PyWunderCam

Athanasios Anastasiou

Sep 19, 2020

CONTENTS:

PyWunderCam is a Python module that enables control of the Wunder 360 S1 panoramic camera from a Python program.

The camera is based on a Rockchip 1108 System on Chip (SoC) running Linux. Within its operating system, it raises three services to serve images and video, control the camera and stream video over a WiFi interface. In addition to these services, the camera presents itself as a standard webcam if connected via USB.

At the time of writing, PyWunderCam interfaces with the first two services and enables functionality that is not possible via the provided mobile phone application.

Streaming video and extended functionality are scheduled for upcoming releases.

CHAPTER

ONE

QUICKSTART

```
from pywundercam import PyWunderCamAuto

camera_control = PyWunderCamAuto()
```

1.1 Single 360 Shot

```
single_photo = camera_control.single_shot()
```

1.2 Continuous (Burst) 360 Shot

```
photos = camera_control.continuous_shot()
```

1.3 Altering the ISO, White Balance and Exposure Compensation Modes

Both of the above functions (`.single_shot()`, `.continuous_shot()`) take optional parameters `iso`, `white_balance_mode` and `exposure_compensation`. For more information on the values of camera state parameters, please see *pywundercam.CamState*.

1.4 Storing Images

To store the result of a single shot:

```
single_photo[0].save_to("MyImage.jpg")
```

To store the result of a continuous (burst) shot:

```
photos[0].save_to("./")
```

Note:

1. In the case of a single image, all that is required is a filename. When saving the result of a continuous (burst) shot, all that is required is a directory in which all files belonging to the same shot will be stored.
2. Depending on file sizes and number of shots (in continuous mode), the file transfers might appear to be inserting a small delay in the whole process.

1.5 Displaying images

`pywundercam` makes use of the `pillow` module and returns images that are ready to be forwarded to Python's ritch ecosystem of image processing modules. A quick way of displaying the image is to use `matplotlib`.

Continuing from the above example, to display the result of a single shot:

```
from matplotlib import pyplot as plt

plt.imshow(single_photo[0].get())
plt.axis("off")
plt.show()
```

And in the case of a continuous (burst) shot, a specific picture out of the set would have to be chosen first:

```
plt.imshow(photos[0].get())
plt.axis("off")
plt.show()
```

This concludes the quickstart guide which makes use of `PyWunderCamAuto`. Although this client allows you to take pictures in single and continuous (burst) modes, the real power of `pywundercam` is in the underlying client object `PyWunderCam` that allows much more fine control over the complete parameter set of the Wunder 360 S1.

You could now browse over to the rest of the documentation sections to learn more about exceptions, hardware, and software design around PyWunderCam.

1.5.1 Basic Concepts

State

PyWunderCam's design is based around the concept of managing **state**. The camera is set to a particular *state* and when *triggered* in that *state* it will carry out an **action** that will likely modify its **state**.

Camera State

Suppose that the camera powers up in *single photo mode* with a fresh SD card and indicates that it has *capacity* for approximately 2000 pictures. In this simple example, **single photo mode** and **capacity** determine the **camera state**. After triggering the camera to take a snapshot, it will have remained in the same mode but its capacity will now have been decreased by 1 frame.

On the Wunder 360 S1, the camera state is defined by 31 parameters such as battery condition, ISO, White Balance and others.

Resource State

This concept of **state** is extended to the camera's file system too.

Continuing with the above example, when the camera powers up, it contains a set of **file resources** such as images and videos. After taking a snapshot (*triggering an action*), more images and videos will be added to its set of files.

Here, the state of the file system is directly equivalent to the files it holds. In general, the structure of the file system would have to be taken into account as well, but the file organisation on cameras is very simple and usually static. On the Wunder 360 S1 for example, there is a top level directory `DCIM/` that is further split into `Images/` and `Videos/` and this file organisation doesn't change (at least by the camera's onboard logic).

Wunder 360 S1 - Hardware

This is a very brief section on the camera's hardware and will be expanded with more details in upcoming versions.

The camera is based on a Rockchip 1108 SoC and boots into Linux. Once the system is up and running, it starts at least 3 processes:

1. A secure shell (SSH) server.
2. A web server (NGINX).
3. A Real Time Streaming Protocol / Real Data Transfer (RTSP / RDT) server.
4. A common "USB camera" service that provides streaming preview.

On the default image that the product was shipped with, the credentials to login to the SSH server are:

```
Username: root
Password: root
```

NGINX starts typically on port 80 and serves part of the camera's filespace as:

```
/
    DCIM/
        Images/
        Videos/
```

Note: PyWunderCam's File I/O is coordinated entirely over this interface.

NGINX also hosts a small Common Gateway Interface (CGI) binary that enables control of the camera over HTTP. The script is called `fcgi_client.cgi` and accepts control commands and returns responses from the camera hardware. It is interesting to note here the bizarre choice of having HTTP GET verbs used for *setting state* for **some** variables but using HTTP POST for others.

The RTSP / RDT server interface would imply that the camera could be used as an IP Camera. However, it is not entirely clear if this service is active throughout the time the camera is on, or it is started as a response to a command.

The common USB webcam interface does enable the Wunder 360 S1 to be connected to a PC (or other device) but it does not "advertise" any parameters other than "brightness". The preview received over USB however **is** responsive to any state changes that might be applied over the WiFi control connection.

1.5.2 Advanced Use

The Quickstart guide made use of a special class called `PyWunderCamAuto`. That class provides two functions (`.single_shot()`, `.continuous_shot()`) that hide the details of communicating with the camera.

`PyWunderCamAuto` is based on the `PyWunderCam` class and that class is allowing much finer control over the functionality of the camera.

The objective of this section is to outline the use of `PyWunderCam` and the way communication is carried out between a client and the camera.

Directing the camera to take pictures / video

The most usual cycle of interaction with the hardware is as follows:

0. Ensure that the computer has joined the network advertised by the camera.
1. Connect to the camera via the `PyWunderCam` client.
2. Get the current camera state.
3. Ensure that it is in the desired... state, depending on what is to be carried out.
4. Trigger the action (e.g. take a snapshot, burst, video, etc)

So, to take a single 360 picture:

```
from pywundercam import PyWunderCam

camera_client = PyWunderCam("192.168.100.1")
camera_state = camera_client.state
# Ensure that the camera is in single picture mode
camera_state.shoot_mode = 0
# Set the camera to the desired state
camera_client.state = camera_state
# Trigger the camera
camera_client.trigger()
```

Once `trigger()` is called, the camera will sound a beep and it will have taken a picture and stored it in its internal SD card.

To take a set of pictures in burst mode, change the `shoot_mode` to `3` and to start (and stop) taking a video, set the `shoot_mode` to `1`.

Retrieving the captured images / videos

This is achieved with a few additional steps that are conceptually very similar to directing the camera to take pictures:

0. Ensure that the computer has joined the network advertised by the camera.
1. Connect to the camera via the `PyWunderCam` client.
2. Get the camera state.
3. *Get the file system state* (**BEFORE** the action)
4. Ensure that camera state is in the desired... state.
5. Trigger the action (e.g. take a snapshot, burst, video, etc)
6. *Get the file system state* (**AFTER** the action)
7. *Retrieve the difference AFTER - BEFORE, to determine which files were generated and download them.*

So, to take a 360 picture and retrieve the image data:

```
from pywundercam import PyWunderCam

camera_client = PyWunderCam("192.168.100.1")
camera_state = camera_client.state
# Ensure that the camera is in single picture mode
camera_state.shoot_mode = 0
# Set the camera to the desired state
camera_client.state = camera_state
# Get a "snapshot" of its file contents BEFORE the shot
contents_before = camera_client.get_resources()
# Trigger the camera
camera_client.trigger()
# Get a "snapshot" of the file contents AFTER the shot
contents_after = camera_client.get_resources()
# Create a new snapshot that only contains the image that was acquired by this action
latest_image = contents_after["images"] - contents_before["images"]
```

Without getting into a lot of detais at this point, `latest_image` will contain only one image. To retrieve it:

```
shot_image = latest_image[0].get()
```

The `get()` function will trigger a file transfer from the camera to the computer over WiFi and depending on the size of the file, it will introduce a small pause.

At the time of writing, `PyWunderCam` serves images as `PIL.Image` objects. Therefore, after `get()`, the image is already in a form that can be passed to other algorithms (e.g. machine vision) for further processing.

Trying to retrieve a video in the same way will **raise an exception**.

Videos can still be transferred over WiFi and stored to the local computer for further processing.

To do this:

```
from pywundercam import PyWunderCam

camera_client = PyWunderCam("192.168.100.1")
camera_state = camera_client.state
# Ensure that the camera is in video mode
camera_state.shoot_mode = 1
# Set the camera to the desired state
camera_client.state = camera_state
# Get a "snapshot" of its file contents BEFORE the shot
contents_before = camera_client.get_resources()
# Trigger the camera
camera_client.trigger()
# Get a "snapshot" of the file contents AFTER the shot
contents_after = camera_client.get_resources()
# Create a new snapshot that only contains the video that was acquired by this action
latest_video = contents_after["videos"] - contents_before["videos"]
latest_video[0].save_to("myvideo.mp4")
```

1.5.3 Details

Camera Control

There are two classes that facilitate camera control:

- *pywundercam.PyWunderCam*

- *pywundercam.CamState*

All interaction with the camera (connecting to different services, exchanging data, etc) is handled by `PyWunderCam`, while state tracking and validation is handled by `CamState`.

`CamState` exposes all camera parameters as class properties and the getters and setters of those properties are doing the marshalling from and to the camera parameter names. To reduce the amount of communications with the camera, all the changes of individual parameters are scheduled in one batch rather than changed immediately.

When the state is obtained with something like:

```
wc = PyWunderCam()
camera_state = wc.camera_state
```

The camera state object is entering *edit mode*. In that mode, any changes to its attributes are simply recorded and put in a queue. If multiple changes are applied to an attribute, only the last change is preserved.

When the state is assigned back to the camera with:

```
wc.camera_state = camera_state
```

All queued camera state changes are applied to the camera and the responses collated to reflect the latest camera state.

Resource I/O

All interaction with the camera's file system occurs over a typical HTTP interface. The onboard camera control service does contain file system functionality (e.g. calls to list, retrieve, delete files) but since an HTTP interface is provided, all file related requests are handled via HTTP "verbs" such as `GET, POST, PUT, DELETE, HEAD`.

There are three classes that facilitate file I/O:

- *pywundercam.ResourceContainer*
- *pywundercam.SingleResource*
- *pywundercam.SequenceResource*

The design here is straightforward, a `ResourceContainer` contains zero or more `SingleResource` or `SequenceResource`. `SingleResource` represents a single file, `SequenceResource` represents mutliple (grouped) files, such as those that result from taking pictures in "Burst" mode. `SequenceResource` is in fact a list of `SingleResource` with a set of convenience functions.

When the file system state is captured with something like:

```
fs_state = wc.get_resources()
```

PyWunderCam returns a dictionary with two top level `ResourceContainer` s. One for images and one for videos.

A `ResourceContainer` is a representation of the camera's file **state** at the moment it was obtained. To discover new files that were created as the result of an action (e.g. taking one or more pictures or videos), simply subtract two `ResourceContainer` objects. Usually these are the **AFTER** the action and **BEFORE** the action was taken objects. This *temporal order* (*AFTER-BEFORE*) needs to be preserved, otherwise the difference is obtained but results to an empty set.

This means that a camera's actions can be scripted and a collection of the files that were generated be returned to the user in one block. For example, the user could obtain a baseline `ResourceContainer`, program the camera to obtain two single shot images and 2 bursts at different rates and finally get a `ResourceContainer` with all the files that where created by this "programmable" type of action.

Resource Metadata

`ResourceContainer` s are relying on regular expressions with named groups to extract metadata from the filename of a resource.

In the case of the Wunder360 S1, image and video filenames follow a simple naming pattern that encodes the type of file resource, the date and time of acquisition, the frame (if acquired in "Burst" mode) and the format of the resource in the extension.

The named regular expression rules used by the Wunder 360 S1 are already encoded in constants `pywundercam.IMG_FILE_RE` and `pywundercam.VID_FILE_RE`.

Metadata for a `SingleResource` are accessed via the `.metadata` read-only attribute.

Resource Transfers

File transfers are handled by `pywundercam.SingleResource.get()` and `pywundercam.SingleResrouce.save_to()`. Specifically, if an image is retrieved, it is served back as a `PIL.Image` object and can be readily used by a wide array of other Python packages. `save_to()` will simply transfer the file and save it locally.

The `SequenceResource` equivalent `save_to()` and `get()` functions have similar side-effects but automatically applied over image sequences.

1.5.4 API

This module implements an interface to a Wunder 360 S1 through Python.

authors Athanasios Anastasiou

date August 2019

Resource I/O

class pywundercam.**SingleResource**(*full_remote_filename*, *metadata=None*)
A single file resource held in the camera's disk space.

This can be a single image or video stored on the camera's SD card.

Note: If you are simply using this package to interface with the camera you do not usually need to instantiate this class directly.

---

property full_remote_filename
Returns the remote filename as this is found on the camera's file space.

get()
Retrieves a resource from the camera and serves it to the application in the right format.

> **Warning:** As of writing this line, anything other than image/jpeg will raise an exception. To save the resource locally, please see SingleResource.save_to().

property metadata
Returns metadata recovered from a resource's filename.

Note:

- This can be `None` if no named regular expression was passed during initialisation of *ResourceContainer*.

save_to(*filename=None*)
Saves a resource to a local path as a binary file.

Parameters **filename** (*str(path)*) – The local filename to save this file to. If `None`, the original name as found on the camera is used.

class pywundercam.**SequenceResource**(*file_io_uri*, *file_list*)
A sequence resource at the camera's memory space.

Sequence resources are produced by the "Continuous" (or "Burst") mode and are basically a set of images that were collected after a single "trigger" action. PyWunderCam will serve these as one resource if a filename regular expression and list of group-by attributes are provided.

Note: This is basically a tuple of SingleResource with a convenience get() to retrieve all resources of the sequence.

---

get()
Returns an array of `PIL.Image` images.

save_to(*directory=None*)
Saves an image sequence to a given directory (or the current one if none is provided).

Parameters **directory** (*str(path)*) – The directory to save the files to.

class pywundercam.**ResourceContainer**(*file_io_uri*, *file_re=None*, *group_by=None*, *order_by=None*)
A ResourceContainer represents a list of files that are accessed via an HTTP interface.

Usually, in devices like cameras, scanners, etc, the file name of an image, encodes a number of metadata, such as time, date, sequence number and others. A file_rule is used to parse these metadata. Based on the assumption that sequences of resources would share part of their filename characteristics, it is possible to group resources that are created as a result of a **single action**. If a "frame" attribute is provided as well, it is also possible to order these resources in the order they were taken.

Camera State

class pywundercam.**CamState**(*camera_data=None*)
Represents all data that capture the camera's state.

The class exposes properties with Pythonic names that are fully documented and ensures that the values that represent the camera's state are valid throughout their round trip to the hardware and back. This class also handles marshalling between the variable names used by the hardware and their Python counterparts.

property auto_shutdown
Minutes to auto-shutdown. Positive Integer. (Read only).

property battery_grid
Battery charge indicator in an arbitrary scale. Integer [0..6]. (Read only).

Note:

- The battery charge indicator on the camera is a 3 bar icon. This indicator goes all the way up to 6.

- The property name on the camera has been mispolled (BatterGird).

property charge_flag
Whether the camera's battery is charging. Bool [0..1]. (Read only).

property exposure_compensation
Exposure compensation in stops. Integer [0..13]. (Read / Write).

Note:

- For this setting to be effective the camera must be in Manual Mode (setting_mode=1).
- Exposure compensation spans a scale of 13 values (1..13) with 0 being the AUTO setting.

property firmware_software_version
Firmware and software version as returned by the camera. String. (Read only).

property hdmi_connect_flag
Whether the HDMI connector is plugged in. Bool [0..1]. (Read only).

Note:

- There is no HDMI connector exposed on the Wunder 360 S1.

property iso
Equivalent ISO in preset values. Integer [0..4]. (Read / Write).

Note:

- For this setting to be effective the camera must be in Manual Mode (setting_mode=1).
- **The preset values are as follows:**
  - 0: AUTO
  - 1: 100
  - 2: 200
  - 3: 400
  - 4: 800

property loop_video_time
Maximum video time in Loop mode in minutes. Integer. (Read only).

property mute
Whether to sound the camera's buzzer. Bool [0..1]. (Read only).

Note:

- This does not seem to be implemented on the Wunder 360 S1. Even if you manage to switch this flag to 0 the buzzer still sounds.

property operations
Returns a list of operations to be sent to the camera hardware so that its state reflects the requested state.

The list is of the format `(command, params)`, where `command` is usually an integer and *params* a dictionary of command specific parameters. `command` and `params` are hardware specific.

property product_model
Product model as returned by the camera. String. (Read only).

Note:

- This will always be "S1" on this camera.

property remain_num
Remaining number of pictures, given the capacity of the SD card. (Read only).

property remain_time
Remaining time for video recording, given the capacity of the SD card, in minutes. (Read only).

property sd_card_plug_flag
Whether an SD card is plugged in the camera and can be used. Integer [0..2]. (Read only).

Note:

- **Values are as follows:**
  - 0: No SD card plugged in
  - 1: SD card plugged in (not necessarily readable)
  - 2: SD card plugged in and readable.

property serial_number
Product serial number as returned by the camera. String. (Read only).

property setting_mode
Whether the camera is in manual or automatic mode. Bool [0..1]. (Read / Write).

Note:

- The photographic settings (iso, white_balance, exposure_compensation) require the camera to be in manual mode.

property shoot_mode
Determines which shoot mode to trigger. Integer [0..6]. (Read / Write).

Note:

- **The shoot modes are as follows:**
  - 0: Photo
  - 1: Video (3K)
  - 2: Timer
  - 3: Continuous (Burst)
  - 4: Time-Lapse
  - 5: Video (60 FPS)
  - 6: Loop
- To stop video recording, simply re-trigger the camera.

property white_balance_mode
White balance in color temperature presets. Integer [0..4]. (Read / Write).

Note:

- For this setting to be effective the camera must be in Manual Mode (setting_mode=1).
- **The temperature presets are as follows:**
  - 0: AUTO
  - 1: 2856K
  - 2: 4000K
  - 3: 5500K

- 4: 6500K

property wifi_pass
The camera's network password. (Read only).

Note:

- The default password is 12345678

property wifi_ssid
The SSID of the WiFi interface that the camera advertises. (Read only).

Note:

- By default, the SSID is Pano_[Camera-Serial-Number].

Camera Control

class pywundercam.**PyWunderCam**(*camera_ip='192.168.100.1'*)
The main client object that communicates with the various services exposed by the camera.

WunderCam handles all hardware requests and data transfers. At the very least, the camera exposes the following services:

1. An NGINX web server on `camera_ip:80`. (Known here as the "File I/O service")
2. A `fcgi_client.cgi` script that handles specific commands towards the hardware. (Known here as the "Control Service")
3. A Real Time Streaming Protocol (RTSP)/Real Data Transport (RDT) service to handle preview video streaming.

Currently, the WunderCam interfaces to the first two services and there are plans to be able to decode individual frames from a stream, at will, in the future.

property camera_ip
Returns the IP that the camera was initialised with.

property camera_state
Prepares and returns the camera state object to the user.

When the object enters the "prepare" state, any variable state changes are logged but **NOT** applied, until the user resets the state back to the camera. At that point, any changes to the state are unrolled, applied and their effect on the camera logged.

get_resources(*img_file_re=re.compile('Img_(?P<year>[0-9][0-9][0-9][0-9])(?P<month>[0-9][0-9])(?P<day>[0-9][0-9])_(?P<hour>[0-9][0-9])(?P<minute>[0-9][0-9])(?P<second>[0-9][0-9])_(?P<frame>[0-9][0-9][0-9])\\.jpg'), vid_file_re=re.compile('Vid_(?P<year>[0-9][0-9][0-9][0-9])(?P<month>[0-9][0-9])(?P<day>[0-9][0-9])_(?P<hour>[0-9][0-9])(?P<minute>[0-9][0-9])(?P<second>[0-9][0-9])_(?P<frame>[0-9][0-9][0-9])\\.mp4'), img_group_by=['year', 'month', 'day', 'hour', 'minute', 'second'], vid_group_by=['year', 'month', 'day', 'hour', 'minute', 'second'], img_order_by='frame'*)
Returns the two resource sets that reside on the camera's file space. One for images and one for videos.

Parameters

- **img_file_re** (*compiled regexp*) – Regular expression to unpack image filename metadata. By default set to the one the Wunder 360 S1 is using.

- **vid_file_re** (*compiled regexp*) – Similarly to `img_file_re` but for the video resource.
- **img_group_by** (*list of str*) – List of named attributes from `img_file_re` to use in distinguishing sequences from singles.
- **vid_group_by** (*list of str*) – Similarly to `img_group_by` but for video resources
- **img_order_by** (*str*) – Named attribute from the image filename regular expression to be used for ordering sequences.

trigger()
Triggers the camera to take an action given its current configuration.

. autoclass:: PyWunderCamAuto

members

Constants

pywundercam.**IMG_FILE_RE = re.compile('Img_(?P<year>[0-9][0-9][0-9][0-9])(?P<month>[0-9][0-**
Named regular expression for decoding metadata from the name of an image file. The named attributes of this regular expression are preserved along with a file resource as metadata.

pywundercam.**VID_FILE_RE = re.compile('Vid_(?P<year>[0-9][0-9][0-9][0-9])(?P<month>[0-9][0-**
Named regular expression for decoding metadata from the name of a video file. The named attributes of this regular expression are preserved along with a file resource as metadata.

pywundercam.**IMG_GROUP_BY = ['year', 'month', 'day', 'hour', 'minute', 'second']**
Attributes to group sets of images by. Images that are shot in quick succession (for example, in "Continuous" (or "Burst") mode). In that case, the images can be grouped by same values in these attributes.

pywundercam.**VID_GROUP_BY = ['year', 'month', 'day', 'hour', 'minute', 'second']**
Similar to `IMG_GROUP_BY` but for videos.

pywundercam.**IMG_ORDER_BY = 'frame'**
Attribute to order image sequences by. This refers to a field of the metadata regular expressions.

pywundercam.**VID_GROUP_BY = ['year', 'month', 'day', 'hour', 'minute', 'second']**
Similar to `IMG_GROUP_BY` but for videos.

Exceptions

This module describes the full exception hierarchy for all possible errors raised by PyWunderCam.

authors Athanasios Anastasiou

date September 2019

exception pywundercam.exceptions.**PyWunderCamCameraNotFoundError**
Raised when the camera hardware is unresponsive at the specified IP

exception pywundercam.exceptions.**PyWunderCamConnectionError**
Raised when a connection error is encountered during a data request to the camera.

exception pywundercam.exceptions.**PyWunderCamContentTypeError**
Raised when a resource's content type cannot be handled.

In the current version of PyWunderCam, this exception is thrown if the content-type of a file transfer is other than image/jpeg.

exception pywundercam.exceptions.**PyWunderCamDataTransferError**
Raised when a data transfer error occurs.

For example, in case the request was met with an HTTP404 error.

exception pywundercam.exceptions.**PyWunderCamError**
Base class for all exceptions originating from the pywundercam module.

exception pywundercam.exceptions.**PyWunderCamFileIOError**
Base class for errors raised during file transfer

exception pywundercam.exceptions.**PyWunderCamSDCardUnusable**
Raised when the SD card is either not plugged in or is not formatted.

exception pywundercam.exceptions.**PyWunderCamStateError**
Base class for camera state related errors.

exception pywundercam.exceptions.**PyWunderCamTimeOutError**
Raised when a data transfer time out occurs.

exception pywundercam.exceptions.**PyWunderCamValueError**
Raised when an attempt is made to set a state variable to an invalid value.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

T

V

W