# Raspberry Pi Synth Eyes

Updated 2023/07/25  (v3.04)

This project was originally intended to provide RGB eye displays for Synths, Protogens and similar robotic costumes which use a display for a face.  The original hardware design used two Unicorn Hat HD panels driven by two linked Raspberry Pi devices.

Since then it has also gained support for driving MAX7219 LED matrices using a single Pi and other display systems may well be added in future.

As such, there are now *several* guides for assembling the hardware.  Feel free to skip over the ones which don’t apply.

# Software Version History

**Version 3.00**

- Initial release, with GIF and scrolling message support, plus GPIO trigger events.  Full feature parity with the earlier 2.xx software.

**Version 3.01**

-Adds smooth scrolling support, procedural blinking, animation looping, colour cycling effect, experimental support for MAX7219 display panels.

**Version 3.02**

-Support lighting arrays (e.g. for horn/ear status lights on a Synth) which can be modulated by a microphone, experimental widescreen MAX7219 driver, improve GPIO pin reservation, add experimental support for servo motors.  Allow 16 actions per expression instead of 8.

**Version 3.03**

-Display rotation support

**Version 3.04**

-Experimental Raspberry Pico support with MAX7219 and Hub75 panel drivers, experimental SDL fullscreen support (for HDMI output), GPIO virtual ground, experimental CAP1188 sensor support

# Unicorn Hat Hardware Setup

For this project you will need the following:

2x Raspberry Pi Zero **without the headers installed**
2x Pimoroni Unicorn Hat HD display panels (make sure it's the HD version)
2x MicroSD cards 2GB or larger
1x 2-pin jumper cap
...and a computer with an SD card reader of some kind.

The Unicorn HD panel is designed to plug directly into a Raspberry Pi. Pretty much any Pi will work, but to save space I recommend using the Pi Zero 2W boards. Development was originally done on the older 1.3 boards and these will work, but the Pi Zero 2 boots a lot faster.

If, down the line I can figure out a way to drive multiple panels off a single board things will be a lot simpler. For now, we need to use two Pi Zeroes, and in order to link the two machines together (and provide inputs for changing expressions) we need to customise the 40-pin headers.

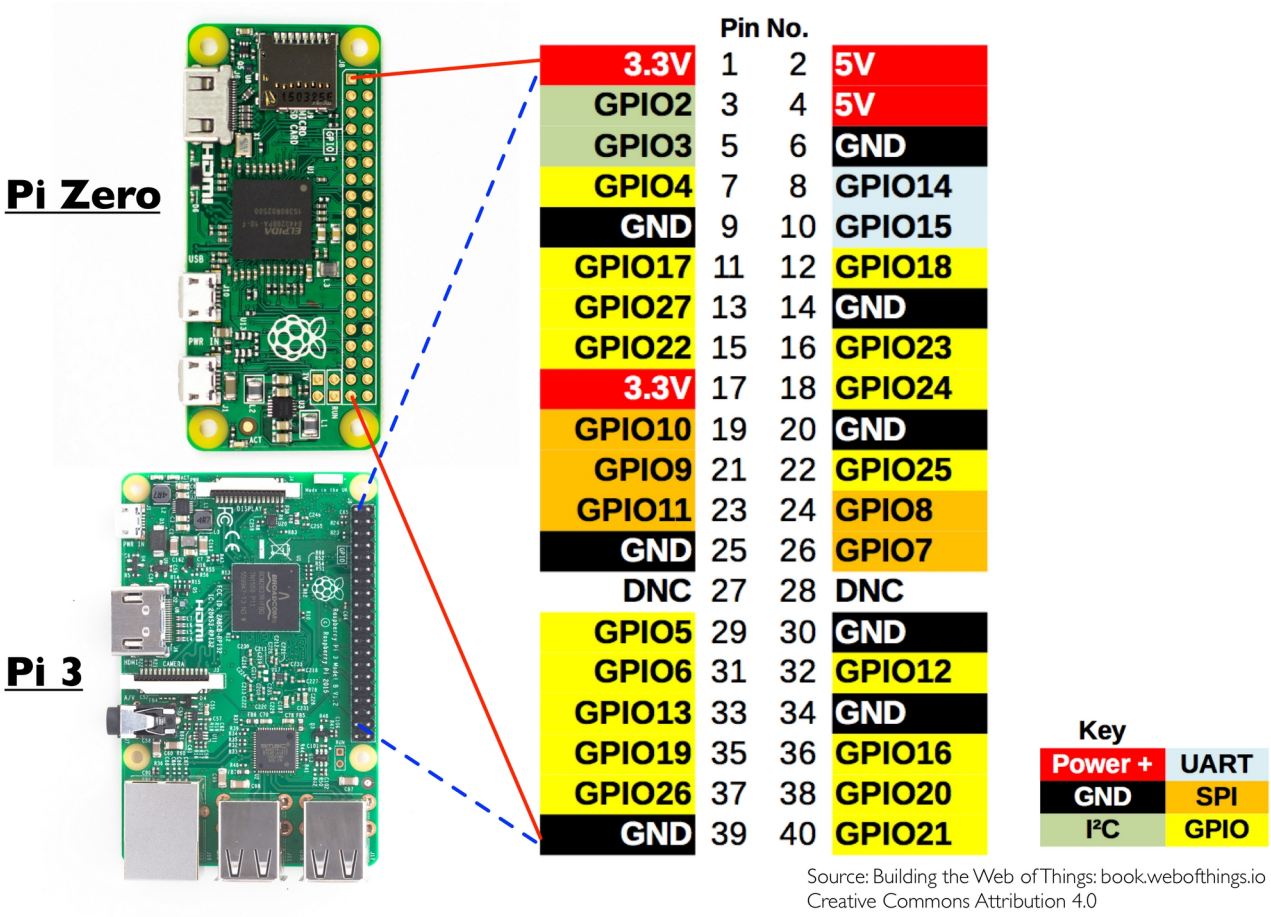
For this reason it is highly recommended that you purchase the boards with the headers provided, but *not* pre-installed (they are usually cheaper like that anyway). Otherwise, you will have to desolder some of the header pins to rearrange them and this is likely to damage the Pi unless you are very careful.

Because we have two separate devices, one of them will need to be the transmitter, and the other one must be configured to listen for commands.

With the W model of Pi Zero it might be possible to do this wirelessly, but I would expect problems if you have two such Synths next to each other – also it is likely to break down in case of heavy interference such as a furry convention full of bluetooth devices.

## Wiring the headers

Here is the pinout for the Raspberry Pi Zero:



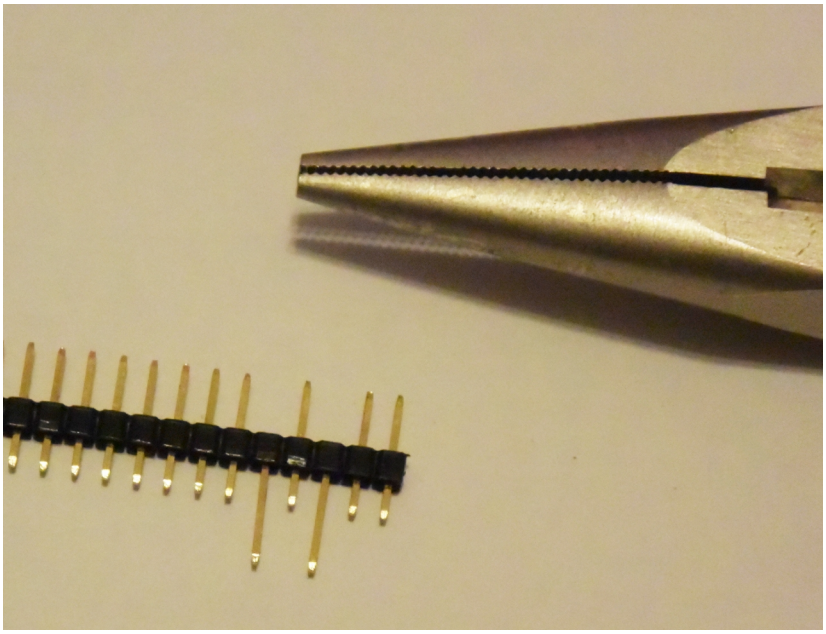

Source: Building the Web of Things: book.webofthings.io
Creative Commons Attribution 4.0

We need to wire both devices slightly differently.

## Wiring the Receiver

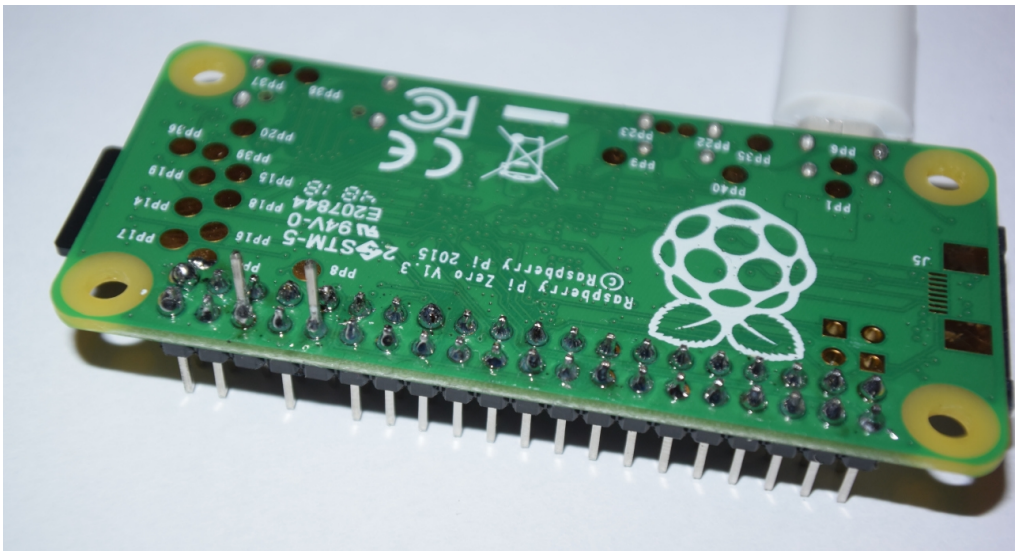
The Receiver is easiest so we'll look at that first. The above picture shows the top of the device – the header is inserted this side so the pins are sticking out the top. However, to provide a link to the other device, we will need to have two pins sticking out the bottom as well.

Use needle-nosed pliers or some other tool to push pins 6 and 10 into the black frame, e.g.



Make absolutely sure that it is pin 6 (GND) and not pin 4 (5V)... If you connect 5V to the other Pi, you will most likely have destroyed that I/O pin and will need to buy a replacement Pi.

Then, solder all 40 header pins onto the board. When it's done, it should look like this:



You may find it helpful to tack an LED to pins 39 and 40 on the back of the Receiver board. This will flash briefly to indicate that an expression change command has been sent, since it may not be obvious to the wearer if they managed to trigger it successfully. Wire negative to pin 39 (GND).

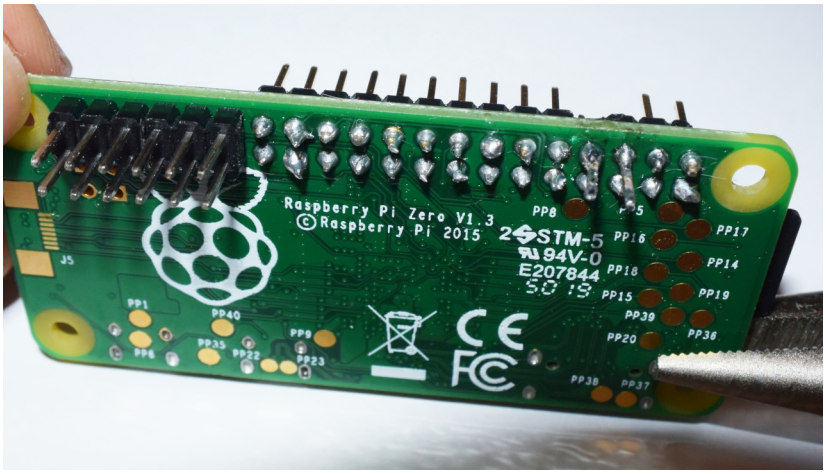
## Wiring the Transmitter

Next, the Transmitter.  This is more complicated.  As before, you will need to push a number of pins through the connector so they come out of the bottom of the device.

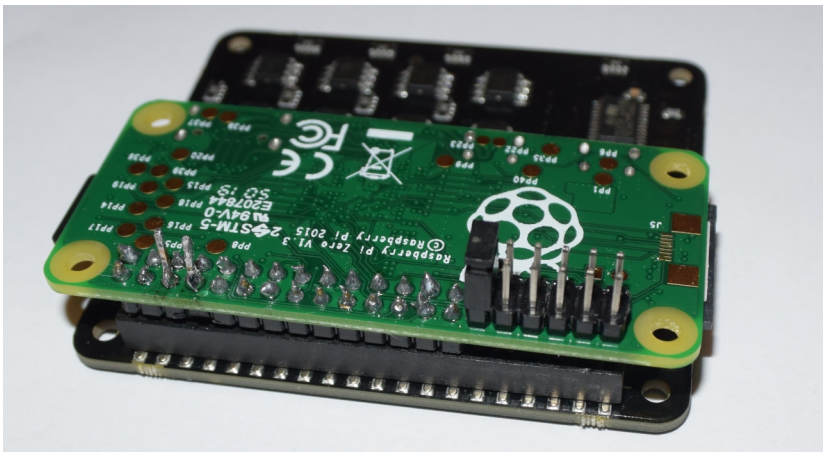
This time we need pins 6 and 8 – which are adjacent to each other.  Again, be sure it's neither of the first two 5V pins since the Pi can't handle 5V logic levels.

1. In addition, we also need access to pins 29-40 – the 6 pairs of pins furthest from the SD card reader.  We need these pins to be poking out the bottom of the circuit board, but because they are all grouped together it is best to cut the header at that point with a pair of sidecutters so that these 12 pins can be soldered on the rear of the board.

Again, solder all 40 pins.  When it's done it should look like this:




At this point you can plug the Unicorn Hat boards into the Pi Zeros, e.g:



Now.  Both Pi Zeros are running the exact same software.  To tell the Transmitter unit that it's going to be the transmitter, we need to bridge pins 29 and 30.  You could solder these two pins together, but I use a jumper cap of the kind often found in Desktop PCs – as shown in the above picture. This will be more future-proof in case we need those pins for some other purpose later on.

## Expression Control Pins

Pins 34 and 39 are ground.  If you temporarily bridge one of these pins to one of the surrounding pins it will trigger an expression change, as with the Arduino software.

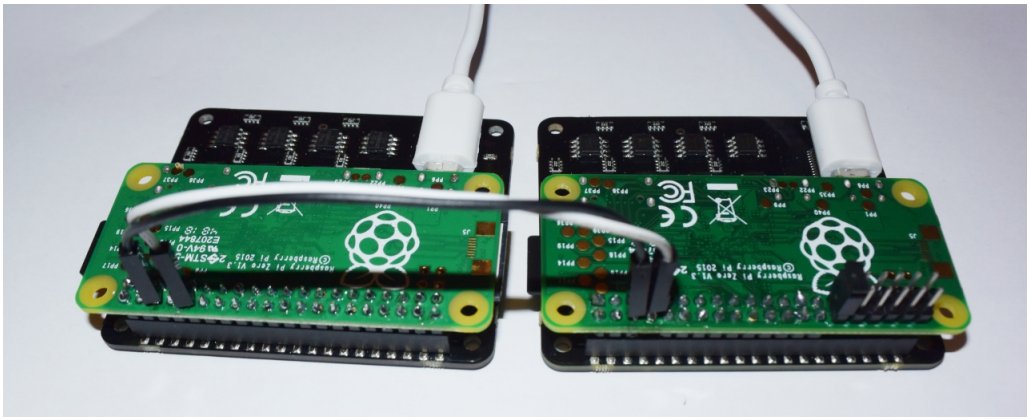
As of this writing, (with v3.00 of the software) the following pins do the following expressions by default, depending on the config file used:

| **Pin** | **Xerian** | **Raptor's Den** |
|---|---|---|
| 29 | - | - |
| 31 | OwO | OwO |
| 32 | Annoyed | Annoyed |
| 33 | Eyeroll | Annoyed (no eyeroll yet) |
| 36 | Fault (X eyes) | Fault (X eyes) |
| 37 | Blushing | Blushing |
| 38 | Happy | Happy |
| 40 | Startled | Startled |

Note that pin 35 is configured for *output* – do not short it to ground!  It can be used to change the colours of the status lights in 'Fault' mode to make them flash red (if the Arduino driving the Synth's status lights is programmed to do this).

# Linking the two devices

The last step hardware-wise is to connect the two Pi Zeroes together.  This is fairly straightforward: Connect pin 6 on one Pi to pin 6 on the other.  Then connect pin 8 on the transmitter to pin 10 on the receiver.  Unless you get them crossed over this should be pretty foolproof.



# MAX7219 Hardware Setup

For this project you will need the following:

1x Raspberry Pi Zero, preferably with headers
2x MAX7219 display panels (32x8 with 4 matrices each – make sure they have sockets!)
plus at least two 5-pin Dupont cables (these usually come with the panels)
1x MicroSD cards 2GB or larger
...and a computer with an SD card reader of some kind.

With the Unicorn hat, the display plugs directly into the Pi. With the MAX7219 panels, they will be plugged into the GPIO header via 5 cables. This means that the entire GPIO header is exposed and we don't have to do the thing where we resolder some of the pins so they're accessible out the back of the device. We also don't need the RX and TX pins since the entire system can run off one board.

**Note that the software will default to the Unicorn display driver!** You will need to edit the config file in order to use the MAX7219 matrix panels, and you will need at least 3.01 of the software, otherwise it will display gibberish. See the software configuration section later.

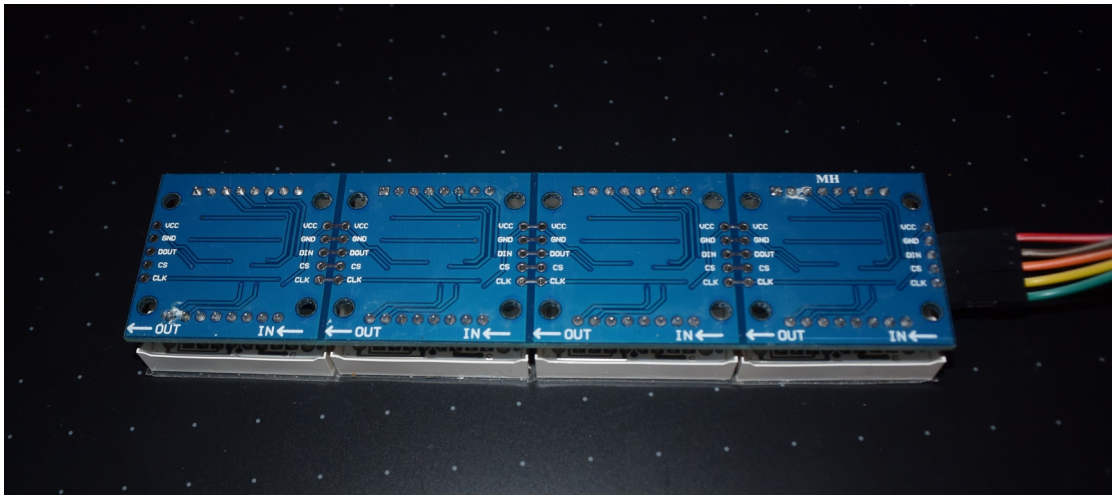
While the Pi side of things requires less soldering than the Unicorn system, the display panels do have to be modified to break them down from 32x8 into 16x16 panels.

With v3.02 there is an experimental driver (MAX7219W) which can run *four* of these 32x8 boards in a twin 32x16 configuration. This requires no modification at all, but the display will be larger (reducing vision in the helmet) and the wiring will be slightly different.

**Note that Raptor's Den has commissioned a set of custom 16x16 PCBs which will not require modification. If these are available, they should save a lot of time and effort.**

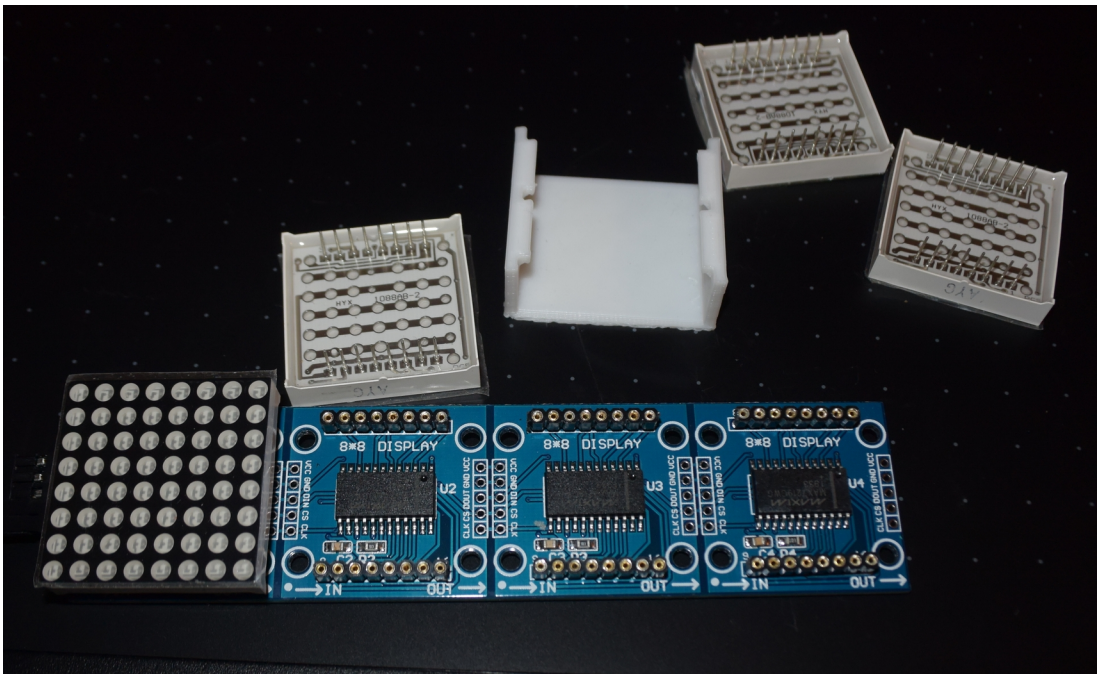
## Modifying the board for 16x16 displays

The MAX7219 display boards consist of four separate modules, each of which has holes in the circuit board for attaching more connectors. Since it is obviously designed to be cut into shorter sections, that is what we will do.

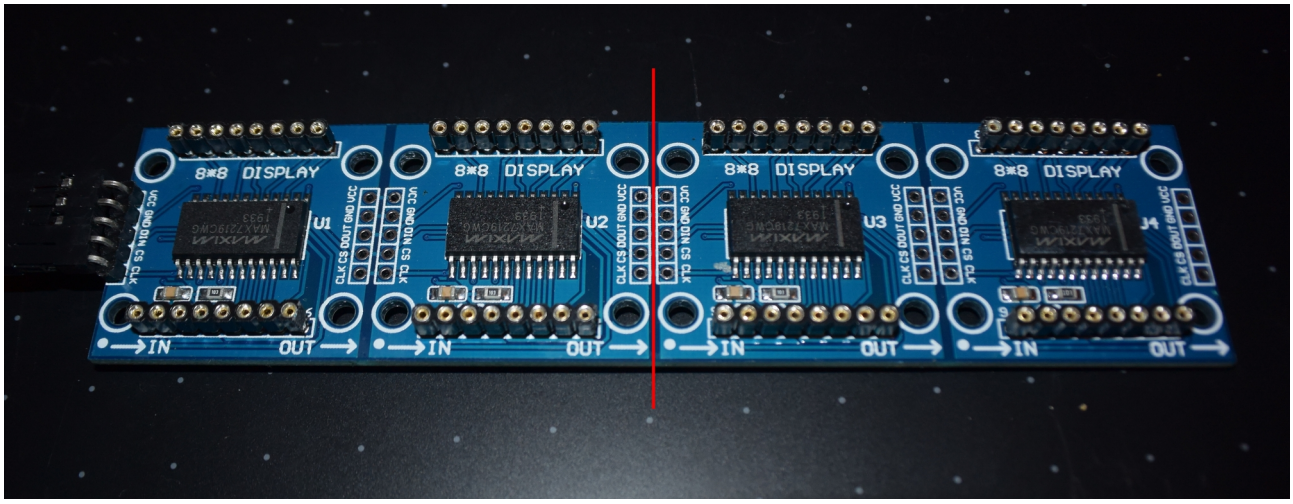


First, remove the matrices from the circuit board. This is easiest if you have a special tool, but can be done simply by levering them out if you're careful not to bend the pins. Mark the panels before removing them so you know which side is up – you can orientate using the text silkscreened onto the circuit board – I usually mark the tops of them.

If you have a 3D printer, a removal tool can be downloaded here:
https://www.thingiverse.com/thing:3282538




When the panels are removed, take a Stanley knife, box-cutter or whatever they're called in your country, and score the thin line dividing the middle two panels. A metal ruler may help.

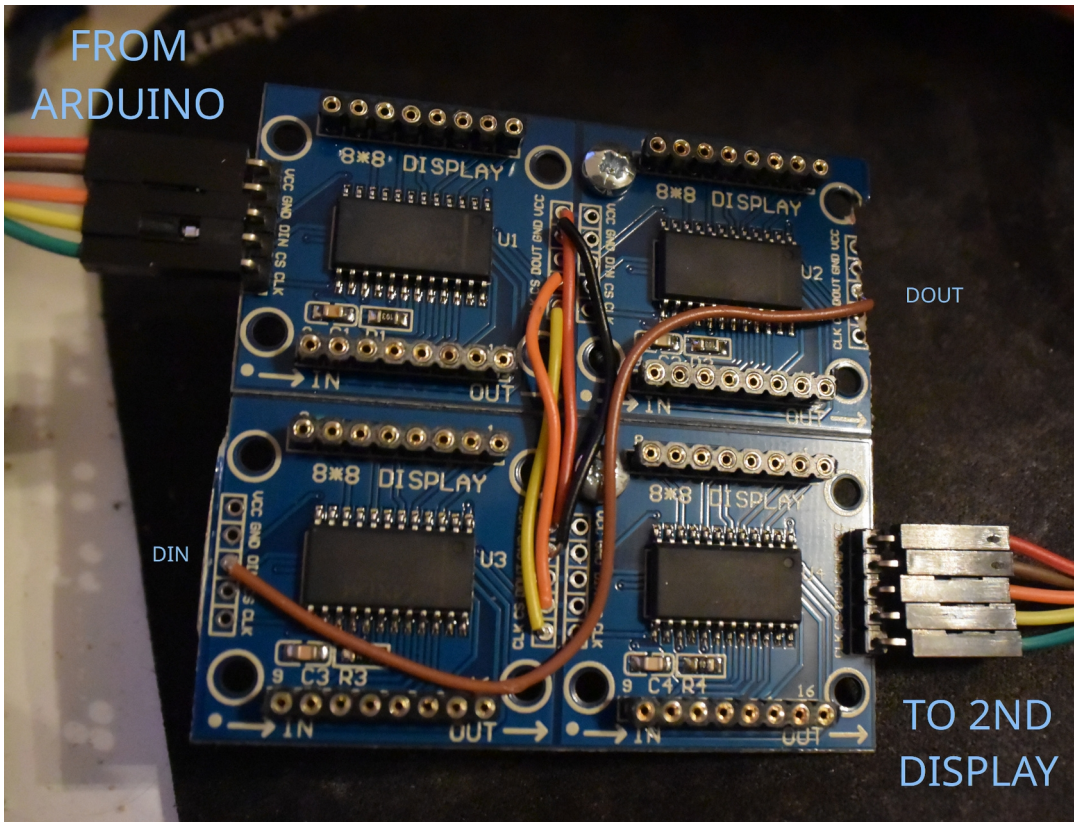



Once you have a suitable line, run the knife back and forth to dig down into the PCB. Do this evenly on both sides until the board is thin enough to snap cleanly in half.

While it is possible to cut them with a saw, note that this will increase the risk of copper particles shorting the connectors and leaving you with a non-working section of the display until this is fixed.

Once the circuit board has been cut in half, you will need to link the two halves together.  One approach is to simply solder 5-pin connectors onto the ends and link them with the Dupont leads that the Arduino and Pi devices typically use.

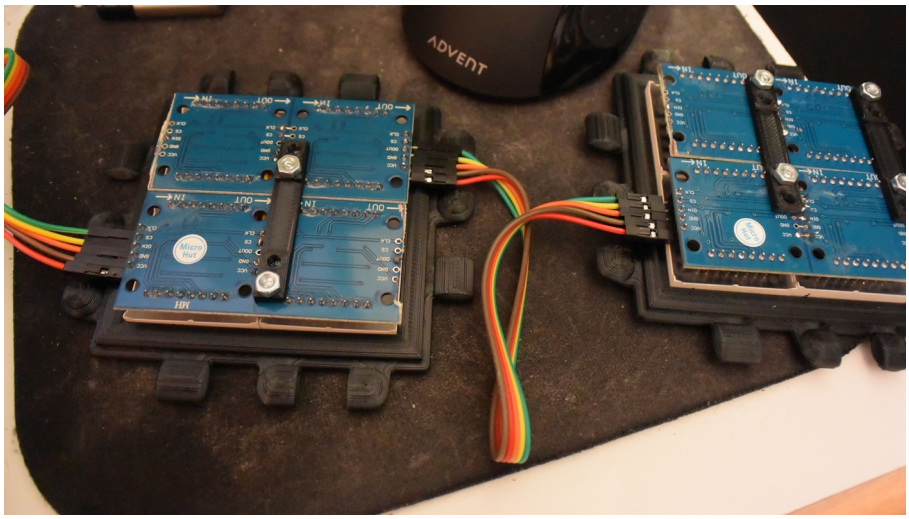
What I tend to do is take 5 sections of insulated wire and solder the two boards together directly. Most of the lines can be wired down the middle of the circuit board, but the DOUT line at the and of the top row *must* be connected to DIN on the bottom left panel, like so:




...the panels can then be fitted back onto the circuit boards.

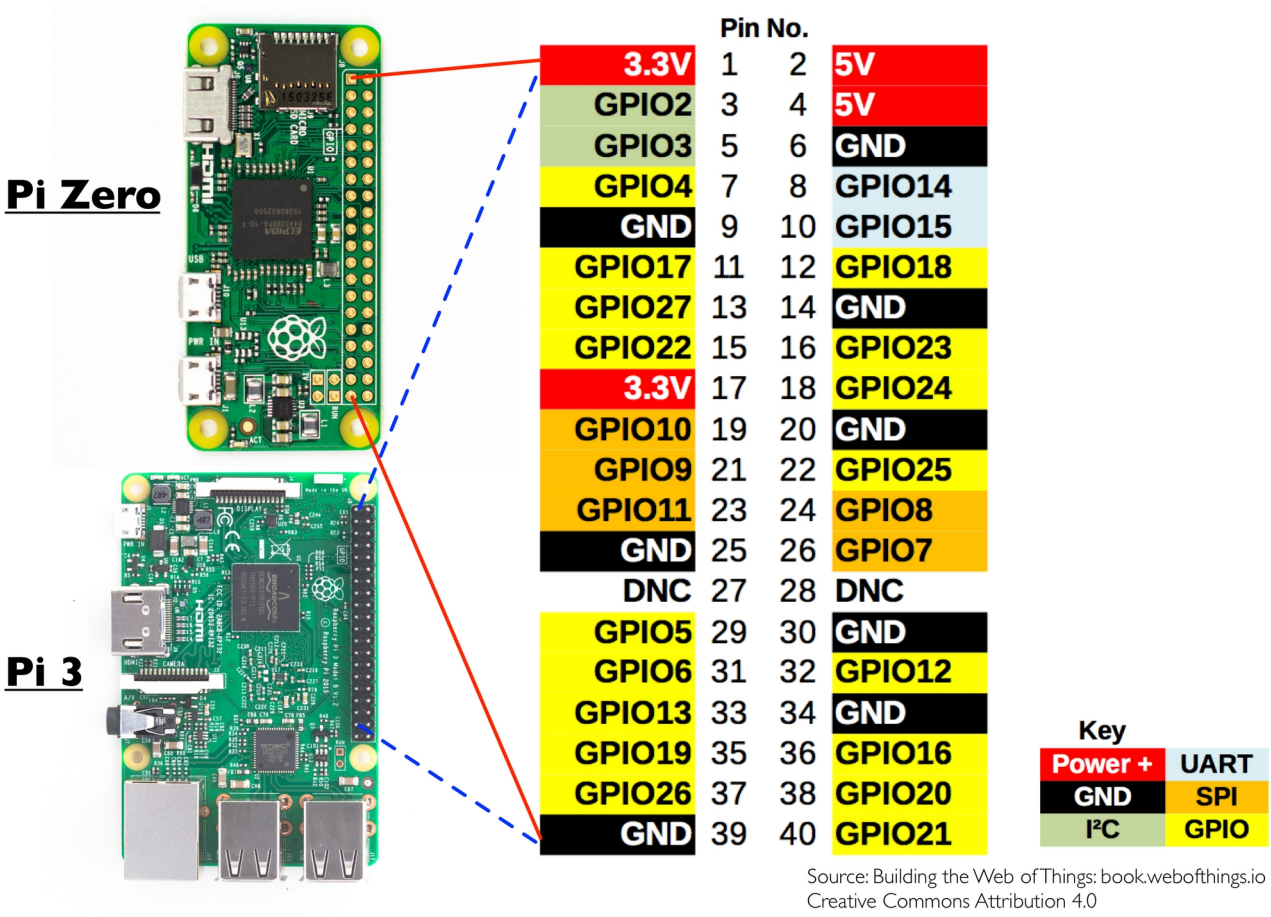
We will also need a way to hold the two halves of the panel together.  This can be done via the screw mounts, or you can 3D-print a friction-fit frame.  Or both.

I use this frame:
https://www.myminifactory.com/object/3d-print-led-matrix-panels-polypanel-electronics-93575
I rescale it to accommodate 16x16 instead of a single 8x8 panel.



## Header connections (16x16)

Here is the pinout for the Raspberry Pi Zero:




Source: Building the Web of Things: book.webofthings.io
Creative Commons Attribution 4.0

The MAX7219 panel needs an SPI connection, so it should be wired as follows:

| **Panel** | **Raspberry Pi** |
|---|---|
| CS (chip select) | 24 |
| CLK (clock) | 23 |
| DIN (Data In) | 19 |
| GND | 25 (or any other ground pin) |

CLK and DIN are fixed by the hardware, but the CS pin can be reassigned if needed by a driver option.

Note that the panel will also require +5V on VCC. Pins 2 and 4 provide a 5v source, but not quite enough current to drive the panels at full brightness. If you hear a whining sound, that means the Raspberry Pi's power regulator is being overloaded and prolonged use may damage the circuitry.

A separate power source is recommended to provide a 5V supply, but make sure to connect ground to both the Raspberry Pi and the display as well as the powerbank or it is likely to malfunction.

# Wiring the board for 32x16 displays

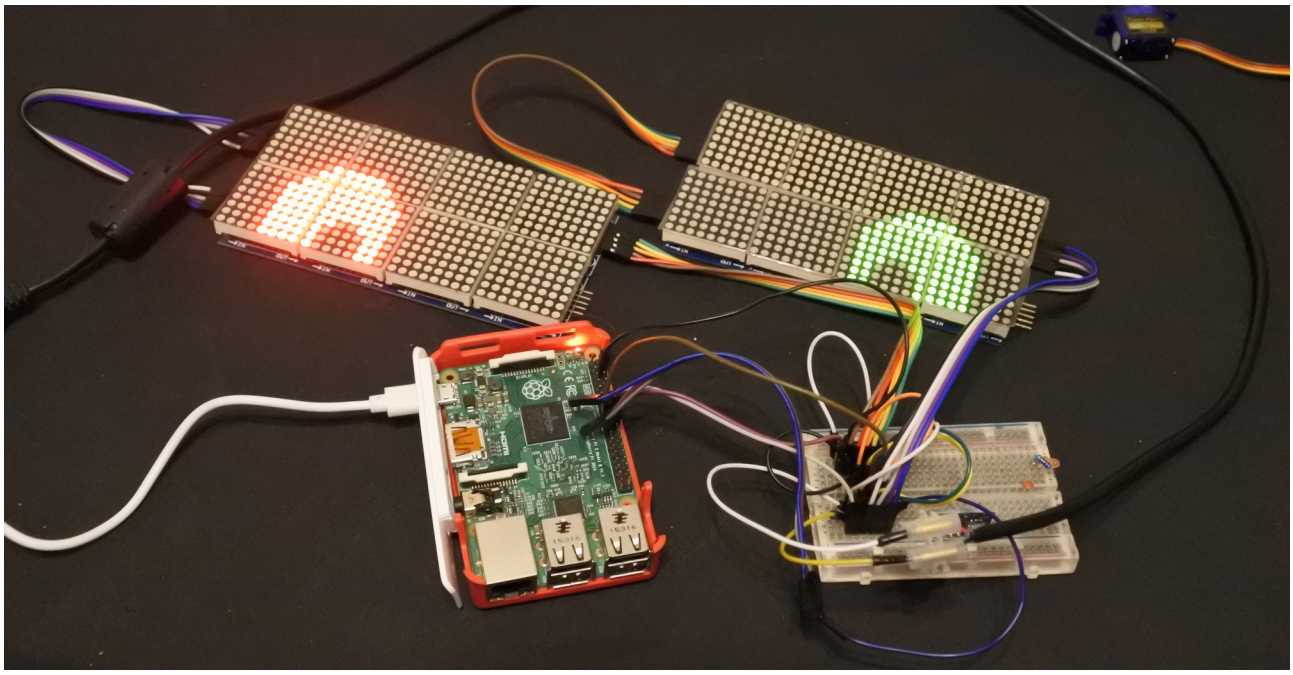
The MAX7219 chipset is capable of addressing up to 8 matrices at once. To exceed this limit we need to wire up the panels in two pairs, both hooked up to the same SPI bus, but independently enabled via the CS line.

This will not require any modification but will need some clever wiring. In addition, the power usage will exceed the capabilities of the Raspberry Pi and an external 5V source is strongly recommended to prevent it crashing from voltage drop or melting the voltage regulator IC.
Make sure to connect ground to both the Raspberry Pi and the display as well as the powerbank or it is likely to malfunction.

Daisy-chaining from panel 2 to panels 3 and 4 is not recommended. Some data lines simply won't be passed through, and the voltage drop is enough to prevent the panels working properly.

It is recommended to split the data lines at source with Y-splitters. I used a breadboard for testing.

The CS lines must be kept separate and not split with a Y adaptor. We need panels 1 and 2 (right eye) to have CS connected to pin 22 and the other pair of panels (3 and 4 – left eye) to have CS connected to pin 18, otherwise both panels will always display the exact same image.
The actual pin assignments for the CS lines can be altered if necessary (see table below).

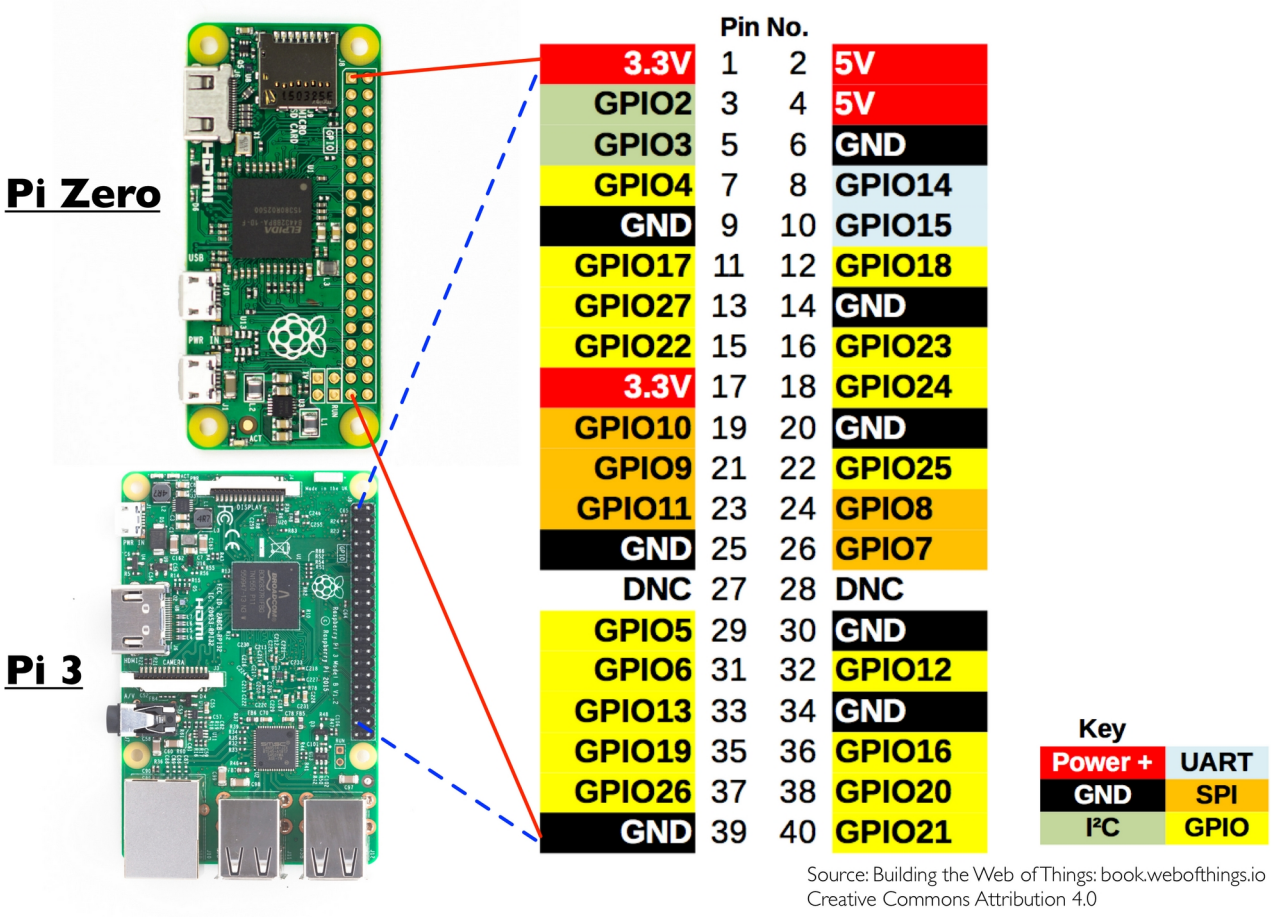


In the above image, the power is being supplied from a powerbank off-shot to the left. It is driving both the Pi and providing a parallel 5V supply to drive the display panels. As with most of the data lines, the power is being split in two and sent to both displays using the breadboard as a multi-way Y-splitter.

The 32x16 configuration is still experimental (16x16 seems to be stable and is a much better bet for production use).

## Header connections (32x16)

Here is the pinout for the Raspberry Pi Zero:




Source: Building the Web of Things: book.webofthings.io
Creative Commons Attribution 4.0

The MAX7219 panels needs an SPI connection, but since we are driving two banks for panels instead of one bank, we'll need *two* CS lines.

Do NOT use pin 24 – in v3.02 there is an issue where it will enable for both eyes resulting in a flickery mess. This may be fixed in a future release.

The defaults are below, but the CS pins can be reassigned if necessary.

| **Panel** | **Raspberry Pi** |
|---|---|
| CS-L (chip select left eye) | 18 |
| CS-R (chip select right eye) | 22 |
| CLK (clock) | 23 |
| DIN (Data In) | 19 |
| GND | 25 (or any other ground pin) |

CLK and DIN are fixed by the hardware, but the CS pins can be reassigned if needed by a driver option.

# Optional Hardware

## Status lights

Starting with v3.02, the software can also drive a string of neopixel LEDs.  Vader-San Synths have a set of status lights on either side of their heads, and the Pi can drive these as well.

The Neopixel string will have three connections:  VCC, GND and DATA.
The DATA pin should be connected to pin 40 on the Raspberry Pi since it needs to use specific features of the hardware.  GND can be attached to any ground pin.

I have been running them off a 3.3V pin on the Pi, although it may require a full 5V depending on the design of the lighting module.

By default the lighting driver will use a Green-Red-Blue lighting module.  If the colours are in a different order on your module, it can be changed via a driver option.

## Microphone

The status lights will pulsate by default.  However you may wish to modulate them with a microphone so the status lights flash when the wearer speaks.

Since the Pi has no ADC, the microphone board will need to have a digital output that triggers when a set level is reached.
If the microphone has adjustable gain it is possible to use the analogue output, as it will count as logic high if the signal level is loud enough.

The microphone should need three connections: VCC, GND and OUT.  Since the microphone will be sending signals back to the Pi it is strongly recommended to power the microphone from a +3V pin, *not* 5V or it may burn out the Pi's GPIO bus.

The OUT pin can be attached to any GPIO pin that is not otherwise being used.

# Software Setup

Now we need to install the OS onto the SD card.  Copying the file won't work, you will need to use something like Raspberry Pi Imager to write the system directly to the SD card.

Download my custom image from the following location:

`https://it-he.org/down/pi-eyes-sdcard-v3.7z`

...it's about 300MB.  Unzip it with 7zip and it will expand to about 2GB.

Inside Raspberry Pi Imager, select 'custom image' (near the end of the list) and choose the 'pi-eyes-minimal.img' file.  Write this to both micro SD cards.

Once the system image has been written to the SD card there will be a small boot partition which will be visible to Windows (the main OS partition isn't because Windows is still stuck in 1989).  If you need updates or customisations, I can provide an updated 'eyes' executable, which you can then copy on to the boot partition to update the software.  You must overwrite the existing 'eyes' file – copying the file with another name won't work.  (Remember to copy the file to both SD cards!)

# Powering up

Now, insert the SD cards into the Pi Zeroes, and give them power via the MicroUSB port marked 'Power' (not the other one, which is for peripherals).

A green light should flash during boot.  As of this writing the OS boot takes about 30 seconds on a an original Pi Zero – the Zero 2 takes more like 13 seconds, a significant improvement.

Once it's finished, the Unicorn display should light up, showing the version number of the synth software for about 2 seconds, and then displaying the eye which should blink on both devices.


During the boot screen one of them should have the word 'rcv' at the bottom, and the other should say 'xmit'.  If they both display the same message, double check that pins 29 and 30 are joined (and only on the Transmitter!).  If the receiver displays the eye but never ever blinks, make sure that the link is OK (pin 6 is connected to pin 6, and pin 8 to pin 10).  The receiver will also be stuck displaying a stationary eye if the receiver is powered on and the transmitter isn't.

# Customisation

The V3 software will look for the file eyeconfig3.txt in the boot partition. It will not read the old eyeconfig.txt file as it works differently. The new config file describes the animations and assigns them to GPIO pins on the hardware.

The actual graphics are loaded from animated GIF files (or static .GIF files for that matter). The exact location is set in the config file, but by default this will be `/boot/videos/raptors_den` . Note that there are currently three sets of video files, for Xerian, Raptor's Den and some miscellaneous sprites for demonstration purposes.

Because of this, it is now possible to customise the display yourself without needing to recompile the software as was the case with SynthEyes2.

The drawback is that the configuration is more complex, and the V3 software is unable to start without it. To assist in setup, the software will display a scrolling error message if there is a major problem with the config file.

There is also a desktop version of the software which can be used for testing. However, as of this writing it is only for Linux – a Windows port may appear in future, otherwise you might be able to run the Linux version through the Windows Subsystem for Linux.

Lines of the config file starting with # will be treated as a comment and ignored. Commented-out examples are provided for most of the supported commands – remove the leading # to enable them.

The next part of the documentation will describe the config file commands. You can probably get started by editing the examples in `/boot/examples/` on the SD card.

# Config File Reference – Hardware Drivers

## Display Panel drivers  (Added in 3.01)

### Max7219 panel driver

```
display: MAX7219
```

‘display:’ specifies the display driver to use.  By default the software will use the Unicorn Hat HD driver.  However if you wish to use MAX7219 LED panels you will need to specify the display driver in the config file.

You can also specify driver options.  For the MAX7219 driver you can specify the CS pin if you do not want to use the default of pin 24.

e.g.

```
display: MAX7219 cs=22
```

...will specify pin 22 as the chip select pin.

### Max7219W panel driver  (New in 3.02)

```
display: MAX7219W
```

The MAX7219W driver is an experimental widescreen version of the MAX7219 driver.  Rather than driving 8 matrices in two 16x16 panels, it drives 16 matrices set up as two 32x16 panels.

‘display:’ specifies the display driver to use.  By default the software will use the Unicorn Hat HD driver.  However if you wish to use widescreen MAX7219 LED panels you will need to specify the display driver in the config file.

You can also specify driver options.  For the MAX7219W driver you can specify both CS pins if you do not want to use the default setup of pin 18 for the left eye and pin 22 for the right.

e.g.

```
display: MAX7219W csl=11,csr=12
```

...will specify pin 11 as the left eye chip select pin, and pin 12 as the CS line for the right eye.

**SDL panel driver**

```
display: SDL
```

‘display:’ specifies the display driver to use. By default the desktop version of the software will use this driver, providing a dual-head setup to emulate the Unicorn Hat display. However, you may wish to specify options.

For the SDL drivers you can specify the width and height of the virtual panel, this is currently restricted to common display panel sizes, e.g. 16, 32 or 64 pixels wide, and 16 or 32 pixels high.

e.g.

```
display: SDL w=32,h=16
```

...will give you a 32x16 display instead of the default 16x16.

**SDLSingle panel driver**

```
display: SDLSingle
```

The SDLSingle driver provides a single-head setup, with one window displaying both eyes, to emulate the behaviour of the MAX7219 display (unlike the MAX display, it is in colour).

‘display:’ specifies the display driver to use. By default the desktop version will use the SDL dual-head driver. However if you wish to use SDL single panel driver you will need to specify the display driver in the config file.

You can also specify driver options. For the SDL drivers you can specify the width and height of the virtual panel, this is currently restricted to common display panel sizes, e.g. 16, 32 or 64 pixels wide, and 16 or 32 pixels high.

e.g.

```
display: SDLSingle w=32,h=16
```

...will create a single 64x16 window, containing both eyes (32x16 pixels each). The default will be a 32x16 window containing two 16x16 eyes.

## Lighting strip drivers  (Added in 3.02)

### WS2811 lighting driver

```
lights: WS2811 6
```

‘lights:’ will specify the lighting strip driver to use, if any.  It also specifies the number of lights in the strip, in this case, six.  If the number is omitted it will default to 8.

The WS2811 driver currently has two options, ‘gpio’ and ‘type’.  ‘gpio’ specifies the GPIO pin, which can be 40, 32 or 12.  40 is recommended and the other pins may not be valid.  If this happens the lights will not be be activated and a warning will be displayed on the HDMI output if a monitor is attached.

The ‘type’ parameter specifies the colour order that the lighting strip is expecting.  It can be one of the following:
rgb, rbg, grb, gbr. brg, bgr, rgbw, rbgw, grbw, gbrw, brgw, bgrw
...the default is grb

e.g.

```
lights: WS2811 12 gpio=40,type=rgbw
```

The light colour and patterns can be set via the lightcolour: lightpattern: lightspeed: and lightbrightness: commands.

Microphone support can be set up using micpin:, micinvert:, micbright: and micdim: commands.

### SDL virtual lighting driver

```
lights: SDLLights 6
```

‘lights:’ will specify the lighting strip driver to use, if any.  It also specifies the number of lights in the strip, in this case, six.  If the number is omitted it will default to 8.

The SDL driver currently has no options, but can be used to emulate a WS2811 strip on the desktop version of the software for testing.

## Servo drivers  (Added in 3.02)

Servo support is experimental – currently only one motor can be driven, support for driving multiple motors may be added in a future release if needed (and if practical).

**Pi Servo driver**

```
servo: PiServo 90
```

'servo:' will specify the servo driver to use, if any.  It also specifies the default angle to move the servo to in degrees (0-359).  If this is omitted it will default to 0.

The PiServo driver currently has one option, 'pwm' which specifies the GPIO pin to use as the servo control line.  This must be a PWM-compatible pin – the default is pin 12, and this is recommended.  Other available pins are 33, 35 and 36.

Note that the servo will draw a *lot* of power while operating.  It is very strongly recommended to use a separate power source for the servo as running it directly off the Pi will cause issues, especially if the Pi is also powering other devices such as a display panel.

Example:

```
servo: PiServo 180 pwm=36
```

The servo motor can be driven via the setservo and seekservo actions.

**Virtual servo driver**

```
servo: TestServo 90
```

'servo:' will specify the servo driver to use, if any.  It also specifies the default angle to move the servo to in degrees (0-359).  If this is omitted it will default to 0.

The test driver currently has no options, but can be used to show the current servo angle on the desktop version of the software for testing.

## Sensor drivers  (Added in 3.04)

Sensor support is experimental – currently only one dedicated sensor can be used, support for driving multiple sensors may be added in a future release if needed (and if practical).

See ‘Sensor-driven expressions’ in the config file reference.

### CAP1188 driver

```
sensordrv: cap1188
```

‘sensordrv:’ will specify the sensor driver to use, if any.  There are also optional parameters to set the I2C address, sensitivity and light control.

The CAP1188 driver will provide 8 independent capacitive touch sensors, which the ‘sensorchannel:’ directive can specify as 0-7, if you want to ensure specific sensors invoke specific animations.

The default address for a CAP1188 board is 0x29.  If you need to change this, use the address= parameter.  Valid addresses for the ADAfruit board are 0x28, 0x29, 0x2a, 0x2b and 0x2c.

The CAP1188 chipset has an 8-stage sensitivity control for the touch sensor array.  Values can be set from 0-7, with 0 being the most sensitive and 7 being least sensitive.  The default value is 0 (most sensitive).

Finally, the ADAfruit board also has 8 LED which light up in response to a sensor being triggered. This is useful for debugging, but the lights can also be disabled via the lights=0 parameter.  Any other numeric value (e.g. lights=9) will be treated as 1 and the lights will be enabled (the default setting).

If the ‘sensor:’ expression is used without a ‘sensorchannel:’ directive in the expression being triggered, it will respond to any one of the 8 touch sensors.

Example:

```
sensordrv: cap1188 address=0x28,sensitivity=5,lights=0
```

### Virtual sensor driver

```
sensordrv: TestSensor
```

‘sensordrv:’ will specify the sensor driver to use, if any.

The test driver is for Linux and reads the keyboard.  It treats ‘A’ as channel 0, ‘B’ as channel 1 and so forth, up to ‘H’ for channel 7.

# Config File Reference – General Setup

## *GIF directory*

```
gifdir: ./videos/raptors_den
gifdir: /boot/videos/raptors_den
```

'gifdir' specifies the folder where the GIF files are stored. Since the SynthEye software runs on Linux, you must use the forward slash separators (e.g. `/boot/myfile.txt` ) rather than the backslash separators used in Windows (e.g. `\boot\myfile.txt` ).

You can have multiple gifdir: commands in the same config file – the last one that exists will be take precedence. This can be useful when testing the animations on the desktop software, since you probably won't have write access to /boot/ on a PC.

The above example does this: ./videos/raptors_den will work on the PC but fail on the Raspberry Pi SD card, likewise /boot/videos/raptors_den will typically only work on the Pi. As a result, the software ignores the one that doesn't work and uses the one that does.

If there there is no valid path, the software will halt with an error message once it tries to load in the animations.
If the gifdir: command is omitted entirely it will default to /boot/

## *Include (Added in 3.02)*

Starting from v3.02 it is now possible to chain config files together.
For instance, if you have multiple configurations for testing and need to switch between them a lot, you could put the configurations in separate files, e.g 'raptorconfig.txt', 'xerianconfig.txt' and then just have the eyeconfig3.txt file containing one line to load in the other config.

It will attempt to load the file from the current directory first, if that fails it will try to the file from /boot/. If that fails too it will ignore it and carry on with the rest of the config file.

e.g.

```
#include: config/raptorconfig.txt
include: config/xerianconfig.txt
```

## *Eye colour*

```
eyecolour: #001122
```

Eyecolour will set the default colour of the eye animations, as with V2.

Note that because V3 uses animated .GIFs, this will only apply to animations which have the ‘drawmode: monochrome’ command dicussed later. Otherwise it will use the built-in colours from the GIF file.

Colours are specified as a hexadecimal red-green-blue triplet, the same as HTML. Xerian’s default warm yellow colour is, e.g.

```
eyecolour: #ff8f00
```

(full red, about half green, no blue). #0080ff will provide a cyan eye.

Most art software such as Krita or GIMP provides a colour picker in this format and there will be colour charts available online as well.
Note, however, that the display’s colour rendering is likely to differ wildly from your monitor, so some experimentation is likely to be needed.

Remember to copy the same config file to both SD cards, unless you want the Synth to have heterochromic eyes.

## *Cooldown*

If the user holds down the expression change switch too long it would immediately retrigger the expression animation, which can look a little strange. To avoid this a cooldown timer has been implemented to help avoid running the same animation twice by accident. (Running a different expression animation should happen immediately – the lockout is for the same animation happening twice).

The duration of the cooldown timer can be changed with the cooldown: command and is measured in seconds. It defaults to 5 seconds. Setting it to 0 should disable the cooldown timer.

e.g.

```
cooldown: 5
```

## *Rotate display (Added in 3.03)*

If you need to use two full-size Raspberry Pi devices instead of Pi Zeroes, you may wish to rotate the display on one of them so that the ports are facing the same direction.

You can enable this rotate180: command. The parameter is a boolean such as ‘true’ or ‘false’, ‘yes’, ‘no’ etc.

e.g.

```
rotate180: true
```

...will flip the display 180 degrees.

Note that this feature is not implemented in all display backends. At the moment only the SDL and Unicorn drivers can do this – the MAX7219 drivers will ignore the flag.

## *Effects*

By default, the eye will tend to be a fixed colour, either using the colours inside the GIF or by overriding them with a single colour.
However it can also be made to cycle through different colours. At present this is primarily intended as a demo mode, though it can be enabled just for certain expressions.

The effect: command determines which effect will be used when the eye is set to do this. Currently there are three effects available and they cannot be combined. The last one will take precedence.

These effects are rainbow_v, rainbow_h and rainbow_o, but more may be added in future.
The ‘Rainbow’ effects will stripe the eye with a rainbow that flows through different colours. It can be striped vertically, horizontally or in a circular effect.

The speed of the colour cycling is an update every 10 milliseconds by default, but can be adjusted with the rainbowspeed: command.

e.g.

```
#effect: rainbow_v
#effect: rainbow_h
effect: rainbow_o

rainbowspeed: 8
```

By default, the gradient will be a rainbow. If you want to change the colours, these can be altered by the setrainbow: command. The gradient has 16 colours in it, so you will need 16 setrainbow: commands to full reprogram it.

The first parameter is the slot from 1-16, the second parameter is the colour in hex as per HTML.
e.g.

```
setrainbow: 1 #ff00ff
   :
setrainbow: 16 #0f000f
```

### *Display brightness (Added in 3.02)*

It is possible to change the brightness of the display panel.

This is done with the ‘brightness:’ command, and it takes a percentage amount.  Higher brightness will provide better visibility outdoors, but also requires more power (i.e. it will reduce battery life).

e.g.

```
brightness: 75
```

## *Lighting strip config (Added in 3.02)*

If a lighting strip has been initialised with the lights: command, you will probably want to at least set the colour of the strip and possibly the brightness as well.

If no lighting strip has been set up, the commands will have no effect.

‘lightcolour:’ (or lightcolor:) sets the default colour of the lighting strip. Like eyecolour: it takes an RGB triplet.

e.g.

```
lightcolour: #ff8f00
```

...will give you a cyan lighting strip. It is also possible to set it to ‘eyecolour’ which will use the eye colour if that has already been set. (It uses the eye colour at the moment the config file was read – if the eye colour is changed later the light colour won’t be updated to match).

‘lightpattern:’ will set the way the lighting strip animates. Currently this can be either ‘triangle’ or ‘saw’. ‘Triangle’ mode will fade the lights in and out in sequence like a Mexican wave, and this is the default.
Setting it to ‘saw’ will fade the lights up to maximum and then drop down to minimum in a sawtooth pattern like fairground lights.

e.g.

```
lightpattern: saw
```

‘lightspeed:’ will set the speed at which the lighting strip animates. This is a delay in milliseconds and the default is 25 ms.

e.g.

```
lightspeed: 12
```

‘lightbrightness:’ will set the brightness of the lighting strip. This is a percentage value and the default is usually 100%, though this may depend on the device driver.

```
lightbrightness: 75
```

## *Microphone support (Added in 3.02)*

The lighting strip can be modulated using a microphone. To enable this you will have to set the GPIO pin to receive the microphone signal level.

‘micpin:’ will specify the pin to monitor for microphone activity.

e.g.

```
micpin: 31
```

Depending on how the microphone module is designed, it may send a logic high or logic low to indicate microphone activity.

By default the software will treat 0V as silence and +3V as speech activity. If you need to invert this, e.g. +3V for silence and 0V for activity, the ‘micinvert:’ command can be used to do this.

e.g.

```
micinvert: true
```

Then we have the question of what the microphone does when speech is detected.

By default it will begin listening for more input for a number of milliseconds. If the microphone reads activity at this point the lighting strip will be set to 100% brightness, if it detects silence it will be set to 10% brightness causing all the lights to flash in sympathy with the wearer’s voice.

It will revert to normal pulsating activity after 500ms, though of course further microphone activity will restart the listening period.

The ‘micbright:’ ‘micdim:’ and ‘micwait:’ commands can be used to tweak this behaviour. ‘micbright:’ will set the brightness level when sound is detected, as mentioned it defaults to 100%. ‘micdim:’ will set the brightness level when no sound is detected, as mentioned it defaults to 10%.

...you could if you prefer, have ‘micdim:’ set to 100% and ‘micbright:’ set to 10% to invert this behaviour.

Finally, ‘micwait:’ will change the length of the detection window in milliseconds. As mentioned, the default value is 500, i.e. half a second.

You can also have the microphone change the light colour using the ‘miccolour:’ (or ‘miccolor:’) command. This takes the usual RGB triplet. If no alternate colour is specified it will use the default colour for the status lights.

e.g.

```
micpin: 31
micbright: 75
micdim: 30
micwait: 350
miccolour: #ff0000
```

## *Transmitter*

**Most people will not need to touch this but it's included for completeness.**

By default the software will boot in Receiver mode unless pin 29 is set to ground via the jumper (see page 4). For testing purposes you may wish to force the Pi into Transmitter mode even if the jumper is not bridged. As mentioned, most people will never need to do this. However, if you do for some reason, the transmitter: command will override the default.

e.g.

```
transmitter: true
```

## *Serial Port*

**Most people will not need to touch this but it's included for completeness.**

By default the two Pis will be linked via the serial port on /dev/ttyAMA0. If this should need to be changed to some other serial device, you can use the serial: command to do so.

e.g.

```
# To override serial port
serial: /dev/ttyS0
```

The serial link speed can also be changed via the baud: command, with the parameter in bits/second. By default this is 19200. Make sure both machines use the same baud rate!

e.g.

```
serial: /dev/ttyS0
baud: 9600
```

## *ACK Light*

When an expression change is triggered on one of the GPIO pins it can be programmed to light up an LED on the Receiver board to signal the user that they've successfully changed the expression. On Xerian's hardware I have added the LED between pins 40 and 39 (GND).

The duration of the acknowledgement LED can be changed with the acktime: command, with the duration in milliseconds. It defaults to 750 milliseconds.

e.g.

```
ackpin: 40
acktime: 650
```

Note: It is also possible to disable the ACK light for specific expressions, e.g. blinking, by adding the 'ack: off' command to the expression definition. See below.

### *Random expression chance*

The system will display the IDLE expression by default, but you can define random expressions as well, e.g. to simulate blinking.

To prevent the face blinking twice in succession, it is possible to configure a percentage chance that it will look for a random expression to play, a random chance on top of a random chance to space things out more.

This defaults to 75%, but can be configured using the randomchance: command.

e.g.

```
randomchance: 60
```

...will reduce the chance to 60%, resulting in longer durations of the idle face.

### *Scrolling speed (Added in 3.01)*

By default scrolling messages wait approximately 40 milliseconds per frame.
You can adjust this with the scrollspeed: command. The parameter is in milliseconds.

e.g.

```
scrollspeed: 160
```

...will slow down the scrolling messages, including error messages.

### *Seamless (Added in 3.01)*

By default the program will always switch back to the Idle expression for a bit after playing an expression to provide a sensible gap between two animations.
However, if you want the expressions to remain locked on the screen until overridden, this effect will cause a brief flicker.

You can disable this behaviour with the seamless: command. The parameter is a boolean such as ‘true’ or ‘false’, ‘yes’, ‘no’ etc.

e.g.

```
seamless: true
```

...will provide proper gapless playback between expressions without running the Idle animation first.

## *Ground (Added in 3.04)*

If you need to add a lot of GPIO triggers, you may find yourself running out of 0V ground pins to complete the circuit with. This command allows you to create virtual grounds by setting a GPIO port to output 0v as a virtual ground.

Note that this is only suitable as a ground for GPIO triggers and should **not** be used as a negative rail for powering other devices, as excess current may damage the microcontroller.
Use the *actual* GND pins for that.

However, for a switch to change expressions by momentarily linking it to ground, this can be a useful option.

The parameter is the pin to tie to ground.

e.g.

```
ground: 7
```

...will lock pin 7 to use as a virtual 0V source for signalling. You can do this as many times as you like, e.g.

```
ground: 5
ground: 7
```

...will set both pins 5 and 7 to be virtual grounds.

# Config File Reference – Animation Setup

While the previous commands are all single-line commands which can be run anywhere, setting up an animation is a more complex task and will require a block of related commands.

The simplest working expression can be set up like this:

```
idle: my_idle_animation
    gif: idle.gif
```

This will create an idle animation called ‘my_idle_animation’, which will display the animated GIF file on the second line.

The exact name given doesn’t really matter, but it’s probably best to give it something appropriate, especially if the GIF files are numbered or are otherwise unclear.
The indentation on the gif: line isn’t necessary (unlike YML or Python) but makes it easier to read.

The *order* of the commands is important, however. Doing ‘gif:’ first and then ‘idle:’ will confuse the program and probably display an error message, as ‘gif:’ is relying on ‘idle:’ to have set things up for it.
Likewise, later commands such as ‘drawmode:’ will only work once both ‘idle:’ and ‘gif:’ have been processed. More on that in due course.

Note that the names are unique... you can’t have two expressions called ‘my_idle_animation’.

## *Declaring a new expression*

Currently the software has supports three types of expression – GIF files, scrolling messages and procedural blinking. It also has four triggers which can cause them to be run – when idle, randomly, triggered by the GPIO pins, and called by a script.

As per the example, we tell it when the expression is going to run first, and then what it will be.

```
idle: my_idle_animation
    gif: idle.gif
```

..but we could also have a random scrolling message, e.g.

```
random: scrolly
    scroll: Hello World
    chance: 50
```

...which will randomly display the message “Hello World” on the display, with a roughly 50% chance.
Or, we could have a blinking animation that is triggered by shorting pin 33 to ground:

```
gpio: eyeroll
       blink: default
       background: my_idle_animation
       pin: 33
```

### *Idle expression*

As we have seen in the examples, you can (and must!) set up an idle expression for the program to work correctly. If this is not done, the program will halt with an error message.

With the current version of the software, there can only be one idle expression. If you declare two of them, it will always pick the first one. This may change in future.

The idle expression is the default expression and will always be displayed when nothing else takes precedence. If a GPIO event triggers a new expression, the idle animation will instantly stop.

The idle: command takes one parameter, the expressions's name. You will also need to specify an GIF, Blink or Scroll command in the following line, but otherwise it's pretty self-contained.

```
idle: my_idle_animation
    gif: idle.gif
```

### *Random expressions*

To allow for blinking and other random events, you can set up randomly-triggered expressions. These are not mandatory but will make the face a lot livelier. If a GPIO event happens, the triggered expression will not play until the random animation has ended.

As with idle, you will need to specify a name, and then a GIF, Blink or scrolling message on the following line. It is also strongly recommended to set the random chance via the chance: command, otherwise it will default to 0% and the expression will never be displayed.

```
random: blinking
    gif: blink.gif
    chance: 50
```

### *GPIO expressions*

So far the face can display a default expression and blink randomly. However, if you want to trigger an expression manually, the easiest way is to use the GPIO pins on the Transmitter board. By briefly shorting one of these pins to ground via a switch, you can tell the software to run a particular expression such as an eye-roll or angry face. If two are triggered in quick succession, the first one will not be interrupted and the second one will only play after it has finished.

As with idle and random expressions, you will need to specify a name, and then a GIF, Blink or scrolling message in the following line. It is also strongly recommended to set the GPIO pin via the pin: command, otherwise it will default to nothing and the expression cannot be triggered.

```
gpio: eyeroll
       gif: eyeroll.gif
       pin: 33
```

The Raspberry Pi has 40 GPIO pins. Of these, only 30-40 are currently supported (others are used by the Pi Hat display). Not all of these can actually be used, since pins 30, 34 and 39 are hardwired to ground and cannot be used for I/O.
Pin 29 is reserved by the software to tell the Transmitter and Receiver boards apart and its use is not recommended. The parameter for pin: is the actual hardware pin number.

### *Sensor-driven expressions*

In 3.04, support was added for hardware sensors such as the CAP1188 touch sensor, which is addressed over an I2C bus instead of using GPIO pins.

The CAP1188 supports up to 8 independent touch sensors, so you may wish to assign different expressions to different channels, e.g. sensor 0 rolls the eyes, sensor 1 shows an exasperated expression, etc.

Alternatively, you may wish any one of them to trigger animations. If no channel is specified and several expressions are set to respond to the same sensor, it will pick one at random.
If you define some for specific channels and some are set to run on any channel, the specific ones will take precedence.

As with the other expressions, you will need to specify a name, and then a GIF, scrolling message or Blink expression on the following line. Nothing else is required, unless you wish to specify a hardware channel. Otherwise it will respond to any sensor.

You must have declared a hardware sensor via the sensordrv: command (see 'Hardware Drivers' at the start of this section). If none is declared, or it cannot be used for some reason, nothing will happen when the sensor is triggered.

As with the Random and GPIO expressions, the animation will not be interrupted by events such as a GPIO signal, and the next animation will not run until the current one has finished.

Here's a couple of examples.

```
sensor: channel2
       gif: eyeroll.gif
       sensorchannel: 2
```

-or-

```
sensor: any1
       gif: eyeroll.gif

sensor: any2
       gif: wink.gif
       sensorchannel: any
```

### *Scripted expressions*

Finally, it is possible to declare an expression that will be played under script control.

At the moment the only scripting support is to allow for actions to be run before or after an expression runs, but this does allow you to chain expressions together so they run in sequence. And this is where the scripted expressions become useful.

As with the other expressions, you will need to specify a name, and then a GIF, scrolling message or Blink expression on the following line. Nothing else is required.

As with the Random and GPIO expressions, the animation will not be interrupted by events such as a GPIO signal, and the next animation will not run until the current one has finished.

Here's an example from the demo files. It has a random expression displaying the eye tinted red, after which it calls the green eye expression, which in turn calls the blue eye expression:

```
random: redeye
       gif: idle.gif
       colour: #ff0000
       chance: 20
       after: chain greeneye

scripted: greeneye
       gif: idle.gif
       colour: #00ff00
       after: chain blueeye

scripted: blueeye
       gif: idle.gif
       colour: #0000ff
```

## *GIF Animations*

The most common type of expression will be a .GIF file. This can be animated or a single frame, though since the expression will end when the animation is finished, still pictures are best used for the idle expression or other similar situations. Starting from 3.01 it is possible to make an animation loop, but this must be set up manually since it is not really how the software was intended to operate.

The Unicorn Hat HD panel is 16x16, which conveniently means that 256 colours are not a limitation since there are only 256 pixels in total. That said, the SynthEye software was really intended to display cartoon graphics rather than full-colour video anyway.

16x16 GIF files are recommended for the Unicorn Hat HD display. Smaller images will be displayed in the top-left corner – larger images will show just the top-left corner of the image.

Note that since the software is displaying two distinct eyes, the Transmitter board will display the eye in its correct orientation. The Receiver board will mirror the eye on the other display so they will both be looking in the same direction. This mirroring can be disabled, e.g. if the animation contains text.

The bundled animations were all created using GIMP 2.10, treating each frame as a distinct image. I have not done extensive testing with files generated by other packages so I can't guarantee that the image loader is perfect, but so far it has handled everything I need it to.

As mentioned, the gif: command must directly follow one of the expression types (idle:, random:, gpio: or scripted:). It takes a filename as its parameter, and this will be loaded from the directory specified by gifdir: (or /boot if none is specified).

e.g.

```
random: blinking
    gif: blink.gif
    chance: 50
```

If the file cannot be found, or is corrupt in some way and cannot be loaded, the program will halt with an error message.

As mentioned, GIF files can have up to 256 colours including a transparent background. By default it will use the colours baked into the GIF file, e.g. if the eye is blue in the GIF file, it will be drawn as blue.
However it is possible to override this, turning the GIF into a monochrome image and rendering it with the colour of your choosing. Transparent sections of the GIF, and colours which map to pitch black (#000000) will be ignored.

This and other effects are documented in the Drawing Effects section below.

### *Scrolling Text*

While .GIF animations are the main use case for the software, it is also possible to make it display scrolling messages. This was originally intended for error messages but it might be useful to do for other reasons, e.g. secret messages that display randomly or under GPIO control.

There is currently a maximum text length of around 1000 characters, and the alphabet is limited to basic A-Z characters, numbers and a few items of punctuation.

As mentioned, the scroll: command must directly follow one of the expression types (idle:, random:, gpio: or scripted:). The rest of the line will be treated as static text to be displayed in the message.

e.g.

```
random: scrolly
    scroll: Hello World
    chance: 50
```

Unless overridden, it will use the default eye colour (yellow-orange) for the text.

New in 3.01, the scrolling speed can be set using the ‘scrollspeed:’ command, though note that this affects all scrolling messages including the error messages at boot. See ‘General setup’ for details.

3.01 also adds the ‘scrolltop:’ command which can be used to set the Y offset of the scrolling message. By default it will be 4 pixels down from the top of the display.

## *Procedural Blinking (Added in 3.01)*

In the original 3.00 release, blinking animations had to be drawn manually, i.e. you needed a .GIF animation with about 32 frames. In 3.01 it is possible to display a still image and have the software perform the blinking animation on it automatically.
This is classed as a different type of expression, not a GIF or a scrolling message. It utilises another expression as the background image. (See also the background: command later on)

As mentioned, the blink: command must directly follow one of the expression types (idle:, random:, gpio: or scripted:). It takes an option as its parameter, specifying the direction of the blink.
e.g.

```
random: blinking
    blink: top
    blinkspeed: 5
    background: idle
    chance: 50
```

The animation will look for an expression called 'idle' and render this on the display. During the course of the animation, a black shutter will descend, wait a short time and then raise again to simulate a blinking eye. The blink speed is a delay measured in milliseconds, 6 is the default.

It is expected that the background image provided should be a .GIF expression, and it will use the first frame of this as the background. Animating backgrounds might be supported at some point in the future.
Using a scrolling message or another blink expression as the background image (or omitting it entirely) is not recommended, as nothing will be displayed at all.

The following blink modes are currently supported:

- top (or 'default') – start at the top and blink downwards
- bottom – start at the bottom and blink upwards
- left – start at the left and blink right
- right – start at the right and blink left
- horizontal (or 'horiz') – blink both left and right and meet in the middle
- vertical (or 'vert') – blink both down and up and meet in the middle
- square (or 'all') – blink inwards from all four sides and meet in the middle

Currently the blinking system will draw a black shutter. Any specified colour or gradient effects will be ignored, although this could be added in future.

Examples:

```
random: blink1
    blink: top
    background: idle
    chance: 50

random: blink2
    blink: horizontal
    background: idle
    chance: 50
```

## *Drawing Effects*

By default, a .GIF animation will be displayed as-is, using the colours from the file, and scrolling text will be displayed in orange.

These can, of course, be overridden, as hinted at earlier.
While .GIF animations are the main use case for the software, it is also possible to make it display scrolling messages. This was originally intended for error messages but it might be useful to do for other reasons, e.g. secret messages that display randomly or under GPIO control.

These commands should follow the ‘gif:’ or ‘scroll:’ lines and will not work correctly otherwise.

## *Draw Mode*

The software currently supports the following drawing modes: colour, monochrome, gradient and flashing.

Colour (or color) is the default drawing mode for GIF files – it will use the colours baked into the GIF. It has no effect on scrolling messages.

Monochrome (or mono) will render a GIF file in a single colour, discarding the colour information from the GIF file entirely. By default this will be the eye colour. Monochrome is the default for scrolling text.

Gradient mode will render the GIF or scrolling message as a colour gradient, typically a rainbow effect. The orientation of the gradient is set with the effect: command (See ‘General Setup’).

Flashing mode works like the old HTML blink tag. It will flash on and off periodically. The speed of this is tied to the rainbowspeed: parameter (See ‘General Setup’) and be about 16 times slower. A separate timer may be implemented in future.

Finally, Cycle mode will draw the eye in monochrome, but step through all the colours of the gradient in turn.

Examples:

```
gpio: citadel
       gif: citadel_guy.gif
       pin: 38
       drawmode: colour
       after: chain callback

gpio: eyeroll
       gif: eyeroll.gif
       pin: 33
       drawmode: monochrome

idle: idle1
       gif: idle.gif
       drawmode: gradient
```

```
gpio: error
       gif: error.gif
       colour: #ff0000
       drawmode: flash
       pin: 36

gpio: cycling
       gif: idle.gif
       drawmode: cycle
       pin: 37
```

### *Colour*

If you set the drawmode to ‘monochrome’, the GIF will be drawn in the default eye colour instead of using the colours baked into the GIF file. As mentioned this is also the default for scrolling text.

However, you may wish to override the default eye colour, e.g. to make the eye turn red temporarily if the character is angry.

This can be accomplished by the colour: (alias color:) command. It takes a colour as its parameter, as usual in the 6-digit HTML format.

If the drawmode was previously set to ‘colour’ (i.e. use the built-in colours), it will changed to ‘monochome’ on the assumption that you probably want to actually see the colour you’ve specified.

Example:

```
random: redeye
       gif: idle.gif
       colour: #ff0000
       chance: 20
       after: chain greeneye

scripted: greeneye
       gif: idle.gif
       colour: #00ff00
       after: chain blueeye

scripted: blueeye
       gif: idle.gif
       colour: #0000ff
```

## *Actions*

It is possible to make an action happen immediately before or after an expression is displayed.

The design goal was to allow GPIO ports to be programmed, so that an Arduino device driving lights elsewhere on the costume could be instructed to change the colour for the duration of the animation and back again afterwards.
This was done in an extensible manner so that other actions could be performed as well. As of v3.02 support was added for the Pi to drive status lights itself, so actions have been added to facilitate this as well.

Actions are set up by using the 'before:' and 'after:' commands. The first parameter will be the command to run, e.g. 'set_gpio', 'clear_gpio' or 'chain' to chain expressions together.
There can also be a parameter for the command itself, e.g. the GPIO pin to set or clear.

**GPIO actions**

With 'set_gpio' and 'clear_gpio' the last parameter will be the GPIO pin to affect.

Note that the Arduino uses pull-up resistors, so if nothing is connected, the GPIO port defaults to logic *high* (3-5V). Because of this, we need to force the logic to be low (0V) to signal it. For this reason, 'set_gpio' is actually setting the pin to 0v instead of 3V. Likewise, 'clear_gpio' will set the pin to +3V, signalling logic high.

It is also important to note that we may wish to affect the GPIO pins on either the Transmitter board or the Receiver board. Hence, it is possible to select the device by prefixing the GPIO pin with R or T. If just the port number is specified, it will assume we mean *both* devices.

e.g.

```
       before: set_gpio R35     # Trigger GPIO on pin 35 of Receiver
       before: set_gpio T35     # Trigger GPIO on pin 35 of Transmitter
       before: set_gpio 35      # Trigger GPIO on pin 35 on both devices
```

It is strongly recommended to specify 'R' or 'T' for event GPIO, since if the pin is already being used for input, trying to write to it may damage the hardware. The software will attempt to detect this and display an error during startup.

I have included example Arduino software that will flash neopixel status lights and set them red in response to a GPIO event.

Examples:

```
gpio: error
       gif: error.gif
       pin: 36
       colour: #ff0000
       drawmode: flash
       before: set_gpio R35     # Set horns red (on Xerian's hardware)
       after: clear_gpio R35    # Set horns back to normal (Xerian's hardware)
```

**Chain action**

By setting a ‘chain’ action it is possible to link to another expression after the first one has finished played. Note that it is possible to set up an infinite loop by doing this, so be careful.

With ‘chain’ the final parameter will be the name of the expression to be played – if it is not found, nothing will happen.

e.g. to make it step through several colours in turn:

```
random: redeye
       gif: idle.gif
       colour: #ff0000
       chance: 20
       after: chain greeneye

scripted: greeneye
       gif: idle.gif
       colour: #00ff00
       after: chain blueeye

scripted: blueeye
       gif: idle.gif
       colour: #0000ff
```

**Wait action (new in 3.02)**

If you are setting GPIO pins you may need to set the pin high, wait a few milliseconds and then clear it again. The ‘wait’ action will provide a suitable delay for this kind of behaviour.

With ‘wait’ the final parameter will be the delay length in milliseconds.

e.g.

```
gpio: error
       gif: error.gif
       pin: 36
       colour: #ff0000
       drawmode: flash
       after: set_gpio R35    # Flip the bit and wait for the device to see it
       after: wait 10         # Wait 10ms
       after: clear_gpio R35  # Clear the bit back again
```

**Lightcolour action (new in 3.02)**

With v3.02 there is built-in support for driving lighting strips on the costume. In previous versions (or with hardware constraints such as the Unicorn Hat) you would use the GPIO pins to control an external device running the lights, but if the Pi is driving the lights itself you will need a way to control them.
The ‘lightcolour’ (or ‘lightcolor’) action will change the colour of the lighting strip (if present). It takes an RGB triplet as its parameter, and this can also be set to ‘lightcolour’ which will set it to the colour specified in the config file, as a handy way of resetting it afterwards.

e.g.

```
gpio: error
       gif: error.gif
       pin: 36
       colour: #ff0000
       drawmode: flash
       before: lightcolour #ff0000       # set it red
       after: lightcolour lightcolour    # set back to default colour
```

**Lightmode action (new in 3.02)**

As well as changing the colour of the status lights you may also wish to change the animation mode so that the lights all flash together rather than the default chasing behaviour.

The ‘lightmode’ action will do this. It takes a keyword as its parameter, which can be either ‘normal’, ‘unison’ or ‘stop’.

‘Normal’ is the default pulsating behaviour. ‘Unison’ will make all the lights flash together. ‘Stop’ will fix all the lights at the current brightness level and stop them animating until it is set back to ‘Normal’ or ‘Unison’ mode. This will not prevent the microphone from modulating the lights if one is enabled.

e.g.

```
gpio: error
       gif: error.gif
       pin: 36
       colour: #ff0000
       drawmode: flash
       before: lightcolour #ff0000       # set it red
       before: lightmode unison          # make them flash together
       after: lightcolour lightcolour    # set back to default colour
       after: lightmode normal           # set back to default animation
```

**SetServo action (new in 3.02)**

If there is a servo motor attached to the Pi, you can use this command to change the angle of the motor. This will perform an immediate movement of the motor to the position indicated.

The parameter is the new motor angle given in degrees (0-359). If you wish it to move slowly over time, use the SeekServo action. Note that not all motors will support the full 360 degrees. Experimentation may be in order.

e.g.

```
gpio: angry
       gif: angry.gif
       pin: 36
       colour: #ff0000
       before: setservo 180              # Move the ears back
       after: setservo 0                 # Move the ears into original position
```

**SeekServo action (new in 3.02)**

If there is a servo motor attached to the Pi, you can use this command to change the angle of the motor. This will gradually rotate the motor to the requested position. The speed can be set using the ServoSpeed action, and the default speed is a 4ms delay between each degree change.

The parameter is the new motor angle given in degrees (0-359). If you wish it to move immediately, use the SetServo action. Note that not all motors will support the full 360 degrees. Experimentation may be in order.

e.g.

```
gpio: angry
       gif: angry.gif
       pin: 36
       colour: #ff0000
       before: servospeed 6              # 6ms delay per degree
       before: seekservo 180             # Move the ears back
       after: setservo 0                 # Move the ears into original position
```

**ServoSpeed action (new in 3.02)**

If there is a servo motor attached to the Pi, you can use this command to change the speed of the servo rotation when using the SeekServo command. SetServo is not affected and will always run at maximum speed.

The parameter is a delay time in milliseconds between each degree change. The default speed is a 4ms delay.

e.g.

```
gpio: angry
       gif: angry.gif
       pin: 36
       colour: #ff0000
       before: servospeed 6              # 6ms delay per degree
       before: seekservo 180             # Move the ears back
       after: setservo 0                 # Move the ears into original position
```

### *Mirror*

As mentioned previously, the software will display expressions as-is on the Transmitter device, and mirror them on the Receiver to avoid giving you a boss-eyed costume.

If the expression contains text or something else which you do not want to be mirrored, you can disable this for a specific expression.

This is done with the command ‘mirror: false’ (or ‘mirror: off’).

Scrolling messages are not normally mirrored – in 3.02 and later you can use ‘mirror: true’ to force them to be mirrored anyway for completeness.

e.g.

```
random: blinking
    gif: blink.gif
    chance: 50
    mirror: false
```

### *ACK disable*

As mentioned in ‘General Setup’ it is possible to flash an LED on the Receiver when an expression change is triggered to acknowledge to the wearer that they’ve successfully changed expression.

By default, this will also light up for expressions such as blinking. Since this may be a problem, you can disable it for certain expressions.

This is done with the command ‘ack: false’ (or ‘ack: off’).

e.g.

```
random: blinking
    gif: blink.gif
    chance: 50
    ack: false
```

### *Background*

You may wish to overlay one expression on top of another. The example that prompted this requirement was blushing – since you may wish to customise the colour of the eye, using the internal colours from the GIF is a problem. Hence, it is better to have the blushing cheek effect on its own, and draw this on top of the default eye. Procedural blinking support needs this too.

The ‘background:’ command is used to do this. It refers to another expression to take the image from, typically the Idle expression. (If you want to use a custom background that is never displayed elsewhere, set up a Scripted expression)

It is intended that the background expression should be a GIF - only the first frame will be displayed as a static backdrop. If you specify a scrolling message or blink, nothing will happen.

Example:

```
gpio: blush
       gif: blush.gif
       pin: 37
       drawmode: colour                # Use GIF's own colours
       background: blush_background    # Draw blush effect on top of this

scripted: blush_background
       gif: blush_eye.gif
       drawmode: monochrome
```

...in this example, the main expression is the cheeks, and the eye is declared as the background in a scripted expression. This is because we need a custom image to raise the eye slightly and make room for the cheeks.

If the eye is not flush with the bottom of the screen, we could just use the Idle expression, e.g.

```
gpio: blush
       gif: blush.gif
       pin: 37
       drawmode: colour      # Use GIF's own colours
       background: idle      # Draw blush effect on top of this
```

### *Interruptible*

With expressions such as the Idle display, we want them to immediately stop when something higher priority comes in, e.g. an expression change via GPIO.

By default, all other animations such as blinking, or the GPIO-triggered expressions themselves will play through completely before yielding, to avoid a sudden jump.

If you want to make an expression (or scrolling message) able to be interrupted, use the ‘interruptible:’ command.

This takes ‘true’ (or ‘on’), or ‘false’ (or ‘off’) as its parameter. It can be used to make any animation or scroller interrupted, or alternatively to stop the Idle animation from accepting interruptions.

Note that making Idle uninterruptible is not recommended as it will introduce latency when switching expressions, but the option is there just in case.

Example:

```
random: blinking
    gif: blink.gif
    chance: 50
    interruptible: true
```

## *Loop (Added in 3.01)*

The software was designed around the principle of displaying a default expression and then running short animations to display an emotion before going back to the default face.

As such, it deliberately ignores the .GIF extension used to make animations loop since it would lock the character into a continuous blinking cycle.

However, you may wish to use static .GIF images and have a new expression remain indefinitely until it's changed to something else. This can be achieved using the 'loop: true' command which will cause the current animation (or static image) to cycle endlessly.

The 'loop:' command will also set the expression to be interruptible, otherwise the display would be stuck on the new expression until you power down.
It is possible to override this by using the 'interruptible: false' command *after* 'loop:' but for the aforementioned reason, this is considered a *bad idea*.

By default, all other animations such as blinking, or the GPIO-triggered expressions themselves will play through completely before yielding, to avoid a sudden jump.

'loop:' will also work on scrolling messages as well as .GIF animations.

Note that any Before or After events will not be looped – the Before events will run before the animation as usual – the After events will only run once the loop has been interrupted, e.g. by a GPIO signal.

By default the software usually switches back to the Idle expression after playing an animation. This can provide a sensible delay between expressions, but can cause flickering if you want to switch between looped animations. You can disable that by using the 'seamless: true' command (see General setup).

Example:

```
gpio: angry
    gif: annoyed.gif
    pin: 37
    loop: true
```

## *Scrolltop (Added in 3.01)*

For a scrolling message, ‘scrolltop:’ sets the height at which the message is displayed. By default it is centred vertically, but if you wish to change this, ‘scrolltop:’ is the command you need.

The height is in pixels from the top of the screen. 0 is the top row and 15 would be the bottom row, though doing this would prevent you from seeing the message.

If the position is negative or greater than 15, it will be ignored. Using ‘scrolltop:’ on a GIF or Blink expression will have no effect.

Example:

```
gpio: scroll
    scroll: Fish on a stick
    pin: 37
    scrolltop: 8
```

## *Blinkspeed (Added in 3.01)*

For a procedural blinking expression, ‘blinkspeed:’ sets the speed at which the blink effect progresses. It is a delay in milliseconds and the default value is 6.

Using ‘blinkspeed:’ on a GIF or Scrolling expression will have no effect.

Example:

```
random: blink
    blink: vertical
    background: eye_picture
    blinkspeed: 8
```

## Source code

The source code and later versions of this document are available at:
`https://github.com/jpmorris33/syntheyes3`