Luma.OLED Documentation

Release 2.2.4

Richard Hull and contributors

Feb 17, 2017

Contents

CHAPTER 1

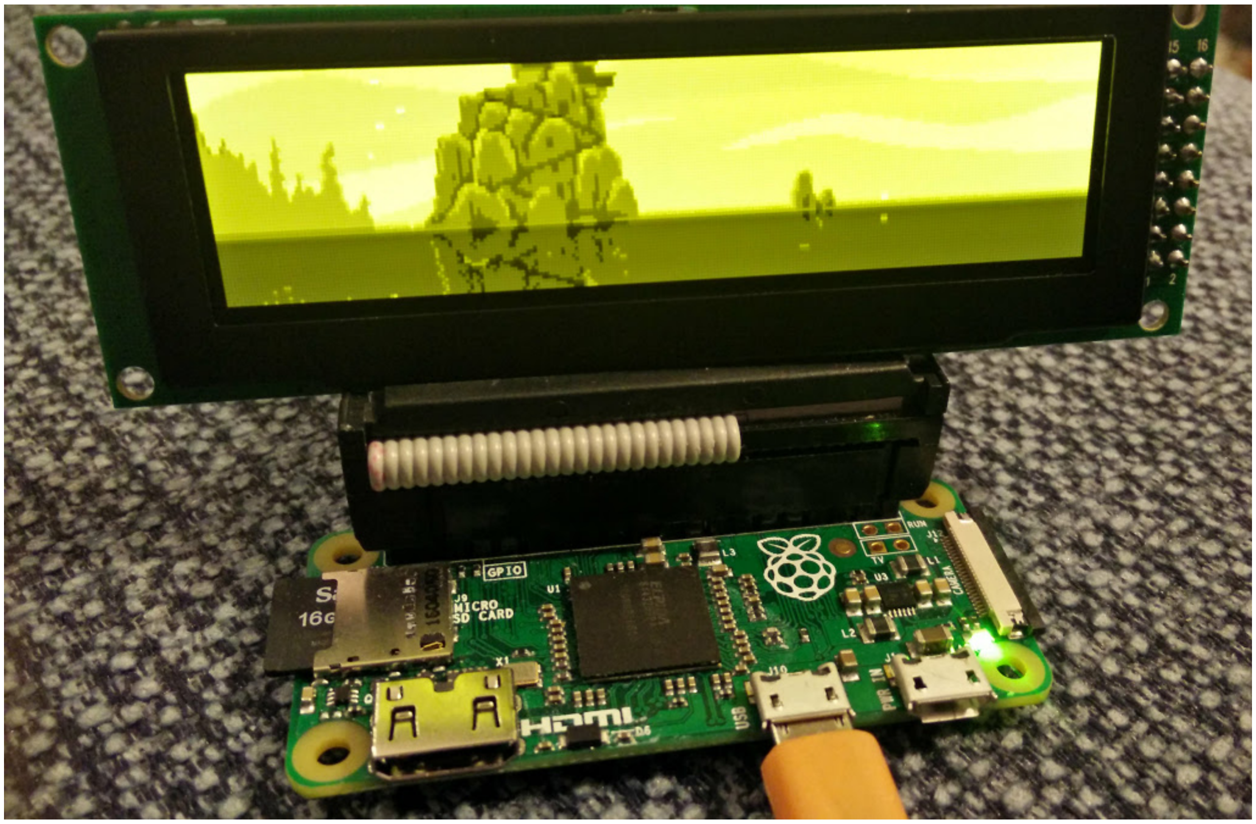
Introduction

Interfacing OLED matrix displays with the SSD1306, SSD1322, SSD1325, SSD1331 or SH1106 driver in Python 2 or 3 using I2C/SPI on the Raspberry Pi and other linux-based single-board computers: the library provides a Pillow-compatible drawing canvas, and other functionality to support:

- scrolling/panning capability,
- terminal-style printing,
- state management,
- color/greyscale (where supported),
- dithering to monochrome

The SSD1306 display pictured below is 128 x 64 pixels, and the board is *tiny*, and will fit neatly inside the RPi case.

```
Ld:0.2 0.2 0.2 Up: 3:00:30
Mem: 225M 86%
SD:  4G 31%
wlan0: Tx6M, Rx4M
```



See also:

Further technical information for the specific devices can be found in the datasheets below:

- `SSD1306`
- `SSD1322`
- `SSD1325`
- `SSD1331`
- `SH1106`

Benchmarks for tested devices can be found in the wiki.

As well as display drivers for various physical OLED devices there are emulators that run in real-time (with pygame) and others that can take screenshots, or assemble animated GIFs, as per the examples below (source code for these is available in the luma.examples git repository:

CHAPTER 2

Python usage

OLED displays can be driven with python using the varous implementations in the *luma.oled.device* package. There are several device classes available and usage is very simple if you have ever used Pillow or PIL.

First, import and initialise the device:

```
from luma.core.serial import i2c, spi
from luma.core.render import canvas
from luma.oled.device import ssd1306, ssd1325, ssd1331, sh1106

# rev.1 users set port=0
# substitute spi(device=0, port=0) below if using that interface
serial = i2c(port=1, address=0x3C)

# substitute ssd1331(...) or sh1106(...) below if using that device
device = ssd1306(serial)
```

The display device should now be configured for use. The specific *luma.oled.device.ssd1306*, *luma.oled.device.ssd1325*, *luma.oled.device.ssd1331*, or *luma.oled.device.sh1106*, classes all expose a `display()` method which takes an image with attributes consistent with the capabilities of the device. However, for most cases, for drawing text and graphics primitives, the canvas class should be used as follows:

```
with canvas(device) as draw:
    draw.rectangle(device.bounding_box, outline="white", fill="black")
    draw.text((30, 40), "Hello World", fill="white")
```

The *luma.core.render.canvas* class automatically creates an `PIL.ImageDraw` object of the correct dimensions and bit depth suitable for the device, so you may then call the usual Pillow methods to draw onto the canvas.

As soon as the with scope is ended, the resultant image is automatically flushed to the device's display memory and the `PIL.ImageDraw` object is garbage collected.

Color Model

Any of the standard `PIL.ImageColor` color formats may be used, but since the SSD1306 and SH1106 OLEDs are monochrome, only the HTML color names `"black"` and `"white"` values should really be used; in fact, by default, any value *other* than black is treated as white. The `luma.core.canvas` object does have a `dither` flag which if set to True, will convert color drawings to a dithered monochrome effect (see the *3d_box.py* example, below).

```
with canvas(device, dither=True) as draw:
    draw.rectangle((10, 10, 30, 30), outline="white", fill="red")
```

There is no such constraint on the SSD1331 OLED which features 16-bit RGB colors: 24-bit RGB images are downsized to 16-bit using a 565 scheme.

The SSD1322 and SSD1325 OLEDs both support 16 greyscale graduations: 24-bit RGB images are downsized to 4-bit using a Luma conversion which is approximately calculated as follows:

```
Y' = 0.299 R' + 0.587 G' + 0.114 B'
```

Landscape / Portrait Orientation

By default the display will be oriented in landscape mode (128x64 pixels for the SSD1306, for example). Should you have an application that requires the display to be mounted in a portrait aspect, then add a `rotate=N` parameter when creating the device:

```
from luma.core.serial import i2c
from luma.core.render import canvas
from luma.oled.device import ssd1306, ssd1325, ssd1331, sh1106

serial = i2c(port=1, address=0x3C)
device = ssd1306(serial, rotate=1)

# Box and text rendered in portrait mode
with canvas(device) as draw:
    draw.rectangle(device.bounding_box, outline="white", fill="black")
    draw.text((10, 40), "Hello World", fill="white")
```

N should be a value of 0, 1, 2 or 3 only, where 0 is no rotation, 1 is rotate 90° clockwise, 2 is 180° rotation and 3 represents 270° rotation.

The `device.size`, `device.width` and `device.height` properties reflect the rotated dimensions rather than the physical dimensions.

Examples

After installing the library, head over to the luma.examples repository, and try running the following examples (and more):

Example	Description
3d_box.py	Rotating 3D box wireframe & color dithering
bounce.py	Display a bouncing ball animation and frames per second
carousel.py	Showcase viewport and hotspot functionality
clock.py	An analog clockface with date & time
colors.py	Color rendering demo
crawl.py	A vertical scrolling demo, which should be familiar
demo.py	Use misc draw commands to create a simple image
game_of_life.py	Conway's game of life
grayscale.py	Greyscale rendering demo
invaders.py	Space Invaders demo
maze.py	Maze generator
perfloop.py	Simple benchmarking utility to measure performance
pi_logo.py	Display the Raspberry Pi logo (loads image as .png)
savepoint.py	Example of savepoint/restore functionality
starfield.py	3D starfield simulation
sys_info.py	Display basic system information
terminal.py	Simple println capabilities
tv_snow.py	Example image-blitting
tweet_scroll.py	Using Twitter's Streaming API to display scrolling notifications
welcome.py	Unicode font rendering & scrolling

Further details of how to run the examples is shown in the example repo's README.

Emulators

There are various display emulators available for running code against, for debugging and screen capture functionality:

- The *luma.core.emulator.capture* device will persist a numbered PNG file to disk every time its `display` method is called.
- The *luma.core.emulator.gifanim* device will record every image when its `display` method is called, and on program exit (or Ctrl-C), will assemble the images into an animated GIF.
- The *luma.core.emulator.pygame* device uses the `pygame` library to render the displayed image to a pygame display surface.

Invoke the demos with:

```
$ python examples/clock.py -d capture
```

or:

```
$ python examples/clock.py -d pygame
```

Note: *Pygame* is required to use any of the emulated devices, but it is **NOT** installed as a dependency by default, and so must be manually installed before using any of these emulation devices (e.g. `pip install pygame`).

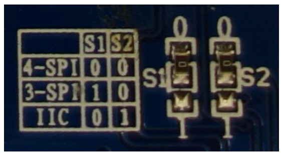
CHAPTER 3

Hardware

Identifying your serial interface

You can determine if you have an I2C or a SPI interface by counting the number of pins on your card. An I2C display will have 4 pins while an SPI interface will have 6 or 7 pins.

If you have a SPI display, check the back of your display for a configuration such as this:



	S1	S2
4-SPI	0	0
3-SPI	1	0
IIC	0	1

For this display, the two 0 Ohm (jumper) resistors have been connected to “0” and the table shows that “0 0” is 4-wire SPI. That is the type of connection that is currently supported by the SPI mode of this library.

A list of tested devices can be found in the wiki.

I2C vs. SPI

If you have not yet purchased your display, you may be wondering if you should get an I2C or SPI display. The basic trade-off is that I2C will be easier to connect because it has fewer pins while SPI may have a faster display update rate due to running at a higher frequency and having less overhead (see benchmarks).

Tips for connecting the display

- If you don’t want to solder directly on the Pi, get 2.54mm 40 pin female single row headers, cut them to length, push them onto the Pi pins, then solder wires to the headers.
- If you need to remove existing pins to connect wires, be careful to heat each pin thoroughly, or circuit board traces may be broken.

- Triple check your connections. In particular, do not reverse VCC and GND.

Pre-requisites

I2C

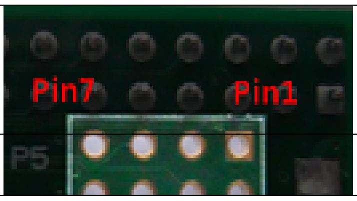
The P1 header pins should be connected as follows:

OLED Pin	Name	Remarks	RPi Pin	RPi Function
1	GND	Ground	P01-6	GND
2	VCC	+3.3V Power	P01-1	3V3
3	SCL	Clock	P01-5	GPIO 3 (SCL)
4	SDA	Data	P01-3	GPIO 2 (SDA)

You can also solder the wires directly to the underside of the RPi GPIO pins.

See also:

Alternatively, on rev.2 RPi's, right next to the male pins of the P1 header, there is a bare P5 header which features I2C channel 0, although this doesn't appear to be initially enabled and may be configured for use with the Camera module.

OLED Pin	Name	Remarks	RPi Pin	RPi Function	Location
1	GND	Ground	P5-07	GND	
2	VCC	+3.3V Power	P5-02	3V3	
3	SCL	Clock	P5-04	GPIO 29 (SCL)	
4	SDA	Data	P5-03	GPIO 28 (SDA)	

Ensure that the I2C kernel driver is enabled:

```
$ dmesg | grep i2c
[    4.925554] bcm2708_i2c 20804000.i2c: BSC1 Controller at 0x20804000 (irq 79)␣
↪(baudrate 100000)
[    4.929325] i2c /dev entries driver
```

or:

```
$ lsmod | grep i2c
i2c_dev                 5769  0
i2c_bcm2708             4943  0
regmap_i2c              1661  3 snd_soc_pcm512x,snd_soc_wm8804,snd_soc_core
```

If you have no kernel modules listed and nothing is showing using `dmesg` then this implies the kernel I2C driver is not loaded. Enable the I2C as follows:

```
$ sudo raspi-config
> Advanced Options > A7 I2C
```

After rebooting re-check that the `dmesg | grep i2c` command shows whether I2C driver is loaded before proceeding. You can also enable I2C manually if the `raspi-config` utility is not available.

Optionally, to improve performance, increase the I2C baudrate from the default of 100KHz to 400KHz by altering `/boot/config.txt` to include:

```
dtparam=i2c_arm=on,i2c_baudrate=400000
```

Then reboot.

Next, add your user to the *i2c* group and install `i2c-tools`:

```
$ sudo usermod -a -G i2c pi
$ sudo apt-get install i2c-tools
```

Logout and in again so that the group membership permissions take effect, and then check that the device is communicating properly (if using a rev.1 board, use 0 for the bus, not 1):

```
$ i2cdetect -y 1
       0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
  00:          -- -- -- -- -- -- -- -- -- -- -- -- --
  10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
  20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
  30: -- -- -- -- -- -- -- -- -- -- -- UU 3c -- -- --
  40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
  50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
  60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
  70: -- -- -- -- -- -- -- --
```

According to the man-page, "UU" means that probing was skipped, because the address was in use by a driver. It suggest that there is a chip at that address. Indeed the documentation for the device indicates it uses two addresses.

SPI

The GPIO pins used for this SPI connection are the same for all versions of the Raspberry Pi, up to and including the Raspberry Pi 3 B.

OLED Pin	Name	Remarks	RPi Pin	RPi Function
1	VCC	+3.3V Power	P01-17	3V3
2	GND	Ground	P01-20	GND
3	D0	Clock	P01-23	GPIO 11 (SCLK)
4	D1	MOSI	P01-19	GPIO 10 (MOSI)
5	RST	Reset	P01-22	GPIO 25
6	DC	Data/Command	P01-18	GPIO 24
7	CS	Chip Select	P01-24	GPIO 8 (CE0)

Note:

- When using the 4-wire SPI connection, Data/Command is an "out of band" signal that tells the controller if you're sending commands or display data. This line is not a part of SPI and the library controls it with a separate GPIO pin. With 3-wire SPI and I2C, the Data/Command signal is sent "in band".
- If you're already using the listed GPIO pins for Data/Command and/or Reset, you can select other pins and pass a `bcm_DC` and/or a `bcm_RST` argument specifying the new *BCM* pin numbers in your serial interface create call.
- The use of the terms 4-wire and 3-wire SPI are a bit confusing because, in most SPI documentation, the terms are used to describe the regular 4-wire configuration of SPI and a 3-wire mode where the input and output lines, MOSI and MISO, have been combined into a single line called SISO. However, in the context of these OLED controllers, 4-wire means MOSI + Data/Command and 3-wire means Data/Command sent as an extra bit over MOSI.
- Because CS is connected to CE0, the display is available on SPI port 0. You can connect it to CE1 to have it available on port 1. If so, pass `port=1` in your serial interface create call.

Enable the SPI port:

```
$ sudo raspi-config
> Advanced Options > A6 SPI
```

If `raspi-config` is not available, enabling the SPI port can be done manually.

Ensure that the SPI kernel driver is enabled:

```
$ ls -l /dev/spi*
crw-rw---- 1 root spi 153, 0 Nov 25 08:32 /dev/spidev0.0
crw-rw---- 1 root spi 153, 1 Nov 25 08:32 /dev/spidev0.1
```

or:

```
$ lsmod | grep spi
spi_bcm2835             6678  0
```

Then add your user to the *spi* and *gpio* groups:

```
$ sudo usermod -a -G spi,gpio pi
```

Log out and back in again to ensure that the group permissions are applied successfully.

CHAPTER 4

Installation

Warning: Ensure that the *Pre-requisites* from the previous section have been performed, checked and tested before proceeding.

Note: The library has been tested against Python 2.7, 3.4 and 3.5.

For **Python3** installation, substitute the following in the instructions below.

- `pip` `pip3`,
- `python` `python3`,
- `python-dev` `python3-dev`,
- `python-pip` `python3-pip`.

It was *originally* tested with Raspbian on a rev.2 model B, with a vanilla kernel version 4.1.16+, and has subsequently been tested on Raspberry Pi model A, model B2 and 3B (Debian Jessie) and OrangePi Zero (Armbian Jessie).

From PyPI

Note: This is the preferred installation mechanism.

Install the latest version of the library directly from PyPI:

```
$ sudo apt-get install python-dev python-pip libfreetype6-dev libjpeg8-dev
$ sudo -H pip install --upgrade pip
$ sudo -H pip install --upgrade luma.oled
```

From source

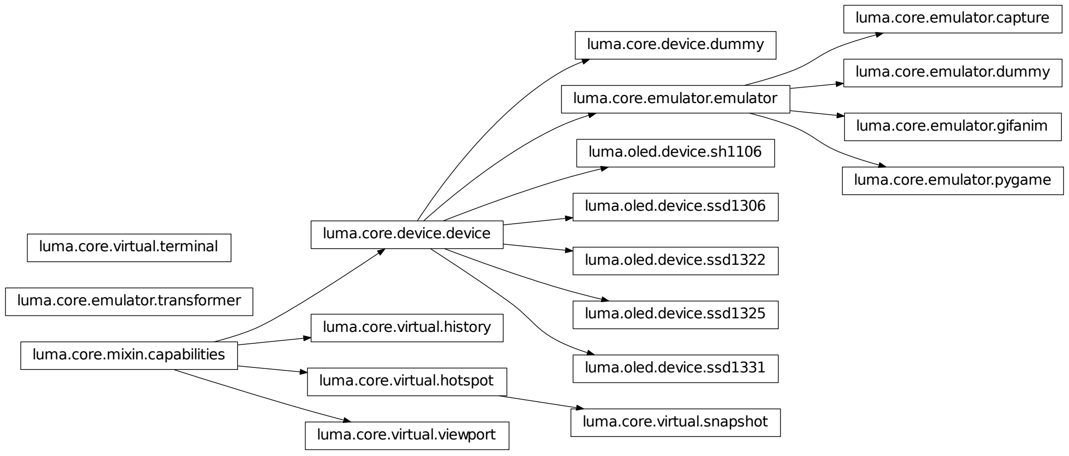
For Python 2, from the bash prompt, enter (for Raspbian, other OSes may be different):

```
$ git clone https://github.com/rm-hull/luma.oled.git
$ cd luma.oled
$ sudo apt-get install python-dev python-pip libfreetype6-dev libjpeg8-dev
$ sudo python setup.py install
```

CHAPTER 5

API Documentation

OLED display driver for SSD1306, SSD1322, SSD1325, SSD1331 and SH1106 devices.


luma.core.emulator.capture
luma.core.device.dummy
luma.core.emulator.dummy
luma.core.emulator.emulator
luma.core.emulator.gifanim
luma.oled.device.sh1106
luma.core.emulator.pygame
luma.oled.device.ssd1306
luma.core.virtual.terminal
luma.core.device.device
luma.oled.device.ssd1322
luma.core.emulator.transformer
luma.oled.device.ssd1325
luma.core.virtual.history
luma.core.mixin.capabilities
luma.oled.device.ssd1331
luma.core.virtual.hotspot
luma.core.virtual.snapshot
luma.core.virtual.viewport

Breaking changes

Warning: Version 2.0.0 was released on 11 January 2017: this came with a rename of the project in github from **ssd1306** to **luma.oled** to reflect the changing nature of the codebase. It introduces some structural changes to the package structure, namely breaking the library up into smaller components and renaming existing packages.

This should largely be restricted to having to update import statements only. To upgrade any existing code that uses the old package structure:

- rename instances of `oled.device` to `luma.oled.device`.
- rename any other usages of `oled.*` to `luma.core.*`.

This breaking change was necessary to be able to add different classes of devices, so that they could reuse core components.

`luma.core.device`

class `luma.core.device.`**`device`**(*const=None*, *serial_interface=None*)

Bases: *`luma.core.mixin.capabilities`*

Base class for display driver classes

Warning: Direct use of the *`command()`* and *`data()`* methods are discouraged: Screen updates should be effected through the *`display()`* method, or preferably with the *`luma.core.render.canvas`* context manager.

`capabilities`(*width*, *height*, *rotate*, *mode='1'*)

`cleanup`()

Attempt to switch the device off or put into low power mode (this helps prolong the life of the device), clear the screen and close resources associated with the underlying serial interface.

This is a managed function, which is called when the python processs is being shutdown, so shouldn't usually need be called directly in application code.

`clear`()

Initializes the device memory with an empty (blank) image.

`command`(**cmd*)

Sends a command or sequence of commands through to the delegated serial interface.

`contrast`(*level*)

Switches the display contrast to the desired level, in the range 0-255. Note that setting the level to a low (or zero) value will not necessarily dim the display to nearly off. In other words, this method is **NOT** suitable for fade-in/out animation.

Parameters **`level`** (*`int`*) – Desired contrast level in the range of 0-255.

`data`(*data*)

Sends a data byte or sequence of data bytes through to the delegated serial interface.

`display`(*image*)

`hide`()

Switches the display mode OFF, putting the device in low-power sleep mode.

`preprocess`(*image*)

`show`()

Sets the display mode ON, waking the device out of a prior low-power sleep mode.

class `luma.core.device.`**`dummy`**(*width=128*, *height=64*, *rotate=0*, *mode='RGB'*, ***kwargs*)

Bases: `luma.core.device.device`

Pseudo-device that acts like a physical display, except that it does nothing other than retain a copy of the displayed image. It is mostly useful for testing. Supports 24-bit color depth.

`capabilities`(*width*, *height*, *rotate*, *mode='1'*)

`cleanup`()

Attempt to switch the device off or put into low power mode (this helps prolong the life of the device), clear the screen and close resources associated with the underlying serial interface.

This is a managed function, which is called when the python processs is being shutdown, so shouldn't usually need be called directly in application code.

`clear`()

Initializes the device memory with an empty (blank) image.

`command`(**cmd*)

Sends a command or sequence of commands through to the delegated serial interface.

`contrast`(*level*)

Switches the display contrast to the desired level, in the range 0-255. Note that setting the level to a low (or zero) value will not necessarily dim the display to nearly off. In other words, this method is **NOT** suitable for fade-in/out animation.

Parameters **`level`** (`int`) – Desired contrast level in the range of 0-255.

`data`(*data*)

Sends a data byte or sequence of data bytes through to the delegated serial interface.

`display`(*image*)

Takes a `PIL.Image` and makes a copy of it for later use/inspection.

`hide`()

Switches the display mode OFF, putting the device in low-power sleep mode.

`preprocess`(*image*)

`show`()

Sets the display mode ON, waking the device out of a prior low-power sleep mode.

luma.core.emulator

class `luma.core.emulator.`**`capture`**(*width=128*, *height=64*, *rotate=0*, *mode='RGB'*, *transform='scale2x'*, *scale=2*, *file_template='luma_{0:06}.png'*, ***kwargs*)

Bases: `luma.core.emulator.emulator`

Pseudo-device that acts like a physical display, except that it writes the image to a numbered PNG file when the `display()` method is called. Supports 24-bit color depth.

`capabilities`(*width*, *height*, *rotate*, *mode='1'*)

`cleanup`()

`clear`()

Initializes the device memory with an empty (blank) image.

`command`(**cmd*)

Sends a command or sequence of commands through to the delegated serial interface.

contrast(*level*)
Switches the display contrast to the desired level, in the range 0-255. Note that setting the level to a low (or zero) value will not necessarily dim the display to nearly off. In other words, this method is **NOT** suitable for fade-in/out animation.

Parameters **level** (`int`) – Desired contrast level in the range of 0-255.

data(*data*)
Sends a data byte or sequence of data bytes through to the delegated serial interface.

display(*image*)
Takes a `PIL.Image` and dumps it to a numbered PNG file.

hide()
Switches the display mode OFF, putting the device in low-power sleep mode.

preprocess(*image*)

show()
Sets the display mode ON, waking the device out of a prior low-power sleep mode.

to_surface(*image*)
Converts a `PIL.Image` into a `pygame.Surface`, transforming it according to the `transform` and `scale` constructor arguments.

class `luma.core.emulator.`**dummy**(*width=128*, *height=64*, *rotate=0*, *mode='RGB'*, *transform='scale2x'*, *scale=2*, ***kwargs*)
Bases: *`luma.core.emulator.emulator`*

Pseudo-device that acts like a physical display, except that it does nothing other than retain a copy of the displayed image. It is mostly useful for testing. Supports 24-bit color depth.

capabilities(*width*, *height*, *rotate*, *mode='1'*)

cleanup()

clear()
Initializes the device memory with an empty (blank) image.

command(**cmd*)
Sends a command or sequence of commands through to the delegated serial interface.

contrast(*level*)
Switches the display contrast to the desired level, in the range 0-255. Note that setting the level to a low (or zero) value will not necessarily dim the display to nearly off. In other words, this method is **NOT** suitable for fade-in/out animation.

Parameters **level** (`int`) – Desired contrast level in the range of 0-255.

data(*data*)
Sends a data byte or sequence of data bytes through to the delegated serial interface.

display(*image*)
Takes a `PIL.Image` and makes a copy of it for later use/inspection.

hide()
Switches the display mode OFF, putting the device in low-power sleep mode.

preprocess(*image*)

show()
Sets the display mode ON, waking the device out of a prior low-power sleep mode.

to_surface(*image*)
Converts a `PIL.Image` into a `pygame.Surface`, transforming it according to the `transform` and `scale` constructor arguments.

class `luma.core.emulator.`**emulator**(*width*, *height*, *rotate*, *mode*, *transform*, *scale*)
Bases: `luma.core.device.device`

Base class for emulated display driver classes

capabilities(*width*, *height*, *rotate*, *mode='1'*)

cleanup()

clear()
Initializes the device memory with an empty (blank) image.

command(**cmd*)
Sends a command or sequence of commands through to the delegated serial interface.

contrast(*level*)
Switches the display contrast to the desired level, in the range 0-255. Note that setting the level to a low (or zero) value will not necessarily dim the display to nearly off. In other words, this method is **NOT** suitable for fade-in/out animation.

Parameters **level** (*int*) – Desired contrast level in the range of 0-255.

data(*data*)
Sends a data byte or sequence of data bytes through to the delegated serial interface.

display(*image*)

hide()
Switches the display mode OFF, putting the device in low-power sleep mode.

preprocess(*image*)

show()
Sets the display mode ON, waking the device out of a prior low-power sleep mode.

to_surface(*image*)
Converts a `PIL.Image` into a `pygame.Surface`, transforming it according to the `transform` and `scale` constructor arguments.

class `luma.core.emulator.`**gifanim**(*width=128*, *height=64*, *rotate=0*, *mode='RGB'*, *transform='scale2x'*, *scale=2*, *filename='luma_anim.gif'*, *duration=0.01*, *loop=0*, *max_frames=None*, ***kwargs*)
Bases: `luma.core.emulator.emulator`

Pseudo-device that acts like a physical display, except that it collects the images when the `display()` method is called, and on exit, assembles them into an animated GIF image. Supports 24-bit color depth, albeit with an indexed color palette.

capabilities(*width*, *height*, *rotate*, *mode='1'*)

cleanup()

clear()
Initializes the device memory with an empty (blank) image.

command(**cmd*)
Sends a command or sequence of commands through to the delegated serial interface.

contrast(*level*)
Switches the display contrast to the desired level, in the range 0-255. Note that setting the level to a low (or

zero) value will not necessarily dim the display to nearly off. In other words, this method is **NOT** suitable for fade-in/out animation.

Parameters **`level`** (`int`) – Desired contrast level in the range of 0-255.

`data`(*data*)
Sends a data byte or sequence of data bytes through to the delegated serial interface.

`display`(*image*)
Takes an image, scales it according to the nominated transform, and stores it for later building into an animated GIF.

`hide`()
Switches the display mode OFF, putting the device in low-power sleep mode.

`preprocess`(*image*)

`show`()
Sets the display mode ON, waking the device out of a prior low-power sleep mode.

`to_surface`(*image*)
Converts a `PIL.Image` into a `pygame.Surface`, transforming it according to the `transform` and `scale` constructor arguments.

`write_animation`()

class `luma.core.emulator.`**`pygame`**(*width=128*, *height=64*, *rotate=0*, *mode='RGB'*, *transform='scale2x'*, *scale=2*, *frame_rate=60*, ***kwargs*)
Bases: `luma.core.emulator.emulator`

Pseudo-device that acts like a physical display, except that it renders to an displayed window. The frame rate is limited to 60FPS (much faster than a Raspberry Pi can acheive, but this can be overridden as necessary). Supports 24-bit color depth.

`pygame` is used to render the emulated display window, and it's event loop is checked to see if the ESC key was pressed or the window was dismissed: if so `sys.exit()` is called.

`capabilities`(*width*, *height*, *rotate*, *mode='1'*)

`cleanup`()

`clear`()
Initializes the device memory with an empty (blank) image.

`command`(**cmd*)
Sends a command or sequence of commands through to the delegated serial interface.

`contrast`(*level*)
Switches the display contrast to the desired level, in the range 0-255. Note that setting the level to a low (or zero) value will not necessarily dim the display to nearly off. In other words, this method is **NOT** suitable for fade-in/out animation.

Parameters **`level`** (`int`) – Desired contrast level in the range of 0-255.

`data`(*data*)
Sends a data byte or sequence of data bytes through to the delegated serial interface.

`display`(*image*)
Takes a `PIL.Image` and renders it to a pygame display surface.

`hide`()
Switches the display mode OFF, putting the device in low-power sleep mode.

`preprocess`(*image*)

show()
Sets the display mode ON, waking the device out of a prior low-power sleep mode.

to_surface(*image*)
Converts a `PIL.Image` into a `pygame.Surface`, transforming it according to the `transform` and `scale` constructor arguments.

class `luma.core.emulator.`**transformer**(*pygame*, *width*, *height*, *scale*)
Bases: `object`

Helper class used to dispatch transformation operations.

identity(*surface*)
Fast scale operation that does not sample the results

led_matrix(*surface*)
Transforms the input surface into an LED matrix (1 pixel = 1 LED)

none(*surface*)
No-op transform - used when `scale` = 1

scale2x(*surface*)
Scales using the AdvanceMAME Scale2X algorithm which does a 'jaggie-less' scale of bitmap graphics.

seven_segment(*surface*)

smoothscale(*surface*)
Smooth scaling using MMX or SSE extensions if available

luma.core.error

Exceptions for this library.

exception `luma.core.error.`**DeviceAddressError**
Bases: *luma.core.error.Error*

Exception raised when an invalid device address is detected.

args

message

exception `luma.core.error.`**DeviceDisplayModeError**
Bases: *luma.core.error.Error*

Exception raised when an invalid device display mode is detected.

args

message

exception `luma.core.error.`**DeviceNotFoundError**
Bases: *luma.core.error.Error*

Exception raised when a device cannot be found.

args

message

exception `luma.core.error.`**DevicePermissionError**
Bases: *luma.core.error.Error*

Exception raised when permission to access the device is denied.

args

message

exception `luma.core.error.`**`Error`**

Bases: `exceptions.Exception`

Base class for exceptions in this library.

args

message

`luma.core.mixin`

class `luma.core.mixin.`**`capabilities`**

Bases: `object`

`capabilities`(*width*, *height*, *rotate*, *mode='1'*)

`clear`()

Initializes the device memory with an empty (blank) image.

`display`(*image*)

`preprocess`(*image*)

`luma.core.render`

class `luma.core.render.`**`canvas`**(*device*, *dither=False*)

Bases: `object`

A canvas returns a properly-sized `PIL.ImageDraw` object onto which the caller can draw upon. As soon as the with-block completes, the resultant image is flushed onto the device.

By default, any color (other than black) will be _generally_ treated as white when displayed on monochrome devices. However, this behaviour can be changed by adding `dither=True` and the image will be converted from RGB space into a 1-bit monochrome image where dithering is employed to differentiate colors at the expense of resolution.

`luma.core.serial`

class `luma.core.serial.`**`i2c`**(*bus=None*, *port=1*, *address=60*)

Bases: `object`

Wrap an I2C interface to provide data and command methods.

Parameters

- **bus** – I2C bus instance.
- **port** (*int*) – I2C port number.
- **address** – I2C address.

Raises

- *`luma.core.error.DeviceAddressError`* – I2C device address is invalid.

- ***luma.core.error.DeviceNotFoundError*** – I2C device could not be found.
- ***luma.core.error.DevicePermissionError*** – Permission to access I2C device denied.

Note:

1. Only one of `bus` OR `port` arguments should be supplied; if both are, then `bus` takes precedence.
2. If `bus` is provided, there is an implicit expectation that it has already been opened.

cleanup()
Clean up I2C resources

command(**cmd*)
Sends a command or sequence of commands through to the I2C address - maximum allowed is 32 bytes in one go.

data(*data*)
Sends a data byte or sequence of data bytes through to the I2C address - maximum allowed in one transaction is 32 bytes, so if data is larger than this, it is sent in chunks.

class `luma.core.serial.`**noop**
Bases: `object`

Does nothing, used for pseudo-devices / emulators / anything really

noop(**args*, ***kwargs*)

class `luma.core.serial.`**spi**(*spi=None*, *gpio=None*, *port=0*, *device=0*, *bus_speed_hz=8000000*, *transfer_size=4096*, *bcm_DC=24*, *bcm_RST=25*)
Bases: `object`

Wraps an SPI interface to provide data and command methods.

Parameters

- **spi** – SPI interface (must be compatible with py-spidev)
- **gpio** – GPIO interface (must be compatible with RPi.GPIO). For slaves that dont need reset or D/C functionality, supply a `noop()` implementation instead.
- **port** (*int*) – SPI port, defaults to 0
- **device** (*int*) – SPI device, defaults to 0
- **bus_speed_hz** – SPI bus speed, defaults to 8MHz
- **transfer_size** – Max bytes to transfer in one go. Some implementations only support maxium of 64 or 128 bytes, whereas RPi/py-spidev supports 4096 (default).
- **bcm_DC** (*int*) – The BCM pin to connect data/command select (DC) to (defaults to 24).
- **bcm_RST** (*int*) – The BCM pin to connect reset (RES / RST) to (defaults to 24).

Raises ***luma.core.error.DeviceNotFoundError*** – SPI device could not be found.

cleanup()
Clean up SPI & GPIO resources

command(**cmd*)
Sends a command or sequence of commands through to the SPI device.

data(*data*)
Sends a data byte or sequence of data bytes through to the SPI device. If the data is more than 4Kb in size, it is sent in chunks.

luma.core.threadpool

class luma.core.threadpool.**threadpool**(*num_threads*)
Pool of threads consuming tasks from a queue

add_task(*func*, **args*, ***kargs*)
Add a task to the queue

wait_completion()
Wait for completion of all the tasks in the queue

class luma.core.threadpool.**worker**(*tasks*)
Bases: threading.Thread

Thread executing tasks from a given tasks queue

daemon
A boolean value indicating whether this thread is a daemon thread (True) or not (False).

This must be set before start() is called, otherwise RuntimeError is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to daemon = False.

The entire Python program exits when no alive non-daemon threads are left.

getName()

ident
Thread identifier of this thread or None if it has not been started.

This is a nonzero integer. See the thread.get_ident() function. Thread identifiers may be recycled when a thread exits and another thread is created. The identifier is available even after the thread has exited.

isAlive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

isDaemon()

is_alive()
Return whether the thread is alive.

This method returns True just before the run() method starts until just after the run() method terminates. The module function enumerate() returns a list of all alive threads.

join(*timeout=None*)
Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call isAlive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

`name`
A string used for identification purposes only.

It has no semantics. Multiple threads may be given the same name. The initial name is set by the constructor.

`run``()`

`setDaemon``(`*daemonic*`)`

`setName``(`*name*`)`

`start``()`
Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

`luma.core.virtual`

`luma.core.virtual.`**`calc_bounds`**`(`*xy*, *entity*`)`
For an entity with width and height attributes, determine the bounding box if were positioned at (x, y).

class `luma.core.virtual.`**`history`**`(`*device*`)`
Bases: *`luma.core.mixin.capabilities`*

Wraps a device (or emulator) to provide a facility to be able to make a savepoint (a point at which the screen display can be "rolled-back" to).

This is mostly useful for displaying transient error/dialog messages which could be subsequently dismissed, reverting back to the previous display.

`capabilities``(`*width*, *height*, *rotate*, *mode='1'*`)`

`clear``()`
Initializes the device memory with an empty (blank) image.

`display``(`*image*`)`

`preprocess``(`*image*`)`

`restore``(`*drop=0*`)`
Restores the last savepoint. If `drop` is supplied and greater than zero, then that many savepoints are dropped, and the next savepoint is restored.

`savepoint``()`
Copies the last displayed image.

class `luma.core.virtual.`**`hotspot`**`(`*width*, *height*, *draw_fn=None*`)`
Bases: *`luma.core.mixin.capabilities`*

A hotspot (*a place of more than usual interest, activity, or popularity*) is a live display which may be added to a virtual viewport - if the hotspot and the viewport are overlapping, then the *`update()`* method will be automatically invoked when the viewport is being refreshed or its position moved (such that an overlap occurs).

You would either:

- create a `hotspot` instance, suppling a render function (taking an `PIL.ImageDraw` object, `width` & `height` dimensions. The render function should draw within a bounding box of (0, 0, width, height), and render a full frame.
- sub-class `hotspot` and override the :func:`should_redraw` and *`update()`* methods. This might be more useful for slow-changing values where it is not necessary to update every refresh cycle, or your implementation is stateful.

`capabilities`(*width*, *height*, *rotate*, *mode='1'*)

`clear`()
Initializes the device memory with an empty (blank) image.

`display`(*image*)

`paste_into`(*image*, *xy*)

`preprocess`(*image*)

`should_redraw`()
Override this method to return true or false on some condition (possibly on last updated member variable) so that for slow changing hotspots they are not updated too frequently.

`update`(*draw*)

`luma.core.virtual.`**`range_overlap`**(*a_min*, *a_max*, *b_min*, *b_max*)
Neither range is completely greater than the other

class `luma.core.virtual.`**`snapshot`**(*width*, *height*, *draw_fn=None*, *interval=1.0*)
Bases: *`luma.core.virtual.hotspot`*

A snapshot is a *type of* hotspot, but only updates once in a given interval, usually much less frequently than the viewport requests refresh updates.

`capabilities`(*width*, *height*, *rotate*, *mode='1'*)

`clear`()
Initializes the device memory with an empty (blank) image.

`display`(*image*)

`paste_into`(*image*, *xy*)

`preprocess`(*image*)

`should_redraw`()
Only requests a redraw after `interval` seconds have elapsed

`update`(*draw*)

class `luma.core.virtual.`**`terminal`**(*device*, *font=None*, *color='white'*, *bgcolor='black'*, *tabstop=4*, *line_height=None*, *animate=True*)
Bases: `object`

Provides a terminal-like interface to a device (or a device-like object that has `mixin.capabilities` characteristics).

`backspace`()
Moves the cursor one place to the left, erasing the character at the current position. Cannot move beyound column zero, nor onto the previous line

`carriage_return`()
Returns the cursor position to the left-hand side without advancing downwards.

clear()
Clears the display and resets the cursor position to (0, 0).

erase()
Erase the contents of the cursor's current postion without moving the cursor's position.

flush()
Cause the current backing store to be rendered on the nominated device.

newline()
Advances the cursor position ot the left hand side, and to the next line. If the cursor is on the lowest line, the displayed contents are scrolled, causing the top line to be lost.

println(*text=''*)
Prints the supplied text to the device, scrolling where necessary. The text is always followed by a newline.

putch(*ch*, *flush=True*)
Prints the specific character, which must be a valid printable ASCII value in the range 32..127 only.

puts(*text*)
Prints the supplied text, handling special character codes for carriage return (r), newline (n), backspace (b) and tab (t).

If the `animate` flag was set to True (default), then each character is flushed to the device, giving the effect of 1970's teletype device.

tab()
Advances the cursor position to the next (soft) tabstop.

class luma.core.virtual.**viewport**(*device*, *width*, *height*)
Bases: *luma.core.mixin.capabilities*

add_hotspot(*hotspot*, *xy*)
Add the hotspot at (x, y). The hotspot must fit inside the bounds of the virtual device. If it does not then an AssertError is raised.

capabilities(*width*, *height*, *rotate*, *mode='1'*)

clear()
Initializes the device memory with an empty (blank) image.

display(*image*)

is_overlapping_viewport(*hotspot*, *xy*)
Checks to see if the hotspot at position (x, y) is (at least partially) visible according to the position of the viewport

preprocess(*image*)

refresh()

remove_hotspot(*hotspot*, *xy*)
Remove the hotspot at (x, y): Any previously rendered image where the hotspot was placed is erased from the backing image, and will be "undrawn" the next time the virtual device is refreshed. If the specified hotspot is not found (x, y), a ValueError is raised.

set_position(*xy*)

`luma.oled.device`

class `luma.oled.device.`**`sh1106`**(*serial_interface=None*, *width=128*, *height=64*, *rotate=0*, ***kwargs*)

Bases: *`luma.core.device.device`*

Encapsulates the serial interface to the monochrome SH1106 OLED display hardware. On creation, an initialization sequence is pumped to the display to properly configure it. Further control commands can then be called to affect the brightness and other settings.

`capabilities`(*width*, *height*, *rotate*, *mode='1'*)

`cleanup`()

Attempt to switch the device off or put into low power mode (this helps prolong the life of the device), clear the screen and close resources associated with the underlying serial interface.

This is a managed function, which is called when the python processs is being shutdown, so shouldn't usually need be called directly in application code.

`clear`()

Initializes the device memory with an empty (blank) image.

`command`(**cmd*)

Sends a command or sequence of commands through to the delegated serial interface.

`contrast`(*level*)

Switches the display contrast to the desired level, in the range 0-255. Note that setting the level to a low (or zero) value will not necessarily dim the display to nearly off. In other words, this method is **NOT** suitable for fade-in/out animation.

Parameters **`level`** (*`int`*) – Desired contrast level in the range of 0-255.

`data`(*data*)

Sends a data byte or sequence of data bytes through to the delegated serial interface.

`display`(*image*)

Takes a 1-bit `PIL.Image` and dumps it to the SH1106 OLED display.

`hide`()

Switches the display mode OFF, putting the device in low-power sleep mode.

`preprocess`(*image*)

`show`()

Sets the display mode ON, waking the device out of a prior low-power sleep mode.

class `luma.oled.device.`**`ssd1306`**(*serial_interface=None*, *width=128*, *height=64*, *rotate=0*, ***kwargs*)

Bases: *`luma.core.device.device`*

Encapsulates the serial interface to the monochrome SSD1306 OLED display hardware. On creation, an initialization sequence is pumped to the display to properly configure it. Further control commands can then be called to affect the brightness and other settings.

`capabilities`(*width*, *height*, *rotate*, *mode='1'*)

`cleanup`()

Attempt to switch the device off or put into low power mode (this helps prolong the life of the device), clear the screen and close resources associated with the underlying serial interface.

This is a managed function, which is called when the python processs is being shutdown, so shouldn't usually need be called directly in application code.

clear()
Initializes the device memory with an empty (blank) image.

command(**cmd*)
Sends a command or sequence of commands through to the delegated serial interface.

contrast(*level*)
Switches the display contrast to the desired level, in the range 0-255. Note that setting the level to a low (or zero) value will not necessarily dim the display to nearly off. In other words, this method is **NOT** suitable for fade-in/out animation.

Parameters **level** (*int*) – Desired contrast level in the range of 0-255.

data(*data*)
Sends a data byte or sequence of data bytes through to the delegated serial interface.

display(*image*)
Takes a 1-bit `PIL.Image` and dumps it to the SSD1306 OLED display.

hide()
Switches the display mode OFF, putting the device in low-power sleep mode.

preprocess(*image*)

show()
Sets the display mode ON, waking the device out of a prior low-power sleep mode.

class luma.oled.device.**ssd1322**(*serial_interface=None*, *width=256*, *height=64*, *rotate=0*, *mode='RGB'*, ***kwargs*)
Bases: *luma.core.device.device*

Encapsulates the serial interface to the 4-bit greyscale SSD1322 OLED display hardware. On creation, an initialization sequence is pumped to the display to properly configure it. Further control commands can then be called to affect the brightness and other settings.

capabilities(*width*, *height*, *rotate*, *mode='1'*)

cleanup()
Attempt to switch the device off or put into low power mode (this helps prolong the life of the device), clear the screen and close resources associated with the underlying serial interface.

This is a managed function, which is called when the python processs is being shutdown, so shouldn't usually need be called directly in application code.

clear()
Initializes the device memory with an empty (blank) image.

command(*cmd*, **args*)
Sends a command and an (optional) sequence of arguments through to the delegated serial interface. Note that the arguments are passed through as data.

contrast(*level*)
Switches the display contrast to the desired level, in the range 0-255. Note that setting the level to a low (or zero) value will not necessarily dim the display to nearly off. In other words, this method is **NOT** suitable for fade-in/out animation.

Parameters **level** (*int*) – Desired contrast level in the range of 0-255.

data(*data*)
Sends a data byte or sequence of data bytes through to the delegated serial interface.

`display`(*image*)
Takes a 1-bit monochrome or 24-bit RGB `PIL.Image` and dumps it to the SSD1322 OLED display, converting the image pixels to 4-bit greyscale using a simplified Luma calculation, based on *Y'=0.299R'+0.587G'+0.114B'*.

`hide`()
Switches the display mode OFF, putting the device in low-power sleep mode.

`preprocess`(*image*)

`show`()
Sets the display mode ON, waking the device out of a prior low-power sleep mode.

class `luma.oled.device.`**`ssd1325`**(*serial_interface=None*, *width=128*, *height=64*, *rotate=0*, *mode='RGB'*, ***kwargs*)
Bases: *`luma.core.device.device`*

Encapsulates the serial interface to the 4-bit greyscale SSD1325 OLED display hardware. On creation, an initialization sequence is pumped to the display to properly configure it. Further control commands can then be called to affect the brightness and other settings.

`capabilities`(*width*, *height*, *rotate*, *mode='1'*)

`cleanup`()
Attempt to switch the device off or put into low power mode (this helps prolong the life of the device), clear the screen and close resources associated with the underlying serial interface.

This is a managed function, which is called when the python processs is being shutdown, so shouldn't usually need be called directly in application code.

`clear`()
Initializes the device memory with an empty (blank) image.

`command`(**cmd*)
Sends a command or sequence of commands through to the delegated serial interface.

`contrast`(*level*)
Switches the display contrast to the desired level, in the range 0-255. Note that setting the level to a low (or zero) value will not necessarily dim the display to nearly off. In other words, this method is **NOT** suitable for fade-in/out animation.

Parameters **`level`** (*`int`*) – Desired contrast level in the range of 0-255.

`data`(*data*)
Sends a data byte or sequence of data bytes through to the delegated serial interface.

`display`(*image*)
Takes a 1-bit monochrome or 24-bit RGB `PIL.Image` and dumps it to the SSD1325 OLED display, converting the image pixels to 4-bit greyscale using a simplified Luma calculation, based on *Y'=0.299R'+0.587G'+0.114B'*.

`hide`()
Switches the display mode OFF, putting the device in low-power sleep mode.

`preprocess`(*image*)

`show`()
Sets the display mode ON, waking the device out of a prior low-power sleep mode.

class `luma.oled.device.`**`ssd1331`**(*serial_interface=None*, *width=96*, *height=64*, *rotate=0*, ***kwargs*)
Bases: *`luma.core.device.device`*

Encapsulates the serial interface to the 16-bit color (5-6-5 RGB) SSD1331 OLED display hardware. On creation, an initialization sequence is pumped to the display to properly configure it. Further control commands can then be called to affect the brightness and other settings.

`capabilities`(*width*, *height*, *rotate*, *mode='1'*)

`cleanup`()

Attempt to switch the device off or put into low power mode (this helps prolong the life of the device), clear the screen and close resources associated with the underlying serial interface.

This is a managed function, which is called when the python processs is being shutdown, so shouldn't usually need be called directly in application code.

`clear`()

Initializes the device memory with an empty (blank) image.

`command`(**cmd*)

Sends a command or sequence of commands through to the delegated serial interface.

`contrast`(*level*)

Switches the display contrast to the desired level, in the range 0-255. Note that setting the level to a low (or zero) value will not necessarily dim the display to nearly off. In other words, this method is **NOT** suitable for fade-in/out animation.

Parameters **`level`** (`int`) – Desired contrast level in the range of 0-255.

`data`(*data*)

Sends a data byte or sequence of data bytes through to the delegated serial interface.

`display`(*image*)

Takes a 24-bit RGB `PIL.Image` and dumps it to the SSD1331 OLED display.

`hide`()

Switches the display mode OFF, putting the device in low-power sleep mode.

`preprocess`(*image*)

`show`()

Sets the display mode ON, waking the device out of a prior low-power sleep mode.

CHAPTER 6

References

- https://learn.adafruit.com/monochrome-oled-breakouts
- https://github.com/adafruit/Adafruit_Python_SSD1306
- http://www.dafont.com/bitmap.php
- http://raspberrypi.znix.com/hipidocs/topic_i2cbus_2.htm
- http://martin-jones.com/2013/08/20/how-to-get-the-second-raspberry-pi-i2c-bus-to-work/
- https://projects.drogon.net/understanding-spi-on-the-raspberry-pi/
- https://pinout.xyz/
- https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi
- http://code.activestate.com/recipes/577187-python-thread-pool/

CHAPTER 7

Contributing

Pull requests (code changes / documentation / typos / feature requests / setup) are gladly accepted. If you are intending to introduce some large-scale changes, please get in touch first to make sure we're on the same page: try to include a docstring for any new method or class, and keep method bodies small, readable and PEP8-compliant. Add tests and strive to keep the code coverage levels high.

GitHub

The source code is available to clone at: https://github.com/rm-hull/luma.oled

Contributors

- Thijs Triemstra (@thijstriemstra)
- Christoph Handel (@fragfutter)
- Boeeerb (@Boeeerb)
- xes (@xes)
- Roger Dahl (@rogerdahl)
- Václav Šmilauer (@eudoxos)
- Claus Bjerre (@bjerrep)

CHAPTER 8

ChangeLog

Version	Description	Date
Upcoming	*TBC*	
2.2.4	• Tweaked SSD1325 init settings & replaced constants • Update dependencies	2017/02/17
2.2.3	• Monochrome rendering on SSD1322 & SSD1325	2017/02/14
2.2.2	• SSD1325 performance improvements (perfloop: 25.50 –> 34.31 FPS) • SSD1331 performance improvements (perfloop: 34.64 –> 51.89 FPS)	2017/02/02
2.2.1	• Support for 256x64 4-bit greyscale OLED (SSD1322) • Improved API documentation (shows inherited members)	2017/01/29
2.1.0	• Simplify/optimize SSD1306 display logic	2017/01/22
2.0.1	• Moved examples to separate git repo • Add notes about breaking changes	2017/01/15
2.0.0	• Package rename to `luma.oled` (**Note:** Breaking changes)	2017/01/11
1.5.0	• Performance improvements	2017/01/09

CHAPTER 9

The MIT License (MIT)

Python Module Index

Index