StreamDeck Package Documentation

Release 0.6.1

Dean Camera

Apr 11, 2020

CONTENTS

ExampleDeck.jpg

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.

PyPi Project Entry

Online Documentation

CHAPTER

ONE

STATUS:

Working - you can enumerate devices, set the brightness of the panel(s), set the images shown on each button, and read the current button states.

Currently the following StreamDeck product variants are supported:

- StreamDeck Original
- StreamDeck Mini
- StreamDeck XL

CHAPTER

TWO

PACKAGE DEPENDENCIES:

The library core has no special dependencies, but does use one of (potentially) several HID backend libraries. You will need to install the dependencies appropriate to your chosen backend.

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.

2.1 HID Backend

The recommended HID backend is the aptly named HID Python library, which should work across the three major (Windows, Mac, Linux) OSes and is the most up-to-date. This can be installed via `pip3 install hid`.

Note that Windows systems requires additional manual installation of a DLL in order to function. The latest source for this DLL is the libUSB GitHub project.

Despite the additional setup, this will give the best results in terms of reliability and performance.

2.2 HIDAPI Backend

Another option is the older HIDAPI Python library, which originates from the same original project as the HID library listed above, but is now entirely unmaintained. This can be installed via `pip3 install hidapi`.

Several users report issues with this backend due to various bugs that have not been patched in the packaged library version, but support for it is included in this library due to its simple one-line setup on all three major platforms.

Note the library package name conflicts with the HID backend above; only one or the other should be installed at the same time.

CHAPTER

THREE

PACKAGE INSTALLATION:

Install the library via pip:

```
pip install streamdeck
```

Alternatively, manually clone the project repository:

```
git clone https://github.com/abcminiuser/python-elgato-streamdeck.git
```

CHAPTER

FOUR

RASPBERRY PI INSTALLATION:

The following script has been verified working on a Raspberry Pi (Model 2 B) running a stock Debian Stretch image, to install all the required dependencies needed by this project:

```
# 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

# Install system packages needed for the Python hidapi package installation
sudo apt install -y libudev-dev libusb-1.0-0-dev

# Install dependencies
pip3 install hidapi
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"
    EOF

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

# Alternatively, install git and check out the repository
#sudo apt install -y git
#git clone https://github.com/abcminiuser/python-elgato-streamdeck.git
```

Note that after adding the `udev` rule, a restart will be required in order for it to take effect and allow access to the StreamDeck device without requiring root privileges.

CHAPTER FIVE

CREDITS:

I've used the reverse engineering notes from this GitHub repository to implement this library. Thanks Alex Van Camp!

Thank you to the following contributors, large and small, for helping with the development and maintenance of this library:

- Aetherdyne
- dirkk0
- Kalle-Wirsch
- pointshader
- spidererrol
- Subsentient
- shanna

If you've contributed in some manner, but I've accidentally missed you in the list above, please let me know.

CHAPTER SIX

LICENSE:

Released under the MIT license:

6.1 Modules: Device Discovery

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

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`**

6.2 Modules: StreamDeck Devices

class `StreamDeck.Devices.StreamDeck.`**`StreamDeck`**`(`*device*`)`
Represents a physically attached original 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 Text corresponding to the specific type of the device.

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.

`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.

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_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** (*function*) – Asyncio loop to dispatch the callback into

abstract set_key_image(*key*, *image*)

Sets the image of a button on the StremDeck 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 `get_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.

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

Bases: *StreamDeck.Devices.StreamDeck.StreamDeck*

Represents a physically attached original StreamDeck 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*.

set_key_image(*key*, *image*)

Sets the image of a button on the StremDeck 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 `get_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.

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

Bases: *StreamDeck.Devices.StreamDeck.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 `get_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.

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

Bases: *StreamDeck.Devices.StreamDeck.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*.

set_key_image(*key*, *image*)
Sets the image of a button on the StremDeck 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 `get_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.

6.3 Modules: Communication Transports

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

class Device
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 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 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()

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 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 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.

class StreamDeck.Transport.HID.**HID**

Bases: *StreamDeck.Transport.Transport.Transport*

USB HID transport layer, using the *hid* Python wrapper. This transport can be used to enumerate and communicate with HID devices.

class Device(*device_info*)

Bases: `StreamDeck.Transport.Transport.Device`

Creates a new HID device instance, used to send and receive HID reports from/to an attached USB HID device.

Parameters **device_info** (*dict()*) – Device information dictionary describing a single unique attached USB HID device.

close()
Closes theHID device for input/output.

See also:

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

connected()
Indicates if the physical HID device this instance is attached to is still connected to the host.
Return type bool
Returns *True* if the device is still connected, *False* otherwise.

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

See also:

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

path()
Retrieves the logical path of the attached HID device within the current system. This can be used to differentiate one HID device from another.
Return type str
Returns Logical device path for 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.

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 USB HID devices on the system.

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(*HID.Device*)

Returns List of discovered USB HID devices.

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.

class StreamDeck.Transport.HIDAPI.**HIDAPI**

Bases: `StreamDeck.Transport.Transport.Transport`

USB HID transport layer, using the *hidapi* Python wrapper. This transport can be used to enumerate and communicate with HID devices.

class Device(*device_info*)

Bases: `StreamDeck.Transport.Transport.Device`

Creates a new HIDAPI device instance, used to send and receive HID reports from/to an attached USB HID device.

Parameters **device_info** (*dict()*) – Device information dictionary describing a single unique attached USB HID device.

close()

Closes theHID device for input/output.

See also:

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

connected()

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

Return type bool

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

open()

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

See also:

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

path()

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

Return type str

Returns Logical device path for 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.

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 USB HID devices on the system.

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(*HIDAPI.Device*)

Returns List of discovered USB HID devices.

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.

6.4 Modules: Image Helpers

StreamDeck.ImageHelpers.PILHelper.**create_image**(*deck*, *background='black'*)

Creates a new PIL Image with the correct image dimensions for the given StreamDeck device's keys.

See also:

See `to_native_format()` method for converting a PIL image instance to the native image format of a given StreamDeck device.

Parameters

- **deck** (`StreamDeck`) – StreamDeck device to generate a compatible image for.
- **background** (*str*) – Background color to use, compatible with *PIL.Image.new()*.

Return type PIL.Image

Returns Created PIL image

`StreamDeck.ImageHelpers.PILHelper.`**`to_native_format`**(*deck*, *image*)

Converts a given PIL image to the native image format for a StreamDeck, suitable for passing to `set_key_image()`.

See also:

See `create_image()` method for creating a PIL image instance for a given StreamDeck device.

Parameters

- **deck** (`StreamDeck`) – StreamDeck device to generate a compatible native image for.
- **image** (*PIL.Image*) – PIL Image to convert to the native StreamDeck image format

Return type enumerable()

Returns Image converted to the given StreamDeck's native format

6.5 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):
    image_format = deck.key_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()))
```

(continues on next page)

(continued from previous page)

```
    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: {} ({}x{} grid)".format(
        deck.key_count(),
        deck.key_layout()[0],
        deck.key_layout()[1]))
    print("\t - Key Images: {}x{} pixels, {} format, rotated {} degrees, {}".format(
        image_format['size'][0],
        image_format['size'][1],
        image_format['format'],
        image_format['rotation'],
        flip_description[image_format['flip']]))


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()
```

6.6 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.
ASSETS_PATH = os.path.join(os.path.dirname(__file__), "Assets")
```

(continues on next page)

(continued from previous page)

```python
# 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):
    # Create new key image of the correct dimensions, black background.
    image = PILHelper.create_image(deck)

    # Resize the source image asset to best-fit the dimensions of a single key,
    # and paste it onto our blank frame centered as closely as possible.
    icon = Image.open(icon_filename).convert("RGBA")
    icon.thumbnail((image.width, image.height - 20), Image.LANCZOS)
    icon_pos = ((image.width - icon.width) // 2, 0)
    image.paste(icon, icon_pos, icon)

    # Load a custom TrueType font and use it to overlay the key index, draw key
    # label onto the image.
    draw = ImageDraw.Draw(image)
    font = ImageFont.truetype(font_filename, 14)
    label_w, label_h = draw.textsize(label_text, font=font)
    label_pos = ((image.width - label_w) // 2, image.height - 20)
    draw.text(label_pos, text=label_text, font=font, fill="white")

    return PILHelper.to_native_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)

    # Generate the custom key with the requested image and label.
```

(continues on next page)

(continued from previous page)

```
    image = render_key_image(deck, key_style["icon"], key_style["font"], key_style[
↪"label"])

    # Update requested key with the generated image.
    deck.set_key_image(key, image)


# Prints key state change information, updates rhe 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":
            # 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):
        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)

        # 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():
            if t is threading.currentThread():
                continue
```

(continues on next page)

(continued from previous page)

```
            if t.is_alive():
                t.join()
```

6.7 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
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, image_filename):
    key_width, key_height = deck.key_image_format()['size']
    key_rows, key_cols = deck.key_layout()

    full_deck_image_size = (key_width * key_cols, key_height * key_rows)

    # Resize the image to suit the StreamDeck's full image size (note: will not
    # preserve the correct aspect ratio).
    image = Image.open(os.path.join(ASSETS_PATH, image_filename)).convert("RGBA")
    return image.resize(full_deck_image_size, Image.LANCZOS)


# 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):
    key_width, key_height = deck.key_image_format()['size']
    key_rows, key_cols = deck.key_layout()

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

(continues on next page)

(continued from previous page)

```
    col = key % key_cols

    # Compute the region of the larger deck image that is occupied by the given
    # key.
    region = (col * key_width, row * key_height, (col + 1) * key_width, (row + 1) *␣
↪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_image(deck)
    key_image.paste(segment)

    return PILHelper.to_native_format(deck, key_image)


# Closes the StreamDeck device on key state change.
def key_change_callback(deck, key, state):
    # 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):
        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)

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

        for k in range(deck.key_count()):
            # Extract out the section of the image that is occupied by the
            # current key.
            key_image = crop_key_image_from_deck_sized_image(deck, image, k)

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

(continues on next page)

(continued from previous page)

```
        for t in threading.enumerate():
            if t is threading.currentThread():
                continue

            if t.is_alive():
                t.join()
```

6.8 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 os
import itertools
import threading
from PIL import Image, ImageSequence
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")

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


# Loads in a source image, extracts out the individual animation frames (if
# any) and returns an infinite generator that returns the next animation frame,
# 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):
        # We need source frames in RGBA format, convert now before we resize it.
        icon_frame = frame.convert("RGBA")
```

(continues on next page)

(continued from previous page)

```
        # Create new key image of the correct dimensions, black background.
        image = PILHelper.create_image(deck)

        # Resize the animation frame to best-fit the dimensions of a single key,
        # and paste it onto our blank frame centered as closely as possible.
        icon_frame.thumbnail(image.size, Image.LANCZOS)
        icon_frame_pos = ((image.width - icon_frame.width) // 2, (image.height - icon_
↪frame.height) // 2)
        image.paste(icon_frame, icon_frame_pos, icon_frame)

        # Store the rendered animation frame in the device's native image
        # format for later use, so we don't need to keep converting it.
        icon_frames.append(PILHelper.to_native_format(deck, image))

    # Return an infinite cycle generator that returns the next animation frame
    # each time it is called.
    return itertools.cycle(icon_frames)


# Closes the StreamDeck device on key state change.
def key_change_callback(deck, key, state):
    # 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):
        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)

        # 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()
```

(continues on next page)

(continued from previous page)

```
        for k in range(deck.key_count()):
            key_images[k] = animations[k % len(animations)]

        # Helper function that is run periodically, to update the images on
        # each key.
        def update_frames():
            try:
                # Update the key images with the next animation frame.
                for key, frames in key_images.items():
                    deck.set_key_image(key, next(frames))

                # Schedule the next periodic animation frame update.
                threading.Timer(1.0 / FRAMES_PER_SECOND, update_frames).start()
            except (IOError, ValueError):
                # Something went wrong (deck closed?) - don't re-schedule the
                # next animation frame.
                pass

        # Kick off the first animation update, which will also schedule the
        # next animation frame.
        update_frames()

        # 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():
            if t is threading.currentThread():
                continue

            if t.is_alive():
                t.join()
```

6.9 Library Changelog

```
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 autoprobing 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.
```

(continues on next page)

(continued from previous page)

```
        - 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.
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX