,
           bool KeepBackground = false);
  myEffect(unsigned int SpeedDelay, npVirtualNeo Target,
           bool KeepBackground = false);
  ~myEffect();
private:
  void doInit();
  bool doUpdate();
  unsigned int lastPosition;
  uint32_t oldColor;
};
#endif // NPMYEFFECT_H
```

myEffect.cpp:

```
#include <stdio.h>

#include "myEffect.h"
//-------------------------------------------------------------------------
myEffect::myEffect(unsigned char red, unsigned char green, unsigned char blue,
                   unsigned int SpeedDelay, npVirtualNeo Target,
                   bool KeepBackground) :
                   npBase(Target, SpeedDelay, red, green, blue, KeepBackground)
{
  init();
  lastPosition = 0;
  oldColor = 0;
}
//-------------------------------------------------------------------------
myEffect::myEffect(unsigned int SpeedDelay, npVirtualNeo Target,
```

```
                    bool KeepBackground) :
                    npBase(Target, SpeedDelay, KeepBackground)
{
  init();
}
//--------------------------------------------------------------------------------
myEffect::~myEffect()
{
}
//--------------------------------------------------------------------------------
void myEffect::doInit()
{
  lastPosition = 0;
  oldColor = 0;
}
//--------------------------------------------------------------------------------
bool myEffect::doUpdate()
{
  int rc = false;
  if (doRainbow) {
    Wheel(((Position * 256 / target->numPixels()) + Position) & 255, &r, &g, &b);
  }
  if (keepBackground) {
    setPixelColor(lastPosition, oldColor);
    oldColor = getPixelColor(Position);
  }
  else setPixelColor(lastPosition, 0, 0, 0);
  rc = setPixelColor(Position, r, g, b);
  lastPosition = Position;
  Position++;
  if (Position >= target->numPixels()) {
    done = true;
    Position = 0;
  }
  return rc;
}
```

List of Figures

Index