102shows Documentation

Release 0.2-post

Simon Leiner

Aug 23, 2017

Contents:

```
   _______  ___        __
  <  / __ \|__ \ _____/ /_  ____ _      _______
  / / / / /__/ // ___/ __ \/ __ \ | /| / / ___/
 / / /_/ // __/(__  ) / / / /_/ / |/ |/ (__  )
/_/\____//____/____/_/ /_/\____/|__/|__/____/
```

Note: This documentation is still not completely finished. If you are missing something, please open an issue.

CHAPTER 1

Using 102shows

Installation

102shows consists of two parts:

- the **lightshow server**, which should run on a Raspberry Pi - it controls the LED strip via SPI - it listens for MQTT messages that tell it which show to start (and what the paramters for the show are)
- the **UI** - it delivers a nice web interface - it sends the MQTT messages to the server

For the two elements to be able to communicate (via MQTT) you need an **MQTT broker**, for example **`mosquitto`**

All of these can run on the same Raspberry Pi but only the server has to.

MQTT broker

If you already have an MQTT broker in your network, you can use it. Else, install **`mosquitto`** via **`sudo apt-get install mosquitto`**. In any case, you will need the host, port (and maybe access credentials) of your MQTT broker for later.

Server

For the latest **stable** release: In the folder you want to install 102shows in, run:

```
wget -q -O 102s-setup.sh https://git.io/vHydu; chmod u+x 102s-setup.sh; ./
↪102s-setup.sh stable; rm 102s-setup.sh
```

This will launch an assistant that will lead you through the installation process.

Installing a development version

The setup script `102s-setup.sh` takes the GitHub branch to clone as an argument. So, if you want to install the latest development version (which resides on the `master` branch), you should run:

```
wget -q -O 102s-setup.sh https://git.io/vHydu; chmod u+x 102s-setup.sh; ./102s-
↪setup.sh master; rm 102s-setup.sh
```

Web UI

1. Prerequisites

The web UI depends on Node-RED with the dashboard add-on.

- Install Node-RED: Follow the Installation Instructions

Raspbian Tip

There is a special simple installation way for the Raspberry Pi:

```
bash <(curl -sL https://raw.githubusercontent.com/node-red/raspbian-deb-package/
↪master/resources/update-nodejs-and-nodered)
```

Warning: If you have installed any version of **`node-red-contrib-ui`**, you have to uninstall it **before** installing **`node-red-dashboard`**.

- Install the Node-RED dashboard add-on:

```
cd ~/.node-red
npm install node-red-dashboard
```

2. Start Node-RED

Execute **`node-red`** on a console. The Node-RED administration interface should now be available on yournoderedhost:1880

Raspbian Tip

If you want Node-RED to automatically start on boot, execute:

```
sudo systemctl enable nodered.service
```

3. Paste the 102shows UI in Node-RED

Copy the contents of ui/nodered.json into the clipboard. Go to the Node-RED admin interface and in the main menu (upper right corner) choose *Import* >> *Clipboard* and paste the code you copied earlier into the window that is opening. Confirm with *Import*

You should now see the flow **LED control**.

Installing a development version

The link to ui/nodered.json above points to the latest `stable` version.

4. Configure the 102shows UI

In the upper left *LED control* there is a node named **global settings**. Double-click on it to open it and modify the preferences in the code so that they match the settings in your server-side `config.py`.

Save with *Done* and hit the red *Deploy* button on the upper right.

5. Have fun

The UI is now available on yournoderedhost:1880/ui and you should be able to control your LED strips from there

Configuration

Todo

Give configuration advice

Running

Server

1. Start the MQTT broker
2. Execute **`/path/to/102shows/server/run.sh`**

Web UI

Just start Node-RED. The panel should appear on yournoderedhost:1880/ui

CHAPTER 2

Supported LED chipsets

APA102 (aka Adafruit DotStar)

The APA102 is an RGB LED with an integrated driver chip that can be addressed via SPI. That makes it ideal for the Raspberry Pi as talking to an SPI device from Python is really easy. Another advantage of this chip is its support for high SPI data rates (for short strips of less than 200 LEDs you can easily do 8 MHz) which results in very high framerates and smooth-looking animations.

You can find cheap strips on AliExpress etc. or buy them at Adafruit - they sell them as DotStar.

This driver was originally written by tinue and can be found here.

class `drivers.apa102.`**`APA102`**(*num_leds*, *max_clock_speed_hz=4000000*, *max_global_brightness=1.0*)

Note: A very brief overview of the APA102

An APA102 LED is addressed with SPI. The bits are shifted in one by one, starting with the least significant bit.

An LED usually just forwards everything that is sent to its data-in to data-out. While doing this, it remembers its own color and keeps glowing with that color as long as there is power.

An LED can be switched to not forward the data, but instead use the data to change it's own color. This is done by sending (at least) 32 bits of zeroes to data-in. The LED then accepts the next correct 32 bit LED frame (with color information) as its new color setting.

After having received the 32 bit color frame, the LED changes color, and then resumes to just copying data-in to data-out.

The really clever bit is this: While receiving the 32 bit LED frame, the LED sends zeroes on its data-out line. Because a color frame is 32 bits, the LED sends 32 bits of zeroes to the next LED. As we have seen above, this means that the next LED is now ready to accept a color frame and update its color.

So that's really the entire protocol:

- Start by sending 32 bits of zeroes. This prepares LED 1 to update its color.
- Send color information one by one, starting with the color for LED 1, then LED 2 etc.
- Finish off by cycling the clock line a few times to get all data to the very last LED on the strip

The last step is necessary, because each LED delays forwarding the data a bit. Imagine ten people in a row. When you yell the last color information, i.e. the one for person ten, to the first person in the line, then you are not finished yet. Person one has to turn around and yell it to person 2, and so on. So it takes ten additional "dummy" cycles until person ten knows the color. When you look closer, you will see that not even person 9 knows the color yet. This information is still with person 2. Essentially the driver sends additional zeroes to LED 1 as long as it takes for the last color frame to make it down the line to the last LED.

Restrictions of this driver:

- strips cannot have more than 1024 LEDs

The constructor initializes the strip connection via SPI

`clear_buffer``()`

Resets all pixels in the color buffer to `(0,0,0)`.

Return type None

`clear_strip``()`

Clears the color buffer, then invokes a blackout on the strip by calling `show()`

Return type None

`close``()`

Closes the SPI connection to the strip.

Return type None

`color_bytes_to_tuple``()`

Converts a 3-byte color value (like `FF001A`) into an RGB color tuple (like `(255, 0, 26)`).

Parameters **`rgb_color`** (`int`) – a 3-byte RGB color value represented as a base-10 integer

Return type `tuple`

Returns color tuple `(red, green, blue)`

`color_tuple_to_bytes`(*green*, *blue*)

Converts an RGB color tuple (like `(255, 0, 26)`) into a 3-byte color value (like `FF001A`)

Parameters

- **`red`** (`float`) – red component of the tuple (`0.0 - 255.0`)
- **`green`** (`float`) – green component of the tuple (`0.0 - 255.0`)
- **`blue`** (`float`) – blue component of the tuple (`0.0 - 255.0`)

Return type `int`

Returns the tuple components joined into a 3-byte value with each byte representing a color component

`freeze``()`

Freezes the strip. All state-changing methods (`on_color_change()` and `on_brightness_change()`) must not do anything anymore and leave the buffer unchanged.

Return type None

`get_pixel`(*led_num*)

Returns the pixel at index `led_num`

Parameters **`led_num`** (`int`) – the index of the pixel you want to get

Return type `tuple`

Returns `(red, green, blue)` as tuple

classmethod **`led_prefix`**(*brightness*)
generates the first byte of a 4-byte SPI message to a single APA102 module

Parameters **`brightness`** (`float`) – float from 0.0 (off) to 1.0 (full brightness)

Return type `int`

Returns the brightness byte

`max_refresh_time_sec` **= 1**
the maximum time the whole strip takes to refresh

`on_brightness_change`(*led_num*)
For the LED at `led_num`, regenerate the prefix and store the new prefix to the message buffer

Parameters **`led_num`** (`int`) – The index of the LED whose prefix should be regenerated

Return type None

`on_color_change`(*led_num*, *red*, *green*, *blue*)
Changes the message buffer after a pixel was changed in the global color buffer. Also, a grayscale correction is performed. To send the message buffer to the strip and show the changes, you must invoke `show()`

Parameters

- **`led_num`** – index of the pixel to be set
- **`red`** (`float`) – red component of the pixel (`0.0 - 255.0`)
- **`green`** (`float`) – green component of the pixel (`0.0 - 255.0`)
- **`blue`** (`float`) – blue component of the pixel (`0.0 - 255.0`)

Return type None

`rotate`(*positions=1*)
Treating the internal leds array as a circular buffer, rotate it by the specified number of positions. The number can be negative, which means rotating in the opposite direction.

Parameters **`positions`** (`int`) – the number of steps to rotate

Return type None

`set_brightness`(*led_num*, *brightness*)
Sets the brightness for a single LED in the strip. A global multiplier is applied.

Parameters

- **`led_num`** (`int`) – the target LED index
- **`brightness`** (`float`) – the desired brightness (`0.0 - 1.0`)

Return type None

`set_global_brightness`(*brightness*)
Sets a global brightness multiplicator which applies to every single LED's brightness.

Parameters **`brightness`** (`float`) – the global brightness (`0.0 - 1.0`) multiplicator to be set

Return type None

`set_global_brightness_percent`(*brightness*)
Just like `set_global_brightness()`, but with a 0-100 percent value.

Parameters **`brightness`** (`float`) – the global brightness (`0.0 - 100.0`) multiplicator to be set

Return type None

`set_pixel`(*led_num*, *red*, *green*, *blue*)
The buffer value of pixel `led_num` is set to `(red, green, blue)`

Parameters

- **`led_num`** (`int`) – index of the pixel to be set
- **`red`** (`float`) – red component of the pixel (`0.0 - 255.0`)
- **`green`** (`float`) – green component of the pixel (`0.0 - 255.0`)
- **`blue`** (`float`) – blue component of the pixel (`0.0 - 255.0`)

Return type None

`set_pixel_bytes`(*led_num*, *rgb_color*)

Changes the pixel `led_num` to the given color **in the buffer**. To send the buffer to the strip and show the changes, invoke `show()`

If you do not know, how the 3-byte `rgb_color` *works, just use* `set_pixel()` .

Parameters

- **`led_num`** (`int`) – index of the pixel to be set
- **`rgb_color`** (`int`) – a 3-byte RGB color value represented as a base-10 integer

Return type None

`show`()

sends the buffered color and brightness values to the strip

Return type None

static **`spi_end_frame`**()

As explained above, dummy data must be sent after the last real color information so that all of the data can reach its destination down the line. The delay is not as bad as with the human example above. It is only 1/2 bit per LED. This is because the SPI clock line needs to be inverted.

Say a bit is ready on the SPI data line. The sender communicates this by toggling the clock line. The bit is read by the LED, and immediately forwarded to the output data line. When the clock goes down again on the input side, the LED will toggle the clock up on the output to tell the next LED that the bit is ready.

After one LED the clock is inverted, and after two LEDs it is in sync again, but one cycle behind. Therefore, for every two LEDs, one bit of delay gets accumulated. For 300 LEDs, 150 additional bits must be fed to the input of LED one so that the data can reach the last LED. In this implementation we add a few more zero bytes at the end, just to be sure.

Ultimately, we need to send additional *num_leds/2* arbitrary data bits, in order to trigger *num_leds/2* additional clock changes. This driver sends zeroes, which has the benefit of getting LED one partially or fully ready for the next update to the strip. An optimized version of the driver could omit the `spi_start_frame()` method if enough zeroes have been sent as part of `spi_end_frame()`.

Return type `list`

Returns The end frame to be sent at the end of each SPI transmission

static **`spi_start_frame`**()

To start a transmission, one must send 32 empty bits

Return type `list`

Returns The 32-bit start frame to be sent at the beginning of a transmission

`sync_down`()

Reads the shared color and brightness buffers and copies them to the local buffers

Return type None

`sync_up`()

Copies the local color and brightness buffers to the shared buffer so other processes can see the current strip state.

Return type None

unfreeze()
Revokes all effects of `freeze()`

Return type None

No LED Strip (Dummy Driver)

class `drivers.dummy.`**DummyDriver**(*num_leds*, *max_clock_speed_hz=4000000*, *max_global_brightness=1.0*)
A Dummy Driver that just shows the LED states on the logger. This can be useful for developing without having a real LED strip at hand.

clear_buffer()
Resets all pixels in the color buffer to `(0,0,0)`.

Return type None

clear_strip()
Clears the color buffer, then invokes a blackout on the strip by calling `show()`

Return type None

color_bytes_to_tuple()
Converts a 3-byte color value (like `FF001A`) into an RGB color tuple (like `(255, 0, 26)`).

Parameters **rgb_color** (`int`) – a 3-byte RGB color value represented as a base-10 integer

Return type `tuple`

Returns color tuple `(red, green, blue)`

color_tuple_to_bytes(*green*, *blue*)
Converts an RGB color tuple (like `(255, 0, 26)`) into a 3-byte color value (like `FF001A`)

Parameters

- **red** (`float`) – red component of the tuple (`0.0 - 255.0`)
- **green** (`float`) – green component of the tuple (`0.0 - 255.0`)
- **blue** (`float`) – blue component of the tuple (`0.0 - 255.0`)

Return type `int`

Returns the tuple components joined into a 3-byte value with each byte representing a color component

freeze()
Freezes the strip. All state-changing methods (`on_color_change()` and `on_brightness_change()`) must not do anything anymore and leave the buffer unchanged.

Return type None

get_pixel(*led_num*)
Returns the pixel at index `led_num`

Parameters **led_num** (`int`) – the index of the pixel you want to get

Return type `tuple`

Returns `(red, green, blue)` as tuple

rotate(*positions=1*)
Treating the internal leds array as a circular buffer, rotate it by the specified number of positions. The number can be negative, which means rotating in the opposite direction.

Parameters **positions** (int) – the number of steps to rotate

Return type None

set_brightness (*led_num*, *brightness*)
Sets the brightness for a single LED in the strip. A global multiplier is applied.

Parameters

- **led_num** (int) – the target LED index
- **brightness** (float) – the desired brightness (`0.0 - 1.0`)

Return type None

set_global_brightness (*brightness*)
Sets a global brightness multiplicator which applies to every single LED's brightness.

Parameters **brightness** (float) – the global brightness (`0.0 - 1.0`) multiplicator to be set

Return type None

set_global_brightness_percent (*brightness*)
Just like `set_global_brightness()`, but with a 0-100 percent value.

Parameters **brightness** (float) – the global brightness (`0.0 - 100.0`) multiplicator to be set

Return type None

set_pixel (*led_num*, *red*, *green*, *blue*)
The buffer value of pixel `led_num` is set to `(red, green, blue)`

Parameters

- **led_num** (int) – index of the pixel to be set
- **red** (float) – red component of the pixel (`0.0 - 255.0`)
- **green** (float) – green component of the pixel (`0.0 - 255.0`)
- **blue** (float) – blue component of the pixel (`0.0 - 255.0`)

Return type None

set_pixel_bytes (*led_num*, *rgb_color*)
Changes the pixel `led_num` to the given color **in the buffer**. To send the buffer to the strip and show the changes, invoke `show()`

If you do not know, how the 3-byte `rgb_color` *works, just use* `set_pixel()` .

Parameters

- **led_num** (int) – index of the pixel to be set
- **rgb_color** (int) – a 3-byte RGB color value represented as a base-10 integer

Return type None

sync_down ()
Reads the shared color and brightness buffers and copies them to the local buffers

Return type None

sync_up ()
Copies the local color and brightness buffers to the shared buffer so other processes can see the current strip state.

Return type None

unfreeze ()
Revokes all effects of `freeze()`

Return type None

CHAPTER 3

Developing for 102shows

MQTT

The 102shows server can be controlled completely via MQTT. On this page, you see the commands it responds to.

Paths

The general scheme is `{prefix}/{sys_name}/show/{show_name}/{command}`

Switching a show

Starting a show

- **topic**: `{prefix}/{sys_name}/show/start`
- **payload**: JSON Object, for example:

  ```
  {
     "name": "name_of_my_show",
     "parameters": {
        "some_time_sec": 3.5,
        "arbitrary_color": [255, 64, 8]
     }
  }
  ```

  The `parameters` block is optional.
- **retained**: no

Stopping a show

- **topic**: `{prefix}/{sys_name}/show/stop`
- **payload**: none needed
- **retained**: no

Response of the system

- **topic**: `{prefix}/{sys_name}/show/current`
- **payload**: show name as string
- **retained**: yes

The system is sending this message every time the current show is changed.

Global brightness

Setting the global brightness

- **topic**: `{prefix}/{sys_name}/global-brightness/set`
- **payload**: string containing a floating-point number between 0.0 and 1.0
- **retained**: no

Response of the system

- **topic**: `{prefix}/{sys_name}/global-brightness/current`
- **payload**: string containing a floating-point number between 0.0 and 1.0
- **retained**: yes

The system is sending this message every time the brightness is changed.

Show-specific parameters

Setting a parameter

- **topic**: `{prefix}/{sys_name}/show/{show-name}/parameters/set`
- **payload**: JSON
- **retained**: no

Response of the system

- **topic**: `{prefix}/{sys_name}/show/{show-name}/parameters/current`
- **payload**: JSON with all the parameters, for example:

  ```
  {
     "some_time_sec": 3.5,
     "arbitrary_color": [255, 64, 8]
  }
  ```

- **retained**: yes

The system is sending this message every time the parameter is changed.

General commands

The MQTT controller listens for the commands `start` and `stop` for all shows, and all shows (should) respond to the `brightness` command. Any other commands (so all except for `start`, `stop` and `brightness`) are up to the individual lightshow.

`start`

Todo

fix method links

The MQTT controller stops (see below) any running show. Then it checks if the given parameters (the JSON payload of the MQTT start message) are valid by invoking `show.check_runnable()`. If the show calls the parameters valid, the controller starts a new process that runs the method `show.run(strip, parameters)`.

`stop`

The MQTT controller asks the lightshow process kindly to join by sending SIGINT to the show process. The Lightshow base template implements a handler for this signal and usually saves the current strip state and joins after a few milliseconds. However, if the process does not join after 1 second, it is terminated by the controller.

`brightness`

This command is handled by lightshows (in earlier versions, the controller handled brightness changes - but two processes accessing the same strip at the same time causes a lot of trouble). They change the brightness of a strip. Payload is a float from 0 to 100.

Lightshow-specific commands

Each lightshow can implement its own commands, like `foo-color`, `velocity` (of an animation) etc. The name of the parameter must not be `start` or `stop`

Lightshows

Formal interface

- **any show should reside in its own file (*aka module*) under `server/lightshows/`** *for example:* `myshow.py`
- **the module must be registered in the list `__all__` in *`lightshows`*** *for example:*

  ```
  __all__ = ['foo', 'bar', 'myshow']
  ```

- **all lightshows should inherit the basic lightshow template under `lightshows.templates.base`** *for example:*

  ```
  from lightshows.templates.base import *

  def MyShow(Lightshow):
      def run(self):
          ...

      def check_runnable(self):
          ...

      def set_parameter(self):
          ...
  ```

- **it must be registered under `shows` in `config` file** *for example:*

```
configuration.shows('MyShow') = myshow.MyShow
```

creating a `lightshows` object

It is really simple:

```
my_show_object = lightshows.__active__.shows['commonnameofthelightshow'](strip,␣
↪parameters)
```

You could access the lightshow class directly, but the 102shows convention is to access the class by its common name in the `shows` array under `lightshows.active`

There are two arguments that you have to pass to the constructor:

- `strip`: A *`drivers.LEDStrip`* object representing your strip
- `parameters`: A `dict` mapping parameter names (of the lightshow) to the parameter values, for example:

```
parameters = {'example_rgb_color': (255,127,8),
              'an_arbitrary_fade_time_sec': 1.5}
```

See also: The documentation of *`lightshows.templates.base.Lightshow`*

Example

a lightweight example is *`lightshows.solidcolor`*

```
# SolidColor
# (c) 2016-2017 Simon Leiner
# licensed under the GNU Public License, version 2

from helpers.color import blend_whole_strip_to_color
from helpers.preprocessors import list_to_tuple
from lightshows.templates.base import *


class SolidColor(Lightshow):
    """\
    The whole strip shines in the same color.

    Parameters:
       =====================================================================
       ||                     ||     python     ||   JSON representation   ||
       ||       color:        ||   3x1 tuple    ||       3x1 array         ||
       =====================================================================
    """

    def init_parameters(self):
        self.register('color', None, verify.rgb_color_tuple, preprocessor=list_to_
↪tuple)

    def check_runnable(self):
        if self.p.value['color'] is None:
            raise InvalidParameters.missing('color')

    def run(self):
        blend_whole_strip_to_color(self.strip, self.p.value['color'])
```

Other templates

Todo

explain other templates

`ColorCycle`

Todo

explain color cycle

CHAPTER 4

Developer Reference

This will give you an overview of all the classes in 102shows.

`mqttcontrol`

The MQTT controller is the essential idea of 102shows: Starting and controlling lightshows via MQTT without making lightshow development very hard.

The MQTT controller takes care of reading the configuration file and initializing the LED strip with the right driver, providing the MQTT interface for starting and stopping shows (of course) and it ensures that only one lightshow is running at the same time. You can think of it as the "main function" of 102shows that is starting and controlling all things that happen.

class `mqttcontrol.`**`MQTTControl`**(*config*)

This class provides function to start/stop the shows under lightshows/ according to the commands it receives via MQTT

`notify_user`(*message*, *qos=0*)

send to the MQTT notification channel: Node-RED will display a toast notification

Parameters

- **`message`** – the text to be displayed
- **`qos`** – MQTT parameter

Return type None

`on_connect`(*client*, *userdata*, *flags*, *rc*)

subscribe to all messages related to this LED installation

`on_message`(*client*, *userdata*, *msg*)

react to a received message and eventually starts/stops a show

`run`()

start the listener

Return type None

`start_show`(*show_name*, *parameters*)

looks for a show, checks if it can run and if so, starts it in an own process

Parameters

- **show_name** (str) – name of the show to be started
- **parameters** (dict) – these are passed to the show

Return type None

stop_controller (*signum=None*, *frame=None*)
what happens if the controller exits

stop_running_show (*timeout_sec=1*)
stops any running show

Parameters **timeout_sec** (float) – time the show process has until it is terminated

Return type None

stop_show (*show_name*)
stops a show with a given name. If this show is not running, the function does nothing.

Parameters **show_name** (str) – name of the show to be stopped

Return type None

drivers

Structure

102shows is designed to work with several types of LED strips. Currently, only APA102 (aka Adafruit DotStar) chips are supported but other chipsets will be included in the future.

There is also a Dummy driver included. It does not control any LED strip. It merely manages similar internal buffers as a "normal" driver and if `drivers.dummy.DummyDriver.show()` is called, it will print the state of all LEDs in the hypothetical strip to the debug output. This is particular useful for tests on a machine with no actual LED strip attached.

To be able to effortlessly switch between drivers, there is a common interface: All drivers should base on the class *`drivers.LEDStrip`* and be located under `/path/to/102shows/server/drivers`.

Note: For 102shows to find and use the driver, it must have an entry in both `drivers.__all__` and `drivers.__active__.drivers`.

Interface

class drivers.**LEDStrip** (*num_leds*, *max_clock_speed_hz=4000000*, *max_global_brightness=1.0*)
This class provides the general interface for LED drivers that the lightshows use. All LED drivers for 102shows should inherit this class. Mind the following:

- Pixel order is `r,g,b`
- Pixel resolution (number of dim-steps per color component) is 8-bit, so minimum brightness is `0` and maximum brightness is `255`

The constructor stores the given parameters and initializes the color and brightness buffers. Drivers can and should extend this method.

Parameters

- **num_leds** (int) – number of LEDs in the strip
- **max_clock_speed_hz** (int) – maximum clock speed (Hz) of the bus

clear_buffer()
Resets all pixels in the color buffer to (0,0,0).

Return type None

clear_strip()
Clears the color buffer, then invokes a blackout on the strip by calling *show()*

Return type None

close()
An abstract method to be overwritten by the drivers.

It should close the bus connection and clean up any remains.

Return type None

static **color_bytes_to_tuple**()
Converts a 3-byte color value (like FF001A) into an RGB color tuple (like (255, 0, 26)).

Parameters **rgb_color** (int) – a 3-byte RGB color value represented as a base-10 integer

Return type tuple

Returns color tuple (red, green, blue)

static **color_tuple_to_bytes**(*green*, *blue*)
Converts an RGB color tuple (like (255, 0, 26)) into a 3-byte color value (like FF001A)

Parameters

- **red** (float) – red component of the tuple (0.0 - 255.0)
- **green** (float) – green component of the tuple (0.0 - 255.0)
- **blue** (float) – blue component of the tuple (0.0 - 255.0)

Return type int

Returns the tuple components joined into a 3-byte value with each byte representing a color component

freeze()
Freezes the strip. All state-changing methods (*on_color_change()* and *on_brightness_change()*) must not do anything anymore and leave the buffer unchanged.

Return type None

get_pixel(*led_num*)
Returns the pixel at index led_num

Parameters **led_num** (int) – the index of the pixel you want to get

Return type tuple

Returns (red, green, blue) as tuple

max_refresh_time_sec = 1
The maximum time (in *seconds*) that a call of *show()* needs to execute. Currently only used in lightshows.templates.base.sleep()

on_brightness_change(*led_num*)
Reacts to a brightness change at led_num by modifying the message buffer

Parameters **led_num** (int) – number of the LED whose brightness was modified

Return type None

on_color_change(*led_num*, *red*, *green*, *blue*)
Changes the message buffer after a pixel was changed in the global color buffer. To send the buffer to the strip and show the changes, you must invoke *show()*

Parameters

- **`led_num`** – index of the pixel to be set
- **`red`** (`float`) – red component of the pixel (`0.0 - 255.0`)
- **`green`** (`float`) – green component of the pixel (`0.0 - 255.0`)
- **`blue`** (`float`) – blue component of the pixel (`0.0 - 255.0`)

Return type None

`rotate`(*positions=1*)

Treating the internal leds array as a circular buffer, rotate it by the specified number of positions. The number can be negative, which means rotating in the opposite direction.

Parameters **`positions`** (`int`) – the number of steps to rotate

Return type None

`set_brightness`(*led_num*, *brightness*)

Sets the brightness for a single LED in the strip. A global multiplier is applied.

Parameters

- **`led_num`** (`int`) – the target LED index
- **`brightness`** (`float`) – the desired brightness (`0.0 - 1.0`)

Return type None

`set_global_brightness`(*brightness*)

Sets a global brightness multiplicator which applies to every single LED's brightness.

Parameters **`brightness`** (`float`) – the global brightness (`0.0 - 1.0`) multiplicator to be set

Return type None

`set_global_brightness_percent`(*brightness*)

Just like *`set_global_brightness()`*, but with a 0-100 percent value.

Parameters **`brightness`** (`float`) – the global brightness (`0.0 - 100.0`) multiplicator to be set

Return type None

`set_pixel`(*led_num*, *red*, *green*, *blue*)

The buffer value of pixel `led_num` is set to `(red, green, blue)`

Parameters

- **`led_num`** (`int`) – index of the pixel to be set
- **`red`** (`float`) – red component of the pixel (`0.0 - 255.0`)
- **`green`** (`float`) – green component of the pixel (`0.0 - 255.0`)
- **`blue`** (`float`) – blue component of the pixel (`0.0 - 255.0`)

Return type None

`set_pixel_bytes`(*led_num*, *rgb_color*)

Changes the pixel `led_num` to the given color **in the buffer**. To send the buffer to the strip and show the changes, invoke *`show()`*

If you do not know, how the 3-byte `rgb_color` *works, just use* *`set_pixel()`* .

Parameters

- **`led_num`** (`int`) – index of the pixel to be set
- **`rgb_color`** (`int`) – a 3-byte RGB color value represented as a base-10 integer

Return type None

show()

Subclasses should overwrite this method

This method should show the buffered pixels on the strip, e.g. write the message buffer to the port on which the strip is connected.

Return type None

sync_down()

Reads the shared color and brightness buffers and copies them to the local buffers

Return type None

sync_up()

Copies the local color and brightness buffers to the shared buffer so other processes can see the current strip state.

Return type None

synced_red_buffer = None

the individual dim factors for each LED (0-1), EXCLUDING the global dim factor

unfreeze()

Revokes all effects of *freeze()*

Return type None

helpers

Overview

This module includes several helpful functions for 102shows to use. Any functionality that could be used in multiple parts of the program should be defined here.

For example:

- checking if color tuples are valid: *helpers.verify.rgb_color_tuple()*
- add two color tuples: *helpers.color.add_tuples()*
- interpreting an incoming MQTT message: *helpers.mqtt*
- parsing the `config.yml` file: *helpers.configparser*

The module also includes some functions that are just too generic to include them in the one place where they are used.

For example:

- getting the 102shows version: *helpers.get_logo()*
- getting the colored 102shows logo: *helpers.get_version()*

helpers.**get_logo**(*filename='../logo'*)

Returns the colored 102shows logo. It is read from `/path/to/102shows/logo`

Parameters **filename** (`str`) – You can specify another logo source file, if you want.

Return type `str`

Returns The logo as a multiline string. The colors are included as escape characters.

helpers.**get_version**(*filename='../version'*)

Returns the current 102shows version as a string that is read from a special version file

Parameters **filename** (`str`) – Name of the version file. If no name is supplied, the standard file `/path/to/102shows/version` will be used

Return type `str`

Returns version string (as in the file))

color

class `helpers.color.`**`SmoothBlend`**(*strip*)

This class lets the user define a specific state of the strip (*`target_colors`*) and then smoothly blends the current state over to the set state.

class **`BlendFunctions`**

Todo

Include blend pictures directly in documentation

An internal class which provides functions to blend between two colors by a parameter fade_progress for `fade_progress == 0` the function should return the start_color for `fade_progress == 1` the function should return the end_color

classmethod **`cubic_blend`**(*start_color*, *end_color*, *fade_progress*)

cubic blend => see https://goo.gl/wZWm07

Return type `tuple`

classmethod **`linear_blend`**(*start_color*, *end_color*, *fade_progress*)

linear blend => see https://goo.gl/lG8RIW

Return type `tuple`

classmethod **`parabolic_blend`**(*start_color*, *end_color*, *fade_progress*)

quadratic blend => see https://goo.gl/hzeFb6

Return type `tuple`

classmethod **`power_blend`**(*power*, *start_color*, *end_color*, *fade_progress*)

blend two colors using a power function, the exponent is set via param power

Return type `tuple`

`SmoothBlend.`**`blend`**(*time_sec=2*, *blend_function=<bound method SmoothBlend.BlendFunctions.linear_blend of <class 'helpers.color.SmoothBlend.BlendFunctions'>>*)

blend the current LED state to the desired state

`SmoothBlend.`**`set_color_for_whole_strip`**(*red*, *green*, *blue*)

set the same color for all LEDs in the strip

`SmoothBlend.`**`set_pixel`**(*led_num*, *red*, *green*, *blue*)

set the desired state of a given pixel after the blending is finished

`SmoothBlend.`**`target_colors`** = **None**

an array of float tuples

`helpers.color.`**`add_tuples`**(*tuple1*, *tuple2*)

Add two tuples component-wise

Parameters

- **`tuple1`** (`tuple`) – summand
- **`tuple2`** (`tuple`) – summand

Returns sum

`helpers.color.`**`blend_whole_strip_to_color`**(*strip*, *color*, *fadetime_sec=2*)

this name is pretty self-explanatory ;-)

Parameters

- **`strip`** (*`LEDStrip`*) – LEDStrip object
- **`color`** (`tuple`) – the color to blend two
- **`fadetime_sec`** (`float`) – the time in seconds to blend in

Return type None

`helpers.color.`**`grayscale_correction`**(*lightness*, *max_in=255.0*, *max_out=255*)

Corrects the non-linear human perception of the led brightness according to the CIE 1931 standard. This is commonly mistaken for gamma correction.[1]

CIE 1931 Lightness correction[2]

The human perception of brightness is not linear to the duty cycle of an LED. The relation between the (perceived) lightness Y and the (technical) lightness L^* was described by the CIE:

$$\text{with} \quad \text{g(t)} = \begin{cases} 3 \cdot \delta^2 \cdot (t - \frac{4}{29}) & t \leq \delta \\ t^3 & t > \delta \end{cases} \quad , \quad \delta = \frac{6}{29}$$

For more efficient computation, these two formulas can be simplified to:

$$Y = \begin{cases} L^*/902.33 & L^* \leq 8 \\ ((L^* + 16)/116)^3 & L^* > 8 \end{cases}$$

$$0 \leq Y \leq 1 \qquad 0 \leq L^* \leq 100$$

Parameters

- **`lightness`** (`float`) – linear brightness value between 0 and max_in
- **`max_in`** (`float`) – maximum value for lightness
- **`max_out`** (`int`) – maximum output integer value (255 for 8-bit LED drivers)

Returns the correct PWM duty cycle for humans to see the desired lightness as integer

`helpers.color.`**`linear_dim`**(*undimmed*, *factor*)

Multiply all components of undimmed with factor

Parameters

- **`undimmed`** (`tuple`) – the vector
- **`factor`** (`float`) – the factor to multiply the components of the vector byy

Return type `tuple`

Returns resulting RGB color vector

`helpers.color.`**`wheel`**(*wheel_pos*)

Get a color from a color wheel: Green -> Red -> Blue -> Green

Parameters **`wheel_pos`** (`float`) – numeric from 0 to 254

Returns RGB color tuple

[1] For more information, read here: https://goo.gl/9Ji129

[2] formula from Wikipedia

configparser

`helpers.configparser.`**`get_configuration`**(*default_filename='defaults.yml'*, *user_filename='config.yml'*)

gets the current configuration, as specified by YAML files

Parameters

- **`default_filename`** (`str`) – name of the default settings file (relative to `configparser.py`)
- **`user_filename`** (`str`) – name of the user settings file (relative to `configparser.py`)

Return type `AttrDict`

Returns settings tree

`helpers.configparser.`**`update_settings_tree`**(*base*, *update*)

For all attributes in `update` override the defaults set in `base` or add them to the tree, if they did not exist in `base`.

Parameters

- **`base`** (`AttrDict`) – default config tree
- **`update`** (`AttrDict`) – "patch" for the default config tree

Return type `AttrDict`

Returns the updated tree

exceptions

see Exceptions (#fixme: link)

mqtt

A couple of helper functions (big surprise!) for MQTTControl

class `helpers.mqtt.`**`TopicAspect`**

information you can get out of an MQTT topic (and on which path hierarchy they are)

`helpers.mqtt.`**`get_from_topic`**(*hierarchy_level*, *topic*)

get the string on a specified hierarchy level

Parameters

- **`hierarchy_level`** (`int`) – integer level
- **`topic`** (`str`) – string to be analyzed

Return type `str`

Returns part-string of the wanted level

`helpers.mqtt.`**`parse_json_safely`**(*payload*)

parse a string as JSON object logs failures as warnings

Parameters **`payload`** (`str`) – string to be parsed

Return type `dict`

Returns parsed JSON object (as dict)

preprocessors

verify

Functions that validate input parameters and exceptions, raising InvalidParameters exceptions if the input does not fit the requirements. #fixme: link to exception

`helpers.verify.`**`boolean`**(*candidate*, *param_name=None*)

a boolean value: True or False

Parameters

- **`candidate`** – the object to be tested
- **`param_name`** (`Optional[str]`) – name of the parameter (to be included in the error message)

`helpers.verify.`**`integer`**(*candidate*, *param_name=None*, *minimum=None*, *maximum=None*)

Parameters

- **`candidate`** – the object to be tested
- **`param_name`** (`Optional[str]`) – name of the parameter (to be included in the error message)
- **`minimum`** (`Optional[float]`) – minimum
- **`maximum`** (`Optional[float]`) – maximum

`helpers.verify.`**`not_negative_integer`**(*candidate*, *param_name=None*)

a not-negative integer => 0,1,2,3,...

Parameters

- **`candidate`** – the object to be tested
- **`param_name`** (`Optional[str]`) – name of the parameter (to be included in the error message)

`helpers.verify.`**`not_negative_numeric`**(*candidate*, *param_name=None*)

a not-negative number => 0 or above

Parameters

- **`candidate`** – the object to be tested
- **`param_name`** (`Optional[str]`) – name of the parameter (to be included in the error message)

`helpers.verify.`**`numeric`**(*candidate*, *param_name=None*, *minimum=None*, *maximum=None*)

number (between minimum and maximum)

Parameters

- **`candidate`** – the object to be tested
- **`param_name`** (`Optional[str]`) – name of the parameter (to be included in the error message)
- **`minimum`** (`Optional[float]`) – minimum (of a closed set)
- **`maximum`** (`Optional[float]`) – maximum (of a closed set)

`helpers.verify.`**`positive_integer`**(*candidate*, *param_name=None*)

a positive integer => greater than 0 => 1 or above

Parameters

- **`candidate`** – the object to be tested

- **param_name** (Optional[str]) – name of the parameter (to be included in the error message)

`helpers.verify.`**`positive_numeric`**(*candidate*, *param_name=None*)

a positive number => greater than 0

Parameters

- **candidate** – the object to be tested
- **param_name** (Optional[str]) – name of the parameter (to be included in the error message)

`helpers.verify.`**`rgb_color_tuple`**(*candidate*, *param_name=None*)

An RGB color tuple. It must contain three integer components between 0 and 255.

Parameters

- **candidate** – the object to be tested
- **param_name** (Optional[str]) – name of the parameter (to be included in the error message)

Tests

Todo

write docstring for this module

`lightshows`

Overview

102shows offers a framework for writing and displaying lightshows. *`lightshows`* includes the code that actually relies on this and displays animations on an LED strip.

Templates

Todo

include link to controller

To make writing lightshows easy and convenient we introduced templates. These provide the interfaces for the controller and generic functionalities.

Basically: The templates are there so that lightshow modules just have to worry about the LED animations, and not about the backgrounds of 102shows

The base template

As the name says, this is the most basic template. All lightshows (and all other templates) rely on this template. It offers quite a lot:

- **The interface to the controller:**
  - `lightshows.base.Lightshow.name()` returns the name of the lightshow

- **`lightshows.base.Lightshow.start()` initializes the show process,** starts the built-in MQTT client and then triggers the start of the animation
- `lightshows.base.Lightshow.stop()` can be called to gracefully end the show
- `lightshows.base.Lightshow.name()`

class `lightshows.templates.base.`**`Lightshow`**(*strip*, *parameters*)

This class defines the interfaces and a few helper functions for lightshows. It is highly recommended to use it as your base class when writing your own show.

Parameters

- **`strip`** (*`LEDStrip`*) – A *`drivers.LEDStrip`* object representing your strip
- **`parameters`** (`dict`) – A `dict` mapping parameter names (of the lightshow) to the parameter values, for example:

```
parameters = {'example_rgb_color': (255,127,8),
              'an_arbitrary_fade_time_sec': 1.5}
```

class **`MQTTListener`**(*lightshow*)

This class collects the functions that receive incoming MQTT messages and parse them as parameter changes.

`parse_message`(*client*, *userdata*, *msg*)

Function to be executed as `on_message` hook of the Paho MQTT client. If the message commands a brightness or parameter change the corresponding hook (`set_brightness()` or `set_parameter()`) is called.

Todo

•include link to the paho mqtt lib
•explain currently unknown parameters

Parameters

- **`client`** – the calling client object
- **`userdata`** – no idea what this does. This is a necessary argument but is not handled in any way in the function.
- **`msg`** – The object representing the incoming MQTT message

Return type None

`set_brightness`(*brightness*)

Limits the brightness value to the maximum brightness that is set in the configuration file, then calls the strip driver's *`drivers.LEDStrip.set_global_brightness()`* function

Parameters **`brightness`** (`float`) – float between 0.0 and 1.0

Return type None

`start_listening`()

If this method is called (e.g. by the show object), incoming MQTT messages will be parsed, given they have the path `$prefix/$sys_name/$show_name/$parameter` `$parameter` and the `$payload` will be given to `lightshow.templates.base.Lightshow.set_parameter()`

Return type None

`stop_listening`()

Ends the connection to the MQTT broker. Messages from the subscribed topics are not parsed anymore.

Return type None

`subscribe`(*client*, *userdata*, *flags*, *rc*)

Function to be executed as `on_connect` hook of the Paho MQTT client. It subscribes to the MQTT paths for brightness changes and parameter changes for the show.

Todo

- include link to the paho mqtt lib
- explain currently unknown parameters

Parameters

- **`client`** – the calling client object
- **`userdata`** – no idea what this does. This is a necessary argument but is not handled in any way in the function.
- **`flags`** – no idea what this does. This is a necessary argument but is not handled in any way in the function.
- **`rc`** – no idea what this does. This is a necessary argument but is not handled in any way in the function.

Return type None

`Lightshow.`**`apply_parameter_set`**(*parameters*)

Applies a set of parameters to the show.

Parameters **`parameters`** (`dict`) – Parameter JSON Object, represented as a Python `dict`

Return type None

Returns `True` if successful, `False` if not

`Lightshow.`**`check_runnable`**()

Todo

include official exception raise notice

Raise an exception (InvalidStrip, InvalidConf or InvalidParameters) if the show is not runnable

`Lightshow.`**`cleanup`**()

This is called before the show gets terminated. Lightshows can use it to clean up resources before their process is killed.

Return type None

`Lightshow.`**`idle_forever`**(*delay_sec=-1*)

Just does nothing and invokes *`drivers.LEDStrip.show()`* until the end of time (or a call of `stop()`)

Parameters **`delay_sec`** (`float`) – Time between two calls of *`drivers.LEDStrip.show()`*

Return type None

`Lightshow.`**`init_parameters`**()

Lightshows can inherit this to set their default parameters. This function is called at initialization of a new show object.

Return type None

`Lightshow.`**`logger`** **= None**

The logger object this show will use for debug output

`Lightshow.`**`mqtt`** **= None**

represents the MQTT connection for parsing parameter changes #FIXME: type annotation

`Lightshow.`**`name`**

The name of the lightshow in lower-cases

Return type `str`

`Lightshow.`**`p`** **= None**

The object that stores all show parameters

`Lightshow.`**`register`**`(`*parameter_name*, *default_val*, *verifier*, *args=None*, *kwargs=None*, *preprocessor=None*`)`

MQTT-settable parameters are stored in `lightshows.templates.base.Lightshow.p.value`. Calling this function will register a new parameter and his verifier in `value` and `verifier`, so the parameter can be set via MQTT and by the controller.

Parameters

- **`parameter_name`** (`str`) – name of the parameter. You access the parameter via self.p.value[parameter_name].
- **`default_val`** – initializer value of the parameter. *Note that this value will not be checked by the verifier function!*
- **`verifier`** – a function that is called before the parameter is set via MQTT. If it raises an InvalidParameters exception, the new value will not be set. #FIXLINK
- **`args`** (`Optional[list]`) – the verifier function will be called as `verifier(new_value, param_name, *args, **kwargs)`
- **`kwargs`** (`Optional[dict]`) – the verifier function will be called via `verifier(new_value, param_name, *args, **kwargs)`
- **`preprocessor`** – before the validation in set_parameter `value = preprocessor(value)` will be called

Return type None

`Lightshow.`**`run`**`()`

The "main" function of the show (obviously this must be re-implemented in child classes)

Return type None

`Lightshow.`**`set_parameter`**`(`*param_name*, *value*, *send_mqtt_update=True*`)`

Take a parameter by name and new value and store it to p.value.

Parameters

- **`param_name`** (`str`) – name of the parameter to be stored
- **`value`** – new value of the parameter to be stored
- **`send_mqtt_update`** (`bool`) – Send the updated parameter array to the MQTT current parameter path after update

Return type None

`Lightshow.`**`sleep`**`(`*time_sec*`)`

Does nothing (but refreshing the strip a few times) for `time_sec` seconds

Parameters **`time_sec`** (`float`) – duration of the break

Return type None

`Lightshow.`**`start`**`()`

invokes the `run()` method and after that synchronizes the shared buffer

Return type None

`Lightshow.`**`stop`**`(`*signum=None*, *frame=None*`)`

Todo

include link for SIGINT

This should be called to stop the show with a graceful ending. It guarantees that the last strip state is uploaded to the global inter-process buffer. This method is called when SIGINT is sent to the show process. The arguments have no influence on the function.

Parameters

- **`signum`** – The integer-code of the signal sent to the show process. This has no influence on how the function works.
- **`frame`** – #fixme

Return type None

`Lightshow.`**`strip`** **= None**

the object representing the LED strip (driver) #FIXME: type annotation

`Lightshow.`**`suicide`**`()`

terminates its own process

Return type None

class `lightshows.templates.base.`**`LightshowParameters`**

A collection of maps for the parameters which store their:

- current values
- preprocessor method references
- verifier method references

`preprocessor` **= None**

maps the show parameter names to their preprocessor functions

`value` **= None**

maps the show parameter names to their current values

`verifier` **= None**

maps the show parameter names to their verifier functions

Exceptions

This module defines some exception classes specific to 102shows:

exception `helpers.exceptions.`**`DescriptiveException`**`(`*value*`)`

This type of exception must contain a value (usually a string) that is used as the string representation of the exception

exception `helpers.exceptions.`**`InvalidConf`**`(`*value*`)`

Use if something in the configuration will not work for what the user has chosen in the config file.

exception `helpers.exceptions.`**`InvalidParameters`**`(`*value*`)`

Use when given parameters (for a lightshow) are not valid

static **`missing`**`()`

Todo

document!

static **`unknown`**`()`

Todo

document!

---

exception `helpers.exceptions.`**`InvalidStrip`**`(`*value*`)`

Use if something is wrong with the strip.

For example: not enough LEDs to run the selected lightshow

CHAPTER 5

Thanks!

- To tinue for the APA102_Pi library. This was the code that 102shows was originally based on.
- The authors and contributors of the libraries that 102shows uses:
  - paho_mqtt
  - spidev
  - PyYAML
  - orderedattrdict
  - coloredlogs
- The people of Sphinx, the great tool that is used for this documentation and the authors and contributors of the plugins for Sphinx that we use:
  - sphinx-autodoc-typehints
  - sphinx_rtd_theme

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

CHAPTER 7

Trouble?

Open an issue on GitHub or write an email to me: 102shows@leiner.me

Python Module Index

Index