Witty Apa Strip

- Documentation under way. See our old more complete doc on https://www.didel.com/RGBstrips.pdf (that must also be updated).

List of functions on Apa102.h lib on https://git.boxtec.ch/didel/Witty

Not well verified, see Apa102.h source

Strip length
@define Npix 8
Brt levels
#define Brt 2

Low level primitives
ApaS8 (byte);
ApaHead();
ApaLed (r,g,b);
ApaTail ();
ApaClear();

0-255 intensity
ApaRGB(r,g,b);

0-15 log intensity
ApaLogRGB(rl,gl,bl);
ApaLogRed(rl); ApaLogGreen(rg); ApaLogBlue(rb);
ApaLogYellow(rgl); ApaLogPink(rbl); ApaLogWhite(wl);

Hue and rainbow
ApaHtoRGB (h); work with ApaHue
ApaHue(h);
ApaRainbow(); optimized for a strip of 8

Special functions (not in Apa102.h)
ShowByte(v8); see WittyDemo2 (demo 5)

Experiments with APA102 RGB mini strip

Which driver?

For the APA102, Pololu propose the driver https://github.com/pololu/apa102-arduino
Our driver is not yet on Git, and it is not a driver, but a set of included files you get loading our examples..
The documented software uses Arduino pins 14/A0 (Ck) and 15/A1 (Da). It is easy to define other pins. You see on the definitions we do not use digitalWrite for speed reason, but of course you can.
RGB smart leds and shift registers are explained on www.didel.com/RGBstrips.pdf
We repeat here only what will be required to understand our programs, the source of which can be found on www.didel.com/RGBstrips.zip

APA102C survey

RGB chips have an interesting internal architecture. Circuits are cascaded, the closer to the controller keeping the first pixel data sent.
There are three families of RGB chips.

1) 2 SPI-like signals (Ck,Data) that is 6 pins package, 6 lines between LEDs
   Circuits are APA102, Sk9822. Libs used here are from Pololu and Didel
2) 1 encoded signal, that is a 4 pins package, 4 lines between LEDs
   Circuits are APA104, WS2812B, SK6812
3) 1 encoded signal plus one "secure" input, that is a 6 pins package, 5 lines between LEDs
   Circuit is WS2813B. For these two cases, libs are NeoPixel and WS28. See www.didel.com/WS28.pdf)

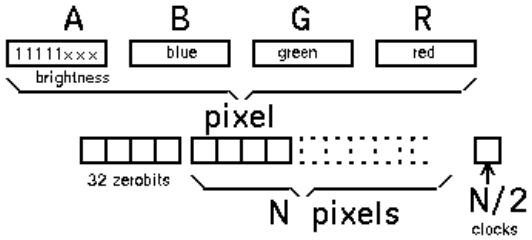
An APA102 strip is a long shift register with a data signal sampled by the positive edge of the clock.
A synchronization word (32 zero bits) and additional final clocks avoid a delay between update sequences, as it must be done with other strips to say the transfer is finished.

SPI can be used. It allows a very fast transfer, not requires for less than 1000 LEDs strips, and brings the drawback of using 2 specific lines of the microcontroller.
Software transfer is quite efficient if Arduino digitaWrite is not used. Definitions and 8-bit serial transfer function is given below. Execution time is less than 5us on an AVR 328 16MHz

```
//Snd 8 bits function for APA102
#define bCk  14 //PORTC
#define bDa  15
#define DaOn  bitSet(PORTC,bDa)
#define DaOff bitClear(PORTC,bDa)
#define CkPulse bitSet(PORTC,bDa); bitClear(PORTC,bDa)
void SetupApa102 () { DDRC |= (1<<bCk)+(1<<bDa) }

void S8 (byte dd) {
  for (byte i=0;i<8;i++) {
    if (dd & 0x80) DaOn;
    else DaOff;
    CkPulse;
  }
}
```

Next picture shows the transfer format. Each LED receives a 4-byte word.
First byte has 3 bits at logic "1" and 5 bit of "brightness", value 0 to 31.
Follows the PWM values for the 3 colors. The shift order depends on the manufacturer, Order of colors in documentation and for listing parameters will always be RGB.
Several clocks must be added the end of the transfer of all the pixels.



Brightness and PWM

APA102 needs a first brightness 5–bit variable as a global control on the three output amplifiers of the LEDs. Values 0 to 31 are not linear and does not change intensity between 0 and 100%. We leave is at value 2 or 5 while developing software inside. The three 8-bit PWM values controls the individual intensity of the LEDS. The resulting action is by no way linear, as it can be tested with the first program. We decide the brightness is a constant parameter BR declared in the beginning, and have only three RGB variables values for every pixel. Parameters are given with the usual order R-G-B. As shown on previous figure, transfer order is different.

Eye sensitivity

The eye responds to an enormous range of light intensity, exceeding 10 units on logarithmic scale. Contrast sensitivity is related when one think to applications.

PWM is linear, the difference of perceived intensity is great between PWM =1 and PWM =6, but not noticable between 250 and 255. A correction must occur to give the impression of a regular increase of intensity.
Neopixel and other libs e.g. strip.setPixelColor() use the linear PWM value, we name LinRGB or RGB(). Our lib allow to use LogRGB(), that access a table to set the PWM values.
The table has 32 entries, 3 bits are lost when accessing the table. Values 0 to 7, 8 to 15, .. 246 to 255 have the same effect.
We duplicate the RGB and Hue functions to offer both approaches. Run the test programs to see the difference.

Buffer or direct transfer

NeoPixel and all libraries use a copy of the LED PWM's in memory and do the update with a global transfer - strip.show(). Transfer time is always maximum, even if a single pixel is modified.
Didel APA lib require to transfer all pixels till the last modified one. Demos programs usually compute the RGB values of consecutive LEDs. NeoPixel copy these values in the buffer and do the transfer when finished. Didel Apa Lib transfers sequentially during computation.

APA102 Didel lib functions

See ApaWitty documentation and demos www.didel.doc/ApaWitty.pdf

Understand we need the flexibility to work on the same strip with different length of LEDs and we do not have a buffer (can be easily added, but the idea is to work with a small microcontroller (AVR Tiny or PIC). Considering the pixel in sequence save a parameter. The other parameter savec is the global brightness BR. It is hence more easy to

We need an initialization all-zero's routine required before each transfer

```
void ApaStart () {
  DaOff;
  for (byte i=0;i<32;i++) { CkPulse; }
}
```

The pixel routine calls four time the `S8(data);` routine

```
void ApaRGB (byte rr,byte gg,byte bb) {
   S8 (0xE+BR); S8 (bb); S8 (gg); S8 (rr);
}
```

BR is a global intensity level, 1 to 31 (usually 1 to 3)

As explained in the detailed doc, N/2 clocks must terminate the sequence

```
void ApaStop (byte nn) {
  DaOff;
  for (byte i=0; i<nn/2; i++) {CkOn;  CkOff;}
}
```

Now there are two cases.

1) The RGB values are fetched or calculated during the transfer. A `while`, a `for` loop or a state machine transfers the pixels
2) A bit-map table contains the value of the pixels. It can be a structure with the RGB bytes indexed by the LED index number. That is one `rgb[N]` table or 3 tables `red(N); green(N); blue(N);` Access does not need to be explained, we guess.

The bitmap is transferred to the strip with an Update function

```
void Update () {
  Start ():
  for (byte i=0;i<N;i++; {
    Send(0xE+BR); Send(blue[i]; Send(green[i]; Send(red[i];
  }
  Stop(N);
}
```

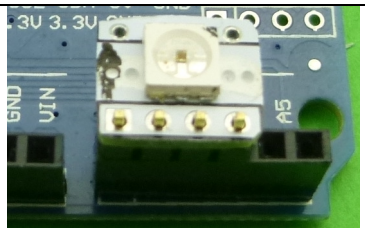
We do not use here a table.

First experiment

Module is connected on pins 14 (Da) 15 (Ck) 16 (+5V) and 17 (Gnd). pinMode notation is also proposed if you are not keen with logic operations (and willing to learn by comparing)

```
#define  bCk 1   //pin 15 A1
#define  bDa 0  //pin  14 A0
#define  b5V 2
#define  bGnd 3
#define CkOn  bitSet (PORTC,bCk)
#define CkOff bitClear (PORTC,bCk)
#define CkPulse bitSet(PORTC,bDa); \
              bitClear(PORTC,bDa)
#define DaOn bitSet (PORTC,bDa)
#define DaOff bitClear (PORTC,bDa)
void SetupApa () {
 DDRC
 |= (1<<bCk)+(1<<bDa)+(1<<b5V)+(1<<bGnd);
 PORTC |= (1<<b5V);
 PORTC &= ~(1<<bGnd);
}
```

If you power the LED from Gnd and +5V, the definitions for bGnd and b5V are not required. If you need more than 10 LEDs and dazzling light, you have to power directly (pins are limited to 20mA).




```
pinMode (14,OUT);
pinMode (15,OUT);
pinMode (16,OUT);
pinMode (17,OUT);
#define CkOn digitalWrite(15,HIGH)
#define CkOff digitalWrite(15,LOW)
#define DaOn digitalWrite(14,HIGH)
#define DaOn digitalWrite(14,LOW)
void SetupApa () {
  pinMode(14,OUTPUT);
  pinMode(15,OUTPUT);
  pinMode(16,OUTPUT);
  pinMode(17,OUTPUT);
  digitalWrite(16,HIGH); // +5V
  digitalWrite(17,LOW);  // Gnd
}
```

Sample program

Exemple 1

```
//Apa102First  784b 9v color first led
#define Npix 4 // Number of pixels in the string (for Clear();
#define  BR 2 // max 31 brigtness
#include "APA102.h"  // define pin here
void setup () {
  SetupApa102();  // clear the first Npix
}
void loop () {
   Start(); RGB (100,0,0); Stop(1);
   delMs (1000);
   Start() RGB (0,100,0); Stop(1);
   delMs (1000);
}
```

Exemple 2 Dim the red color – compare lin and log

```
// LinVsLog.ino  Compare lin and log dimming on first two LEDs
// Increase/decrease by steps of 8  -- ok
#define Npix 6 // Number of pixels in the string
#define  BR 2 // max 31 brigtness
#include "Apa102.h"  // define pin
void setup () {
  SetupApa102();
  Clear();
}
byte red; byte i;
void loop () {
  while ((i+=8)<248) {  // increase
    Start(); RGB (i,0,0); LogRGB (i,0,0); Stop(2); delay(100);
  }
  delMs (1000);
  while (i>0) { i-=8;   // decrease
    Start(); RGB(i,0,0); LogRGB(i,0,0); Stop(2); delay(100);
  }
  delMs (1000);
}
```

Exemple 3 Rainbow

Appendix

Library functions

```
#define  Npix 6  // strip length
#define BR 2  // brightness
void Start () {
  DaOff;
```

```
  for (byte i=0;i<32;i++) { CkPulse; }
}
void RGB (byte rr,byte gg,byte bb) {
   S8 (0xE+BR); S8 (bb); S8 (gg); S8 (rr);
}
void Stop (byte nn) {
  DaOff;
  for (byte i=0; i<nn/2; i++) {CkOn;  CkOff; }
}

byte red[Npix], green[Npix], blue[Npix],
void Update () {
  Start ():
  for (byte i=0;i<Npix;i++; {
    S8(0xE+BR); S8(blue(i); S8(green(i); S8(red(i);
  }
  Stop(Npix);
}
```

Other tests

We just need to document the main program. You have a strip of 10 correctly connected.
Let us test the 3 colors on the strip.and answer the question

Have all LED the same intensity with the same parameters?
Have the RGB LEDs the same physiological intensity with the same PWM values?
Is the perceived light intensity proportional to PWM?
What is the effect of brightness BR (0 to 31)?

```
//ApaTest1.ino   Compare LED intensity at different pwm
#define Npix 6   //number of LEDS
#define BR 2  //brightness 0..31
#include "APA102.h"
void setup () {  SetupApa(); }

byte pwm = 20;  // try 0, 1, 10, 40, 100, 200, 250
void loop () {
  Start():
  for (byte i=0;i<N;i++;) { SendPixel(pwm,0,0); }
  SendLast(N);
delay(1000);
  Start():
  for (byte i=0;i<N;i++;) { SendPixel(0,pwm,0); }
  SendLast(N);
delay(1000);
  Start();
  for (byte i=0;i<N;i++;) { SendPixel(0,0,pwm); }
  SendLast(N);
delay(1000);
}
}
```

Mode direct – vous définissez au fur et à mesure ce que vous transférez

Ce que le strip demande c'est dans l'ordre
Sstart(aa); synchro de début de bloc avec paramètre d'intensité
Srgb (rr,gg.bb); Scolor(rr,gg,bb); Shue(hh); pour chaque pixels (boucle for)
S20colors? S20hue?
Sstop(nn); bloc de fin avec nombre de pixels
Fonctions auxiliaires
Exemple 1 Variation du rouge – comparaison direct et log

Mode tampon – vous préparez en mémoire et mettez à jour

Il faut définir la variable intensité et le tableau des pixels, vu comme une une suite des trois bytes rgb. byte TaStrip[nn]; R=0 G=1 B=2
Pour accéder dans au vert du 3e élément on écrit TaStrip[2+G];
On met à jour avec Sshow(); ou S20show();

Xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

from?? Fichier apa102producdescription détruit le 190325

APA 102 Product Description:

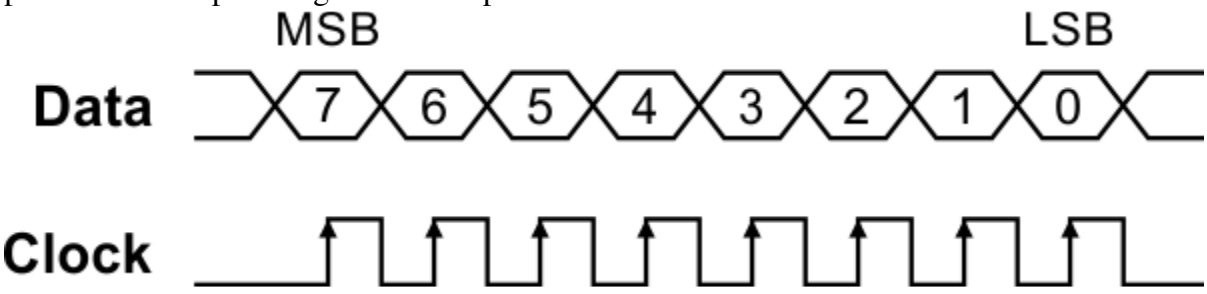
Dimensions: 5mm x 5mm x 1.4mm
Voltage: 5.0V
Current 20ma per color, 60mA total at full brigtness
Viewing angle: 120 degree.
APA102 for the three-color RGB LED dimming control string Then IC, using the CMOS process,
providing three-color RGB LED output driver to adjust the pouput with 256 gray-scale and
32 brightness adjustment APA102 with two-output way,the CLK signal by synchronizatioAPA102 LED CHIP n,
so that the crystal cascadePiece of output movements synchronized.

Features

CMOS process, low coltage, low power consumption
Synchronous of two-lane
Choose postive output or negative output RGB tri-
color LED out, 8 Bit(256level) color Set, 5Bit (32 level) brightness adjustment
Built-20mA constant current output
With self-detection signal
Built-in support for continuous oscillation PWM output can be maintained Static Screen

Protocol

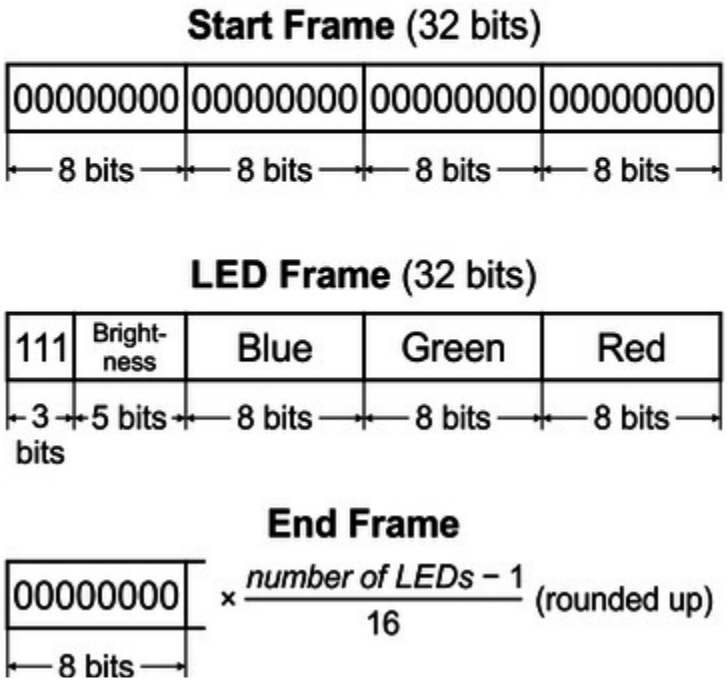
These LED strips are controlled through an SPI protocol on the data and clock input lines. The protocol is documented in the APA102C datasheet (1MB pdf), but we describe it below with some modifications that we have found to work better.
The default, idle state of the clock signal line is low, and the data signal is read on each rising edge of the clock. To update the LED colors, you need to toggle the clock line while driving the data line with the value of each bit to send; this can be done through software (bit-banging), or it can be handled by a hardware SPI peripheral in a microcontroller. There is no minimum clock frequency, although using a lower frequency means that it will take longer to update the entire sequence of LEDs (especially when controlling a long strip or many strips are chained together), so you will probably want to use the fastest practical clock speed to get the best update rate.



APA102C control signal timing diagram.

The data for each LED is encoded as a sequence of 32 bits (4 bytes) called an LED frame. The first three bits of the LED frame should be ‘1’. The next 5 bits are a “global”, color-independent brightness value (0–31) that is applied equally to all three color channels. The remaining 24 bits are the color values, in BGR (blue-green-red) order. Each color value uses 8 bits (0–255). The most significant bit of each value is transmitted first. The first LED frame transmitted applies to the LED that is closest to the data input connector, while the second color transmitted applies to the next LED in the strip, and so on.
To update all the LEDs in the strip, you should send a “start frame” of 32 ‘0’ bits, then a 32-bit “LED frame” for each LED, and finally an “end frame”. If you send fewer LED frames than the number of LEDs on the strip, then some LEDs near the end of the strip will not be updated.
The APA102 datasheet recommends that the end frame be composed of 32 ‘1’ bits, but we have found this does not work reliably in certain situations and can sometimes lead to glitches. This can be avoided by using an end frame that consists of at least $(n–1)$ extra clock edges, where n is the number of LEDs, with ‘0’ on the data line. It is often easiest to round up to a multiple of 16 clock edges so that you are counting bytes instead (there are 2 clock edges in a bit and 8 bits in a byte); you would therefore send $((n–1)/16)$

bytes (rounded up to the next whole number). For a more detailed explanation, see the comments in the source code of our APA102 Arduino library, discussed below.



APA102C data format.

For example, to update all 30 LEDs on a 1-meter strip, you should send a 32-bit start frame, thirty 32-bit LED frames, and a 16-bit end frame, for a total of 1008 bits (126 bytes). If multiple strips are chained together with their data connectors, they can be treated as one longer strip and updated the same way (two chained 1-meter strips behave the same as one 2-meter strip).

Each RGB LED receives data on its data input line and passes data on to the next LED using its data output line. The update rate is generally limited only by the speed of the controller; our Arduino library below can update 60 LEDs in about 1.43 milliseconds, so it is possible to update nearly 700 LEDs at 60 Hz. However, constant updates are not necessary; the LED strip can hold its state indefinitely as long as power remains connected.

Note: The minimum logic high threshold for the data and clock signals is 3.5 V, so you should use level-shifters if you want to control these strips from 3.3 V systems. It might be possible to control them with 3.3 V signals directly, but using the strip out of spec like this could lead to unexpected problems.

Sample code

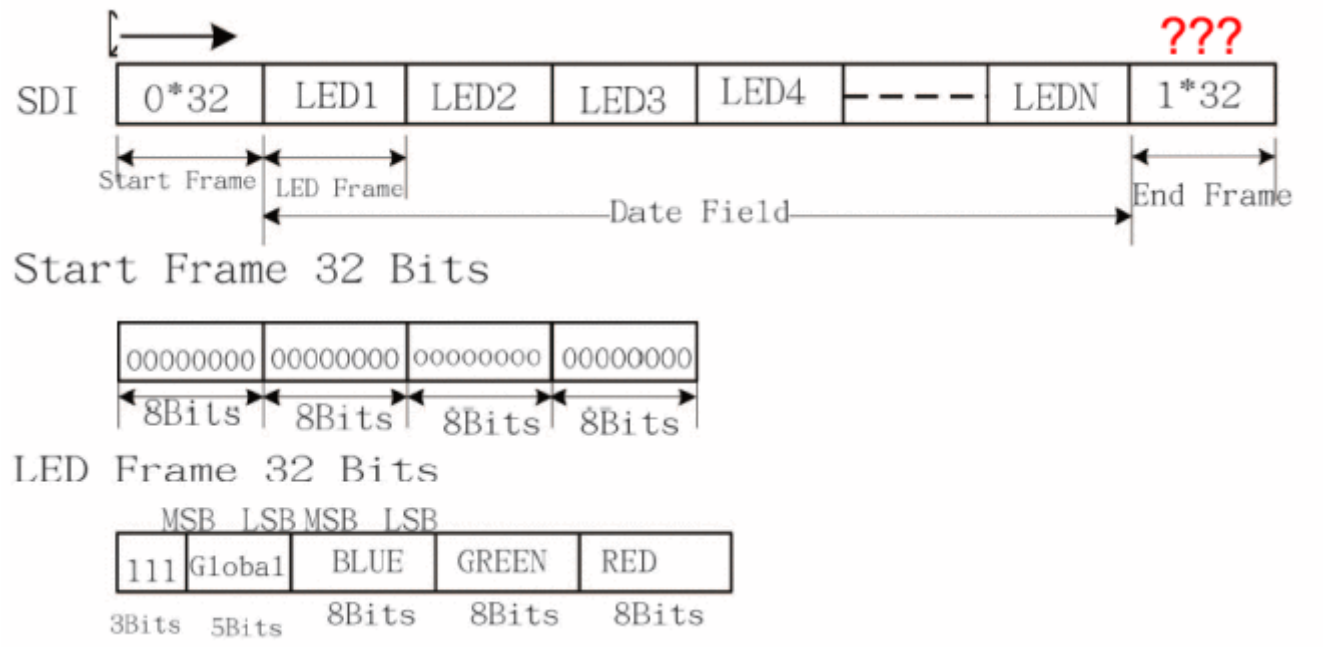
To help you get started quickly, we provide an APA102 Arduino library (it also works with our Arduino-compatible A-Star modules).

Understanding the APA102 “Superled”

November 30, 2014 33 Comments

A couple of weeks ago I reported about a new type of RGB-LED with integrated controller, the APA102. One of the interesting new features of this device is a two-wire SPI interface instead of the proprietary one-wire protocol of the more common WS2812. Many microcontrollers have hardware SPI functions, which allow easy control of these LEDs, as opposed to timing critical bit banging. But it turned out this was not the end of the story. As pointed out by Bernd in a comment, there is some discrepancy between the datasheet and the actual behavior of the devices when it comes to the “end frame”. Reason enough to subject the APA102 to more scrutiny.

The diagram below summarizes the APA102 protocol as found in the data sheet.



To investigate the functionality of the AP102, I connected an ATtiny85 to the clock and data input lines and used a logic analyzer to probe both the input and output lines. The microcontroller was programmed to output various bit patterns as described below.
I focused my investigation on four areas:

1. Behavior of the “Start Frame”
2. Function of the “111” bits in the “LED Frame”
3. How data is forwarded to the next device
4. Impact of the “End Frame”

The Start Frame

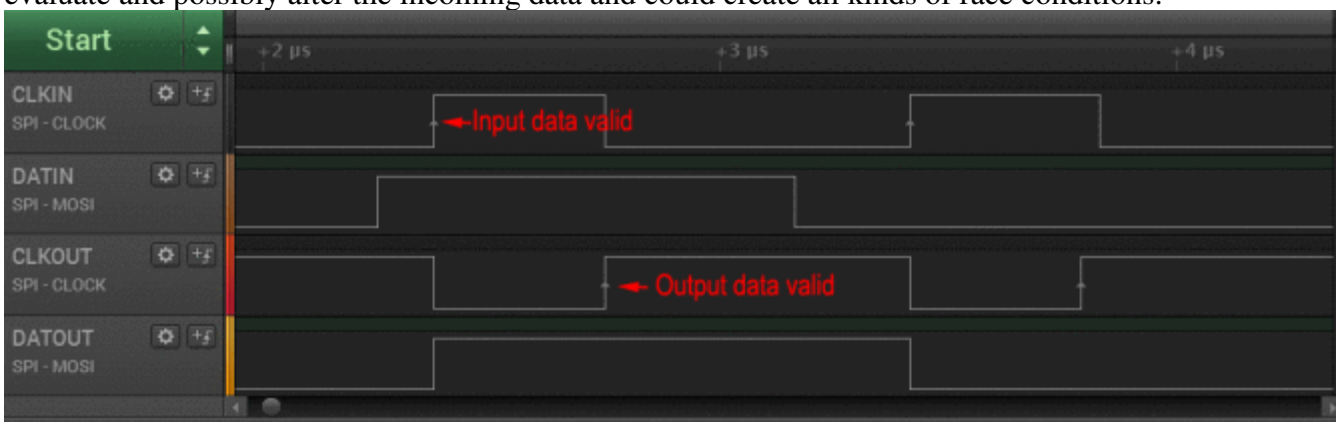
I varied the number of zero bits in the start frame from below to above 32. It turns out that a minimum of 32 zeroes are required to initiate an update. Increasing the number of zeroes does not have any impact.
The LED frame is identified by the first one bit following the start frame.
The LED output color is updated immediately after the first valid LED frame. This is quite interesting, since it means that almost arbitrary update rates of the APA102 are possible. However, this may lead to a “staggered” update for longer strings, where the first LEDs in a string are updated earlier than the later ones. The best way to work around this is to use a sufficiently high SPI clock rate.

The LED Frame

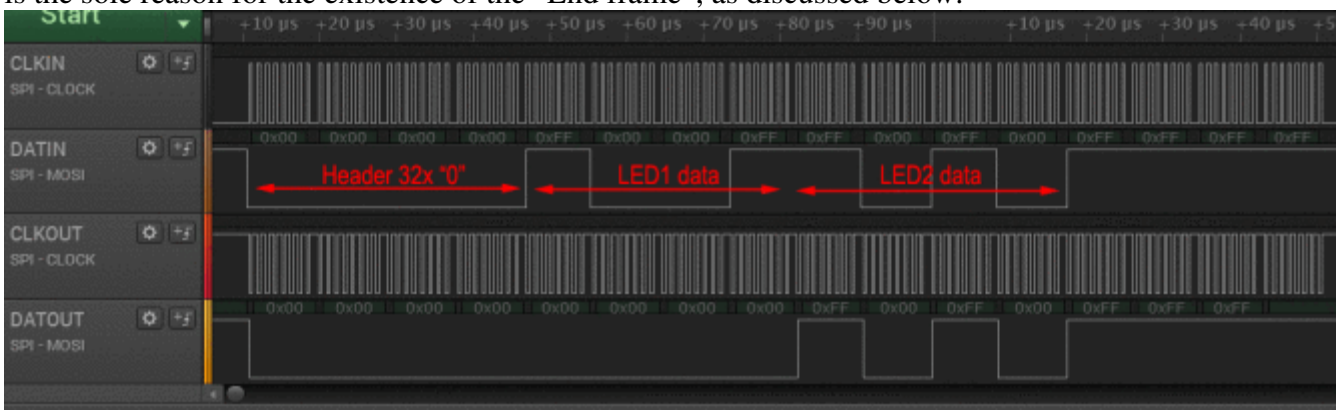
As noted above, the most significant bit of the LED frame has to be “1”, since it is used to identify the start of the frame. It appears that the next two bits serve no function and can have arbitrary values. To stay compliant with the data sheet, it makes sense to set them to “1”, though.

Data forwarding

The APA102 receives a valid SPI signal and outputs a valid SPI signal to the next devices. By definition, the data line is valid only during the rise edge of the clock signal. This resulted in an interesting problem for the APA102 designers. Simply feeding the input signal to the output would not leave enough time to evaluate and possibly alter the incoming data and could create all kinds of race conditions.



To work around this issue, the APA102 delays the data on the output by half a cycle. As shown in the figure above, this is accomplished by inverting the incoming clock signal at the output. The data output is forwarded during the rising edge of the incoming clock, but only becomes valid for the next device at the rising edge of the outgoing clock.
This design is quite ingenuous as it does not require any internal clock source. It does, however, have implications for the protocol: Since the data for each subsequent LED is delayed by half a clock cycle, but the clock is not, additional clock cycles have to be fed to the string even after all data has been sent. This is the sole reason for the existence of the “End frame”, as discussed below.



The diagram above shows how entire LED frames are forwarded from one device to the next one. Once a device detects a start frame (more than 31 zero bits), it will interpret the next “1” bit as start of its own LED frame. 32 bits are clocked into the PWM registers, while zeroes are pushed to the output. After the entire LED frame as been read, any subsequent data is simply forwarded until another start frame is detected.

The End Frame

As we have learned above, the only function of the “End frame” is to supply more clock pulses to the string until the data has permeated to the last LED. The number of clock pulses required is exactly half the total number of LEDs in the string. The recommended end frame length of 32 is only sufficient for strings up to 64 LEDs. This was first pointed out by Bernd in a comment. It should not matter, whether the end frame consists of ones or zeroes. Just don’t mix them.
Furthermore, omitting the end frame will not mean that data from the update is discarded. Instead it will be loaded in to the PWM registers at the start of the next update.

Summary

In summary, each update of an APA102 based LED string should consist of the following:

1. A start frame of 32 zero bits (<0x00> <0x00> <0x00> <0x00>)
2. A 32 bit LED frame for each LED in the string (<0xE0+brightness> <blue> <green> <red>)
3. An end frame consisting of at least (n/2) bits of 1, where n is the number of LEDs in the string.

Unlike the WS2812 protocol, no waiting period is required before the next update. As discussed before, I strongly suggest to only use the full brightness setting (31) to reduce flicker.
I have no recommendation for a maximum or minimum SPI clock speed. There is no specification for this in the datasheet. So far, it seems that the LED is able to handle any clock setting that is thrown at it. I had no issues with 4 MHz and others have successfully tested 10 MHz and above.

Light weight APA102 library

I uploaded an initial release of the líght_apa102 library based on above findings, a companion to the light_ws2812 lib. You can find the code on github.