
python-elgato-streamdeck

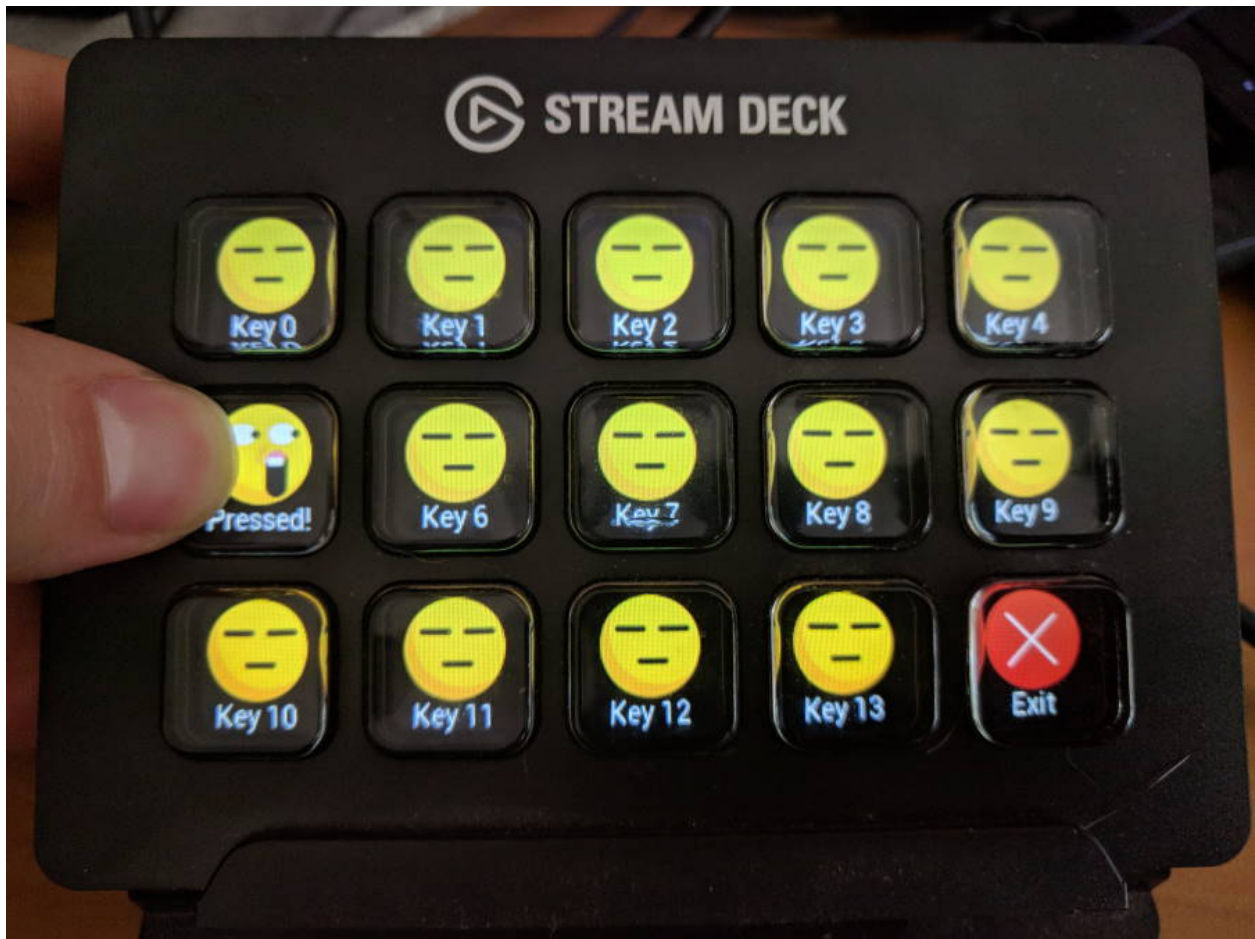
Release 0.9.5

Dean Camera

Dec 23, 2023

INSTALLATION AND SETUP

1	Index	3
1.1	Library Installation	3
1.2	Modules: Device Discovery	4
1.3	Modules: StreamDeck Devices	5
1.4	Modules: Communication Transports	17
1.5	Modules: Image Helpers	22
1.6	Example Script: Device Information	22
1.7	Example Script: Basic Usage	23
1.8	Example Script: Pedal	26
1.9	Example Script: StreamDeck +	27
1.10	Example Script: Tiled Image	30
1.11	Example Script: Animated Images	33
1.12	Source Code	36
1.13	Changelog	37
1.14	License	39
2	Indices and tables	41
	Python Module Index	43
	Index	45



This is an open source Python 3 library to control an [Elgato Stream Deck](#) directly, without the official software. This can allow you to create your own custom front-ends, such as a custom control front-end for home automation software.

1.1 Library Installation

To install this library via the *pip* package manager, simply run `pip install streamdeck` from a terminal.

The included examples require the PIL fork *pillow*, although it can be swapped out if desired by the user application for any other image manipulation library. This can also be installed with *pip* via `pip install pillow`.

1.1.1 HID Backend Setup

The library core is structured so that it can use one of (potentially) several alternative HID backend libraries for the actual low level device communications. You will need to install the dependencies appropriate to your chosen backend for the library to work correctly.

Backends:

Default LibUSB HIDAPI Backend

This is the default and recommended backend - a cross platform library for communicating with HID devices. Most systems will have this as a system package available for easy installation.

Windows

Windows systems requires additional manual installation of a DLL in order to function. The latest source for the `hidapi.dll` DLL is the [releases page of the libUSB GitHub project](#).

Place the DLL into a folder that has been added to your system `%PATH%` directory list (typically this includes the `C:\Windows\System32` folder but adding a new path would be recommended instead of modifying your Windows directory).

Ensure you use the correct DLL version for your installed Python version; i.e. if you are using 32-bit Python, install the 32-bit `hidapi.dll`.

MacOS (Darwin)

On MacOS systems, you can choose to either compile the [HIDAPI project](#) yourself, or install it via one of the multiple third party package managers (e.g. `brew install hidapi`, when using Homebrew).

Linux (Ubuntu/Debian)

On Linux, the `libhidapi-libusb0` package is required can can be installed via the system's package manager.

The following script has been verified working on a Raspberry Pi (Models 2B and 4B) running a stock Debian Buster image, to install all the required dependencies needed by this project on a fresh system:

```
# Ensure system is up to date, upgrade all out of date packages
sudo apt update && sudo apt dist-upgrade -y

# Install the pip Python package manager
sudo apt install -y python3-pip python3-setuptools

# Install system packages needed for the default LibUSB HIDAPI backend
sudo apt install -y libudev-dev libusb-1.0-0-dev libhidapi-libusb0

# Install system packages needed for the Python Pillow package installation
sudo apt install -y libjpeg-dev zlib1g-dev libopenjp2-7 libtiff5

# Install python library dependencies
pip3 install wheel
pip3 install pillow

# Add udev rule to allow all users non-root access to Elgato StreamDeck devices:
sudo tee /etc/udev/rules.d/10-streamdeck.rules << EOF
    SUBSYSTEMS=="usb", ATTRS{idVendor}=="0fd9", GROUP="users", TAG+="uaccess"
EOF

# Reload udev rules to ensure the new permissions take effect
sudo udevadm control --reload-rules

# Install the latest version of the StreamDeck library via pip
pip3 install streamdeck
```

Note that after adding the udev rules, you will need to remove and re-attach any existing StreamDeck devices to ensure they adopt the new permissions. This should allow you to access StreamDeck devices *without* needing root permissions.

1.2 Modules: Device Discovery

1.2.1 Device Manager

```
class StreamDeck.DeviceManager.DeviceManager(transport=None)
```

Bases: object

Central device manager, to enumerate any attached StreamDeck devices. An instance of this class must be created in order to detect and use any StreamDeck devices.

Creates a new StreamDeck DeviceManager, used to detect attached StreamDeck devices.

Parameters

transport (*str*) – name of the the specific HID transport back-end to use, None to auto-probe.

enumerate()

Detect attached StreamDeck devices.

Return type

list(*StreamDeck*)

Returns

list of StreamDeck instances, one for each detected device.

exception StreamDeck.DeviceManager.ProbeError

Bases: Exception

Exception thrown when attempting to probe for attached StreamDeck devices, but no suitable valid transport was found.

1.3 Modules: StreamDeck Devices

1.3.1 StreamDeck (Abstract Base)

class StreamDeck.Devices.StreamDeck.DialEventType(*value*)

Bases: Enum

Type of event that has occurred for a Dial.

class StreamDeck.Devices.StreamDeck.StreamDeck(*device*)

Bases: ABC

Represents a physically attached StreamDeck device.

close()

Closes the device for input/output.

See also:

See [open\(\)](#) for the corresponding open method.

connected()

Indicates if the physical StreamDeck device this instance is attached to is still connected to the host.

Return type

bool

Returns

True if the deck is still connected, *False* otherwise.

deck_type()

Retrieves the model of Stream Deck.

Return type

str

Returns

String containing the model name of the StreamDeck device.

dial_count()

Retrieves number of physical dials on the attached StreamDeck device.

Return type

int

Returns

Number of physical dials

dial_states()

Retrieves the current states of the dials (pressed or not) on the Stream Deck

Return type

list(bool)

Returns

List describing the current states of each of the dials on the device (*True* if the dial is being pressed, *False* otherwise).

abstract get_firmware_version()

Gets the firmware version of the attached StreamDeck.

Return type

str

Returns

String containing the firmware version of the attached device.

abstract get_serial_number()

Gets the serial number of the attached StreamDeck.

Return type

str

Returns

String containing the serial number of the attached device.

id()

Retrieves the physical ID of the attached StreamDeck. This can be used to differentiate one StreamDeck from another.

Return type

str

Returns

Identifier for the attached device.

is_open()

Indicates if the StreamDeck device is currently open and ready for use.

Return type

bool

Returns

True if the deck is open, *False* otherwise.

is_touch()

Returns whether the Stream Deck can receive touch events

Return type

bool

Returns

True if the deck can receive touch events, *False* otherwise

is_visual()

Returns whether the Stream Deck has a visual display output.

Return type

bool

Returns

True if the deck has a screen, *False* otherwise.

key_count()

Retrieves number of physical buttons on the attached StreamDeck device.

Return type

int

Returns

Number of physical buttons.

key_image_format()

Retrieves the image format accepted by the attached StreamDeck device. Images should be given in this format when setting an image on a button.

See also:

See [set_key_image\(\)](#) method to update the image displayed on a StreamDeck button.

Return type

dict()

Returns

Dictionary describing the various image parameters (size, image format, image mirroring and rotation).

key_layout()

Retrieves the physical button layout on the attached StreamDeck device.

Return type

(int, int)

Return (rows, columns)

Number of button rows and columns.

key_states()

Retrieves the current states of the buttons on the StreamDeck.

Return type

list(bool)

Returns

List describing the current states of each of the buttons on the device (*True* if the button is being pressed, *False* otherwise).

open()

Opens the device for input/output. This must be called prior to setting or retrieving any device state.

See also:

See [close\(\)](#) for the corresponding close method.

product_id()

Retrieves the product ID attached StreamDeck. This can be used to determine the exact type of attached StreamDeck.

Return type

int

Returns

Product ID of the attached device.

abstract reset()

Resets the StreamDeck, clearing all button images and showing the standby image.

abstract set_brightness(*percent*)

Sets the global screen brightness of the StreamDeck, across all the physical buttons.

Parameters

percent (*int/float*) – brightness percent, from [0-100] as an *int*, or normalized to [0.0-1.0] as a *float*.

set_dial_callback(*callback*)

Sets the callback function called each time there is an interaction with a dial on the StreamDeck.

Note: This callback will be fired from an internal reader thread. Ensure that the given callback function is thread-safe.

Note: Only one callback can be registered at one time.

See also:

See [`set_dial_callback_async\(\)`](#) method for a version compatible with Python 3 *asyncio* asynchronous functions.

Parameters

callback (*function*) – Callback function to fire each time a button state changes.

set_dial_callback_async(*async_callback*, *loop=None*)

Sets the asynchronous callback function called each time there is an interaction with a dial on the StreamDeck. The given callback should be compatible with Python 3's *asyncio* routines.

Note: The asynchronous callback will be fired in a thread-safe manner.

Note: This will override the callback (if any) set by [`set_dial_callback\(\)`](#).

Parameters

- **async_callback** (*function*) – Asynchronous callback function to fire each time a button state changes.
- **loop** (*asyncio.loop*) – Asyncio loop to dispatch the callback into

set_key_callback(*callback*)

Sets the callback function called each time a button on the StreamDeck changes state (either pressed, or released).

Note: This callback will be fired from an internal reader thread. Ensure that the given callback function is thread-safe.

Note: Only one callback can be registered at one time.

See also:

See [set_key_callback_async\(\)](#) method for a version compatible with Python 3 *asyncio* asynchronous functions.

Parameters

callback (*function*) – Callback function to fire each time a button state changes.

set_key_callback_async(*async_callback*, *loop=None*)

Sets the asynchronous callback function called each time a button on the StreamDeck changes state (either pressed, or released). The given callback should be compatible with Python 3's *asyncio* routines.

Note: The asynchronous callback will be fired in a thread-safe manner.

Note: This will override the callback (if any) set by [set_key_callback\(\)](#).

Parameters

- **async_callback** (*function*) – Asynchronous callback function to fire each time a button state changes.
- **loop** (*asyncio.loop*) – Asyncio loop to dispatch the callback into

abstract set_key_image(*key*, *image*)

Sets the image of a button on the StreamDeck to the given image. The image being set should be in the correct format for the device, as an enumerable collection of bytes.

See also:

See [key_image_format\(\)](#) method for information on the image format accepted by the device.

Parameters

- **key** (*int*) – Index of the button whose image is to be updated.
- **image** (*enumerable*) – Raw data of the image to set on the button. If *None*, the key will be cleared to a black color.

set_poll_frequency(*hz*)

Sets the frequency of the button polling reader thread, determining how often the StreamDeck will be polled for button changes.

A higher frequency will result in a higher CPU usage, but a lower latency between a physical button press and a event from the library.

Parameters

hz (*int*) – Reader thread frequency, in Hz (1-1000).

set_touchscreen_callback(*callback*)

Sets the callback function called each time there is an interaction with a touchscreen on the StreamDeck.

Note: This callback will be fired from an internal reader thread. Ensure that the given callback function is thread-safe.

Note: Only one callback can be registered at one time.

See also:

See [set_touchscreen_callback_async\(\)](#) method for a version compatible with Python 3 *asyncio* asynchronous functions.

Parameters

callback (*function*) – Callback function to fire each time a button state changes.

set_touchscreen_callback_async(*async_callback*, *loop=None*)

Sets the asynchronous callback function called each time there is an interaction with the touchscreen on the StreamDeck. The given callback should be compatible with Python 3's *asyncio* routines.

Note: The asynchronous callback will be fired in a thread-safe manner.

Note: This will override the callback (if any) set by [set_touchscreen_callback\(\)](#).

Parameters

- **async_callback** (*function*) – Asynchronous callback function to fire each time a button state changes.
- **loop** (*asyncio.loop*) – Asyncio loop to dispatch the callback into

abstract set_touchscreen_image(*image*, *x_pos=0*, *y_pos=0*, *width=0*, *height=0*)

Draws an image on the touchscreen in a certain position. The image should be in the correct format for the devices, as an enumerable collection of bytes.

See also:

See [touchscreen_image_format\(\)](#) method for information on the image format accepted by the device.

Parameters

- **image** (*enumerable*) – Raw data of the image to set on the button. If *None*, the touchscreen will be cleared.
- **x_pos** (*int*) – Position on x axis of the image to draw
- **y_pos** (*int*) – Position on y axis of the image to draw

- **width** (*int*) – width of the image
- **height** (*int*) – height of the image

touchscreen_image_format()

Retrieves the image format accepted by the touchscreen of the Stream Deck. Images should be given in this format when drawing on touchscreen.

See also:

See [set_touchscreen_image\(\)](#) method to draw an image on the StreamDeck touchscreen.

Return type

dict()

Returns

Dictionary describing the various image parameters (size, image format).

vendor_id()

Retrieves the vendor ID attached StreamDeck. This can be used to determine the exact type of attached StreamDeck.

Return type

int

Returns

Vendor ID of the attached device.

class StreamDeck.Devices.StreamDeck.TouchscreenEventType(*value*)

Bases: Enum

Type of event that has occurred for a Touchscreen.

1.3.2 StreamDeck Original

class StreamDeck.Devices.StreamDeckOriginal.StreamDeckOriginal(*device*)

Bases: [StreamDeck](#)

Represents a physically attached StreamDeck Original device.

get_firmware_version()

Gets the firmware version of the attached StreamDeck.

Return type

str

Returns

String containing the firmware version of the attached device.

get_serial_number()

Gets the serial number of the attached StreamDeck.

Return type

str

Returns

String containing the serial number of the attached device.

reset()

Resets the StreamDeck, clearing all button images and showing the standby image.

set_brightness(*percent*)

Sets the global screen brightness of the StreamDeck, across all the physical buttons.

Parameters

percent (*int/float*) – brightness percent, from [0-100] as an *int*, or normalized to [0.0-1.0] as a *float*.

set_key_image(*key*, *image*)

Sets the image of a button on the StreamDeck to the given image. The image being set should be in the correct format for the device, as an enumerable collection of bytes.

See also:

See `key_image_format()` method for information on the image format accepted by the device.

Parameters

- **key** (*int*) – Index of the button whose image is to be updated.
- **image** (*enumerable*) – Raw data of the image to set on the button. If *None*, the key will be cleared to a black color.

set_touchscreen_image(*image*, *x_pos*=0, *y_pos*=0, *width*=0, *height*=0)

Draws an image on the touchscreen in a certain position. The image should be in the correct format for the devices, as an enumerable collection of bytes.

See also:

See `touchscreen_image_format()` method for information on the image format accepted by the device.

Parameters

- **image** (*enumerable*) – Raw data of the image to set on the button. If *None*, the touchscreen will be cleared.
- **x_pos** (*int*) – Position on x axis of the image to draw
- **y_pos** (*int*) – Position on y axis of the image to draw
- **width** (*int*) – width of the image
- **height** (*int*) – height of the image

1.3.3 StreamDeck Mini

class `StreamDeck.Devices.StreamDeckMini.StreamDeckMini` (*device*)

Bases: `StreamDeck`

Represents a physically attached StreamDeck Mini device.

get_firmware_version()

Gets the firmware version of the attached StreamDeck.

Return type

`str`

Returns

String containing the firmware version of the attached device.

get_serial_number()

Gets the serial number of the attached StreamDeck.

Return type

str

Returns

String containing the serial number of the attached device.

reset()

Resets the StreamDeck, clearing all button images and showing the standby image.

set_brightness(*percent*)

Sets the global screen brightness of the StreamDeck, across all the physical buttons.

Parameters

percent (*int/float*) – brightness percent, from [0-100] as an *int*, or normalized to [0.0-1.0] as a *float*.

set_key_image(*key, image*)

Sets the image of a button on the StreamDeck to the given image. The image being set should be in the correct format for the device, as an enumerable collection of bytes.

See also:

See `key_image_format()` method for information on the image format accepted by the device.

Parameters

- **key** (*int*) – Index of the button whose image is to be updated.
- **image** (*enumerable*) – Raw data of the image to set on the button. If *None*, the key will be cleared to a black color.

set_touchscreen_image(*image, x_pos=0, y_pos=0, width=0, height=0*)

Draws an image on the touchscreen in a certain position. The image should be in the correct format for the devices, as an enumerable collection of bytes.

See also:

See `touchscreen_image_format()` method for information on the image format accepted by the device.

Parameters

- **image** (*enumerable*) – Raw data of the image to set on the button. If *None*, the touchscreen will be cleared.
- **x_pos** (*int*) – Position on x axis of the image to draw
- **y_pos** (*int*) – Position on y axis of the image to draw
- **width** (*int*) – width of the image
- **height** (*int*) – height of the image

1.3.4 StreamDeck XL

class StreamDeck.Devices.StreamDeckXL.**StreamDeckXL**(*device*)

Bases: *StreamDeck*

Represents a physically attached StreamDeck XL device.

get_firmware_version()

Gets the firmware version of the attached StreamDeck.

Return type

str

Returns

String containing the firmware version of the attached device.

get_serial_number()

Gets the serial number of the attached StreamDeck.

Return type

str

Returns

String containing the serial number of the attached device.

reset()

Resets the StreamDeck, clearing all button images and showing the standby image.

set_brightness(*percent*)

Sets the global screen brightness of the StreamDeck, across all the physical buttons.

Parameters

percent (*int/float*) – brightness percent, from [0-100] as an *int*, or normalized to [0.0-1.0] as a *float*.

set_key_image(*key, image*)

Sets the image of a button on the StreamDeck to the given image. The image being set should be in the correct format for the device, as an enumerable collection of bytes.

See also:

See `key_image_format()` method for information on the image format accepted by the device.

Parameters

- **key** (*int*) – Index of the button whose image is to be updated.
- **image** (*enumerable*) – Raw data of the image to set on the button. If *None*, the key will be cleared to a black color.

set_touchscreen_image(*image, x_pos=0, y_pos=0, width=0, height=0*)

Draws an image on the touchscreen in a certain position. The image should be in the correct format for the devices, as an enumerable collection of bytes.

See also:

See `touchscreen_image_format()` method for information on the image format accepted by the device.

Parameters

- **image** (*enumerable*) – Raw data of the image to set on the button. If *None*, the touchscreen will be cleared.
- **x_pos** (*int*) – Position on x axis of the image to draw
- **y_pos** (*int*) – Position on y axis of the image to draw
- **width** (*int*) – width of the image
- **height** (*int*) – height of the image

1.3.5 StreamDeck Pedal

class StreamDeck.Devices.StreamDeckPedal.**StreamDeckPedal**(*device*)

Bases: *StreamDeck*

Represents a physically attached StreamDeck Pedal device.

get_firmware_version()

Gets the firmware version of the attached StreamDeck.

Return type

str

Returns

String containing the firmware version of the attached device.

get_serial_number()

Gets the serial number of the attached StreamDeck.

Return type

str

Returns

String containing the serial number of the attached device.

reset()

Resets the StreamDeck, clearing all button images and showing the standby image.

set_brightness(*percent*)

Sets the global screen brightness of the StreamDeck, across all the physical buttons.

Parameters

percent (*int/float*) – brightness percent, from [0-100] as an *int*, or normalized to [0.0-1.0] as a *float*.

set_key_image(*key*, *image*)

Sets the image of a button on the StreamDeck to the given image. The image being set should be in the correct format for the device, as an enumerable collection of bytes.

See also:

See `key_image_format()` method for information on the image format accepted by the device.

Parameters

- **key** (*int*) – Index of the button whose image is to be updated.
- **image** (*enumerable*) – Raw data of the image to set on the button. If *None*, the key will be cleared to a black color.

set_touchscreen_image(*image*, *x_pos*=0, *y_pos*=0, *width*=0, *height*=0)

Draws an image on the touchscreen in a certain position. The image should be in the correct format for the devices, as an enumerable collection of bytes.

See also:

See `touchscreen_image_format()` method for information on the image format accepted by the device.

Parameters

- **image** (*enumerable*) – Raw data of the image to set on the button. If *None*, the touchscreen will be cleared.
- **x_pos** (*int*) – Position on x axis of the image to draw
- **y_pos** (*int*) – Position on y axis of the image to draw
- **width** (*int*) – width of the image
- **height** (*int*) – height of the image

1.3.6 StreamDeck Plus

class StreamDeck.Devices.StreamDeckPlus.**StreamDeckPlus**(*device*)

Bases: [StreamDeck](#)

get_firmware_version()

Gets the firmware version of the attached StreamDeck.

Return type

str

Returns

String containing the firmware version of the attached device.

get_serial_number()

Gets the serial number of the attached StreamDeck.

Return type

str

Returns

String containing the serial number of the attached device.

reset()

Resets the StreamDeck, clearing all button images and showing the standby image.

set_brightness(*percent*)

Sets the global screen brightness of the StreamDeck, across all the physical buttons.

Parameters

percent (*int/float*) – brightness percent, from [0-100] as an *int*, or normalized to [0.0-1.0] as a *float*.

set_key_image(*key*, *image*)

Sets the image of a button on the StreamDeck to the given image. The image being set should be in the correct format for the device, as an enumerable collection of bytes.

See also:

See `key_image_format()` method for information on the image format accepted by the device.

Parameters

- **key** (*int*) – Index of the button whose image is to be updated.
- **image** (*enumerable*) – Raw data of the image to set on the button. If *None*, the key will be cleared to a black color.

set_touchscreen_image(*image*, *x_pos=0*, *y_pos=0*, *width=0*, *height=0*)

Draws an image on the touchscreen in a certain position. The image should be in the correct format for the devices, as an enumerable collection of bytes.

See also:

See `touchscreen_image_format()` method for information on the image format accepted by the device.

Parameters

- **image** (*enumerable*) – Raw data of the image to set on the button. If *None*, the touchscreen will be cleared.
- **x_pos** (*int*) – Position on x axis of the image to draw
- **y_pos** (*int*) – Position on y axis of the image to draw
- **width** (*int*) – width of the image
- **height** (*int*) – height of the image

1.4 Modules: Communication Transports

1.4.1 Transport (Abstract Base)

class StreamDeck.Transport.Transport.Transport

Bases: ABC

Base transport layer, representing an abstract communication back-end which can be used to discovery attached StreamDeck devices.

class Device

Bases: ABC

Base connection device, representing an abstract connected device which can be communicated with by an upper layer high level protocol.

abstract close()

Closes the device for input/output.

See also:

See [open\(\)](#) for the corresponding open method.

abstract connected()

Indicates if the physical device object this instance is attached to is still connected to the host.

Return type

bool

Returns

True if the device is still connected, *False* otherwise.

abstract is_open()

Indicates if the physical device object this instance is attached to has been opened by the host.

Return type

bool

Returns

True if the device is open, *False* otherwise.

abstract open()

Opens the device for input/output. This must be called prior to sending or receiving any reports.

See also:

See [`close\(\)`](#) for the corresponding close method.

abstract path()

Retrieves the logical path of the attached device within the current system. This can be used to uniquely differentiate one device from another.

Return type

str

Returns

Logical device path for the attached device.

abstract product_id()

Retrieves the product ID value of the attached device.

Return type

int

Returns

Product ID of the attached device.

abstract read(*length*)

Performs a blocking read of a HID In report from the open HID device.

Parameters

length (*int*) – Maximum length of the In report to read.

Return type

list(byte)

Returns

List of bytes containing the read In report. The first byte of the report will be the Report ID of the report that was read.

abstract read_feature(*report_id*, *length*)

Reads a HID Feature report from the open HID device.

Parameters

- **report_id** (*int*) – Report ID of the report being read.
- **length** (*int*) – Maximum length of the Feature report to read.

Return type

list(byte)

Returns

List of bytes containing the read Feature report. The first byte of the report will be the Report ID of the report that was read.

abstract vendor_id()

Retrieves the vendor ID value of the attached device.

Return type

int

Returns

Vendor ID of the attached device.

abstract write(payload)

Sends a HID Out report to the open HID device.

Parameters

payload (*enumerable()*) – Enumerate list of bytes to send to the device, as an Out report. The first byte of the report should be the Report ID of the report being sent.

Return type

int

Returns

Number of bytes successfully sent to the device.

abstract write_feature(payload)

Sends a HID Feature report to the open HID device.

Parameters

payload (*enumerable()*) – Enumerate list of bytes to send to the device, as a feature report. The first byte of the report should be the Report ID of the report being sent.

Return type

int

Returns

Number of bytes successfully sent to the device.

abstract enumerate(vid, pid)

Enumerates all available devices on the system using the current transport back-end.

Parameters

- **vid** (*int*) – USB Vendor ID to filter all devices by, *None* if the device list should not be filtered by vendor.
- **pid** (*int*) – USB Product ID to filter all devices by, *None* if the device list should not be filtered by product.

Return type

list(*Transport.Device*)

Returns

List of discovered devices that are available through this transport back-end.

abstract static probe()

Attempts to determine if the back-end is installed and usable. It is expected that probe failures throw exceptions detailing their exact cause of failure.

exception StreamDeck.Transport.Transport.TransportError

Bases: *Exception*

Exception thrown when attempting to access a device using a backend transport that has failed (for example, if the requested device could not be accessed).

1.4.2 ‘LibUSB HIDAPI’ Library Transport

class StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI

Bases: *Transport*

USB HID transport layer, using the LibUSB HIDAPI dynamically linked library directly via ctypes.

class Device(hidapi, device_info)

Bases: *Device*

close()

Closes the device for input/output.

See also:

See `open()` for the corresponding open method.

connected()

Indicates if the physical device object this instance is attached to is still connected to the host.

Return type

bool

Returns

True if the device is still connected, *False* otherwise.

is_open()

Indicates if the physical device object this instance is attached to has been opened by the host.

Return type

bool

Returns

True if the device is open, *False* otherwise.

open()

Opens the device for input/output. This must be called prior to sending or receiving any reports.

See also:

See `close()` for the corresponding close method.

path()

Retrieves the logical path of the attached device within the current system. This can be used to uniquely differentiate one device from another.

Return type

str

Returns

Logical device path for the attached device.

product_id()

Retrieves the product ID value of the attached device.

Return type

int

Returns

Product ID of the attached device.

read(length)

Performs a blocking read of a HID In report from the open HID device.

Parameters

length (*int*) – Maximum length of the In report to read.

Return type

list(byte)

Returns

List of bytes containing the read In report. The first byte of the report will be the Report ID of the report that was read.

read_feature(report_id, length)

Reads a HID Feature report from the open HID device.

Parameters

- **report_id** (*int*) – Report ID of the report being read.
- **length** (*int*) – Maximum length of the Feature report to read.

Return type

list(byte)

Returns

List of bytes containing the read Feature report. The first byte of the report will be the Report ID of the report that was read.

vendor_id()

Retrieves the vendor ID value of the attached device.

Return type

int

Returns

Vendor ID of the attached device.

write(payload)

Sends a HID Out report to the open HID device.

Parameters

payload (*enumerable()*) – Enumerate list of bytes to send to the device, as an Out report. The first byte of the report should be the Report ID of the report being sent.

Return type

int

Returns

Number of bytes successfully sent to the device.

write_feature(payload)

Sends a HID Feature report to the open HID device.

Parameters

payload (*enumerable()*) – Enumerate list of bytes to send to the device, as a feature report. The first byte of the report should be the Report ID of the report being sent.

Return type

int

Returns

Number of bytes successfully sent to the device.

enumerate(vid, pid)

Enumerates all available devices on the system using the current transport back-end.

Parameters

- **vid** (*int*) – USB Vendor ID to filter all devices by, *None* if the device list should not be filtered by vendor.
- **pid** (*int*) – USB Product ID to filter all devices by, *None* if the device list should not be filtered by product.

Return typelist(*Transport.Device*)**Returns**

List of discovered devices that are available through this transport back-end.

static probe()

Attempts to determine if the back-end is installed and usable. It is expected that probe failures throw exceptions detailing their exact cause of failure.

1.5 Modules: Image Helpers

1.5.1 PIL Image Helper

1.6 Example Script: Device Information

The following is a complete example script to enumerate any available StreamDeck devices and print out all information on the device's location and image format.

```
#!/usr/bin/env python3

#      Python Stream Deck Library
#      Released under the MIT license
#
#      dean [at] fourwalledcubicle [dot] com
#      www.fourwalledcubicle.com
#

# Example script that prints out information about any discovered StreamDeck
# devices to the console.

from StreamDeck.DeviceManager import DeviceManager

# Prints diagnostic information about a given StreamDeck.
def print_deck_info(index, deck):
    key_image_format = deck.key_image_format()
    touchscreen_image_format = deck.touchscreen_image_format()

    flip_description = {
        (False, False): "not mirrored",
        (True, False): "mirrored horizontally",
        (False, True): "mirrored vertically",
        (True, True): "mirrored horizontally/vertically",
    }

    print("Deck {} - {}".format(index, deck.deck_type()))
    print("\t - ID: {}".format(deck.id()))
    print("\t - Serial: '{}'.format(deck.get_serial_number()))
    print("\t - Firmware Version: '{}'.format(deck.get_firmware_version()))
    print("\t - Key Count: {} (in a {}x{} grid)".format(
        deck.key_count(),
        deck.key_layout()[0],
        deck.key_layout()[1]))
    if deck.is_visual():
        print("\t - Key Images: {}x{} pixels, {} format, rotated {} degrees, {}".format(
            key_image_format['size'][0],
            key_image_format['size'][1],
            key_image_format['format'],
            key_image_format['rotation'],
            flip_description[key_image_format['flip']]))
```

(continues on next page)

(continued from previous page)

```

    if deck.is_touch():
        print("\t - Touchscreen: {}x{} pixels, {} format, rotated {} degrees, {}".
            ↪format(
                touchscreen_image_format['size'][0],
                touchscreen_image_format['size'][1],
                touchscreen_image_format['format'],
                touchscreen_image_format['rotation'],
                flip_description[touchscreen_image_format['flip']]))
    else:
        print("\t - No Visual Output")

if __name__ == "__main__":
    streamdecks = DeviceManager().enumerate()

    print("Found {} Stream Deck(s).\n".format(len(streamdecks)))

    for index, deck in enumerate(streamdecks):
        deck.open()
        deck.reset()

        print_deck_info(index, deck)

        deck.close()

```

1.7 Example Script: Basic Usage

The following is a complete example script to connect to attached StreamDeck devices, display custom image/text graphics on the buttons and respond to press events.

```

#!/usr/bin/env python3

#         Python Stream Deck Library
#         Released under the MIT license
#
#   dean [at] fourwalledcubicle [dot] com
#       www.fourwalledcubicle.com
#

# Example script showing basic library usage - updating key images with new
# tiles generated at runtime, and responding to button state change events.

import os
import threading

from PIL import Image, ImageDraw, ImageFont
from StreamDeck.DeviceManager import DeviceManager
from StreamDeck.ImageHelpers import PILHelper

# Folder location of image assets used by this example.

```

(continues on next page)

(continued from previous page)

```

ASSETS_PATH = os.path.join(os.path.dirname(__file__), "Assets")

# Generates a custom tile with run-time generated text and custom image via the
# PIL module.
def render_key_image(deck, icon_filename, font_filename, label_text):
    # Resize the source image asset to best-fit the dimensions of a single key,
    # leaving a margin at the bottom so that we can draw the key title
    # afterwards.
    icon = Image.open(icon_filename)
    image = PILHelper.create_scaled_key_image(deck, icon, margins=[0, 0, 20, 0])

    # Load a custom TrueType font and use it to overlay the key index, draw key
    # label onto the image a few pixels from the bottom of the key.
    draw = ImageDraw.Draw(image)
    font = ImageFont.truetype(font_filename, 14)
    draw.text((image.width / 2, image.height - 5), text=label_text, font=font, anchor="ms
↪", fill="white")

    return PILHelper.to_native_key_format(deck, image)

# Returns styling information for a key based on its position and state.
def get_key_style(deck, key, state):
    # Last button in the example application is the exit button.
    exit_key_index = deck.key_count() - 1

    if key == exit_key_index:
        name = "exit"
        icon = "{}.png".format("Exit")
        font = "Roboto-Regular.ttf"
        label = "Bye" if state else "Exit"
    else:
        name = "emoji"
        icon = "{}.png".format("Pressed" if state else "Released")
        font = "Roboto-Regular.ttf"
        label = "Pressed!" if state else "Key {}".format(key)

    return {
        "name": name,
        "icon": os.path.join(ASSETS_PATH, icon),
        "font": os.path.join(ASSETS_PATH, font),
        "label": label
    }

# Creates a new key image based on the key index, style and current key state
# and updates the image on the StreamDeck.
def update_key_image(deck, key, state):
    # Determine what icon and label to use on the generated key.
    key_style = get_key_style(deck, key, state)

```

(continues on next page)

(continued from previous page)

```

# Generate the custom key with the requested image and label.
image = render_key_image(deck, key_style["icon"], key_style["font"], key_style["label"]
↪")

# Use a scoped-with on the deck to ensure we're the only thread using it
# right now.
with deck:
    # Update requested key with the generated image.
    deck.set_key_image(key, image)

# Prints key state change information, updates the key image and performs any
# associated actions when a key is pressed.
def key_change_callback(deck, key, state):
    # Print new key state
    print("Deck {} Key {} = {}".format(deck.id(), key, state), flush=True)

    # Update the key image based on the new key state.
    update_key_image(deck, key, state)

    # Check if the key is changing to the pressed state.
    if state:
        key_style = get_key_style(deck, key, state)

        # When an exit button is pressed, close the application.
        if key_style["name"] == "exit":
            # Use a scoped-with on the deck to ensure we're the only thread
            # using it right now.
            with deck:
                # Reset deck, clearing all button images.
                deck.reset()

                # Close deck handle, terminating internal worker threads.
                deck.close()

if __name__ == "__main__":
    streamdecks = DeviceManager().enumerate()

    print("Found {} Stream Deck(s).\n".format(len(streamdecks)))

    for index, deck in enumerate(streamdecks):
        # This example only works with devices that have screens.
        if not deck.is_visual():
            continue

        deck.open()
        deck.reset()

        print("Opened '{}' device (serial number: '{}', fw: '{}')".format(
            deck.deck_type(), deck.get_serial_number(), deck.get_firmware_version()
        ))

```

(continues on next page)

(continued from previous page)

```

# Set initial screen brightness to 30%.
deck.set_brightness(30)

# Set initial key images.
for key in range(deck.key_count()):
    update_key_image(deck, key, False)

# Register callback function for when a key state changes.
deck.set_key_callback(key_change_callback)

# Wait until all application threads have terminated (for this example,
# this is when all deck handles are closed).
for t in threading.enumerate():
    try:
        t.join()
    except RuntimeError:
        pass

```

1.8 Example Script: Pedal

The following is an example script to connect to the Stream Deck Pedal, and print whenever its buttons are pushed and released.

```

#!/usr/bin/env python3

#      Python Stream Deck Library
#      Released under the MIT license
#
#      dean [at] fourwalledcubicle [dot] com
#      www.fourwalledcubicle.com
#

# Example script showing basic library usage, printing button presses. This
# example only shows key events, and is intended to demonstrate how to get
# such events from device that lack screens, i.e. the StreamDeck Pedal.

import threading

from StreamDeck.DeviceManager import DeviceManager

def key_change_callback(deck, key, state):
    print("Deck {} Key {} = {}".format(deck.id(), key, "down" if state else "up"),
        flush=True)

if __name__ == "__main__":
    streamdecks = DeviceManager().enumerate()

```

(continues on next page)

(continued from previous page)

```

print("Found {} Stream Deck(s).\n".format(len(streamdecks)))

for index, deck in enumerate(streamdecks):
    deck.open()

    print("Opened '{}' device (serial number: '{}', fw: '{}')".format(
        deck.deck_type(), deck.get_serial_number(), deck.get_firmware_version()
    ))

    # Register callback function for when a key state changes.
    deck.set_key_callback(key_change_callback)

    # Wait until all application threads have terminated (for this example,
    # this is when all deck handles are closed).
    for t in threading.enumerate():
        try:
            t.join()
        except RuntimeError:
            pass

```

1.9 Example Script: StreamDeck +

The following is a complete example script to display an image on the touchscreen of the Stream Deck +, and react to dial events.

```

#!/usr/bin/env python3

#         Python Stream Deck Library
#     Released under the MIT license
#
#
# Example script showing some Stream Deck + specific functions

import os
import threading
import io

from PIL import Image
from StreamDeck.DeviceManager import DeviceManager
from StreamDeck.Devices.StreamDeck import DialEventType, TouchscreenEventType

# Folder location of image assets used by this example.
ASSETS_PATH = os.path.join(os.path.dirname(__file__), "Assets")

# image for idle state
img = Image.new('RGB', (120, 120), color='black')
released_icon = Image.open(os.path.join(ASSETS_PATH, 'Released.png')).resize((80, 80))
img.paste(released_icon, (20, 20), released_icon)

```

(continues on next page)

(continued from previous page)

```

img_byte_arr = io.BytesIO()
img.save(img_byte_arr, format='JPEG')
img_released_bytes = img_byte_arr.getvalue()

# image for pressed state
img = Image.new('RGB', (120, 120), color='black')
pressed_icon = Image.open(os.path.join(ASSETS_PATH, 'Pressed.png')).resize((80, 80))
img.paste(pressed_icon, (20, 20), pressed_icon)

img_byte_arr = io.BytesIO()
img.save(img_byte_arr, format='JPEG')
img_pressed_bytes = img_byte_arr.getvalue()

# callback when buttons are pressed or released
def key_change_callback(deck, key, key_state):
    print("Key: " + str(key) + " state: " + str(key_state))

    deck.set_key_image(key, img_pressed_bytes if key_state else img_released_bytes)

# callback when dials are pressed or released
def dial_change_callback(deck, dial, event, value):
    if event == DialEventType.PUSH:
        print(f"dial pushed: {dial} state: {value}")
        if dial == 3 and value:
            deck.reset()
            deck.close()
        else:
            # build an image for the touch lcd
            img = Image.new('RGB', (800, 100), 'black')
            icon = Image.open(os.path.join(ASSETS_PATH, 'Exit.png')).resize((80, 80))
            img.paste(icon, (690, 10), icon)

            for k in range(0, deck.DIAL_COUNT - 1):
                img.paste(pressed_icon if (dial == k and value) else released_icon, (30_
↪ + (k * 220), 10),
                        pressed_icon if (dial == k and value) else released_icon)

            img_byte_arr = io.BytesIO()
            img.save(img_byte_arr, format='JPEG')
            img_byte_arr = img_byte_arr.getvalue()

            deck.set_touchscreen_image(img_byte_arr, 0, 0, 800, 100)
    elif event == DialEventType.TURN:
        print(f"dial {dial} turned: {value}")

# callback when lcd is touched
def touchscreen_event_callback(deck, evt_type, value):
    if evt_type == TouchscreenEventType.SHORT:
        print("Short touch @ " + str(value['x']) + "," + str(value['y']))

```

(continues on next page)

(continued from previous page)

```

elif evt_type == TouchscreenEventType.LONG:

    print("Long touch @ " + str(value['x']) + "," + str(value['y']))

elif evt_type == TouchscreenEventType.DRAG:

    print("Drag started @ " + str(value['x']) + "," + str(value['y']) + " ended @ " +
    ↪+ str(value['x_out']) + "," + str(value['y_out']))

if __name__ == "__main__":
    streamdecks = DeviceManager().enumerate()

    print("Found {} Stream Deck(s).\n".format(len(streamdecks)))

    for index, deck in enumerate(streamdecks):
        # This example only works with devices that have screens.

        if deck.DECK_TYPE != 'Stream Deck +':
            print(deck.DECK_TYPE)
            print("Sorry, this example only works with Stream Deck +")
            continue

        deck.open()
        deck.reset()

        deck.set_key_callback(key_change_callback)
        deck.set_dial_callback(dial_change_callback)
        deck.set_touchscreen_callback(touchscreen_event_callback)

        print("Opened '{}' device (serial number: '{}')".format(deck.deck_type(), deck.
        ↪get_serial_number()))

        # Set initial screen brightness to 30%.
        deck.set_brightness(100)

        for key in range(0, deck.KEY_COUNT):
            deck.set_key_image(key, img_released_bytes)

        # build an image for the touch lcd
        img = Image.new('RGB', (800, 100), 'black')
        icon = Image.open(os.path.join(ASSETS_PATH, 'Exit.png')).resize((80, 80))
        img.paste(icon, (690, 10), icon)

        for dial in range(0, deck.DIAL_COUNT - 1):
            img.paste(released_icon, (30 + (dial * 220), 10), released_icon)

        img_bytes = io.BytesIO()
        img.save(img_bytes, format='JPEG')
        touchscreen_image_bytes = img_bytes.getvalue()

```

(continues on next page)

(continued from previous page)

```

deck.set_touchscreen_image(touchscreen_image_bytes, 0, 0, 800, 100)

# Wait until all application threads have terminated (for this example,
# this is when all deck handles are closed).
for t in threading.enumerate():
    try:
        t.join()
    except RuntimeError:
        pass

```

1.10 Example Script: Tiled Image

The following is a complete example script to display a larger image across a StreamDeck, by sectioning up the image into key-sized tiles, and displaying them individually onto each key.

```

#!/usr/bin/env python3

# Python Stream Deck Library
# Released under the MIT license
#
# dean [at] fourwalledcubicle [dot] com
# www.fourwalledcubicle.com
#

# Example script showing how to tile a larger image across multiple buttons, by
# first generating an image suitable for the entire deck, then cropping out and
# applying key-sized tiles to individual keys of a StreamDeck.

import os
import threading

from PIL import Image, ImageOps
from StreamDeck.DeviceManager import DeviceManager
from StreamDeck.ImageHelpers import PILHelper

# Folder location of image assets used by this example.
ASSETS_PATH = os.path.join(os.path.dirname(__file__), "Assets")

# Generates an image that is correctly sized to fit across all keys of a given
# StreamDeck.
def create_full_deck_sized_image(deck, key_spacing, image_filename):
    key_rows, key_cols = deck.key_layout()
    key_width, key_height = deck.key_image_format()['size']
    spacing_x, spacing_y = key_spacing

    # Compute total size of the full StreamDeck image, based on the number of
    # buttons along each axis. This doesn't take into account the spaces between
    # the buttons that are hidden by the bezel.
    key_width *= key_cols

```

(continues on next page)

(continued from previous page)

```

key_height *= key_rows

# Compute the total number of extra non-visible pixels that are obscured by
# the bezel of the StreamDeck.
spacing_x *= key_cols - 1
spacing_y *= key_rows - 1

# Compute final full deck image size, based on the number of buttons and
# obscured pixels.
full_deck_image_size = (key_width + spacing_x, key_height + spacing_y)

# Resize the image to suit the StreamDeck's full image size. We use the
# helper function in Pillow's ImageOps module so that the image's aspect
# ratio is preserved.
image = Image.open(os.path.join(ASSETS_PATH, image_filename)).convert("RGBA")
image = ImageOps.fit(image, full_deck_image_size, Image.LANCZOS)
return image

# Crops out a key-sized image from a larger deck-sized image, at the location
# occupied by the given key index.
def crop_key_image_from_deck_sized_image(deck, image, key_spacing, key):
    key_rows, key_cols = deck.key_layout()
    key_width, key_height = deck.key_image_format()['size']
    spacing_x, spacing_y = key_spacing

    # Determine which row and column the requested key is located on.
    row = key // key_cols
    col = key % key_cols

    # Compute the starting X and Y offsets into the full size image that the
    # requested key should display.
    start_x = col * (key_width + spacing_x)
    start_y = row * (key_height + spacing_y)

    # Compute the region of the larger deck image that is occupied by the given
    # key, and crop out that segment of the full image.
    region = (start_x, start_y, start_x + key_width, start_y + key_height)
    segment = image.crop(region)

    # Create a new key-sized image, and paste in the cropped section of the
    # larger image.
    key_image = PILHelper.create_key_image(deck)
    key_image.paste(segment)

    return PILHelper.to_native_key_format(deck, key_image)

# Closes the StreamDeck device on key state change.
def key_change_callback(deck, key, state):
    # Use a scoped-with on the deck to ensure we're the only thread using it
    # right now.

```

(continues on next page)

(continued from previous page)

```

with deck:
    # Reset deck, clearing all button images.
    deck.reset()

    # Close deck handle, terminating internal worker threads.
    deck.close()

if __name__ == "__main__":
    streamdecks = DeviceManager().enumerate()

    print("Found {} Stream Deck(s).\n".format(len(streamdecks)))

    for index, deck in enumerate(streamdecks):
        # This example only works with devices that have screens.
        if not deck.is_visual():
            continue

        deck.open()
        deck.reset()

        print("Opened '{}' device (serial number: '{}')".format(deck.deck_type(), deck.
↳get_serial_number()))

        # Set initial screen brightness to 30%.
        deck.set_brightness(30)

        # Approximate number of (non-visible) pixels between each key, so we can
        # take those into account when cutting up the image to show on the keys.
        key_spacing = (36, 36)

        # Load and resize a source image so that it will fill the given
        # StreamDeck.
        image = create_full_deck_sized_image(deck, key_spacing, "Harold.jpg")

        print("Created full deck image size of {}x{} pixels.".format(image.width, image.
↳height))

        # Extract out the section of the image that is occupied by each key.
        key_images = dict()
        for k in range(deck.key_count()):
            key_images[k] = crop_key_image_from_deck_sized_image(deck, image, key_
↳spacing, k)

        # Use a scoped-with on the deck to ensure we're the only thread
        # using it right now.
        with deck:
            # Draw the individual key images to each of the keys.
            for k in range(deck.key_count()):
                key_image = key_images[k]

                # Show the section of the main image onto the key.

```

(continues on next page)

(continued from previous page)

```

        deck.set_key_image(k, key_image)

    # Register callback function for when a key state changes.
    deck.set_key_callback(key_change_callback)

    # Wait until all application threads have terminated (for this example,
    # this is when all deck handles are closed).
    for t in threading.enumerate():
        try:
            t.join()
        except RuntimeError:
            pass

```

1.11 Example Script: Animated Images

The following is a complete example script to connect to attached StreamDeck devices, and display animated graphics on the keys.

```

#!/usr/bin/env python3

#         Python Stream Deck Library
#     Released under the MIT license
#
#   dean [at] fourwalledcubicle [dot] com
#       www.fourwalledcubicle.com
#

# Example script showing one way to display animated images using the
# library, by pre-rendering all the animation frames into the StreamDeck
# device's native image format, and displaying them with a periodic
# timer.

import itertools
import os
import threading
import time

from fractions import Fraction
from PIL import Image, ImageSequence
from StreamDeck.DeviceManager import DeviceManager
from StreamDeck.ImageHelpers import PILHelper
from StreamDeck.Transport.Transport import TransportError

# Folder location of image assets used by this example.
ASSETS_PATH = os.path.join(os.path.dirname(__file__), "Assets")

# Animation frames per second to attempt to display on the StreamDeck devices.
FRAMES_PER_SECOND = 30

```

(continues on next page)

(continued from previous page)

```

# Loads in a source image, extracts out the individual animation frames (if
# any) and returns a list of animation frames in the StreamDeck device's
# native image format.
def create_animation_frames(deck, image_filename):
    icon_frames = list()

    # Open the source image asset.
    icon = Image.open(os.path.join(ASSETS_PATH, image_filename))

    # Iterate through each animation frame of the source image
    for frame in ImageSequence.Iterator(icon):
        # Create new key image of the correct dimensions, black background.
        frame_image = PILHelper.create_scaled_key_image(deck, frame)

        # Pre-convert the generated image to the native format of the StreamDeck
        # so we don't need to keep converting it when showing it on the device.
        native_frame_image = PILHelper.to_native_key_format(deck, frame_image)

        # Store the rendered animation frame for later user.
        icon_frames.append(native_frame_image)

    # Return the decoded list of frames - the caller will need to decide how to
    # sequence them for display.
    return icon_frames

# Closes the StreamDeck device on key state change.
def key_change_callback(deck, key, state):
    # Use a scoped-with on the deck to ensure we're the only thread using it
    # right now.
    with deck:
        # Reset deck, clearing all button images.
        deck.reset()

        # Close deck handle, terminating internal worker threads.
        deck.close()

if __name__ == "__main__":
    streamdecks = DeviceManager().enumerate()

    print("Found {} Stream Deck(s).\n".format(len(streamdecks)))

    for index, deck in enumerate(streamdecks):
        # This example only works with devices that have screens.
        if not deck.is_visual():
            continue

        deck.open()
        deck.reset()

        print("Opened '{}' device (serial number: '{}')".format(deck.deck_type(), deck.

```

(continues on next page)

(continued from previous page)

```

↪get_serial_number()))

    # Set initial screen brightness to 30%.
    deck.set_brightness(30)

    # Pre-render a list of animation frames for each source image, in the
    # native display format so that they can be quickly sent to the device.
    print("Loading animations...")
    animations = [
        create_animation_frames(deck, "Elephant_Walking_animated.gif"),
        create_animation_frames(deck, "RGB_color_space_animated_view.gif"),
        create_animation_frames(deck, "Simple_CV_Joint_animated.gif"),
    ]
    print("Ready.")

    # Create a mapping of StreamDeck keys to animation image sets that will
    # be displayed.
    key_images = dict()
    for k in range(deck.key_count()):
        # Each key gets an infinite cycle generator bound to the animation
        # frames, so it will loop the animated sequence forever.
        key_images[k] = itertools.cycle(animations[k % len(animations)])

    # Helper function that will run a periodic loop which updates the
    # images on each key.
    def animate(fps):
        # Convert frames per second to frame time in seconds.
        #
        # Frame time often cannot be fully expressed by a float type,
        # meaning that we have to use fractions.
        frame_time = Fraction(1, fps)

        # Get a starting absolute time reference point.
        #
        # We need to use an absolute time clock, instead of relative sleeps
        # with a constant value, to avoid drifting.
        #
        # Drifting comes from an overhead of scheduling the sleep itself -
        # it takes some small amount of time for `time.sleep()` to execute.
        next_frame = Fraction(time.monotonic())

        # Periodic loop that will render every frame at the set FPS until
        # the StreamDeck device we're using is closed.
        while deck.is_open():
            try:
                # Use a scoped-with on the deck to ensure we're the only
                # thread using it right now.
                with deck:
                    # Update the key images with the next animation frame.
                    for key, frames in key_images.items():
                        deck.set_key_image(key, next(frames))
            except TransportError as err:

```

(continues on next page)

(continued from previous page)

```

    print("TransportError: {}".format(err))
    # Something went wrong while communicating with the device
    # (closed?) - don't re-schedule the next animation frame.
    break

    # Set the next frame absolute time reference point.
    #
    # We are running at the fixed `fps`, so this is as simple as
    # adding the frame time we calculated earlier.
    next_frame += frame_time

    # Knowing the start of the next frame, we can calculate how long
    # we have to sleep until its start.
    sleep_interval = float(next_frame) - time.monotonic()

    # Schedule the next periodic frame update.
    #
    # `sleep_interval` can be a negative number when current FPS
    # setting is too high for the combination of host and
    # StreamDeck to handle. If this is the case, we skip sleeping
    # immediately render the next frame to try to catch up.
    if sleep_interval >= 0:
        time.sleep(sleep_interval)

    # Kick off the key image animating thread.
    threading.Thread(target=animate, args=[FRAMES_PER_SECOND]).start()

    # Register callback function for when a key state changes.
    deck.set_key_callback(key_change_callback)

    # Wait until all application threads have terminated (for this example,
    # this is when all deck handles are closed).
    for t in threading.enumerate():
        try:
            t.join()
        except RuntimeError:
            pass

```

1.12 Source Code

Source code is available on [Github](#). Bug reports, patches, suggestions and other contributions welcome.

1.13 Changelog

Version 0.9.5:

- Added support for the StreamDeck Plus.

Version 0.9.4:

- Updated Windows HIDAPI backend to attempt to load from the local working directory.
- Added detection for MacOS Homebrew installations of the libhidapi back-end library.

Version 0.9.3:

- Added support for a new sub-variant of the StreamDeck XL.

Version 0.9.2:

- Added support for a new sub-variant of the StreamDeck Mini.

Version 0.9.1:

- Transport errors now trigger a closing of the underlying StreamDeck device, so further API calls will throw correctly (and `is_open()` will return `False`).
- Updated animated example script to use separate cycle generators for each key, so the animations play at the correct rate regardless of key count.
- Added support for the StreamDeck pedal.
- Added new `is_visual()` function.

Version 0.9.0:

- Added new `set_poll_frequency()` function.
- Added new `is_open()` function.
- Fixed a possible internal thread join error when a deck object was closed.

Version 0.8.5:

- Add support for the new StreamDeck MK2.

Version 0.8.4:

- Updated animated example script to attempt to maintain a constant FPS, regardless of rendering time.
- Fixed a race condition in the LibUSB HIDAPI transport backend that could cause crashes when a device was closed.

Version 0.8.3:

- Altered LibUSB transport workaround to only apply on Mac.
- Fixed internal `_extract_string()` method to discard all data after the first NUL byte, fixing corrupt serial number strings being returned in some cases.
- Set minimum Python version to 3.8, as some of the library uses newer syntax/core library features.

Version 0.8.2:

- Added new `PILHelper.create_scaled_image()` function to easily generate scaled/padded key images for a given deck.
- Updated LibUSB transport backend so that device paths are returned as UTF-8 strings, not raw bytes.

- Updated version/serial number string extraction from StreamDecks so that invalid characters are substituted, rather than raising a `UnicodeDecodeError` error.
- Added LibUSB transport workaround for a bug on Mac platforms when using older versions of the library.

Version 0.8.1:

- Fixed memory leak in LibUSB HIDAPI transport backend.

Version 0.8.0:

- Fix random crashes in LibUSB HIDAPI transport backend on Windows, as the API is not thread safe.
- Added support for atomic updates of StreamDeck instances via the Python `with` scope syntax.

Version 0.7.3:

- Fix crash in new LibUSB HIDAPI transport backend on systems with multiple connected StreamDeck devices.
- Fix crash in new LibUSB HIDAPI transport backend when `connected()` was called on a StreamDeck instance.

Version 0.7.2:

- Documentation restructuring to move installation out of the readme and into the library documentation.

Version 0.7.1:

- Cleaned up new LibUSB HIDAPI transport backend, so that it only searches for OS-specific library files.
- Fixed minor typo in the libUSB HIDAPI transport backend probe failure message.

Version 0.7.0:

- Removed old HID and HIDAPI backends, added new `ctypes` based LibUSB-HIDAPI backend replacement.

Version 0.6.3:

- Added support for the new V2 hardware revision of the StreamDeck Original.

Version 0.6.2:

- Fixed broken StreamDeck XL communications on Linux.
- Added blacklist for the `libhidapi-hidraw` system library which breaks StreamDeck Original communications.

Version 0.6.1:

- Fixed broken HIDAPI backend probing.
- Fixed double-open of HID backend devices causing connection issues on some platforms.

Version 0.6.0:

- Added support for the HID Python package. This new HID backend is strongly recommended over the old HIDAPI backend.
- Added auto-probing of installed backends, if no specific transport is supplied when constructing a `DeviceManager` instance.

Version 0.5.1:

- Fixed StreamDeck XL reporting swapped rows/columns count.
- Fixed StreamDeck XL failing to report correct serial number and firmware version.

Version 0.5.0:

- Fixed StreamDeck devices occasionally showing partial old frames on initial connection.
- Removed support for RAW pixel images, StreamDeck Mini and Original take BMP images.
- Removed width and height information from Deck key image dict, now returned as size tuple entry.

Version 0.4.0:

- Added StreamDeck XL support.

Version 0.3.2:

- Fixed StreamDeck Mini key images not updating under some circumstances.

Version 0.3.1:

- Added animated image example script.

Version 0.3:

- Remapped StreamDeck key indexes so that key 0 is located on the physical top-left of all supported devices.

Version 0.2.4:

- Added new `StreamDeck.get_serial_number()` function.
- Added new `StreamDeck.get_firmware_version()` function.

Version 0.2.3:

- Added new `StreamDeck.ImageHelpers` modules for easier key image generation.

1.14 License

Released under the MIT license below.

Permission to use, copy, modify, **and** distribute this software **and** its documentation **for any** purpose **is** hereby granted without fee, provided that the above copyright notice appear **in all** copies **and** that both that the copyright notice **and** this permission notice **and** warranty disclaimer appear **in** supporting documentation, **and** that the name of the author **not** be used **in** advertising **or** publicity pertaining to distribution of the software without specific, written prior permission.

The author disclaims **all** warranties **with** regard to this software, including **all** implied warranties of merchantability **and** fitness. In no event shall the author be liable **for any** special, indirect **or** consequential damages **or any** damages whatsoever resulting **from loss** of use, data **or** profits, whether **in** an action of contract, negligence **or** other tortious action, arising out of **or in** connection **with** the use **or** performance of this software.

INDICES AND TABLES

- `genindex`
- `modindex`

PYTHON MODULE INDEX

S

`StreamDeck.DeviceManager`, 4
`StreamDeck.Devices.StreamDeck`, 5
`StreamDeck.Devices.StreamDeckMini`, 12
`StreamDeck.Devices.StreamDeckOriginal`, 11
`StreamDeck.Devices.StreamDeckPedal`, 15
`StreamDeck.Devices.StreamDeckPlus`, 16
`StreamDeck.Devices.StreamDeckXL`, 14
`StreamDeck.Transport.LibUSBHIDAPI`, 19
`StreamDeck.Transport.Transport`, 17

INDEX

C

close() (*StreamDeck.Devices.StreamDeck.StreamDeck* method), 5

close() (*StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI.Device* method), 19

close() (*StreamDeck.Transport.Transport.Transport.Device* method), 17

connected() (*StreamDeck.Devices.StreamDeck.StreamDeck* method), 5

connected() (*StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI.Device* method), 20

connected() (*StreamDeck.Transport.Transport.Transport.Device* method), 17

get_firmware_version() (*StreamDeck.Devices.StreamDeckOriginal.StreamDeckOriginal* method), 11

get_firmware_version() (*StreamDeck.Devices.StreamDeckPedal.StreamDeckPedal* method), 15

get_firmware_version() (*StreamDeck.Devices.StreamDeckPlus.StreamDeckPlus* method), 16

get_firmware_version() (*StreamDeck.Devices.StreamDeckXL.StreamDeckXL* method), 14

get_serial_number() (*StreamDeck.Devices.StreamDeck.StreamDeck* method), 6

D

deck_type() (*StreamDeck.Devices.StreamDeck.StreamDeck* method), 5

DeviceManager (class in *StreamDeck.DeviceManager*), 4

dial_count() (*StreamDeck.Devices.StreamDeck.StreamDeck* method), 5

dial_states() (*StreamDeck.Devices.StreamDeck.StreamDeck* method), 6

DialEventType (class in *StreamDeck.Devices.StreamDeck*), 5

get_serial_number() (*StreamDeck.Devices.StreamDeckMini.StreamDeckMini* method), 13

get_serial_number() (*StreamDeck.Devices.StreamDeckOriginal.StreamDeckOriginal* method), 11

get_serial_number() (*StreamDeck.Devices.StreamDeckPedal.StreamDeckPedal* method), 15

get_serial_number() (*StreamDeck.Devices.StreamDeckPlus.StreamDeckPlus* method), 16

get_serial_number() (*StreamDeck.Devices.StreamDeckXL.StreamDeckXL* method), 14

E

enumerate() (*StreamDeck.DeviceManager.DeviceManager* method), 5

enumerate() (*StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI* method), 21

enumerate() (*StreamDeck.Transport.Transport.Transport* method), 19

id() (*StreamDeck.Devices.StreamDeck.StreamDeck* method), 6

is_open() (*StreamDeck.Devices.StreamDeck.StreamDeck* method), 6

is_open() (*StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI.Device* method), 20

is_open() (*StreamDeck.Transport.Transport.Transport.Device* method), 17

is_touch() (*StreamDeck.Devices.StreamDeck.StreamDeck* method), 6

G

get_firmware_version() (*StreamDeck.Devices.StreamDeck.StreamDeck* method), 6

get_firmware_version() (*StreamDeck.Devices.StreamDeckMini.StreamDeckMini* method), 12

`is_visual()` (*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 7

K

`key_count()` (*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 7

`key_image_format()` (*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 7

`key_layout()` (*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 7

`key_states()` (*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 7

L

`LibUSBHIDAPI` (class
StreamDeck.Transport.LibUSBHIDAPI),
19

`LibUSBHIDAPI.Device` (class
StreamDeck.Transport.LibUSBHIDAPI),
19

M

module

StreamDeck.DeviceManager, 4
StreamDeck.Devices.StreamDeck, 5
StreamDeck.Devices.StreamDeckMini, 12
StreamDeck.Devices.StreamDeckOriginal, 11
StreamDeck.Devices.StreamDeckPedal, 15
StreamDeck.Devices.StreamDeckPlus, 16
StreamDeck.Devices.StreamDeckXL, 14
StreamDeck.Transport.LibUSBHIDAPI, 19
StreamDeck.Transport.Transport, 17

O

`open()` (*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 7

`open()` (*StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI.Device*
method), 20

`open()` (*StreamDeck.Transport.Transport.Transport.Device*
method), 18

P

`path()` (*StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI.Device*
method), 20

`path()` (*StreamDeck.Transport.Transport.Transport.Device*
method), 18

`probe()` (*StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI*
static method), 21

`probe()` (*StreamDeck.Transport.Transport.Transport*
static method), 19

`ProbeError`, 5

`product_id()` (*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 7

`product_id()` (*StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI.D*
method), 20

`product_id()` (*StreamDeck.Transport.Transport.Transport.Device*
method), 18

R

`read()` (*StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI.Device*
method), 20

`read()` (*StreamDeck.Transport.Transport.Transport.Device*
method), 18

`read_feature()` (*StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI*
method), 20

`read_feature()` (*StreamDeck.Transport.Transport.Transport.Device*
method), 18

in `reset()` (*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 8

`reset()` (*StreamDeck.Devices.StreamDeckMini.StreamDeckMini*
method), 13

in `reset()` (*StreamDeck.Devices.StreamDeckOriginal.StreamDeckOriginal*
method), 11

`reset()` (*StreamDeck.Devices.StreamDeckPedal.StreamDeckPedal*
method), 15

`reset()` (*StreamDeck.Devices.StreamDeckPlus.StreamDeckPlus*
method), 16

`reset()` (*StreamDeck.Devices.StreamDeckXL.StreamDeckXL*
method), 14

S

`set_brightness()` (*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 8

`set_brightness()` (*StreamDeck.Devices.StreamDeckMini.StreamDeckMi*
method), 13

`set_brightness()` (*StreamDeck.Devices.StreamDeckOriginal.StreamDe*
method), 12

`set_brightness()` (*StreamDeck.Devices.StreamDeckPedal.StreamDeckP*
method), 15

`set_brightness()` (*StreamDeck.Devices.StreamDeckPlus.StreamDeckPlu*
method), 16

`set_brightness()` (*StreamDeck.Devices.StreamDeckXL.StreamDeckXL*
method), 14

`set_dial_callback()`
(*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 8

`set_dial_callback_async()`
(*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 8

`set_key_callback()` (*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 8

`set_key_callback_async()`
(*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 9

`set_key_image()` (*StreamDeck.Devices.StreamDeck.StreamDeck*
method), 9

set_key_image() (StreamDeck.Devices.StreamDeckMini.StreamDeckMini
 method), 13
 set_key_image() (StreamDeck.Devices.StreamDeckOriginal.StreamDeckOriginal
 method), 12
 set_key_image() (StreamDeck.Devices.StreamDeckPedal.StreamDeckPedal), 12
 method), 15
 set_key_image() (StreamDeck.Devices.StreamDeckPlus.StreamDeckPlus), 11
 method), 16
 set_key_image() (StreamDeck.Devices.StreamDeckXL.StreamDeckXL), 14
 method), 14
 set_poll_frequency()
 (StreamDeck.Devices.StreamDeck.StreamDeck
 method), 9
 set_touchscreen_callback()
 (StreamDeck.Devices.StreamDeck.StreamDeck
 method), 10
 set_touchscreen_callback_async()
 (StreamDeck.Devices.StreamDeck.StreamDeck
 method), 10
 set_touchscreen_image()
 (StreamDeck.Devices.StreamDeck.StreamDeck
 method), 10
 set_touchscreen_image()
 (StreamDeck.Devices.StreamDeckMini.StreamDeckMini
 method), 13
 set_touchscreen_image()
 (StreamDeck.Devices.StreamDeckOriginal.StreamDeckOriginal
 method), 12
 set_touchscreen_image()
 (StreamDeck.Devices.StreamDeckPedal.StreamDeckPedal
 method), 15
 set_touchscreen_image()
 (StreamDeck.Devices.StreamDeckPlus.StreamDeckPlus
 method), 17
 set_touchscreen_image()
 (StreamDeck.Devices.StreamDeckXL.StreamDeckXL
 method), 14
 StreamDeck (class in StreamDeck.Devices.StreamDeck),
 5
 StreamDeck.DeviceManager
 module, 4
 StreamDeck.Devices.StreamDeck
 module, 5
 StreamDeck.Devices.StreamDeckMini
 module, 12
 StreamDeck.Devices.StreamDeckOriginal
 module, 11
 StreamDeck.Devices.StreamDeckPedal
 module, 15
 StreamDeck.Devices.StreamDeckPlus
 module, 16
 StreamDeck.Devices.StreamDeckXL
 module, 14
 StreamDeck.Transport.LibUSBHIDAPI

T
 touchscreen_image_format()
 (StreamDeck.Devices.StreamDeck.StreamDeck
 method), 11
 TouchscreenEventType (class in
 StreamDeck.Devices.StreamDeck), 11
 Transport (class in StreamDeck.Transport.Transport),
 17
 Transport.Device (class in
 StreamDeck.Transport.Transport), 17
 TransportError, 19

V
 vendor_id() (StreamDeck.Devices.StreamDeck.StreamDeck
 method), 11
 vendor_id() (StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI.Device
 method), 21
 vendor_id() (StreamDeck.Transport.Transport.Transport.Device
 method), 18

W
 write() (StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI.Device
 method), 21
 write() (StreamDeck.Transport.Transport.Transport.Device
 method), 18
 write_feature() (StreamDeck.Transport.LibUSBHIDAPI.LibUSBHIDAPI.Device
 method), 21
 write_feature() (StreamDeck.Transport.Transport.Transport.Device
 method), 19