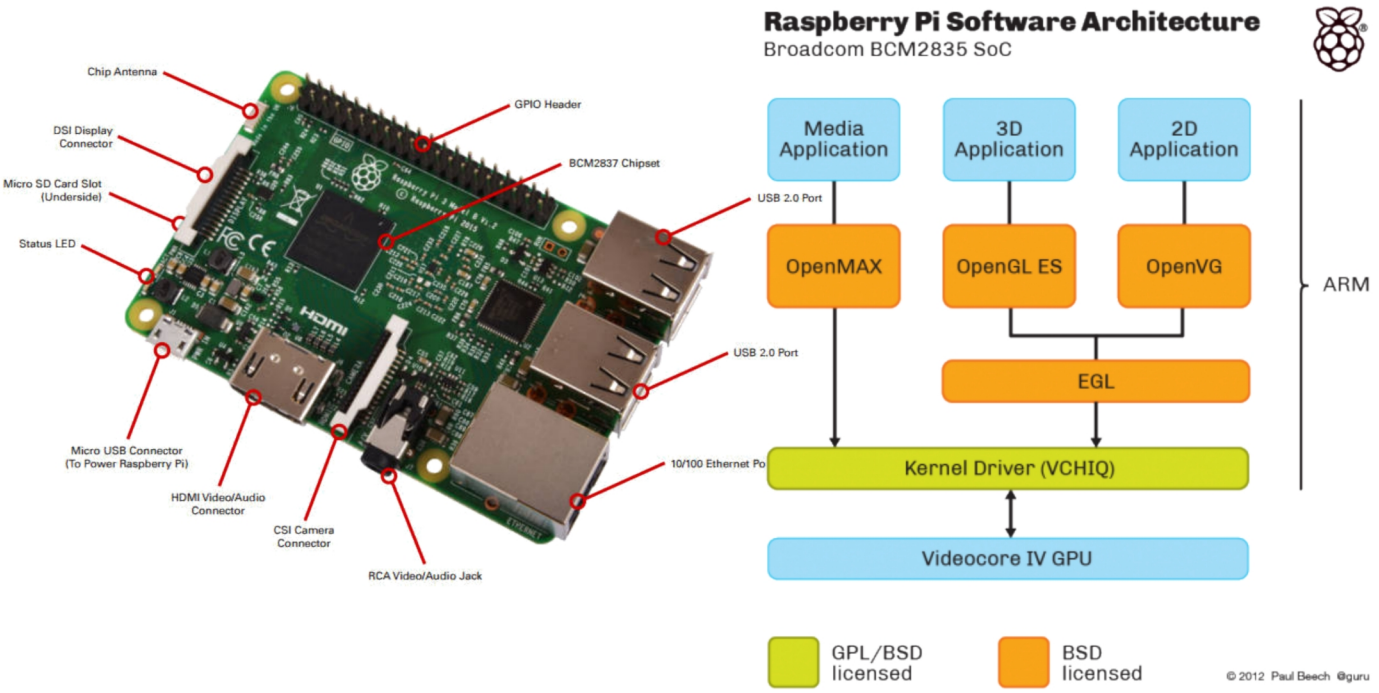


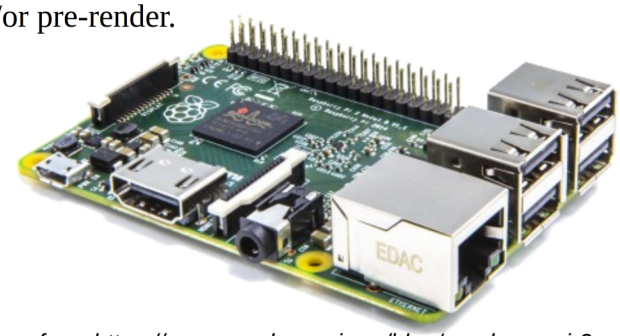
from http://www.raspberrypi.org

Table of Contents

Version 1.0 4/2/2017

1. Introduction

The Raspberry Pi is a nice little single-board computer, but even the RPi 2 with quad cores can be a little anemic for the heavy graphics and math needed with DIYC sequencing (especially effects generation), unless you use all those cores and/or pre-render.



Raspberry Pi 2 - image from https://www.raspberrypi.org/blog/raspberry-pi-2-on-sale/

The on-board Broadcom GPU is rated up to a theoretical *24 GFLOPS*[1]. That is less powerful than other GPUs[2], but it's still a lot of bandwidth for a $35 USD device. I've run strings of individually addressable WS2811/12B pixels using a Microchip PIC16F1688[3] (the heart of the original DIYC Renard controller[4]) which runs at a mere *4.6 MIPS* (18.4 MHz clock). By comparison, the RPi GPU at 24 GFLOPS is over *5,000x faster* (ignoring the architectural differences), so the RPi GPU should be phenomenal at driving smart LED pixels like the WS281X. I set out to test that theory ...

Several others have pioneered the use of a GPU to drive WS281X pixels. In particular, Steve Hardy did some great work on driving WS281X strings using a VGA card[5]. Now, how to do that with a RPi? Obviously we can use an HDMI-to-VGA adapter and the same resistor ladder technique, but that's a bunch of extra parts and it relies on an analog circuit so it's sensitive to resistor tolerances.

It turns out that the RPi provides an *on-board high-speed (up to 500 MHz[6]) 24-bit parallel port!* The GPIO pins can be configured to output the video signal *before* it goes to the HDMI port, allowing massive data output streaming. So, this past Christmas season I ran my display that way - I configured the RPi GPIO pins for VGA output and then drove strings of WS281X pixels directly from the GPU data stream - no data transfer latency, no ethernet ports and cabling, no controller in the mix, just the RPi and WS281X pixels (well, okay, I used a voltage shifter circuit as well). The actual sequencing looked jittery[7], but that was due to my sloppy software (I ran out of prep time ☹) and not because of any hardware control problems. This setup can refresh WS281X pixels at 60 FPS, but I only used 30 FPS.

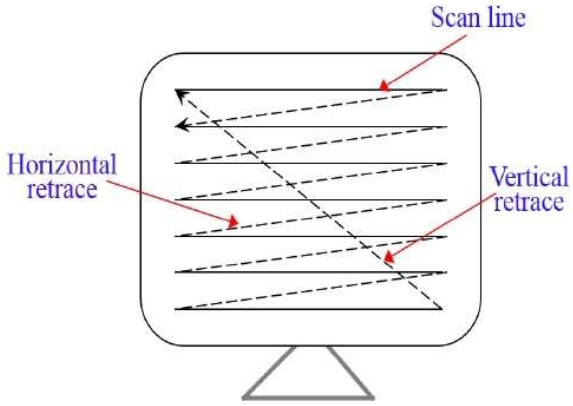
The remainder of this document will describe how I used the GPU to drive multiple WS281X strings. The discussion will be focused primarily on the GPU-related steps with only brief coverage of the background info – that info is already described extensively out on the web so I won't try to repeat it all.

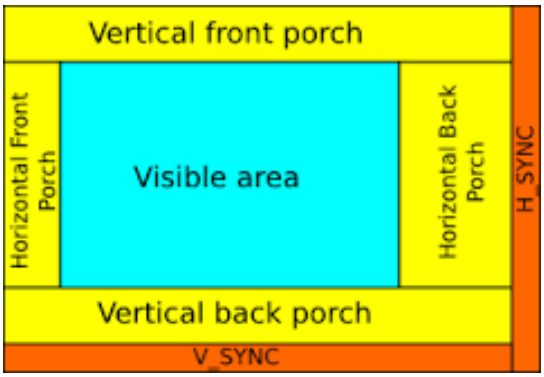
---

1 See GPU performance at https://rpiplayground.wordpress.com/2014/05/03/hacking-the-gpu-for-fun-and-profit-pt-1/
2 See comparison at http://www.geeks3d.com/20150603/quick-benchmark-of-the-raspberry-pi-2-gpu-videocore-iv/
3 Demo at https://www.youtube.com/watch?v=7N23VzZ6srE
4 DIYC Renard described at https://www.doityourselfchristmas.com/wiki/index.php?title=Renard
5 See his article at https://stevehardyblog.wordpress.com/2016/01/02/ws2811-christmas-lighting-using-vga/
6 Max resolution 2560 x 1600 at 120 Hz ≈ 500 MHz pixel clock; see http://elinux.org/RPiconfig#Video_mode_options
7 Demo at https://www.youtube.com/watch?v=pym_7Z5yTqk

2. Theory of Operation

Before looking at how the GPU can be used to control WS281X pixels, first let's look at how it normally displays an image on the screen. Today's screens resemble CRTs from years back, in which an electron beam swept rapidly back and forth, from top to bottom according to the path shown below on the left:





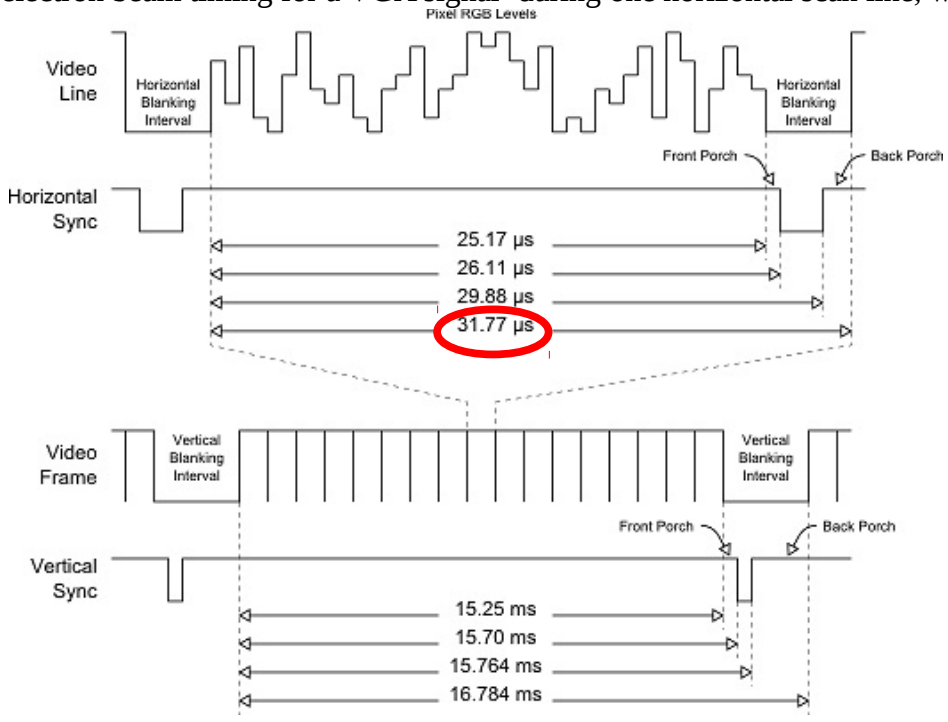
https://www.tutorialspoint.com/computer_graphics/computer_graphics_quick_guide.htm *http://nathandumont.com/blog/vga-primer*

Due to distortion near the edges, only part of the screen is used for the image. The unused areas serve special purposes as labeled in the diagram above on the right. In particular, the horizontal and vertical Sync pulses are used to reset the position of the electron beam (corresponding to the "retrace" lines).

Now if we look at the electron beam timing for a VGA signal[8] during one horizontal scan line, we see:

Timing for one horizontal row:

Timing for one entire frame:



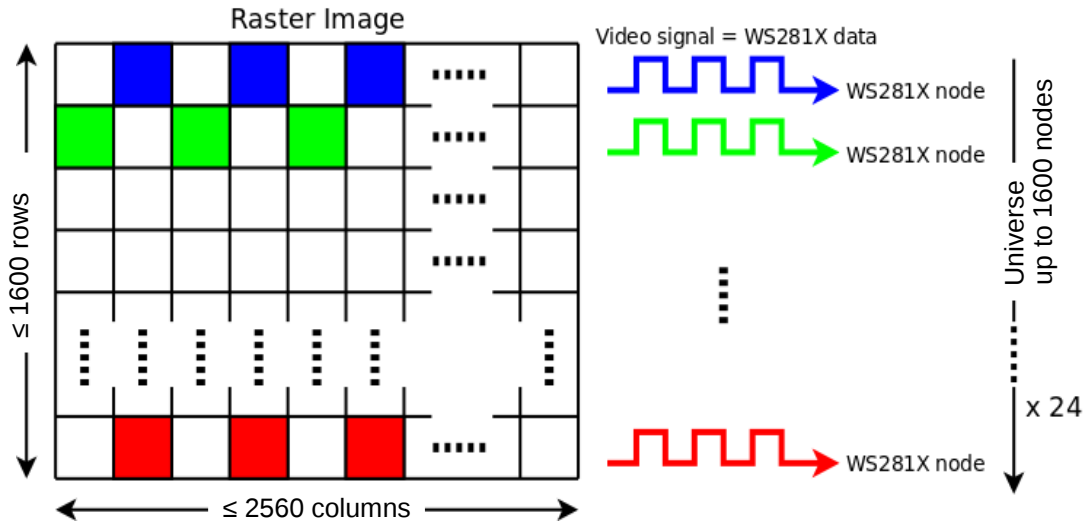
http://electronics.stackexchange.com/questions/166681/how-exactly-does-a-vga-cable-work

8 Industry standard VGA timing: 640 x 480 @ 60 Hz, see http://tinyvga.com/vga-timing/640x480@60Hz

Notice that a horizontal scan line takes 31.77 usec – approximately the same time as a WS281X pixel at 800 KHz (24 x 1.25 usec = 30 usec). Most display cards allow timing to be adjusted ± to accommodate different monitors or screen formats. *That's the magic ingredient* - the VGA signal can be adjusted to *exactly* match the 30 usec needed by the WS281X protocol!

Once the horizontal timing is correct, we simply need to draw an image representing a WS281X data signal onto the screen, and then send the video signal to WS281X LED pixels instead. In fact, if we draw one image per display line, we can control as many WS281X pixels as the screen height allows, without the need for an external WS281X controller. There are a few little complications such as the horizontal and vertical Sync pulses and signal voltage levels to deal with, but those are easily overcome.

The RPi 2 even allows the VGA or HDMI signal to be output as a 24-bit RGB value onto the GPIO pins. This means we no longer need an analog resistor ladder to recover the data, and we can directly control up to *24 strings* of WS281X LEDs (one string on each GPIO pin). With a max display resolution of 2560 x 1600 at 60 Hz, this means the RPi GPU can directly control up to 24 strings ("universes") of 1600 pixels each = 38,400 WS281X pixels[9] at 60 FPS, without an external controller:



Up to 24 "universes" of 1600 nodes = 38,400 WS281X pixels (= 115,200 channels)

Now each WS281X pixel only needs 24 bits of data, so with 2560 pixels per scan line on the screen there would actually be 2560 / 24 ≈ 106x as much data coming from the GPU as is needed. An external latched 1-of-106 decoder could therefore be added to each GPIO pin to control up to a total of 24 x 106 universes of 1600 nodes = 4,070,400 WS281X pixels[10]! (theoretically)

I did not actually try to run with external decoders to verify the ≈ 4 mega-pixel limit - the ≈ 40K pixel count available without external decoders was already more than I needed. ☺ However, I did start experimenting with a ≈ $10 USD Xilinx CPLD board as a 1-of-30 latched decoder, which gives ≈ 30 universes from one GPIO pin so I think the ≈ 4 M-pixel limit is actually possible with a bigger decoder.

9 By adjusting the vertical sync and front/back porch, you can even go a little higher than this.

10 Based on pixel count. However, refresh rate or pixel clock would need to be adjusted to maintain 30 usec per scan line.

3. Detailed Steps

Below are the high-level steps that I used[11] to set up the RPi GPU to drive WS281X pixels:
1. Install an O/S
2. Install Node.js (or other tools)
3. Install dependent libraries (non-RPi, optional)
4. Choose display parameters (number of "universes", max "universe" size, FPS, etc)
5. Configure RPi GPIO pins for video output
6. Connect WS281X pixels directly to the GPIO pins (might need voltage shifter)
7. Create and run a sequence

The main steps of interest for setting up the RPi GPU to control WS281X pixels are steps 4 - 6, but I will briefly cover the other setup steps as well for completeness.

If you just want to try out the GPU concept first on a regular desktop of laptop PC without a RPi, just use steps 2, 3, and 7 only. Then you can go back through the entire list of above steps with a RPi.

3.1. Install O/S

There are several good O/S choices for the RPi. The most popular appear to be Raspbian (a trimmed down Linux), other flavors of Linux, Windows 10, and some embedded O/S choices. I chose Ubuntu MATE because it runs on Intel (desktops and laptops) and ARM v7 (RPi 2) - I wanted the same O/S across machines, for consistency. The remainder of this document will assume the 16.04.2 LTS version because of the long term support ("LTS"), even though version 17.04 was already out as of this writing.

Most of the RPi setup procedures seem to be oriented around Raspbian and/or NOOBS (New Out Of the Box Software) which is an O/S installer, but I did not need those since I would be installing a different O/S. I downloaded Ubuntu MATE and copied it to the SD card using info from the following links:

> https://ubuntu-mate.org/raspberry-pi/
> https://ubuntu-mate.org/download/
> https://www.raspberrypi.org/documentation/installation/installing-images/README.md

Running the RPi O/S from a USB stick is reportedly faster and less prone to corruption than running it from an SD card, so then I put the O/S onto a USB stick. The complete process to do that is shown at:

> https://www.youtube.com/watch?v=tSGA-63FniU

However, since I had already copied the O/S onto an SD card, I just made another copy of it onto the USB stick. It's redundant, but it gave me a fall-back copy in case the USB copy becomes damaged.

There is no BIOS on the RPi, so it reads **cmdline.txt** from the PI_BOOT partition to find the O/S. I used one of the steps from the YouTube procedure above to tell the RPi to load the O/S from the second partition of the USB stick instead of the SD card. The relevant change to **cmdline.txt** was as follows:

> ... root=**/dev/sda2** ...

11 I've taken the liberty of rearranging them into a more logical order; I didn't follow that order the first time around. ☹

I also wanted to do a headless install (no monitor or keyboard), but that required a network cable to the router so instead I just used the regular startup procedure with a monitor and keyboard.

I powered up the RPi and answered all the questions. It started up successfully and even connected to the wifi network. Then it downloaded a bunch of updates (over wifi) and got a package error, but it was able to show the login prompt and I was able to log in. After login, I opened a terminal window and ran **raspi-config** as suggested by the Ubuntu MATE install procedure (first link, above). Under Boot options, I turned off the GUI to conserve resources, and under Interface options I enabled SSH to allow remote access. I also used **ifconfig** to get the IP address – easier to do this before rebooting. 😄

After reboot, the RPi started in console mode, and I was also able to log in with **ssh** remotely over wifi, which was very convenient. To do this, you need to find the IP address of the RPi (MAC address is also useful in case of **arp** problems). There are various ways to do that, as described here:

https://www.raspberrypi.org/documentation/remote-access/ip-address.md

After logging in, I used the following commands to update the software and get a required library:

```
sudo  apt-get  update
sudo  apt-get  upgrade
sudo  apt-get  install  libfreeimage-dev
```

It's probably also worth updating the firmware. The following links provide more info about that:

https://linhost.info/2015/05/update-the-firmware-on-a-raspberry-pi/
https://github.com/raspberrypi/firmware/
https://github.com/raspberrypi/userland/

One further adjustment I made was to create a RAM disk. This seemed like a good idea for reducing wear and tear on the USB stick. I used 200 MB instead of 1 GB, but the procedure is described at:

http://www.hecticgeek.com/2015/12/create-ram-disk-ubuntu-linux/

BTW, the RPi seems to have an annoying habit of refusing to connect after a while (related to an **arp** problem). If **ssh** does not respond and "arp -n" shows that the RPi MAC address is not in the routing table, then try the following command to manually restore it:

```
sudo  arp  -s  192.168.1.7  74:da:38:7e:1c:74          (substitute correct IP + MAC address)
```

3.2. Install Node.js

As with the O/S, there are several good choices for sequencing software. **xLights**[12] is a great choice, or you can write your own using languages such as Python[13], Perl, C++, or just about anything else. I chose Node.js and JavaScript because I wanted a simple, flexible, light-weight and efficient platform that didn't require specialized development tools – only a text editor is needed. I also wanted something easily understood and customized by others with minimal programming experience, so they could freely experiment with their own new effects. (I also plan to use a web based UI myself, so JavaScript seemed

12 xLights documentation and download info is at http://xlights.org/
13 Used by Steve Hardy; link given earlier.

like a logical choice to handle browser and server sides). The remainder of this document will assume the 6.10.0 LTS version of Node.js, even though there is a version 7.6.0 already out as of this writing.

The simplest way to install Node.js is with **apt-get**, but the version packaged with Ubuntu MATE was a few versions back (4.2.6 according to **apt-cache policy**). I wanted to run a more recent version, so I downloaded and unpacked the current LTS binaries from **nodejs.org** using the following commands:

```
wget  https://nodejs.org/dist/v6.10.0/node-v6.10.0-linux-armv7l.tar.xz
tar  -xvf  node*.tar.xz
```

However, the binaries did not include a script to actually install. Then I found a Node.js installer script[14] on **github** showing how to install the Node.js binaries so I used the following commands based on that:

```
sudo  mv  node*7l  /opt/nodejs/
sudo  ln  -s  /opt/nodejs/bin/node  /usr/local/bin/node
sudo  ln  -s  /opt/nodejs/bin/npm  /usr/local/bin/npm
```

This gave me an installed copy of Node.js and **npm**, but not **nvm**. To get an installed Node.js that is controlled by **nvm** (if you care), the following commands could be used instead of the above commands:

```
sudo  apt-get  install  build-essential  libssl-dev  curl
curl  -sL  https://raw.githubusercontent.com/creationix/nvm/v0.31.0/install.sh  -o  install_nvm.sh
bash  install_nvm.sh
source  ~/.profile                          #or log out + in
nvm  ls-remote                              #shows which versions are available
nvm  install 6.10.0
nvm  use 6.10.0
node  -v
nvm  ls                                     #show versions that are installed
nvm  alias  default  6.10.0                 #set default version (optional)
```

3.3. Non-RPi Libraries (optional)

For development/test purposes, I also installed Node.js and **npm** on a non-RPi machine. In order to run the demo programs (described later), the following additional libraries needed to be manually installed:

- GLEW (http://glew.sourceforge.net/)
  GLEW handles OpenGL extensions.
- GLFW (http://www.glfw.org/)
  GLFW opens a window, creates an OpenGL context, and manages input.
- AntTweakBar (http://www.antisphere.com/Wiki/tools:anttweakbar)
  AntTweakBar adds a light GUI to interactively tweak parameters on-screen.

These libraries are cross-platform and should work anywhere that supports OpenGL or OpenGL ES, but I only needed to install them on my non-RPi dev machine. (separate install not needed on the RPi)

To install them on Ubuntu MATE, I used the following steps:

14 See https://github.com/audstanley/NodeJs-Raspberry-Pi

```
#install missing dependencies:
sudo apt-get install libxrandr-dev libxinerama-dev libxcursor-dev libfreeimage-dev libxi-dev
sudo apt-get install build-essential libxmu-dev libxi-dev libgl-dev libosmesa-dev

#GLEW:
wget https://sourceforge.net/projects/glew/files/glew/2.0.0/glew-2.0.0.tgz
tar -xf glew*.tgz
cd glew-2.0.0
make
sudo make install
sudo mv /usr/lib64/* /usr/lib
cd ..

#GLFW:
wget https://github.com/glfw/glfw/releases/download/3.2.1/glfw-3.2.1.zip
unzip glfw*.zip
cd glfw-3.2.1
cmake .
make
sudo make install
cd ..

#AntTweakBar:
wget https://sourceforge.net/projects/anttweakbar/files/latest/download?source=dlp
mv *dlp anttweakbar.zip
unzip ant*.zip
cd AntTweakBar
cd src
make
sudo cp ../include/* /usr/local/include
sudo cp ../lib/libAntTweakBar.* /usr/local/lib
sudo ldconfig
cd ..
```

3.4. Choose Display Parameters

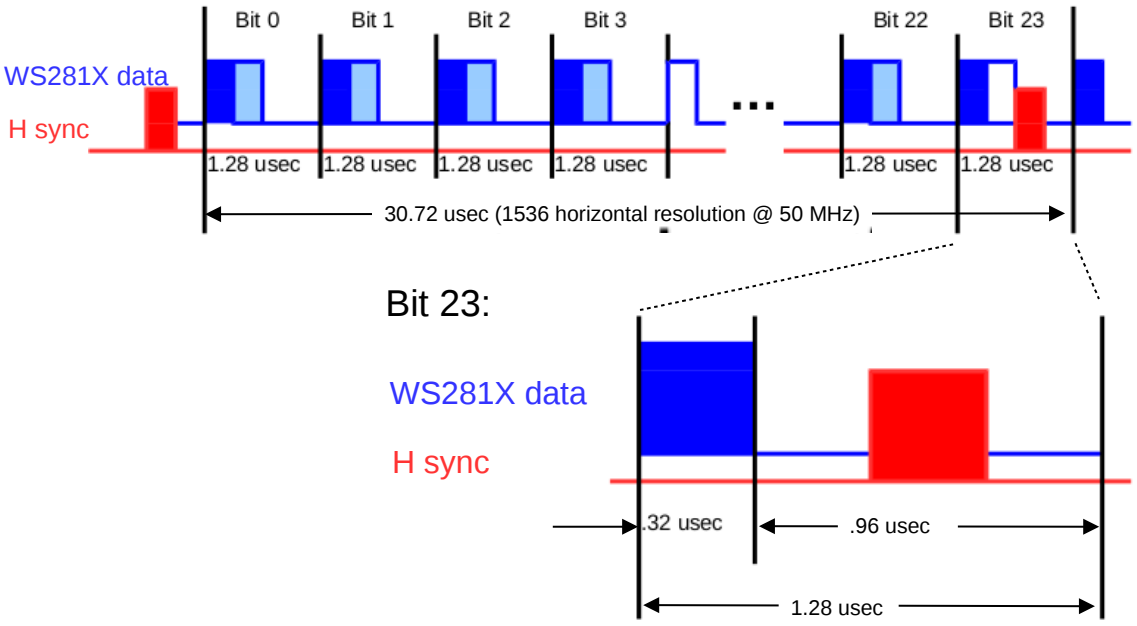
Various RPi GPU configuration parameters affect the video signal timing and must be chosen to accommodate the target number of WS281X pixels while still conforming to the timing spec:

What	I used	Purpose/Explanation
Pixel clock	50 MHz	Display pixel clock frequency
Display width	1536	Horizontal display resolution (including H sync + front/back porch); should be a multiple of 24 for WS281X protocol
Display height	1152	Vertical display resolution (including V sync + front/back porch); value - 2 determines max WS281X string length (“universe” size) and FPS
H retrace	48	12+12+24 = sync + front/back porch; sync should be ≤ 25% WS281X bit time
V retrace	48	12+12+24 = sync + front/back porch; must be ≥ 2 (50 usec WS281X latch)
FPS	30	1536 x 1152 @ 50 MHz ≈ 28.26 FPS (nominally 30 FPS)

There can be multiple choices for achieving any given target number of WS281X pixels. The following reasoning led to my choice of the values shown above:

- My 11 foot M-tree had 36 x 32 = 1152 pixels (I only set up 34 x 32 = 1088 of those). I wanted to run the whole tree using one data line (just because I could ☺), so I chose my "universe" size (display height) to be 1152 nodes (should have been 1154 due to 2 node times used for 50 usec latch signal).
- I read about the GPU automatically correcting the aspect ratio and I didn't want that to happen, so I chose a display width of 1536 (maintains a 4:3 aspect ratio). This is also a multiple of 24 (bits).
- With no external memory buffer, displaying 1536 pixels in a horizontal scan time of 30 usec (for WS281X protocol) requires a pixel clock of 51.2 MHz. I rounded this down to 50 MHz for a nice round number, which was still within the WS281X timing spec[15] (30.72 usec instead of 30 usec).
- Sending 24 WS281X data bits using a horizontal display resolution of 1536 uses 64 display pixels per WS281X bit. The WS281X spec requires the first ≈ 25% of the bit time to be high, which limits the H sync + front/back porch time to the remaining 75% of the bit time, or 48 display pixels.
- Based on other HDMI timing examples, I chose H front porch = H sync = 12, H back porch = 24.
- For consistency I also chose V front porch = V sync = 12, V back porch = 24.

Using the values shown above, the GPU will generate WS281X timing as follows:



WS281X signal waveform, H sync positioned to avoid interfering with high part of last data bit

The H sync and front/back porch must fall within the low portion of the WS281X bit time in order to not interfere with the WS281X data stream (video signal is low during Sync periods). This leads to the requirement that H sync + front/back porch total time be ≤ 75% of the WS281X bit time (the trailing 0.96 usec of the bit time in the above timing diagram).

15 https://cdn-shop.adafruit.com/datasheets/WS2811.pdf and https://cdn-shop.adafruit.com/datasheets/WS2812.pdf

FWIW, some additional notes:

- H Sync appears to have a 1 KHz rounded amplitude when H porch + Sync ≈ 20. It's a lot squarer when ≈ 176. (using my cheapo 'scope). This suggests that higher is better?
- There doesn't seem to be any VGA output with pixel clock < 32 MHz[16]?
- A target row time of ≈ 30 usec with a "universe" size of ≥ 1K rows limits frame rate to ≤ 33 Hz
- H blank will interfere with LSB of each WS281X node; limit H blank to 4% of row time (1/24)
- V blank should be >= 2 row times (≥ 50 usec)
- VGA 640x480 @60 Hz with a 27.175 MHz pixel clock meets the target row time
- SVGA 800x600 @56 Hz with 36 MHz pixel clock also meets target row time

3.5. Configure GPIO pins

GPU configuration is controlled by **/boot/config.txt**, which is read when the RPi boots up. This file can be edited with any text editor. The various parameters are described here:

https://www.raspberrypi.org/documentation/configuration/config-txt.md
https://www.raspberrypi.org/documentation/hardware/raspberrypi/dpi/README.md
https://www.raspberrypi.org/documentation/configuration/device-tree.md

After choosing GPU display parameters (previous section), I made the following changes to **config.txt**:

- Disable SPI and I2C (to avoid pin conflicts)
- Enable DPI24 device tree overlay (sends VGA signal to GPIO pins)
- Set display resolution and sync using values from previous section (controls video signal timing)
- Adjust other settings (GPU memory, framebuffer, etc)

I did this by adding a section of new entries at the end of the **config.txt** file:

```
#WS281X settings:
gpu_mem=128
[0xDEADBEEF]
dtparam=spi=off
dtparam=i2c_arm=off
hdmi_force_hotplug=1
hdmi_ignore_edid=0xa5000080
dpi_group=2
dpi_mode=87
hdmi_timings=1488 0 12 12 24   1104 0 12 12 24    0 0 0 30 0 50000000 1
#? hdmi_drive=4
#? hdmi_ignore_cec_init=1
#x dpi_output_format=0x6f007
####dpi_output_format=0x117
dpi_output_format=7
#? sdtv_aspect=1

#dtdebug=on
# dtoverlay=vga666                (this one might also work)
dtoverlay=dpi24
```

16 Possibly related to pixel clock limitations at: https://www.raspberrypi.org/forums/viewtopic.php?f=29&t=112735

```
enable_dpi_lcd=1
display_default_lcd=1
#? config_hdmi_boost=4

disable_overscan=1
overscan_left=0
overscan_right=0
overscan_top=0
overscan_bottom=0
framebuff_width=1488
framebuff_height=1104

#eof
```

These are in the **config-example.txt** file in **github**. Note that duplicate entries override previous entries, so I put this at the end of the file and didn't need to edit entries earlier in the file.

Setting **dpi_output_format** to 0x117 instead of 7 can be useful for timing debug – the H sync and V sync will be sent to GPIO 2 and 3, which can then be used as triggers when using an oscilloscope. You would normally turn these off and then be able to use those GPIO pins for other purposes.

There is a choice of whether to send 16, 18, or 24 bits of VGA/HDMI data to the GPIO pins (I used 24):

- RGB565 format = uses 16 GPIO pins, leaves 8 for other purposes
- RGB666 format = uses 18 GPIO pins, leaves 6 for other purposes
- RGB888 format = uses 24 GPIO pins, leaves none (well, I guess GPIO 2 and 3 could be used)

The **hdmi_timings** line is where the parameters from the previous section go. They might also go in the **framebuff_*** entries. (I'm not sure if those are needed).

I added the **[0xDEADBEEF]** line to more easily allow switching the GPU back and forth between WS281X timing and regular HDMI timing (so a monitor can be connected). Comment out that line (by putting a "#" in front of it) for the WS281X settings to take effect, or uncomment it (remove the "#") to disable the WS281X settings.

The RPi must be rebooted after changing the **/boot/config.txt** file. After rebooting, the following commands will check the current GPU display settings (for the settings shown earlier, the expected output is in ***blue***):

```
vcgencmd  get_mem  gpu                  shows amount of memory for GPU
gpu=128M
/opt/vc/bin/tvservice  -s               shows current display resolution
state 0x400000 [LCD], 1488x1104 @ 0.00Hz, progressive
sudo  vcdbg  reloc  |  awk  '/ref/{sum+=$12}END{print  sum}'                    shows memory used[17]
5259776  or  6704758
```

BTW, if the HDMI display goes blank (due to power saving or other reasons), it can be forced to turn back on using the following commands:

17 For additional memory usage info, see https://www.raspberrypi.org/forums/viewtopic.php?f=67&t=23185

```
tvservice  -o
tvservice  -p
```

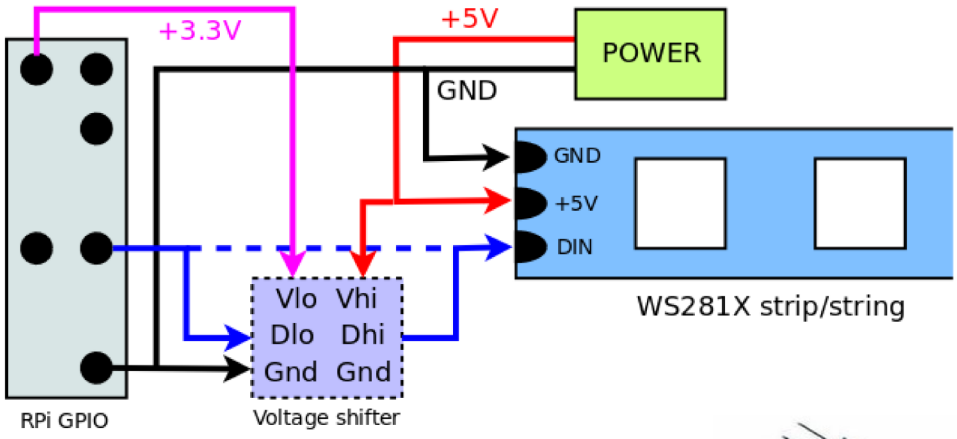
3.6. Connect WS281X pixels

After the GPU has been configured for WS281X timing, screen pixels will be sent to the GPIO pins according to the following pin-outs:

Mode	RGB bits	GPIO																							
		27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4
1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	565	-	-	-	-	-	-	-	-	7	6	5	4	3	7	6	5	4	3	2	7	6	5	4	3
3	565	-	-	-	7	6	5	4	3	-	-	7	6	5	4	3	2	-	-	-	7	6	5	4	3
4	565	-	-	7	6	5	4	3	-	-	-	7	6	5	4	3	2	-	-	7	6	5	4	3	-
5	666	-	-	-	-	-	-	7	6	5	4	3	2	7	6	5	4	3	2	7	6	5	4	3	2
6	666	-	-	7	6	5	4	3	2	-	-	7	6	5	4	3	2	-	-	7	6	5	4	3	2
7	888	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0

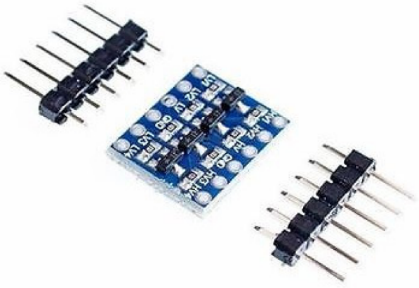
GPIO VGA pin-out from https://www.raspberrypi.org/documentation/hardware/raspberrypi/dpi/dpi-packing.png

During initial testing, I was able to connect a piece of WS281X LED strip directly to a GPIO pin without the need for a voltage level shifter (**blue dashed line** below) - probably because the first WS281X pixel was very close to the RPi. However, for longer strings/strips further away, I found that a voltage level shifter circuit (shaded, below) was needed:



GPIO connected directly to WS281X, or through a voltage level shifter

I used MOSFET voltage level shifters similar to the following, which I found on **ebay.com** for ≈ $1 USD:



http://www.ebay.com/itm/IIC-I2C-Logic-Level-Converter-Bi-Directional-Module-5V-to-3-3V-For-Arduino/172347428527

Another option is to run the WS281X pixels at a lower voltage – say, in series with a heavy duty diode. For example, a 1N000# style diode would drop the voltage by about 0.7V. Running the WS281X pixels at 5 – 0.7 = 4.3V would drop the voltage level threshold for a logic "1" to 4.3 * 0.7 ≈ 3V, which is below the 3.3V sent out on the GPIO pins. However, this would leave less tolerance on the signal voltage, so the supply voltage would need to be injected more frequently on longer WS281X strings/strips to avoid the pinkish tint on full-white pixels. Several other voltage shifting options are also available[18].

3.7. Install/Run a Sequence

Once the hardware is all set up (or if you are just trying out the GPU approach without any WS281X LEDs or RPi hardware), it's time to have some fun and try out some sequencing! I used JavaScript and a simple text editor, but you can use whatever software or tools you want. The description below assumes JavaScript/Node.js, and the **config.txt** settings described earlier.

I wrote a few little test programs, which I've put out on **github** if anyone would like to play with them. They demonstrate the basic techniques and serve as sample/demo programs. Use the following commands to install the **git** command line interface and then the sample programs:

```
sudo  apt-get  install  git
git  clone  https://github.com/djulien/FriendsWithGpu-WS281X.git
cd  FriendsWithGpu-WS281X
npm  install
```

The demo programs are all run from the command line, from within the the **FriendsWithGpu-WS281X** folder created by the above commands. They can also be used *without a RPi or WS281X pixels*, if you just want to experiment with the idea of using the GPU from a regular desktop or laptop PC. On a non-RPi machine, the additional libraries described a few sections back will need to be installed first.

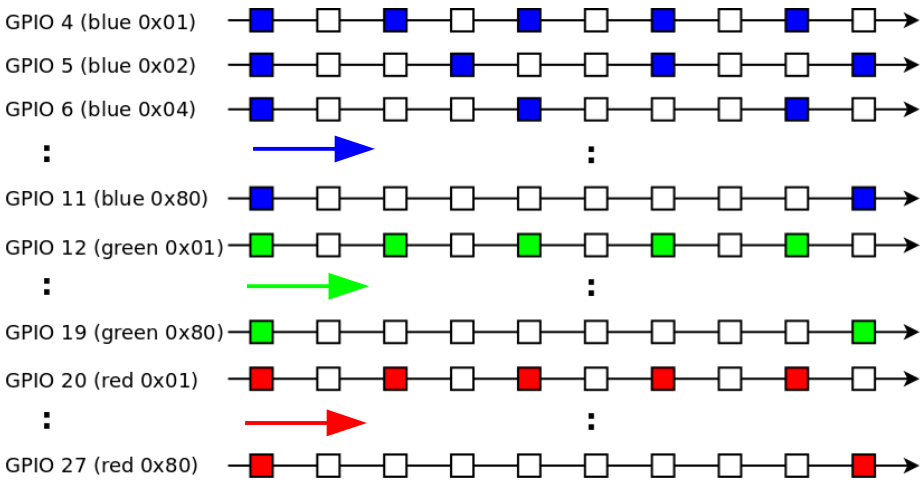
3.7.1. RGB Finder

The first demo program verifies that the config settings and hardware are working, and also serves as "proof" that the GPU technique actually works. ☺ To run the program, enter the following command line from within the **FriendsWithGpu-WS281X** folder that was created in the previous section:

```
DEBUG=*  demos/rgb-finder.js                    (omit "DEBUG=*" to get less info on the console)
```

This will send a different animated pattern to each GPIO pin, according to the pin-out chart shown earlier (using RGB888 format):

18 Several options are described at http://ww1.microchip.com/downloads/en/DeviceDoc/chapter%208.pdf

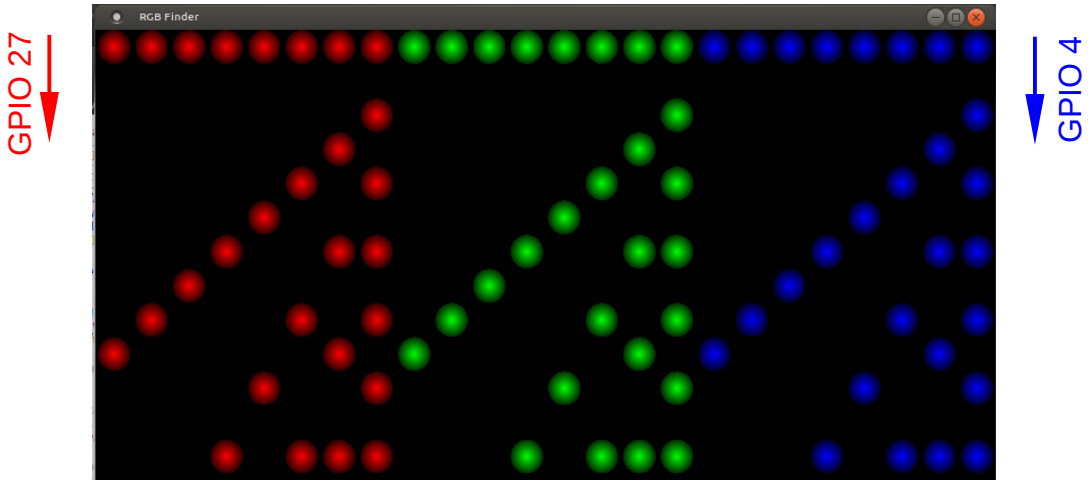


RGB Finder pattern – different animated pattern on each GPIO pin

The color of the LEDs indicates which byte within the 24-bit RGB value is being used, and the gap size between lit LEDs indicates the power of 2 or the bit position (1 … 8) – see above, and then compare to the pin-out chart in the previous section. This gives a different pattern on each GPIO pin. I found it very convenient to just connect a string of WS281X LEDs to any of the GPIO pins and then know which GPIO pin it was. 😄

The pattern is also animated (moving down the screen and "away" from the GPU), to help verify if there are any intermittent hardware issues (also allows the CPU load to be measured). The speed of animation can be adjusted with the SPEED constant near the start of the file. With the **config.txt** settings shown earlier, the animated pattern will show on the first 1102 WS281X pixels of each universe[19].

If the RGB Finder program detects that video output is not redirected to the GPIO pins or it is not even running on a RPi, it will just simulate the WS281X pixels on the screen. This allows the JavaScript logic and OpenGL interface to the GPU to be tested:



RGB Finder pattern – WS281X LEDs simulated on screen

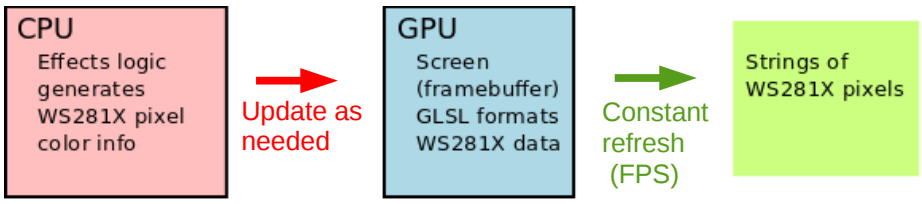
19 Other protocols such as E1.31 or DMX use this term to refer to groups of connected RGB pixels, so I'll use it too. 😄

In this case, strings of WS281X pixels are represented as columns of glowing LEDs on the screen, with each column of lights connected to a different GPIO pin. Due to display limitations, only the first 24 LEDs of each string are shown (abbreviated in screen shot above) - this can be adjusted with the **VGROUP** constant near the start of the file. The LEDs at the top of the screen are closest to the GPU, so the animated pattern will travel down the screen ("away" from the GPU).

The display columns are in left-to-right order according to the pin-out chart shown in the previous section – GPIO 27 is at the left, and GPIO 4 is at the right.

This demo program is structured like traditional sequencing software, in that the "effects" (the graphics patterns) are generated by logic running directly on the CPU, and the colors are then packaged and sent to the WS281X pixels (via the GPU through the GPIO pins). In this particular case, the effects are written in JavaScript, but any other language could be used.

Within the program, the GPU is abstracted as a rectangular canvas, and the effects logic just turns pixels on/off as desired using (x, y) coordinates. While the CPU is generating the RGB Finder patterns, the GPU constantly refreshes the screen or WS281X LEDs at ≈ 30 FPS (if the example **config.txt** settings shown earlier are used). In this respect, the GPU is acting like a constant refresh "WS281X dongle":



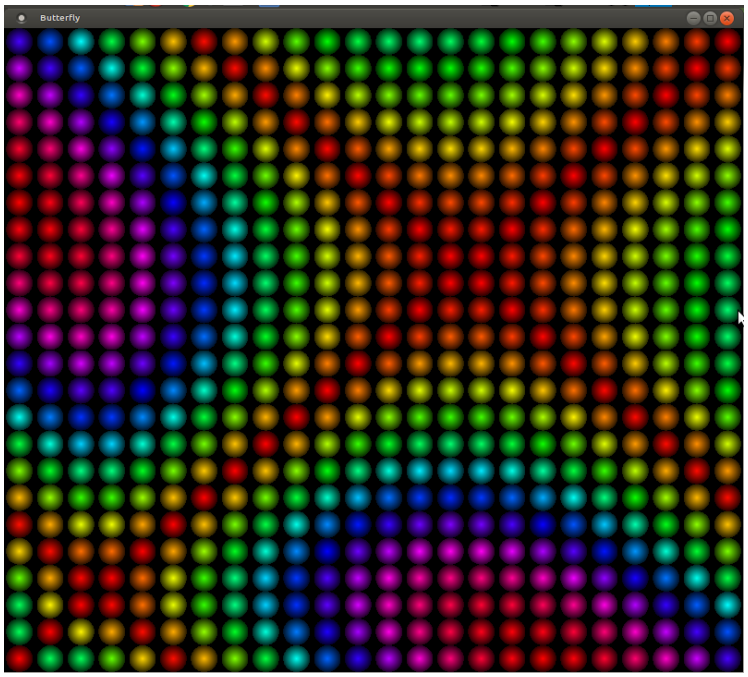
RGB Finder architecture – effects generated on CPU

3.7.2. Butterfly

The second demo program generates the equivalent of the xLights Butterfly effect (with pre-selected options). To run the program, use the following command line:

```
DEBUG=*  demos/butterfly.js                    (omit "DEBUG=*" to get less info on the console)
```

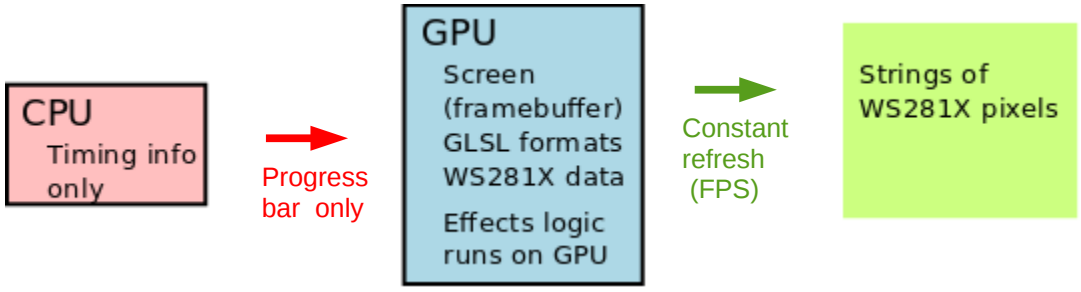
As with the previous demo program, the Butterfly program will send the generated patterns to the GPIO pins if running on a RPi with video output redirected, or just to the screen otherwise:



Butterfly effect – WS281X LEDs simulated on screen

In either case, Butterfly assumes a 24 x 24 grid so it will only light up the first 24 WS281X pixels on each GPIO pin. This can be easily changed to drive a larger rectangular grid or from fewer GPIO pins – I just didn't bother setting it up that way for this demo.

Internally, the Butterfly demo program operates very differently from the previous test program. The CPU only sends minimal frame timing info to the GPU (for the progress bar), while the actual Butterfly effect is generated entirely on the GPU. As a result, the CPU usage is considerably lower than if the patterns had been generated on the CPU. You can see this by commenting out the **OPTS.gpufx** constant near the start of the file – when commented out, the effects are generated on the CPU (using ≈ 14% of one RPi core), and when uncommented, the effects are generated on the GPU (using ≈ 4% of one RPi core). This particular effect is not very demanding, but on a more complex effect the difference in CPU load would be more significant.



Butterfly architecture – effect generated on GPU, only progress bar timing from CPU

3.7.3. One by One Stepper

I've included the third demo program for anyone who wants to look into the details of the WS281X formatting a little more, and what the GPU is actually doing to generate those 24 parallel WS281X data signals. To run the program, enter the following command line from within the folder created earlier:

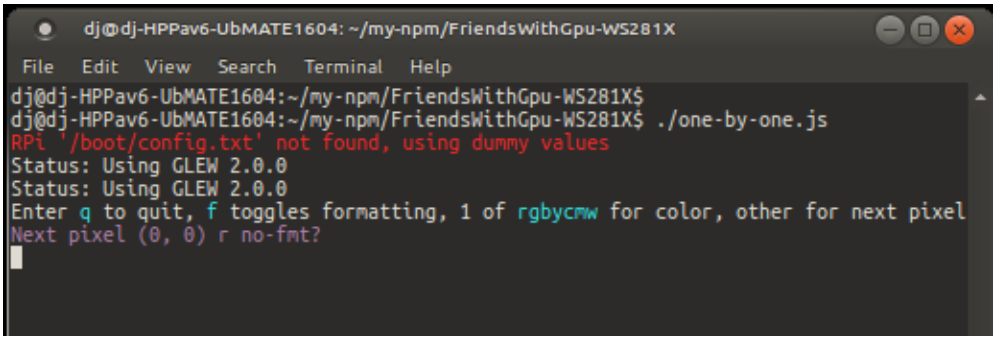
```
DEBUG=*  demos/one-by-one.js                    (omit "DEBUG=*" to get less info on the console)
```

The purpose of this demo is to turn on one WS281X LED at a time, using the keyboard to single step through each successive LED. As with the other demo programs, if the screen is not redirected to the GPIO pins or it's not even running on a RPi, it will simulate the LEDs on the screen instead. In fact, this demo program is actually more interesting when output is *not* being redirected to the GPIO pins or running on a RPi – it's helpful to see how the formatted WS281X data appears on screen rather than just sending it out to the WS281X LEDs.

When the program starts, it will display an intro screen for 10 seconds[20] (if the screen has not been redirected to the GPIO pins). During the wait, a progress bar will be displayed along the lower edge:

One-by-one title screen, progress bar while waiting

After the 10 seconds, it will clear the screen and then prompt for keyboard input before turning on any WS281X pixels:



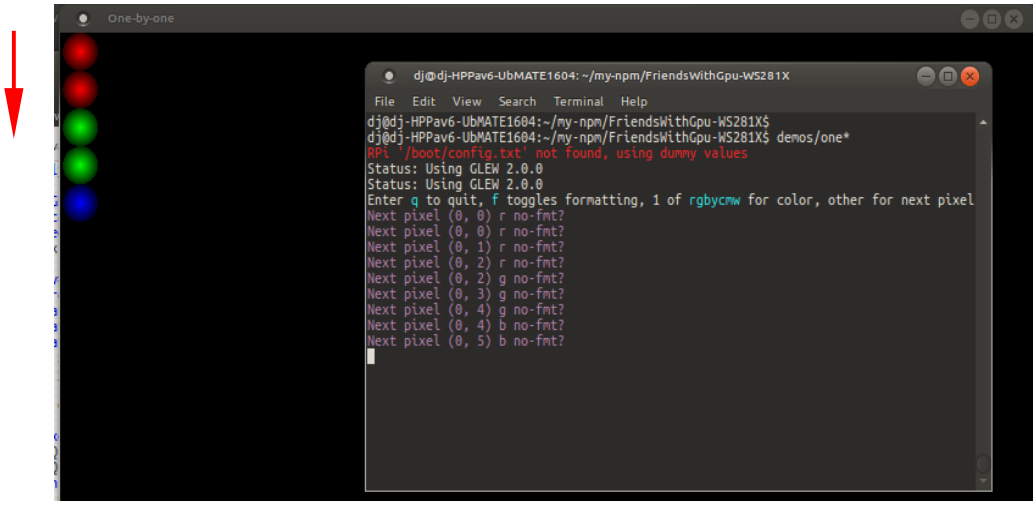
Console prompt before turning on any LEDs

20 No real reason. I just needed an image load function for other sequencing, so I thought I'd include it in the demo. 😊

The “Next pixel” prompt shows which (x, y) LED will be turned on next, with (0, 0) representing the top left corner. Any of the following single-letter commands can be entered in response:

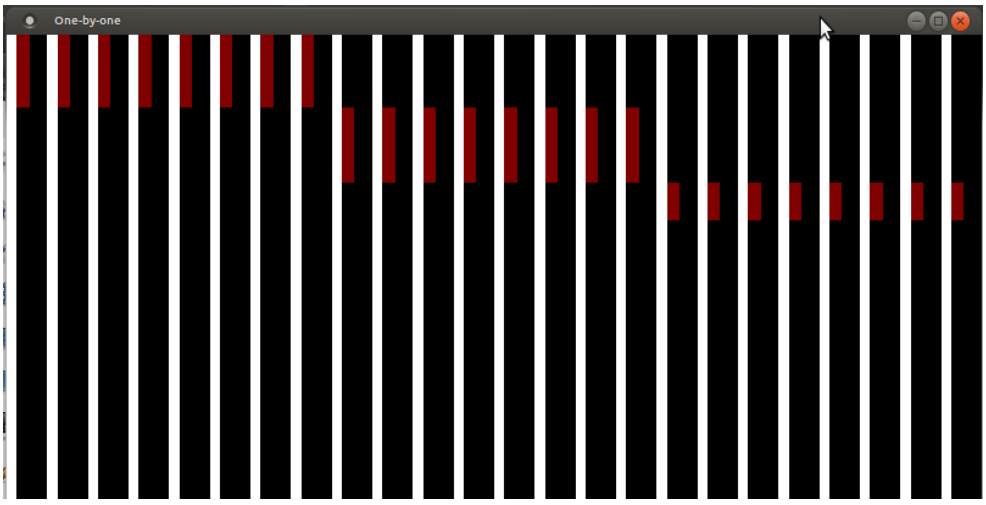
- r/g/b/y/c/m/w Selects color of next WS281X LED. I only implemented the primary colors, since their RGB values are easier to recognize (3-byte combinations of 0xFF and 0x00).
- f Toggles WS281X formatting. This is very useful for comparing the simulated LEDs to the actual formatted WS281X data. Switch back and forth to compare.
- q Quit
- all else Turn on next LED using the currently selected color.

For example, if you enter the command sequence “r, Enter, Enter, g, Enter, Enter, b, Enter”, the 1st and 2nd LEDs will be set to red, the 3rd and 4th to green, and the 5th one to blue:



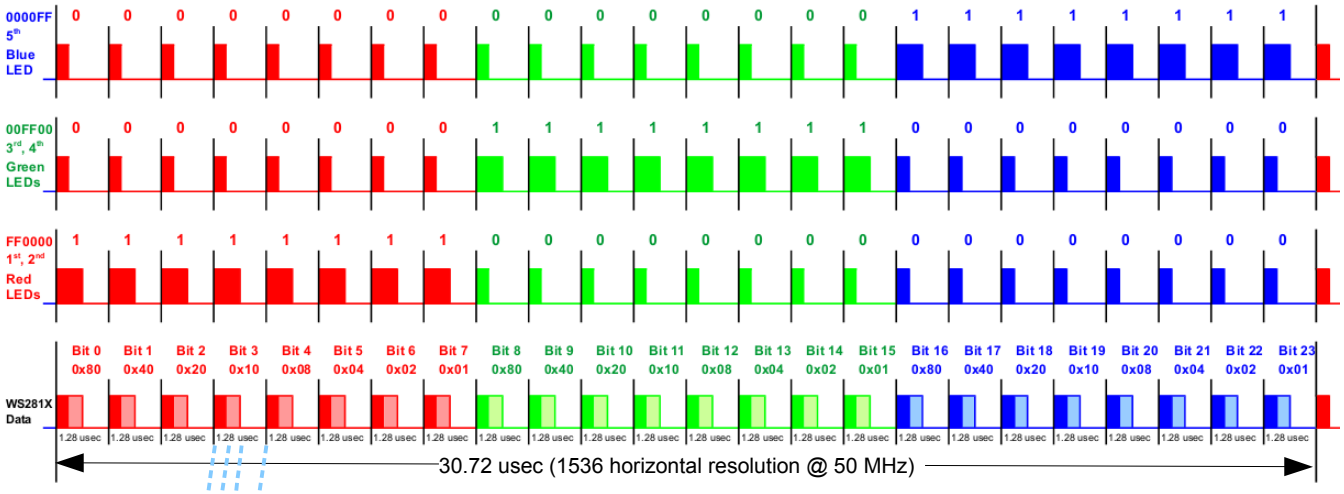
Set LED colors while single-stepping

Now for the interesting part - if you then hit the “f” key, it will toggle the WS281X formatting on/off:



WS281X formatting turned on while single-stepping

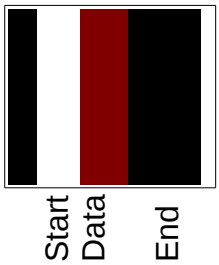
Referring back to the VGA timing diagram in Chapter 2 shows how the WS281X signal on the screen corresponds to the data stream expected by the WS281X LEDs:



Red, green, blue WS281X data patterns

For example, the 1st and 2nd LEDs on the screen are red, so the WS281X pixels need 24 bits sent to them as 0xFF0000. The "0" bits have a narrow logic high while the "1" bits have a wider logic high so we see 8 wide bits followed by 16 narrow bits:

Red LED bit pattern shown on screen – 0xFF0000 (8 wide bits, then 16 narrow bits)



You'll notice that each bit starts with a white area, followed by a variable color area, and ending with a black area – this is because all 24 GPIO pins must be set high at the start of the bit time (0xFFFFFF is white on the screen) and low at the end of the bit time (0x000000 is black on the screen), but in between some GPIO pins might be high and some might be low, depending on the LED color in each universe. In the example above, the first universe has a red LED and other universes are dark, so the RGB value on the screen is 0x800000. (Open a screen shot in an image editor to verify this). For the second universe, the color would have been 0x400000, or for the 8th universe it would have been 0x010000, etc.

There are many other subtleties in this and the other demo programs, which I am not going to try to cover here – there are just too many details, and I'm trying to cut down on excessive documentation. ☺ Hopefully I've covered enough info to give the general idea. If you have any questions, feel free to ask. BTW, to get additional on-screen debug info, set the **WS281X_DEBUG** option near the top of the file.

4. Conclusions

This was an interesting experiment, but very successful. It was somewhat challenging to figure out how to set things up initially, but once it was all set up I found it very easy to connect WS281X pixels to the

RPi, and to create and change my display sequences (JavaScript and text editor). I only got the rudimentary stuff working, but now that I'm past that point it should be easier for the next display season.

Some possible extensions to this approach are:

- Build a full sequence player using the GPU; also an OpenGL-based designer/preview tool (3D)
- a GLSL Hacker or Jupyter Notebook style sandbox for 3D models and effects
- CPLD-based external decoder (FPGA is probably overkill)
- RPi Hat with voltage level shifters and/or RS485 drivers

I've done some initial prototyping on those – they are all doable, but if anyone would like to help, please feel free! ☺

A couple of other interesting observations during this project ... I don't mean to imply that infinite God can be put in a box, but I did notice how the comparison of GPU vs. refreshing the WS281X LEDs, Jesus is always there and listening to us, and He has infinite, parallel compute resources compared to our slow, single-threaded selfs.

5. Revision history

Version	Date	Purpose
1.0	4/2/2017	First version finally written up and published after some code clean-up

-eof-