Picamera Documentation

Release 1.3

Dave Hughes

March 24, 2014

This package provides a pure Python interface to the Raspberry Pi camera module for Python 2.7 (or above) and Python 3.2 (or above).

The code is licensed under the BSD license. Packages can be downloaded from the project homepage on PyPI. The source code can be obtained from GitHub, which also hosts the bug tracker. The documentation (which includes installation, quick-start examples, and lots of code recipes) can be read on ReadTheDocs.

Table of Contents

1.1 Python 2.7+ Installation

There are several ways to install picamera under Python 2.7 (or above), each with their own advantages and disadvantages. Have a read of the sections below and select an installation method which conforms to your needs.

1.1.1 Raspbian installation

If you are using the Raspbian distro, it is best to install picamera using the system's package manager: apt. This will ensure that picamera is easy to keep up to date, and easy to remove should you wish to do so. It will also make picamera available for all users on the system. To install picamera using apt simply:

```
$ sudo apt-get install python-picamera
```

To upgrade your installation when new releases are made you can simply use apt's normal upgrade procedure:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

If you ever need to remove your installation:

```
$ sudo apt-get remove python-picamera
```

1.1.2 User installation

This is the simplest (non-apt) form of installation, but bear in mind that it will only work for the user you install under. For example, if you install as the `pi` user, you will only be able to use picamera as the `pi` user. If you run python as root (e.g. with `sudo python`) it will not find the module. See *System installation* below if you require a root installation.

To install as your current user:

```
$ sudo apt-get install python-pip
$ pip install --user picamera
```

Note that `pip` is **not** run with sudo; this is deliberate. To upgrade your installation when new releases are made:

```
$ pip install --user -U picamera
```

If you ever need to remove your installation:

```
$ pip uninstall picamera
```

1.1.3 System installation

A system installation will make picamera accessible to all users (in contrast to the user installation). It is as simple to perform as the user installation and equally easy to keep updated. To perform the installation:

```
$ sudo apt-get install python-pip
$ sudo pip install picamera
```

To upgrade your installation when new releases are made:

```
$ sudo pip install -U picamera
```

If you ever need to remove your installation:

```
$ sudo pip uninstall picamera
```

1.1.4 Virtualenv installation

If you wish to install picamera within a virtualenv (useful if you're working on several Python projects with potentially conflicting dependencies, or you just like keeping things separate and easily removable):

```
$ sudo apt-get install python-pip python-virtualenv
$ virtualenv sandbox
$ source sandbox/bin/activate
(sandbox) $ pip install picamera
```

Bear in mind that each time you want to use picamera you will need to activate the virtualenv before running Python:

```
$ source sandbox/bin/activate
(sandbox) $ python
>>> import picamera
```

To upgrade your installation, make sure the virtualenv is activated and just use pip:

```
$ source sandbox/bin/activate
(sandbox) $ pip install -U picamera
```

To remove your installation simply blow away the virtualenv:

```
$ rm -fr ~/sandbox/
```

1.1.5 Development installation

If you wish to develop picamera itself, it is easiest to obtain the source by cloning the GitHub repository and then use the "develop" target of the Makefile which will install the package as a link to the cloned repository allowing in-place development (it also builds a tags file for use with vim/emacs with exuberant's ctags utility). The following example demonstrates this method within a virtual Python environment:

```
$ sudo apt-get install build-essential git git-core exuberant-ctags \
    python-virtualenv
$ virtualenv sandbox
$ source sandbox/bin/activate
(sandbox) $ git clone https://github.com/waveform80/picamera.git
(sandbox) $ cd picamera
(sandbox) $ make develop
```

To pull the latest changes from git into your clone and update your installation:

```
$ source sandbox/bin/activate
(sandbox) $ cd picamera
(sandbox) $ git pull
(sandbox) $ make develop
```

To remove your installation blow away the sandbox and the checkout:

```
$ rm -fr ~/sandbox/ ~/picamera/
```

For anybody wishing to hack on the project please understand that although it is technically written in pure Python, heavy use of `ctypes` is involved so the code really doesn't look much like Python - more a sort of horrid mish-mash of C and Python. The project currently consists of a class (`PiCamera`) which is a re-implementation of high-level bits of the `raspistill` and `raspivid` commands using the `ctypes` based `libmmal` header conversion, plus a set of (currently undocumented) encoder classes which re-implement the encoder callback configuration in the aforementioned binaries.

Even if you don't feel up to hacking on the code, I'd love to hear suggestions from people of what you'd like the API to look like (even if the code itself isn't particularly pythonic, the interface should be)!

1.2 Python 3.2+ Installation

There are several ways to install picamera under Python 3.2 (or above), each with their own advantages and disadvantages. Have a read of the sections below and select an installation method which conforms to your needs.

1.2.1 Raspbian installation

If you are using the Raspbian distro, it is best to install picamera using the system's package manager: apt. This will ensure that picamera is easy to keep up to date, and easy to remove should you wish to do so. It will also make picamera available for all users on the system. To install picamera using apt simply:

```
$ sudo apt-get install python3-picamera
```

To upgrade your installation when new releases are made you can simply use apt's normal upgrade procedure:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

If you ever need to remove your installation:

```
$ sudo apt-get remove python3-picamera
```

1.2.2 User installation

This is the simplest (non-apt) form of installation (though it's also complex to uninstall should you wish to do so later), but bear in mind that it will only work for the user you install under. For example, if you install as the `pi` user, you will only be able to use picamera as the `pi` user. If you run python as root (e.g. with `sudo python3`) it will not find the module. See *System installation* below if you require a root installation.

To install as your current user:

```
$ sudo apt-get install python3-setuptools
$ easy_install3 --user picamera
```

Note that `easy_install3` is **not** run with sudo; this is deliberate. To upgrade your installation when new releases are made:

```
$ easy_install3 --user -U picamera
```

If you ever need to remove your installation:

```
$ rm -fr ~/.local/lib/python3.*/site-packages/picamera-*
$ sed -i -e '/^\.\/picamera-/d' ~/.local/lib/python3.*/site-packages/easy-install.pth
```

Note: If the removal looks horribly complex, that's because it is! This is the reason Python devs tend to prefer virtualenvs. However, I suspect it's unlikely that most users will actually care about removing picamera - it's a tiny package and has no dependencies so leaving it lying around shouldn't cause any issues even if you don't use it anymore.

1.2.3 System installation

A system installation will make picamera accessible to all users (in contrast to the user installation). It is as simple to perform as the user installation and equally easy to keep updated but unfortunately, is also difficult to remove. To perform the installation:

```
$ sudo apt-get install python3-setuptools
$ sudo easy_install3 picamera
```

To upgrade your installation when new releases are made:

```
$ sudo easy_install3 -U picamera
```

If you ever need to remove your installation:

```
$ sudo rm -fr /usr/local/lib/python3.*/dist-packages/picamera-*
$ sudo sed -i -e '/^\.\/picamera-/d' /usr/local/lib/python3.*/dist-packages/easy-install.pth
```

Warning: Please be careful when running commands like `rm -fr` as root. With a simple slip (e.g. changing the final "-" to a space), such a command will very quickly delete a lot of things you probably don't want deleted (including most of your operating system if you're unlucky enough to be in the root directory). Double check what you've typed before hitting Enter!

1.2.4 Virtualenv installation

If you wish to install picamera within a virtualenv (useful if you're working on several Python projects with potentially conflicting dependencies, or you just like keeping things separate and easily removable):

```
$ sudo apt-get install python3-setuptools python-virtualenv
$ virtualenv -p python3 sandbox
$ source sandbox/bin/activate
(sandbox) $ easy_install picamera
```

Bear in mind that each time you want to use picamera you will need to activate the virtualenv before running Python:

```
$ source sandbox/bin/activate
(sandbox) $ python
>>> import picamera
```

To upgrade your installation, make sure the virtualenv is activated and just use easy_install:

```
$ source sandbox/bin/activate
(sandbox) $ easy_install -U picamera
```

To remove your installation simply blow away the virtualenv:

```
$ rm -fr ~/sandbox/
```

1.2.5 Development installation

If you wish to develop picamera itself, it is easiest to obtain the source by cloning the GitHub repository and then use the "develop" target of the Makefile which will install the package as a link to the cloned repository allowing in-place development (it also builds a tags file for use with vim/emacs with exuberant's ctags utility). The following example demonstrates this method within a virtual Python environment:

```
$ sudo apt-get install build-essential git git-core exuberant-ctags \
    python-virtualenv
$ virtualenv -p python3 sandbox
$ source sandbox/bin/activate
(sandbox) $ git clone https://github.com/waveform80/picamera.git
(sandbox) $ cd picamera
(sandbox) $ make develop
```

To pull the latest changes from git into your clone and update your installation:

```
$ source sandbox/bin/activate
(sandbox) $ cd picamera
(sandbox) $ git pull
(sandbox) $ make develop
```

To remove your installation blow away the sandbox and the checkout:

```
$ rm -fr ~/sandbox/ ~/picamera/
```

For anybody wishing to hack on the project please understand that although it is technically written in pure Python, heavy use of `ctypes` is involved so the code really doesn't look much like Python - more a sort of horrid mish-mash of C and Python. The project currently consists of a class (`PiCamera`) which is a re-implementation of high-level bits of the `raspistill` and `raspivid` commands using the `ctypes` based `libmmal` header conversion, plus a set of (currently undocumented) encoder classes which re-implement the encoder callback configuration in the aforementioned binaries.

Even if you don't feel up to hacking on the code, I'd love to hear suggestions from people of what you'd like the API to look like (even if the code itself isn't particularly pythonic, the interface should be)!

1.3 Quick Start

Start a preview for 10 seconds with the default settings:

```
import time
import picamera

camera = picamera.PiCamera()
try:
    camera.start_preview()
    time.sleep(10)
    camera.stop_preview()
finally:
    camera.close()
```

Note that you should always ensure you call `close()` on the PiCamera object to clean up resources.

The following example demonstrates that Python's `with` statement can be used to achieve this implicitly; when the `with` block ends, `close()` will be called implicitly:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.start_preview()
```

```
    time.sleep(10)
    camera.stop_preview()
```

The following example shows that certain properties can be adjusted "live" while a preview is running. In this case, the brightness is increased steadily during display:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.start_preview()
    try:
        for i in range(100):
            camera.brightness = i
            time.sleep(0.2)
    finally:
        camera.stop_preview()
```

The next example demonstrates setting the camera resolution (this can only be done when the camera is not recording) to 640x480, then starting a preview and a recording to a disk file:

```
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (640, 480)
    camera.start_preview()
    camera.start_recording('foo.h264')
    camera.wait_recording(60)
    camera.stop_recording()
    camera.stop_preview()
```

Note that `wait_recording()` is used above instead of `time.sleep()`. This method checks for errors (e.g. out of disk space) while the recording is running and raises an exception if one occurs. If `time.sleep()` was used instead the exception would be raised by `stop_recording()` but only after the full waiting time had run.

This example demonstrates starting a preview, setting some parameters and then capturing an image while the preview is running:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (1280, 720)
    camera.start_preview()
    camera.exposure_compensation = 2
    camera.exposure_mode = 'spotlight'
    camera.meter_mode = 'matrix'
    camera.image_effect = 'gpen'
    # Give the camera some time to adjust to conditions
    time.sleep(2)
    camera.capture('foo.jpg')
    camera.stop_preview()
```

The following example customizes the Exif tags to embed in the image before calling `capture()`:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (2592, 1944)
    camera.start_preview()
    time.sleep(2)
    camera.exif_tags['IFD0.Artist'] = 'Me!'
    camera.exif_tags['IFD0.Copyright'] = 'Copyright (c) 2013 Me!'
```

```
    camera.capture('foo.jpg')
    camera.stop_preview()
```

See the documentation for `exif_tags` for a complete list of the supported tags.

The next example demonstrates capturing a series of images as a numbered series with a one minute delay between each capture using the `capture_continuous()` method:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (1280, 720)
    camera.start_preview()
    time.sleep(1)
    for i, filename in enumerate(camera.capture_continuous('image{counter:02d}.jpg')):
        print('Captured image %s' % filename)
        if i == 100:
            break
        time.sleep(60)
    camera.stop_preview()
```

This example demonstrates capturing low resolution JPEGs extremely rapidly using the video-port capability of the `capture_sequence()` method. The framerate of the captures is displayed afterward:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (640, 480)
    camera.start_preview()
    start = time.time()
    camera.capture_sequence((
        'image%03d.jpg' % i
        for i in range(120)
        ), use_video_port=True)
    print('Captured 120 images at %.2ffps' % (120 / (time.time() - start)))
    camera.stop_preview()
```

This example demonstrates capturing an image in raw RGB format:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (1024, 768)
    camera.start_preview()
    time.sleep(2)
    camera.capture('image.data', 'rgb')
```

1.4 Basic Recipes

The following recipes should be reasonably accessible to Python programmers of all skill levels. Please feel free to suggest enhancements or additional recipes.

1.4.1 Capturing to a file

Capturing an image to a file is as simple as specifying the name of the file as the output of whatever capture() method you require:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (1024, 768)
    camera.start_preview()
    # Camera warm-up time
    time.sleep(2)
    camera.capture('foo.jpg')
```

Note that files opened by picamera (as in the case above) will be flushed and closed so that when the capture() method returns, the data should be accessible to other processes.

1.4.2 Capturing to a stream

Capturing an image to a file-like object (a `socket()`, a `io.BytesIO` stream, an existing open file object, etc.) is as simple as specifying that object as the output of whatever capture() method you're using:

```
import io
import time
import picamera

# Create an in-memory stream
my_stream = io.BytesIO()
with picamera.PiCamera() as camera:
    camera.start_preview()
    # Camera warm-up time
    time.sleep(2)
    camera.capture(my_stream, 'jpeg')
```

Note that the format is explicitly specified in the case above. The `BytesIO` object has no filename, so the camera can't automatically figure out what format to use.

One thing to bear in mind is that (unlike specifying a filename), the stream is *not* automatically closed after capture; picamera assumes that since it didn't open the stream it can't presume to close it either. In the case of file objects this can mean that the data doesn't actually get written to the disk until the object is explicitly closed:

```
import time
import picamera

# Explicitly open a new file called my_image.jpg
my_file = open('my_image.jpg', 'wb')
with picamera.PiCamera() as camera:
    camera.start_preview()
    time.sleep(2)
    camera.capture(my_file)
# Note that at this point the data is in the file cache, but may
# not actually have been written to disk yet
my_file.close()
# Now the file has been closed, other processes should be able to
# read the image successfully
```

Note that in the case above, we didn't have to specify the format as the camera interrogated the `my_file` object for its filename (specifically, it looks for a `name` attribute on the provided object).

1.4.3 Capturing to a PIL Image

This is a variation on *Capturing to a stream*. First we'll capture an image to a `BytesIO` stream (Python's in-memory stream class), then we'll rewind the position of the stream to the start, and read the stream into a PIL Image object:

```
import io
import time
import picamera
from PIL import Image

# Create the in-memory stream
stream = io.BytesIO()
with picamera.PiCamera() as camera:
    camera.start_preview()
    time.sleep(2)
    camera.capture(stream, format='jpeg')
# "Rewind" the stream to the beginning so we can read its content
stream.seek(0)
image = Image.open(stream)
```

1.4.4 Capturing to an OpenCV object

This is another variation on *Capturing to a stream*. First we'll capture an image to a `BytesIO` stream (Python's in-memory stream class), then convert the stream to a numpy array and read the array with OpenCV:

```
import io
import time
import picamera
import cv2
import numpy as np

# Create the in-memory stream
stream = io.BytesIO()
with picamera.PiCamera() as camera:
    camera.start_preview()
    time.sleep(2)
    camera.capture(stream, format='jpeg')
# Construct a numpy array from the stream
data = np.fromstring(stream.getvalue(), dtype=np.uint8)
# "Decode" the image from the array, preserving colour
image = cv2.imdecode(data, 1)
# OpenCV returns an array with data in BGR order. If you want RGB instead
# use the following...
image = image[:, :, ::-1]
```

1.4.5 Capturing resized images

Sometimes, particularly in scripts which will perform some sort of analysis or processing on images, you may wish to capture smaller images than the current resolution of the camera. Although such resizing can be performed using libraries like PIL or OpenCV, it is considerably more efficient to have the Pi's GPU perform the resizing when capturing the image. This can be done with the *resize* parameter of the `capture()` methods:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (1024, 768)
    camera.start_preview()
    # Camera warm-up time
    time.sleep(2)
    camera.capture('foo.jpg', resize=(320, 240))
```

The *resize* parameter can also be specified when recording video with the `start_recording()` method.

1.4.6 Capturing timelapse sequences

The simplest way to capture long time-lapse sequences is with the `capture_continuous()` method. With this method, the camera captures images continually until you tell it to stop. Images are automatically given unique names and you can easily control the delay between captures. The following example shows how to capture images with a 5 minute delay between each shot:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.start_preview()
    time.sleep(2)
    for filename in camera.capture_continuous('img{counter:03d}.jpg'):
        print('Captured %s' % filename)
        time.sleep(300) # wait 5 minutes
```

However, you may wish to capture images at a particular time, say at the start of every hour. This simply requires a refinement of the delay in the loop (the `datetime` module is slightly easier to use for calculating dates and times; this example also demonstrates the `timestamp` template in the captured filenames):

```
import time
import picamera
from datetime import datetime, timedelta

def wait():
    # Calculate the delay to the start of the next hour
    next_hour = (datetime.now() + timedelta(hour=1)).replace(
        minute=0, second=0, microsecond=0)
    delay = (next_hour - datetime.now()).seconds
    time.sleep(delay)

with picamera.PiCamera() as camera:
    camera.start_preview()
    wait()
    for filename in camera.capture_continuous('img{timestamp:%Y-%m-%d-%H-%M}.jpg'):
        print('Captured %s' % filename)
        wait()
```

1.4.7 Capturing to a network stream

This is a variation of *Capturing timelapse sequences*. Here we have two scripts: a server (presumably on a fast machine) which listens for a connection from the Raspberry Pi, and a client which runs on the Raspberry Pi and sends a continual stream of images to the server. Firstly the server script (which relies on PIL for reading JPEGs, but you could replace this with any other suitable graphics library, e.g. OpenCV or GraphicsMagick):

```
import io
import socket
import struct
from PIL import Image

# Start a socket listening for connections on 0.0.0.0:8000 (0.0.0.0 means
# all interfaces)
server_socket = socket.socket()
server_socket.bind(('0.0.0.0', 8000))
server_socket.listen(0)

# Accept a single connection and make a file-like object out of it
connection = server_socket.accept()[0].makefile('rb')
try:
    while True:
```

```
        # Read the length of the image as a 32-bit unsigned int. If the
        # length is zero, quit the loop
        image_len = struct.unpack('<L', connection.read(4))[0]
        if not image_len:
            break
        # Construct a stream to hold the image data and read the image
        # data from the connection
        image_stream = io.BytesIO()
        image_stream.write(connection.read(image_len))
        # Rewind the stream, open it as an image with PIL and do some
        # processing on it
        image_stream.seek(0)
        image = Image.open(image_stream)
        print('Image is %dx%d' % image.size)
        image.verify()
        print('Image is verified')
finally:
    connection.close()
    server_socket.close()
```

Now for the client side of things, on the Raspberry Pi:

```
import io
import socket
import struct
import time
import picamera

# Connect a client socket to my_server:8000 (change my_server to the
# hostname of your server)
client_socket = socket.socket()
client_socket.connect(('my_server', 8000))

# Make a file-like object out of the connection
connection = client_socket.makefile('wb')
try:
    with picamera.PiCamera() as camera:
        camera.resolution = (640, 480)
        # Start a preview and let the camera warm up for 2 seconds
        camera.start_preview()
        time.sleep(2)

        # Note the start time and construct a stream to hold image data
        # temporarily (we could write it directly to connection but in this
        # case we want to find out the size of each capture first to keep
        # our protocol simple)
        start = time.time()
        stream = io.BytesIO()
        for foo in camera.capture_continuous(stream, 'jpeg'):
            # Write the length of the capture to the stream and flush to
            # ensure it actually gets sent
            connection.write(struct.pack('<L', stream.tell()))
            connection.flush()
            # Rewind the stream and send the image data over the wire
            stream.seek(0)
            connection.write(stream.read())
            # If we've been capturing for more than 30 seconds, quit
            if time.time() - start > 30:
                break
            # Reset the stream for the next capture
            stream.seek(0)
            stream.truncate()
    # Write a length of zero to the stream to signal we're done
```

```
    connection.write(struct.pack('<L', 0))
finally:
    connection.close()
    client_socket.close()
```

The server script should be run first to ensure there's a listening socket ready to accept a connection from the client script.

1.4.8 Recording video to a file

Recording a video to a file is simple:

```
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (640, 480)
    camera.start_recording('my_video.h264')
    camera.wait_recording(60)
    camera.stop_recording()
```

Note that we use `wait_recording()` in the example above instead of `time.sleep()` which we've been using in the image capture recipes above. The `wait_recording()` method is similar in that it will pause for the number of seconds specified, but unlike `time.sleep()` it will continually check for recording errors (e.g. an out of disk space condition) while it is waiting. If we had used `time.sleep()` instead, such errors would only be raised by the `stop_recording()` call (which could be long after the error actually occurred).

1.4.9 Recording video to a stream

This is very similar to *Recording video to a file*:

```
import io
import picamera

stream = io.BytesIO()
with picamera.PiCamera() as camera:
    camera.resolution = (640, 480)
    camera.start_recording(stream, format='h264', quantization=23)
    camera.wait_recording(15)
    camera.stop_recording()
```

Here, we've set the *quantization* parameter which will cause the video encoder to use VBR (variable bit-rate) encoding. This can be considerably more efficient especially in mostly static scenes (which can be important when recording to memory, as in the example above). Quantization values (for the H.264 format) can be between 0 and 40, where 0 represents the highest possible quality, and 40 the lowest. Typically, a value in the range of 20-25 provides reasonable quality for reasonable bandwidth.

1.4.10 Recording over multiple files

If you wish split your recording over multiple files, you can use the `split_recording()` method to accomplish this:

```
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (640, 480)
    camera.start_recording('1.h264')
    camera.wait_recording(5)
    for i in range(2, 11):
        camera.split_recording('%d.h264' % i)
```

```
            camera.wait_recording(5)
        camera.stop_recording()
```

This should produce 10 video files named `1.h264`, `2.h264`, etc. each of which is approximately 5 seconds long (approximately because the `split_recording()` method will only split files at a key-frame).

The `record_sequence()` method can also be used to achieve this with slightly cleaner code:

```
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (640, 480)
    for filename in camera.record_sequence(
            '%d.h264' % i for i in range(1, 11)):
        camera.wait_recording(5)
```

New in version 0.8.

Changed in version 1.3: The `record_sequence()` method was introduced in version 1.3

1.4.11 Recording to a circular stream

This is similar to *Recording video to a stream* but uses a special kind of in-memory stream provided by the picamera library. The `PiCameraCircularIO` class implements a ring buffer based stream, specifically for video recording. This enables you to keep an in-memory stream containing the last *n* seconds of video recorded (where *n* is determined by the bitrate of the video recording and the size of the ring buffer underlying the stream).

A typical use-case for this sort of storage is security applications where one wishes to detect motion and only record to disk the video where motion was detected. This example keeps 20 seconds of video in memory until the `write_now` function returns `True` (in this implementation, this is random but one could, for example, replace this with some sort of motion detection algorithm). Once `write_now` returns `True`, the script waits 10 more seconds (so that the buffer contains 10 seconds of video from before the event, and 10 seconds after) and writes the resulting video to disk before going back to waiting:

```
import io
import random
import picamera

def write_now():
    # Randomly return True (like a fake motion detection routine)
    return random.randint(0, 10) == 0

def write_video(stream):
    print('Writing video!')
    with stream.lock:
        # Find the first header frame in the video
        for frame in stream.frames:
            if frame.header:
                stream.seek(frame.position)
                break
        # Write the rest of the stream to disk
        with io.open('motion.h264', 'wb') as output:
            output.write(stream.read())

with picamera.PiCamera() as camera:
    stream = picamera.PiCameraCircularIO(camera, seconds=20)
    camera.start_recording(stream, format='h264')
    try:
        while True:
            camera.wait_recording(1)
            if write_now():
                # Keep recording for 10 seconds and only then write the
```

```
                # stream to disk
                camera.wait_recording(10)
                write_video(stream)
    finally:
        camera.stop_recording()
```

In the above script we use the threading lock in the `lock` attribute to prevent the camera's background writing thread from changing the stream while our own thread reads from it (as the stream is a circular buffer, a write can remove information that is about to be read). If we had stopped recording while writing we could eliminate the `with stream.lock` line in the `write_video` function.

Note: Note that *at least* 20 seconds of video are in the stream. This is an estimate only; if the H.264 encoder requires less than the specified bitrate (17Mbps by default) for recording the video, then more than 20 seconds of video will be available in the stream.

New in version 1.0.

1.4.12 Recording to a network stream

This is similar to *Recording video to a stream* but instead of an in-memory stream like `BytesIO`, we will use a file-like object created from a `socket()`. Unlike the example in *Capturing to a network stream* we don't need to complicate our network protocol by writing things like the length of images. This time we're sending a continual stream of video frames (which necessarily incorporates such information, albeit in a much more efficient form), so we can simply dump the recording straight to the network socket.

Firstly, the server side script which will simply read the video stream and pipe it to a media player for display:

```
import socket
import subprocess

# Start a socket listening for connections on 0.0.0.0:8000 (0.0.0.0 means
# all interfaces)
server_socket = socket.socket()
server_socket.bind(('0.0.0.0', 8000))
server_socket.listen(0)

# Accept a single connection and make a file-like object out of it
connection = server_socket.accept()[0].makefile('rb')
try:
    # Run a viewer with an appropriate command line. Uncomment the mplayer
    # version if you would prefer to use mplayer instead of VLC
    cmdline = ['vlc', '--demux', 'h264', '-']
    #cmdline = ['mplayer', '-fps', '31', '-cache', '1024', '-']
    player = subprocess.Popen(cmdline, stdin=subprocess.PIPE)
    while True:
        # Repeatedly read 1k of data from the connection and write it to
        # the media player's stdin
        data = connection.read(1024)
        if not data:
            break
        player.stdin.write(data)
finally:
    connection.close()
    server_socket.close()
    player.terminate()
```

Note: If you run this script on Windows you will probably need to provide a complete path to the VLC or mplayer executable. If you run this script on Mac OS X, and are using Python installed from MacPorts, please ensure you have also installed VLC or mplayer from MacPorts.

Note: You will probably notice several seconds of latency with this setup. This is normal and is because media players buffer several seconds to guard against unreliable network streams.

Some media players (notably mplayer in this case) permit the user to skip to the end of the buffer (press the right cursor key in mplayer), reducing the latency by increasing the risk that delayed / dropped network packets will interrupt the playback.

Now for the client side script which simply starts a recording over a file-like object created from the network socket:

```
import socket
import time
import picamera

# Connect a client socket to my_server:8000 (change my_server to the
# hostname of your server)
client_socket = socket.socket()
client_socket.connect(('my_server', 8000))

# Make a file-like object out of the connection
connection = client_socket.makefile('wb')
try:
    with picamera.PiCamera() as camera:
        camera.resolution = (640, 480)
        # Start a preview and let the camera warm up for 2 seconds
        camera.start_preview()
        time.sleep(2)
        # Start recording, sending the output to the connection for 60
        # seconds, then stop
        camera.start_recording(connection, format='h264')
        camera.wait_recording(60)
        camera.stop_recording()
finally:
    connection.close()
    client_socket.close()
```

It should also be noted that the effect of the above is much more easily achieved (at least on Linux) with a combination of `netcat` and the `raspivid` executable. For example:

```
server-side: nc -l 8000 | vlc --demux h264 -
client-side: raspivid -w 640 -h 480 -t 60000 -o - | nc my_server 8000
```

However, this recipe does serve as a starting point for video streaming applications. For example, it shouldn't be terribly difficult to extend the recipe above to permit the server to control some aspects of the client's video stream.

1.4.13 Controlling the LED

In certain circumstances, you may find the camera module's red LED a hindrance. For example, in the case of automated close-up wild-life photography, the LED may scare off animals. It can also cause unwanted reflected red glare with close-up subjects.

One trivial way to deal with this is simply to place some opaque covering on the LED (e.g. blue-tack or electricians tape). However, provided you have the RPi.GPIO package installed, and provided your Python process is running with sufficient privileges (typically this means running as root with `sudo python`), you can also control the LED via the `led` attribute:

```
import picamera

with picamera.PiCamera() as camera:
    # Turn the camera's LED off
```

```
    camera.led = False
    # Take a picture while the LED remains off
    camera.capture('foo.jpg')
```

Warning: Be aware when you first use the LED property it will set the GPIO library to Broadcom (BCM) mode with `GPIO.setmode(GPIO.BCM)` and disable warnings with `GPIO.setwarnings(False)`. The LED cannot be controlled when the library is in BOARD mode.

1.5 Advanced Recipes

The following recipes involve advanced techniques and may not be "beginner friendly". Please feel free to suggest enhancements or additional recipes.

1.5.1 Unencoded image capture (YUV format)

If you want images captured without loss of detail (due to JPEG's lossy compression), you are probably better off exploring PNG as an alternate image format (PNG uses lossless compression). However, some applications (particularly scientific ones) simply require the image data in numeric form. For this, the `'yuv'` format is provided:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (100, 100)
    camera.start_preview()
    time.sleep(2)
    camera.capture('image.data', 'yuv')
```

The specific YUV format used is YUV420 (planar). This means that the Y (luminance) values occur first in the resulting data and have full resolution (one 1-byte Y value for each pixel in the image). The Y values are followed by the U (chrominance) values, and finally the V (chrominance) values. The UV values have one quarter the resolution of the Y components (4 1-byte Y values in a square for each 1-byte U and 1-byte V value).

It is also important to note that when outputting to unencoded formats, the camera rounds the requested resolution. The horizontal resolution is rounded up to the nearest multiple of 32 pixels, while the vertical resolution is rounded up to the nearest multiple of 16 pixels. For example, if the requested resolution is 100x100, the capture will actually contain 128x112 pixels worth of data, but pixels beyond 100x100 will be uninitialized.

Given that the YUV420 format contains 1.5 bytes worth of data for each pixel (a 1-byte Y value for each pixel, and 1-byte U and V values for every 4 pixels), and taking into account the resolution rounding, the size of a 100x100 YUV capture will be:

The first 14336 bytes of the data (128*112) will be Y values, the next 3584 bytes (128*112/4) will be U values, and the final 3584 bytes will be the V values.

The following code demonstrates capturing YUV image data, loading the data into a set of numpy arrays, and converting the data to RGB format in an efficient manner:

```
from __future__ import division

import time
import picamera
import numpy as np

width = 100
height = 100
stream = open('image.data', 'wb')
```

```
# Capture the image in YUV format
with picamera.PiCamera() as camera:
    camera.resolution = (width, height)
    camera.start_preview()
    time.sleep(2)
    camera.capture(stream, 'yuv')
# Rewind the stream for reading
stream.seek(0)
# Calculate the actual image size in the stream (accounting for rounding
# of the resolution)
fwidth = (width + 31) // 32 * 32
fheight = (height + 15) // 16 * 16
# Load the Y (luminance) data from the stream
Y = np.fromfile(stream, dtype=np.uint8, count=fwidth*fheight).\
        reshape((fheight, fwidth))
# Load the UV (chrominance) data from the stream, and double its size
U = np.fromfile(stream, dtype=np.uint8, count=(fwidth//2)*(fheight//2)).\
        reshape((fheight//2, fwidth//2)).\
        repeat(2, axis=0).repeat(2, axis=1)
V = np.fromfile(stream, dtype=np.uint8, count=(fwidth//2)*(fheight//2)).\
        reshape((fheight//2, fwidth//2)).\
        repeat(2, axis=0).repeat(2, axis=1)
# Stack the YUV channels together, crop the actual resolution, convert to
# floating point for later calculations, and apply the standard biases
YUV = np.dstack((Y, U, V))[:height, :width, :].astype(np.float)
YUV[:, :, 0]  = YUV[:, :, 0]  - 16   # Offset Y by 16
YUV[:, :, 1:] = YUV[:, :, 1:] - 128  # Offset UV by 128
# YUV conversion matrix from ITU-R BT.601 version (SDTV)
#              Y       U       V
M = np.array([[1.164,  0.000,  1.596],    # R
              [1.164, -0.392, -0.813],    # G
              [1.164,  2.017,  0.000]])   # B
# Take the dot product with the matrix to produce RGB output, clamp the
# results to byte range and convert to bytes
RGB = YUV.dot(M.T).clip(0, 255).astype(np.uint8)
```

Alternatively, see *Unencoded image capture (RGB format)* for a method of having the camera output RGB data directly.

Note: Capturing so-called "raw" formats (`'yuv'`, `'rgb'`, etc.) does not provide the raw bayer data from the camera's sensor. Rather, it provides access to the image data after GPU processing, but before format encoding (JPEG, PNG, etc). Currently, the only method of accessing the raw bayer data is via the *bayer* parameter to the `capture()` method.

Changed in version 1.0: The `raw_format` attribute is now deprecated, as is the `'raw'` format specification for the `capture()` method. Simply use the `'yuv'` format instead, as shown in the code above.

1.5.2 Unencoded image capture (RGB format)

The RGB format is rather larger than the YUV format discussed in the section above, but is more useful for most analyses. To have the camera produce output in RGB format, you simply need to specify `'rgb'` as the format for the `capture()` method instead:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (100, 100)
    camera.start_preview()
    time.sleep(2)
    camera.capture('image.data', 'rgb')
```

The size of RGB data can be calculated similarly to YUV captures. Firstly round the resolution appropriately (see *Unencoded image capture (YUV format)* for the specifics), then multiply the number of pixels by 3 (1 byte of red, 1 byte of green, and 1 byte of blue intensity). Hence, for a 100x100 capture, the amount of data produced is:

The resulting RGB data is interleaved. That is to say that the red, green and blue values for a given pixel are grouped together, in that order. The first byte of the data is the red value for the pixel at (0, 0), the second byte is the green value for the same pixel, and the third byte is the blue value for that pixel. The fourth byte is the red value for the pixel at (1, 0), and so on.

Loading the resulting RGB data into a numpy array is simple:

```
from __future__ import division

width = 100
height = 100
stream = open('image.data', 'wb')
# Capture the image in RGB format
with picamera.PiCamera() as camera:
    camera.resolution = (width, height)
    camera.start_preview()
    time.sleep(2)
    camera.capture(stream, 'rgb')
# Rewind the stream for reading
stream.seek(0)
# Calculate the actual image size in the stream (accounting for rounding
# of the resolution)
fwidth = (width + 31) // 32 * 32
fheight = (height + 15) // 16 * 16
# Load the data in a three-dimensional array and crop it to the requested
# resolution
image = np.fromfile(stream, dtype=uint8).\
        reshape((fheight, fwidth, 3))[:height, :width, :]
# If you wish, the following code will convert the image's bytes into
# floating point values in the range 0 to 1 (a typical format for some
# sorts of analysis)
image = image.astype(np.float, copy=False)
image = image / 255.0
```

Changed in version 1.0: The `raw_format` attribute is now deprecated, as is the `'raw'` format specification for the `capture()` method. Simply use the `'rgb'` format instead, as shown in the code above.

Warning: You may find RGB captures rather slow. If this is the case, please try the `'rgba'` format instead. The reason for this is that GPU component that picamera uses to perform RGB conversion doesn't support RGB output, only RGBA. As a result, RGBA data can be written directly, but picamera has to spend time stripping out the (unused) alpha byte from RGBA if RGB format is requested. A similar situation exists for the BGR and BGRA formats.

1.5.3 Rapid capture and processing

The camera is capable of capturing a sequence of images extremely rapidly by utilizing its video-capture capabilities with a JPEG encoder (via the `use_video_port` parameter). However, there are several things to note about using this technique:

- When using video-port based capture only the video recording area is captured; in some cases this may be smaller than the normal image capture area (see dicussion in *Camera Modes*).
- No Exif information is embedded in JPEG images captured through the video-port.

- Captures typically appear "granier" with this technique. The author is not aware of the exact technical reasons why this is so, but suspects that some part of the image processing pipeline that is present for still captures is not used when performing still captures through the video-port.

All capture methods support the `use_video_port` option, but the methods differ in their ability to rapidly capture sequential frames. So, whilst `capture()` and `capture_continuous()` both support `use_video_port`, `capture_sequence()` is by far the fastest method (because it does not re-initialize an encoder prior to each capture). Using this method, the author has managed 30fps JPEG captures at a resolution of 1024x768.

By default, `capture_sequence()` is particular suited to capturing a fixed number of frames rapidly, as in the following example which captures a "burst" of 5 images:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (1024, 768)
    camera.framerate = 30
    camera.start_preview()
    time.sleep(2)
    camera.capture_sequence([
        'image1.jpg',
        'image2.jpg',
        'image3.jpg',
        'image4.jpg',
        'image5.jpg',
        ])
```

We can refine this slightly by using a generator expression to provide the filenames for processing instead of specifying every single filename manually:

```
import time
import picamera

frames = 60

with picamera.PiCamera() as camera:
    camera.resolution = (1024, 768)
    camera.framerate = 30
    camera.start_preview()
    # Give the camera some warm-up time
    time.sleep(2)
    start = time.time()
    camera.capture_sequence([
        'image%02d.jpg' % i
        for i in range(frames)
        ], use_video_port=True)
    finish = time.time()
print('Captured %d frames at %.2ffps' % (
    frames,
    frames / (finish - start)))
```

However, this still doesn't let us capture an arbitrary number of frames until some condition is satisfied. To do this we need to use a generator function to provide the list of filenames (or more usefully, streams) to the `capture_sequence()` method:

```
import time
import picamera

frames = 60

def filenames():
    frame = 0
```

```
    while frame < frames:
        yield 'image%02d.jpg' % frame
        frame += 1

with picamera.PiCamera() as camera:
    camera.resolution = (1024, 768)
    camera.framerate = 30
    camera.start_preview()
    # Give the camera some warm-up time
    time.sleep(2)
    start = time.time()
    camera.capture_sequence(filenames(), use_video_port=True)
    finish = time.time()
print('Captured %d frames at %.2ffps' % (
    frames,
    frames / (finish - start)))
```

The major issue with capturing this rapidly is that the Raspberry Pi's IO bandwidth is extremely limited. As a format, JPEG is considerably less efficient than the H.264 video format (which is to say that, for the same number of bytes, H.264 will provide considerably better quality over the same number of frames).

At higher resolutions (beyond 800x600) you are likely to find you cannot sustain 30fps captures to the Pi's SD card for very long (before exhausting the disk cache). In other words, if you are intending to perform processing on the frames after capture, you may be better off just capturing video and decoding frames from the resulting file rather than dealing with individual JPEG captures.

However, if you can perform your processing fast enough, you may not need to involve the disk at all. Using a generator function, we can maintain a queue of objects to store the captures, and have parallel threads accept and process the streams as captures come in. Provided the processing runs at a faster frame rate than the captures, the encoder won't stall and nothing ever need hit the disk.

Please note that the following code involves some fairly advanced techniques (threading and all its associated locking fun is typically not a "beginner friendly" subject, not to mention generator expressions):

```
import io
import time
import threading
import picamera

# Create a pool of image processors
done = False
lock = threading.Lock()
pool = []

class ImageProcessor(threading.Thread):
    def __init__(self):
        super(ImageProcessor, self).__init__()
        self.stream = io.BytesIO()
        self.event = threading.Event()
        self.terminated = False
        self.start()

    def run(self):
        # This method runs in a separate thread
        global done
        while not self.terminated:
            if self.event.wait(1):
                try:
                    self.stream.seek(0)
                    # Read the image and do some processing on it
                    #Image.open(self.stream)
                    #...
                    #...
```

```
                    # Set done to True if you want the script to terminate
                    # at some point
                    #done=True
                finally:
                    # Reset the stream and event
                    self.stream.seek(0)
                    self.stream.truncate()
                    self.event.clear()
                    # Return ourselves to the pool
                    with lock:
                        pool.append(self)

def streams():
    while not done:
        with lock:
            processor = pool.pop()
        yield processor.stream
        processor.event.set()

with picamera.PiCamera() as camera:
    pool = [ImageProcessor() for i in range (4)]
    camera.resolution = (640, 480)
    # Set the framerate appropriately; too fast and the image processors
    # will stall the image pipeline and crash the script
    camera.framerate = 10
    camera.start_preview()
    time.sleep(2)
    camera.capture_sequence(streams(), use_video_port=True)

# Shut down the processors in an orderly fashion
while pool:
    with lock:
        processor = pool.pop()
    processor.terminated = True
    processor.join()
```

New in version 0.5.

1.5.4 Rapid capture and streaming

Following on from *Rapid capture and processing*, we can combine the video-port capture technique with *Capturing to a network stream*. The server side script doesn't change (it doesn't really care what capture technique is being used - it just reads JPEGs off the wire). The changes to the client side script can be minimal at first - just add `use_video_port=True` to the `capture_continuous()` call:

```
import io
import socket
import struct
import time
import picamera

client_socket = socket.socket()
client_socket.connect(('my_server', 8000))
connection = client_socket.makefile('wb')
try:
    with picamera.PiCamera() as camera:
        camera.resolution = (640, 480)
        time.sleep(2)
        start = time.time()
        stream = io.BytesIO()
        # Use the video-port for captures...
```

```
        for foo in camera.capture_continuous(stream, 'jpeg',
                                             use_video_port=True):
            connection.write(struct.pack('<L', stream.tell()))
            connection.flush()
            stream.seek(0)
            connection.write(stream.read())
            if time.time() - start > 30:
                break
            stream.seek(0)
            stream.truncate()
    connection.write(struct.pack('<L', 0))
finally:
    connection.close()
    client_socket.close()
```

Using this technique, the author can manage about 14fps of streaming at 640x480. One deficiency of the script above is that it interleaves capturing images with sending them over the wire (although we deliberately don't flush on sending the image data). Potentially, it would be more efficient to permit image capture to occur simultaneously with image transmission. We can attempt to do this by utilizing the background threading techniques from the final example in *Rapid capture and processing*.

Once again, please note that the following code involves some quite advanced techniques and is not "beginner friendly":

```
import io
import socket
import struct
import time
import threading
import picamera

client_socket = socket.socket()
client_socket.connect(('spider', 8000))
connection = client_socket.makefile('wb')
try:
    connection_lock = threading.Lock()
    pool = []
    pool_lock = threading.Lock()

    class ImageStreamer(threading.Thread):
        def __init__(self):
            super(ImageStreamer, self).__init__()
            self.stream = io.BytesIO()
            self.event = threading.Event()
            self.terminated = False
            self.start()

        def run(self):
            # This method runs in a background thread
            while not self.terminated:
                if self.event.wait(1):
                    try:
                        with connection_lock:
                            connection.write(struct.pack('<L', self.stream.tell()))
                            connection.flush()
                            self.stream.seek(0)
                            connection.write(self.stream.read())
                    finally:
                        self.stream.seek(0)
                        self.stream.truncate()
                        self.event.clear()
                        with pool_lock:
                            pool.append(self)
```

```
    count = 0
    start = time.time()
    finish = time.time()

    def streams():
        global count, finish
        while finish - start < 30:
            with pool_lock:
                streamer = pool.pop()
            yield streamer.stream
            streamer.event.set()
            count += 1
            finish = time.time()

    with picamera.PiCamera() as camera:
        pool = [ImageStreamer() for i in range(4)]
        camera.resolution = (640, 480)
        # Set the framerate appropriately; too fast and we'll starve the
        # pool of streamers and crash the script
        camera.framerate = 15
        camera.start_preview()
        time.sleep(2)
        camera.capture_sequence(streams(), 'jpeg', use_video_port=True)

    # Shut down the streamers in an orderly fashion
    while pool:
        with pool_lock:
            streamer = pool.pop()
        streamer.terminated = True
        streamer.join()

    # Write the terminating 0-length to the connection to let the server
    # know we're done
    with connection_lock:
        connection.write(struct.pack('<L', 0))

finally:
    connection.close()
    client_socket.close()

print('Sent %d images in %.2f seconds at %.2ffps' % (
    count, finish-start, count / (finish-start)))
```

The author's tests with the script above haven't yielded substantial improvements over the former script using `capture_continuous()`, but the reason for this is not currently clear. Suggestions for further improvements are welcomed!

New in version 0.5.

1.5.5 Capturing images whilst recording

The camera is capable of capturing still images while it is recording video. However, if one attempts this using the stills capture mode, the resulting video will have dropped frames during the still image capture. This is because regular stills require a mode change, causing the dropped frames (this is the flicker to a higher resolution that one sees when capturing while a preview is running).

However, if the *use_video_port* parameter is used to force a video-port based image capture (see *Rapid capture and processing*) then the mode change does not occur, and the resulting video will not have dropped frames:

```
import picamera
```

```
with picamera.PiCamera() as camera:
    camera.resolution = (800, 600)
    camera.start_preview()
    camera.start_recording('foo.h264')
    camera.wait_recording(10)
    camera.capture('foo.jpg', use_video_port=True)
    camera.wait_recording(10)
    camera.stop_recording()
```

The above code should produce a 20 second video with no dropped frames, and a still frame from 10 seconds into the video.

New in version 0.8.

1.5.6 Splitting to/from a circular stream

This example builds on the one in *Recording to a circular stream* and the one in *Capturing images whilst recording* to demonstrate the beginnings of a security application. As before, a `PiCameraCircularIO` instance is used to keep the last few seconds of video recorded in memory. While the video is being recorded, video-port-based still captures are taken to provide a motion detection routine with some input (the actual motion detection algorithm is left as an exercise for the reader).

Once motion is detected, the last 10 seconds of video are written to disk, and video recording is split to another disk file to proceed until motion is no longer detected. Once motion is no longer detected, we split the recording back to the in-memory ring-buffer:

```
import io
import random
import picamera
from PIL import Image

prior_image = None

def detect_motion(camera):
    global prior_image
    stream = io.BytesIO()
    camera.capture(stream, format='jpeg', use_video_port=True)
    stream.seek(0)
    if prior_image is None:
        prior_image = Image.open(stream)
        return False
    else:
        current_image = Image.open(stream)
        # Compare current_image to prior_image to detect motion. This is
        # left as an exercise for the reader!
        result = random.randint(0, 10) == 0
        # Once motion detection is done, make the prior image the current
        prior_image = current_image
        return result

def write_video(stream):
    # Write the entire content of the circular buffer to disk. No need to
    # lock the stream here as we're definitely not writing to it
    # simultaneously
    with io.open('before.h264', 'wb') as output:
        for frame in stream.frames:
            if frame.header:
                stream.seek(frame.position)
                break
        while True:
            buf = stream.read1()
            if not buf:
```

```
                break
            output.write(buf)
    # Wipe the circular stream once we're done
    stream.seek(0)
    stream.truncate()

with picamera.PiCamera() as camera:
    camera.resolution = (1280, 720)
    stream = picamera.PiCameraCircularIO(camera, seconds=10)
    camera.start_recording(stream, format='h264')
    try:
        while True:
            camera.wait_recording(1)
            if detect_motion(camera):
                print('Motion detected!')
                # As soon as we detect motion, split the recording to
                # record the frames "after" motion
                camera.split_recording('after.h264')
                # Write the 10 seconds "before" motion to disk as well
                write_video(stream)
                # Wait until motion is no longer detected, then split
                # recording back to the in-memory circular buffer
                while detect_motion(camera):
                    camera.wait_recording(1)
                print('Motion stopped!')
                camera.split_recording(stream)
    finally:
        camera.stop_recording()
```

This example also demonstrates writing the circular buffer to disk in an efficient manner using the `read1()` method (as opposed to `read()`).

Note: Note that `read1()` does not guarantee to return the number of bytes requested, even if they are available in the underlying stream; it simply returns as many as are available from a single chunk up to the limit specified.

New in version 1.0.

1.6 Camera Hardware

This chapter attempts to provide an overview of the operation of the camera under various conditions, as well as to provide an introduction to the low level software interface that picamera utilizes.

1.6.1 Camera Modes

The Pi's camera has a discrete set of input modes which are as follows:

Resolution	Aspect Ratio	Framerates	Video	Image	FoV
2592x1944	4:3	1-15fps	x	x	Full
1296x972	4:3	1-42fps	x		Full
1296x730	16:9	1-49fps	x		Full
640x480	4:3	42.1-60fps	x		Full
640x480	4:3	60.1-90fps	x		Full
1920x1080	16:9	1-30fps	x		Partial

Note: This table is accurate as of firmware revision #656. Firmwares prior to this had a more restricted set of modes, and all video modes had partial FoV. Please use `sudo rpi-update` to upgrade to the latest firmware.

Modes with full field of view (FoV) capture from the whole area of the camera's sensor (2592x1944 pixels). Modes with partial FoV only capture from the center 1920x1080 pixels. The difference between these areas is shown in the illustration below:

Which input mode is used cannot be *directly* controlled, but is selected based on the requested `resolution` and `framerate`. The rules governing which input mode is selected are as follows:

- The mode must be acceptable. Video modes can be used for video recording, or for image captures from the video port (i.e. when *use_video_port* is `True` in calls to the various capture methods). Image captures when *use_video_port* is `False` must use an image mode (of which only one currently exists).
- The closer the requested `resolution` is to the mode's resolution the better, but downscaling from a higher input resolution is preferable to upscaling from a lower input resolution.
- The requested `framerate` should be within the range of the input mode. Note that this is not a hard restriction (it is possible, but unlikely, for the camera to select a mode that does not support the requested framerate).
- The closer the aspect ratio of the requested `resolution` is to the mode's resolution, the better. Attempts to set resolutions with aspect ratios other than 4:3 or 16:9 (which are the only ratios directly supported by the modes in the table above) will choose the mode which maximizes the resulting FoV.

A few examples are given below to clarify the operation of this heuristic:

- If you set the `resolution` to 1024x768 (a 4:3 aspect ratio), and `framerate` to anything less than 42fps, the 1296x976 mode will be selected, and the camera will downscale the result to 1024x768.
- If you set the `resolution` to 1280x720 (a 16:9 wide-screen aspect ratio), and `framerate` to anything less than 49fps, the 1296x730 mode will be selected and downscaled appropriately.
- Setting `resolution` to 1920x1080 and `framerate` to 30fps exceeds the resolution of both the 1296x730 and 1296x976 modes (i.e. they would require upscaling), so the 1920x1080 mode is selected instead, although it has a reduced FoV.

- A `resolution` of 800x600 and a `framerate` of 60fps will select the 640x480 60fps mode, even though it requires upscaling because the algorithm considers the framerate to take precedence in this case.
- Any attempt to capture an image without using the video port will (temporarily) select the 2592x1944 mode while the capture is performed (this is what causes the flicker you sometimes see when a preview is running while a still image is captured).

1.6.2 Under the Hood

This section attempts to provide detail of what picamera is doing "under the hood" in response to various method calls.

The Pi's camera has three ports, the still port, the video port, and the preview port. The following sections describe how these ports are used by picamera and how they influence the camera's resolutions.

The Still Port

Firstly, the still port. Whenever this is used to capture images, it (briefly) forces the camera's mode to the only supported still mode (see *Camera Modes*) so that images are captured using the full area of the sensor. It also appears to perform a considerable amount of post-processing on captured images so that they appear higher quality.

The still port is used by the various `capture()` methods when their `use_video_port` parameter is `False` (which it is by default).

The Video Port

The video port is somewhat simpler in that it never changes the camera's mode. The video port is used by the `start_recording()` method (for recording video), and is also used by the various `capture()` methods when their `use_video_port` parameter is `True`. Images captured from the video port tend to have a "grainy" appearance, much more akin to a video frame than the images captured by the still port (the author suspects the still port may be taking an average of several frames).

The Preview Port

The preview port operates more or less identically to the video port. As the preview port is never used for encoding we won't mention it further in this section.

Encoders

The camera provides various encoders which can be attached to the still and video ports for the purpose of producing output (e.g. JPEG images or H.264 encoded video). A port can have a single encoder attached to it at any given time (or nothing if the port is not in use).

Encoders are connected directly to the still port. For example, when capturing a picture using the still port, the camera's state conceptually moves through these states:

As you have probably noticed in the diagram above, the video port is a little more complex. In order to permit simultaneous video recording and image capture via the video port, a "splitter" component is permanently connected to the video port by picamera, and encoders are in turn attached to one of its four output ports (numbered 0, 1, 2, and 3). Hence, when recording video the camera's setup looks like this:

And when simultaneously capturing images via the video port whilst recording, the camera's configuration moves through the following states:

When the `resize` parameter is passed to one of the aforementioned methods, a resizer component is placed between the camera's ports and the encoder, causing the output to be resized before it reaches the encoder. This is particularly useful for video recording, as the H.264 encoder cannot cope with full resolution input. Hence, when performing full frame video recording, the camera's setup looks like this:

Finally, when performing raw captures an encoder is (naturally) not required. Instead data is taken directly from the camera's ports. When raw YUV format is requested no components are attached to the ports at all (as the ports are configured for YUV output at all times). When another raw format like RGBA is requested, a resizer is used (with its output resolution set to the input resolution, unless the `resize` option is specified with something different), and its output format is set to the requested raw format:

Please note that even the description above is almost certainly far removed from what actually happens at the camera's ISP level. Rather, what has been described in this section is how the MMAL library exposes the camera to applications which utilize it (these include the picamera library, along with the official *raspistill* and *raspivid* applications).

In other words, by using picamera you are passing through (at least) two abstraction layers which necessarily obscure (but hopefully simplify) the "true" operation of the camera.

1.7 API Reference

This package primarily provides the `PiCamera` class which is a pure Python interface to the Raspberry Pi's camera module.

1.7.1 PiCamera

class `picamera.`**`PiCamera`**

Provides a pure Python interface to the Raspberry Pi's camera module.

Upon construction, this class initializes the camera. As there is only a single camera supported by the Raspberry Pi, this means that only a single instance of this class can exist at any given time (it is effectively a singleton class although it is not implemented as such).

No preview or recording is started automatically upon construction. Use the `capture()` method to capture images, the `start_recording()` method to begin recording video, or the `start_preview()` method to start live display of the camera's input.

Several attributes are provided to adjust the camera's configuration. Some of these can be adjusted while a recording is running, like `brightness`. Others, like `resolution`, can only be adjusted when the camera is idle.

When you are finished with the camera, you should ensure you call the `close()` method to release the camera resources (failure to do this leads to GPU memory leaks):

```
camera = PiCamera()
try:
    # do something with the camera
    pass
finally:
    camera.close()
```

The class supports the context manager protocol to make this particularly easy (upon exiting the `with` statement, the `close()` method is automatically called):

```
with PiCamera() as camera:
    # do something with the camera
    pass
```

capture(*output*, *format=None*, *use_video_port=False*, *resize=None*, *splitter_port=0*, ***options*)
Capture an image from the camera, storing it in *output*.

If *output* is a string, it will be treated as a filename for a new file which the image will be written to. Otherwise, *output* is assumed to a be a file-like object and the image data is appended to it (the implementation only assumes the object has a `write()` method - no other methods will be called).

If *format* is `None` (the default), the method will attempt to guess the required image format from the extension of *output* (if it's a string), or from the *name* attribute of *output* (if it has one). In the case that the format cannot be determined, a `PiCameraValueError` will be raised.

If *format* is not `None`, it must be a string specifying the format that you want the image written to. The format can be a MIME-type or one of the following strings:

- `'jpeg'` - Write a JPEG file
- `'png'` - Write a PNG file
- `'gif'` - Write a GIF file
- `'bmp'` - Write a Windows bitmap file
- `'yuv'` - Write the raw image data to a file in YUV420 format
- `'rgb'` - Write the raw image data to a file in 24-bit RGB format
- `'rgba'` - Write the raw image data to a file in 32-bit RGBA format
- `'bgr'` - Write the raw image data to a file in 24-bit BGR format
- `'bgra'` - Write the raw image data to a file in 32-bit BGRA format
- `'raw'` - Deprecated option for raw captures; the format is taken from the deprecated `raw_format` attribute

The *use_video_port* parameter controls whether the camera's image or video port is used to capture images. It defaults to `False` which means that the camera's image port is used. This port is slow but produces better quality pictures. If you need rapid capture up to the rate of video frames, set this to `True`.

When *use_video_port* is `True`, the *splitter_port* parameter specifies the port of the video splitter that the image encoder will be attached to. This defaults to `0` and most users will have no need to specify anything different. This parameter is ignored when *use_video_port* is `False`. See *Under the Hood* for more information about the video splitter.

If *resize* is not `None` (the default), it must be a two-element tuple specifying the width and height that the image should be resized to.

> **Warning:** If *resize* is specified, or *use_video_port* is `True`, Exif metadata will **not** be included in JPEG output. This is due to an underlying firmware limitation.

Certain file formats accept additional options which can be specified as keyword arguments. Currently, only the `'jpeg'` encoder accepts additional options, which are:

- *quality* - Defines the quality of the JPEG encoder as an integer ranging from 1 to 100. Defaults to 85.
- *thumbnail* - Defines the size and quality of the thumbnail to embed in the Exif metadata. Specifying `None` disables thumbnail generation. Otherwise, specify a tuple of `(width, height, quality)`. Defaults to `(64, 48, 35)`.
- *bayer* - If `True`, the raw bayer data from the camera's sensor is included in the Exif metadata.

Note: The so-called "raw" formats listed above (`'yuv'`, `'rgb'`, etc.) do not represent the raw bayer data from the camera's sensor. Rather they provide access to the image data after GPU processing, but before format encoding (JPEG, PNG, etc). Currently, the only method of accessing the raw bayer data is via the *bayer* parameter described above.

Changed in version 1.0: The *resize* parameter was added, and raw capture formats can now be specified directly

Changed in version 1.3: The *splitter_port* parameter was added, and *bayer* was added as an option for the `'jpeg'` format

capture_continuous(*output*, *format=None*, *use_video_port=False*, *resize=None*, *splitter_port=0*, ***options*)

Capture images continuously from the camera as an infinite iterator.

This method returns an infinite iterator of images captured continuously from the camera. If *output* is a string, each captured image is stored in a file named after *output* after substitution of two values with the `format()` method. Those two values are:

- `{counter}` - a simple incrementor that starts at 1 and increases by 1 for each image taken
- `{timestamp}` - a `datetime` instance

The table below contains several example values of *output* and the sequence of filenames those values could produce:

output Value	Filenames	Notes
`'image{counter}.jpg'`	image1.jpg, image2.jpg, image3.jpg, ...	
`'image{counter:02d}.jpg'`	image01.jpg, image02.jpg, image03.jpg, ...	
`'image{timestamp}.jpg'`	image2013-10-05 12:07:12.346743.jpg, image2013-10-05 12:07:32.498539, ...	1.
`'image{timestamp:%H-%M-%S-%f}.jpg'`	image12-10-02-561527.jpg, image12-10-14-905398.jpg	1.
`'{timestamp:%H%M%S}-{counter:03d}.jpg'`	121002-001.jpg, 121013-002.jpg, 121014-003.jpg, ...	2.

1. Note that because timestamp's default output includes colons (:), the resulting filenames are not suitable for use on Windows. For this reason (and the fact the default contains spaces) it is strongly recommended you always specify a format when using `{timestamp}`.
2. You can use both `{timestamp}` and `{counter}` in a single format string (multiple times too!) although this tends to be redundant.

If *output* is not a string, it is assumed to be a file-like object and each image is simply written to this object sequentially. In this case you will likely either want to write something to the object between the images to distinguish them, or clear the object between iterations.

The *format*, *use_video_port*, *splitter_port*, *resize*, and *options* parameters are the same as in `capture()`.

For example, to capture 60 images with a one second delay between them, writing the output to a series of JPEG files named image01.jpg, image02.jpg, etc. one could do the following:

```
import time
import picamera
with picamera.PiCamera() as camera:
    camera.start_preview()
    try:
        for i, filename in enumerate(camera.capture_continuous('image{counter:02d}.jpg'))
            print(filename)
            time.sleep(1)
            if i == 59:
                break
```

```
    finally:
        camera.stop_preview()
```

Alternatively, to capture JPEG frames as fast as possible into an in-memory stream, performing some processing on each stream until some condition is satisfied:

```
import io
import time
import picamera
with picamera.PiCamera() as camera:
    stream = io.BytesIO()
    for foo in camera.capture_continuous(stream, format='jpeg'):
        # Truncate the stream to the current position (in case
        # prior iterations output a longer image)
        stream.truncate()
        stream.seek(0)
        if process(stream):
            break
```

Changed in version 1.0: The *resize* parameter was added, and raw capture formats can now be specified directly

Changed in version 1.3: The *splitter_port* parameter was added

capture_sequence(*outputs*, *format='jpeg'*, *use_video_port=False*, *resize=None*, *splitter_port=0*, ***options*)

Capture a sequence of consecutive images from the camera.

This method accepts a sequence or iterator of *outputs* each of which must either be a string specifying a filename for output, or a file-like object with a `write` method. For each item in the sequence or iterator of outputs, the camera captures a single image as fast as it can.

The *format*, *use_video_port*, *splitter_port*, *resize*, and *options* parameters are the same as in `capture()`, but *format* defaults to `'jpeg'`. The format is **not** derived from the filenames in *outputs* by this method.

For example, to capture 3 consecutive images:

```
import time
import picamera
with picamera.PiCamera() as camera:
    camera.start_preview()
    time.sleep(2)
    camera.capture_sequence([
        'image1.jpg',
        'image2.jpg',
        'image3.jpg',
        ])
    camera.stop_preview()
```

If you wish to capture a large number of images, a list comprehension or generator expression can be used to construct the list of filenames to use:

```
import time
import picamera
with picamera.PiCamera() as camera:
    camera.start_preview()
    time.sleep(2)
    camera.capture_sequence([
        'image%02d.jpg' % i
        for i in range(100)
        ])
    camera.stop_preview()
```

More complex effects can be obtained by using a generator function to provide the filenames or output objects.

Changed in version 1.0: The *resize* parameter was added, and raw capture formats can now be specified directly

Changed in version 1.3: The *splitter_port* parameter was added

close()
Finalizes the state of the camera.

After successfully constructing a PiCamera object, you should ensure you call the close() method once you are finished with the camera (e.g. in the finally section of a try..finally block). This method stops all recording and preview activities and releases all resources associated with the camera; this is necessary to prevent GPU memory leaks.

record_sequence(*outputs*, *format='h264'*, *resize=None*, *splitter_port=1*, ***options*)
Record a sequence of video clips from the camera.

This method accepts a sequence or iterator of *outputs* each of which must either be a string specifying a filename for output, or a file-like object with a write method.

The method acts as an iterator itself, yielding each item of the sequence in turn. In this way, the caller can control how long to record to each item by only permitting the loop to continue when ready to switch to the next output.

The *format*, *splitter_port*, *resize*, and *options* parameters are the same as in start_recording(), but *format* defaults to 'h264'. The format is **not** derived from the filenames in *outputs* by this method.

For example, to record 3 consecutive 10-second video clips, writing the output to a series of H.264 files named clip01.h264, clip02.h264, and clip03.h264 one could use the following:

```
import picamera
with picamera.PiCamera() as camera:
    for filename in camera.record_sequence([
            'clip01.h264',
            'clip02.h264',
            'clip03.h264']):
        print('Recording to %s' % filename)
        camera.wait_recording(10)
```

Alternatively, a more flexible method of writing the previous example (which is easier to expand to a large number of output files) is by using a generator expression as the input sequence:

```
import picamera
with picamera.PiCamera() as camera:
    for filename in camera.record_sequence(
            'clip%02d.h264' % i for i in range(3)):
        print('Recording to %s' % filename)
        camera.wait_recording(10)
```

More advanced techniques are also possible by utilising infinite sequences, such as those generated by itertools.cycle(). In the following example, recording is switched between two in-memory streams. Whilst one stream is recording, the other is being analysed. The script only stops recording when a video recording meets some criteria defined by the process function:

```
import io
import itertools
import picamera
with picamera.PiCamera() as camera:
    analyse = None
    for stream in camera.record_sequence(
            itertools.cycle((io.BytesIO(), io.BytesIO()))):
        if analyse is not None:
            if process(analyse):
```

```
                break
            analyse.seek(0)
            analyse.truncate()
        camera.wait_recording(5)
        analyse = stream
```

New in version 1.3.

split_recording(*output*, *splitter_port=1*)
Continue the recording in the specified output; close existing output.

When called, the video encoder will wait for the next appropriate split point (an inline SPS header), then will cease writing to the current output (and close it, if it was specified as a filename), and continue writing to the newly specified *output*.

If *output* is a string, it will be treated as a filename for a new file which the video will be written to. Otherwise, *output* is assumed to be a file-like object and the video data is appended to it (the implementation only assumes the object has a `write()` method - no other methods will be called).

The *splitter_port* parameter specifies which port of the video splitter the encoder you wish to change outputs is attached to. This defaults to `1` and most users will have no need to specify anything different. Valid values are between `0` and `3` inclusive.

Note that unlike `start_recording()`, you cannot specify format or options as these cannot be changed in the middle of recording. Only the new *output* can be specified. Furthermore, the format of the recording is currently limited to H264, *inline_headers* must be `True`, and *bitrate* must be non-zero (CBR mode) when `start_recording()` is called (this is the default).

Changed in version 1.3: The *splitter_port* parameter was added

start_preview()
Displays the preview window.

This method starts a new preview running at the configured resolution (see `resolution`). Most camera properties can be modified "live" while the preview is running (e.g. `brightness`). The preview overrides whatever is currently visible on the display. More specifically, the preview does not rely on a graphical environment like X-Windows (it can run quite happily from a TTY console); it is simply an overlay on the Pi's video output.

To stop the preview and reveal the display again, call `stop_preview()`. The preview can be started and stopped multiple times during the lifetime of the `PiCamera` object.

Note: Because the preview typically obscures the screen, ensure you have a means of stopping a preview before starting one. If the preview obscures your interactive console you won't be able to Alt+Tab back to it as the preview isn't in a window. If you are in an interactive Python session, simply pressing Ctrl+D usually suffices to terminate the environment, including the camera and its associated preview.

---

start_recording(*output*, *format=None*, *resize=None*, *splitter_port=1*, ***options*)
Start recording video from the camera, storing it in *output*.

If *output* is a string, it will be treated as a filename for a new file which the video will be written to. Otherwise, *output* is assumed to be a file-like object and the video data is appended to it (the implementation only assumes the object has a `write()` method - no other methods will be called).

If *format* is `None` (the default), the method will attempt to guess the required video format from the extension of *output* (if it's a string), or from the *name* attribute of *output* (if it has one). In the case that the format cannot be determined, a `PiCameraValueError` will be raised.

If *format* is not `None`, it must be a string specifying the format that you want the image written to. The format can be a MIME-type or one of the following strings:

- `'h264'` - Write an H.264 video stream
- `'mjpeg'` - Write an M-JPEG video stream

If *resize* is not `None` (the default), it must be a two-element tuple specifying the width and height that the video recording should be resized to. This is particularly useful for recording video using the full area of the camera sensor (which is not possible without down-sizing the output).

The *splitter_port* parameter specifies the port of the built-in splitter that the video encoder will be attached to. This defaults to `1` and most users will have no need to specify anything different. If you wish to record multiple (presumably resized) streams simultaneously, specify a value between `0` and `3` inclusive for this parameter, ensuring that you do not specify a port that is currently in use.

Certain formats accept additional options which can be specified as keyword arguments. The `'h264'` format accepts the following additional options:

- *profile* - The H.264 profile to use for encoding. Defaults to 'high', but can be one of 'baseline', 'main', 'high', or 'constrained'.
- *intra_period* - The key frame rate (the rate at which I-frames are inserted in the output). Defaults to 0, but can be any positive 32-bit integer value representing the number of frames between successive I-frames.
- *inline_headers* - When `True`, specifies that the encoder should output SPS/PPS headers within the stream to ensure GOPs (groups of pictures) are self describing. This is important for streaming applications where the client may wish to seek within the stream, and enables the use of `split_recording()`. Defaults to `True` if not specified.

All formats accept the following additional options:

- *bitrate* - The bitrate at which video will be encoded. Defaults to 17000000 (17Mbps) if not specified. A value of 0 implies VBR (variable bitrate) encoding. The maximum value is 25000000 (25Mbps).
- *quantization* - When *bitrate* is zero (for variable bitrate encodings), this parameter specifies the quality that the encoder should attempt to maintain.

  For the `'h264'` format, use values between 10 and 40 where 10 is extremely high quality, and 40 is extremely low (20-25 is usually a reasonable range for H.264 encoding). Note that `split_recording()` cannot be used in VBR mode.

Changed in version 1.0: The *resize* parameter was added, and `'mjpeg'` was added as a recording format

Changed in version 1.3: The *splitter_port* parameter was added

stop_preview()
Closes the preview window display.

If `start_preview()` has previously been called, this method shuts down the preview display which generally results in the underlying TTY becoming visible again. If a preview is not currently running, no exception is raised - the method will simply do nothing.

stop_recording(*splitter_port=1*)
Stop recording video from the camera.

After calling this method the video encoder will be shut down and output will stop being written to the file-like object specified with `start_recording()`. If an error occurred during recording and `wait_recording()` has not been called since the error then this method will raise the exception.

The *splitter_port* parameter specifies which port of the video splitter the encoder you wish to stop is attached to. This defaults to `1` and most users will have no need to specify anything different. Valid values are between `0` and `3` inclusive.

Changed in version 1.3: The *splitter_port* parameter was added

wait_recording(*timeout=0*, *splitter_port=1*)
Wait on the video encoder for timeout seconds.

It is recommended that this method is called while recording to check for exceptions. If an error occurs during recording (for example out of disk space), an exception will only be raised when the `wait_recording()` or `stop_recording()` methods are called.

If `timeout` is 0 (the default) the function will immediately return (or raise an exception if an error has occurred).

The *splitter_port* parameter specifies which port of the video splitter the encoder you wish to wait on is attached to. This defaults to `1` and most users will have no need to specify anything different. Valid values are between `0` and `3` inclusive.

Changed in version 1.3: The *splitter_port* parameter was added

ISO

Retrieves or sets the apparent ISO setting of the camera.

When queried, the `ISO` property returns the ISO setting of the camera, a value which represents the sensitivity of the camera to light. Lower ISO speeds (e.g. 100) imply less sensitivity than higher ISO speeds (e.g. 400 or 800). Lower sensitivities tend to produce less "noisy" (smoother) images, but operate poorly in low light conditions.

When set, the property adjusts the sensitivity of the camera. Valid values are between 0 (auto) and 800. The actual value used when ISO is explicitly set will be one of the following values (whichever is closest): 100, 200, 320, 400, 500, 640, 800.

ISO can be adjusted while previews or recordings are in progress. The default value is 0 which means the ISO is automatically set according to image-taking conditions.

Note: With ISO settings other than 0 (auto), the `exposure_mode` property becomes non-functional.

---

awb_mode

Retrieves or sets the auto-white-balance mode of the camera.

When queried, the `awb_mode` property returns a string representing the auto-white-balance setting of the camera. The possible values can be obtained from the `PiCamera.AWB_MODES` attribute.

When set, the property adjusts the camera's auto-white-balance mode. The property can be set while recordings or previews are in progress. The default value is `'auto'`.

brightness

Retrieves or sets the brightness setting of the camera.

When queried, the `brightness` property returns the brightness level of the camera as an integer between 0 and 100. When set, the property adjusts the brightness of the camera. Brightness can be adjusted while previews or recordings are in progress. The default value is 50.

closed

Returns `True` if the `close()` method has been called.

color_effects

Retrieves or sets the current color effect applied by the camera.

When queried, the `color_effects` property either returns `None` which indicates that the camera is using normal color settings, or a `(u, v)` tuple where `u` and `v` are integer values between 0 and 255.

When set, the property changes the color effect applied by the camera. The property can be set while recordings or previews are in progress. For example, to make the image black and white set the value to `(128, 128)`. The default value is `None`.

contrast

Retrieves or sets the contrast setting of the camera.

When queried, the `contrast` property returns the contrast level of the camera as an integer between -100 and 100. When set, the property adjusts the contrast of the camera. Contrast can be adjusted while previews or recordings are in progress. The default value is 0.

crop

Retrieves or sets the crop applied to the camera's input.

When queried, the `crop` property returns a `(x, y, w, h)` tuple of floating point values ranging from 0.0 to 1.0, indicating the proportion of the image to include in the output (the "Region of Interest" or ROI). The default value is `(0.0, 0.0, 1.0, 1.0)` which indicates that everything should be included. The property can be set while recordings or previews are in progress.

exif_tags

Holds a mapping of the Exif tags to apply to captured images.

Note: Please note that Exif tagging is only supported with the `jpeg` format.

By default several Exif tags are automatically applied to any images taken with the `capture()` method: `IFD0.Make` (which is set to `RaspberryPi`), `IFD0.Model` (which is set to `RP_OV5647`), and three timestamp tags: `IFD0.DateTime`, `EXIF.DateTimeOriginal`, and `EXIF.DateTimeDigitized` which are all set to the current date and time just before the picture is taken.

If you wish to set additional Exif tags, or override any of the aforementioned tags, simply add entries to the exif_tags map before calling `capture()`. For example:

```
camera.exif_tags['IFD0.Copyright'] = 'Copyright (c) 2013 Foo Industries'
```

The Exif standard mandates ASCII encoding for all textual values, hence strings containing non-ASCII characters will cause an encoding error to be raised when `capture()` is called. If you wish to set binary values, use a `bytes()` value:

```
camera.exif_tags['EXIF.UserComment'] = b'Something containing\x00NULL characters'
```

Warning: Binary Exif values are currently ignored; this appears to be a libmmal or firmware bug.

You may also specify datetime values, integer, or float values, all of which will be converted to appropriate ASCII strings (datetime values are formatted as `YYYY:MM:DD HH:MM:SS` in accordance with the Exif standard).

The currently supported Exif tags are:

Group	Tags
IFD0, IFD1	ImageWidth, ImageLength, BitsPerSample, Compression, PhotometricInterpretation, ImageDescription, Make, Model, StripOffsets, Orientation, SamplesPerPixel, RowsPerString, StripByteCounts, Xresolution, Yresolution, PlanarConfiguration, ResolutionUnit, TransferFunction, Software, DateTime, Artist, WhitePoint, PrimaryChromaticities, JPEGInterchangeFormat, JPEGInterchangeFormatLength, YcbCrCoefficients, YcbCrSubSampling, YcbCrPositioning, ReferenceBlackWhite, Copyright
EXIF	ExposureTime, FNumber, ExposureProgram, SpectralSensitivity, ISOSpeedRatings, OECF, ExifVersion, DateTimeOriginal, DateTimeDigitized, ComponentsConfiguration, CompressedBitsPerPixel, ShutterSpeedValue, ApertureValue, BrightnessValue, ExposureBiasValue, MaxApertureValue, SubjectDistance, MeteringMode, LightSource, Flash, FocalLength, SubjectArea, MakerNote, UserComment, SubSecTime, SubSecTimeOriginal, SubSecTimeDigitized, FlashpixVersion, ColorSpace, PixelXDimension, PixelYDimension, RelatedSoundFile, FlashEnergy, SpacialFrequencyResponse, FocalPlaneXResolution, FocalPlaneYResolution, FocalPlaneResolutionUnit, SubjectLocation, ExposureIndex, SensingMethod, FileSource, SceneType, CFAPattern, CustomRendered, ExposureMode, WhiteBalance, DigitalZoomRatio, FocalLengthIn35mmFilm, SceneCaptureType, GainControl, Contrast, Saturation, Sharpness, DeviceSettingDescription, SubjectDistanceRange, ImageUniqueID
GPS	GPSVersionID, GPSLatitudeRef, GPSLatitude, GPSLongitudeRef, GPSLongitude, GPSAltitudeRef, GPSAltitude, GPSTimeStamp, GPSSatellites, GPSStatus, GPSMeasureMode, GPSDOP, GPSSpeedRef, GPSSpeed, GPSTrackRef, GPSTrack, GPSImgDirectionRef, GPSImgDirection, GPSMapDatum, GPSDestLatitudeRef, GPSDestLatitude, GPSDestLongitudeRef, GPSDestLongitude, GPSDestBearingRef, GPSDestBearing, GPSDestDistanceRef, GPSDestDistance, GPSProcessingMethod, GPSAreaInformation, GPSDateStamp, GPSDifferential
EINT	InteroperabilityIndex, InteroperabilityVersion, RelatedImageFileFormat, RelatedImageWidth, RelatedImageLength

exposure_compensation

Retrieves or sets the exposure compensation level of the camera.

When queried, the `exposure_compensation` property returns an integer value between -25 and 25 indicating the exposure level of the camera. Larger values result in brighter images.

When set, the property adjusts the camera's exposure compensation level. The property can be set while recordings or previews are in progress. The default value is `0`.

exposure_mode

Retrieves or sets the exposure mode of the camera.

When queried, the `exposure_mode` property returns a string representing the exposure setting of the camera. The possible values can be obtained from the `PiCamera.EXPOSURE_MODES` attribute.

When set, the property adjusts the camera's exposure mode. The property can be set while recordings or previews are in progress. The default value is `'auto'`.

frame

Retrieves information about the current frame recorded from the camera.

When video recording is active (after a call to `start_recording()`), this attribute will return a `PiVideoFrame` tuple containing information about the current frame that the camera is recording.

If multiple video recordings are currently in progress (after multiple calls to `start_recording()` with different values for the `splitter_port` parameter), this attribute will return a `dict` mapping active port numbers to a `PiVideoFrame` tuples.

Querying this property when the camera is not recording will result in an exception.

Note: There is a small window of time when querying this attribute will return `None` after calling

start_recording(). If this attribute returns None, this means that the video encoder has been initialized, but the camera has not yet returned any frames.

---

framerate

Retrieves or sets the framerate at which video-port based image captures, video recordings, and previews will run.

When queried, the framerate property returns the rate at which the camera's video and preview ports will operate as a Fraction instance which can be easily converted to an int or float.

Note: For backwards compatibility, a derivative of the Fraction class is actually used which permits the value to be treated as a tuple of (numerator, denominator).

When set, the property reconfigures the camera so that the next call to recording and previewing methods will use the new framerate. The framerate can be specified as an int, float, Fraction, or a (numerator, denominator) tuple. The camera must not be closed, and no recording must be active when the property is set.

Note: This attribute, in combination with resolution, determines the mode that the camera operates in. The actual sensor framerate and resolution used by the camera is influenced, but not directly set, by this property. See *Camera Modes* for more information.

---

hflip

Retrieves or sets whether the camera's output is horizontally flipped.

When queried, the hflip property returns a boolean indicating whether or not the camera's output is horizontally flipped. The property can be set while recordings or previews are in progress. The default value is False.

image_effect

Retrieves or sets the current image effect applied by the camera.

When queried, the image_effect property returns a string representing the effect the camera will apply to captured video. The possible values can be obtained from the PiCamera.IMAGE_EFFECTS attribute.

When set, the property changes the effect applied by the camera. The property can be set while recordings or previews are in progress, but only certain effects work while recording video (notably 'negative' and 'solarize'). The default value is 'none'.

led

Sets the state of the camera's LED via GPIO.

If a GPIO library is available (only RPi.GPIO is currently supported), and if the python process has the necessary privileges (typically this means running as root via sudo), this property can be used to set the state of the camera's LED as a boolean value (True is on, False is off).

Note: This is a write-only property. While it can be used to control the camera's LED, you cannot query the state of the camera's LED using this property.

---

meter_mode

Retrieves or sets the metering mode of the camera.

When queried, the meter_mode property returns the method by which the camera determines the exposure as one of the following strings:

Value	Description
'average'	The camera measures the average of the entire scene.
'spot'	The camera measures the center of the scene.
'backlit'	The camera measures a larger central area, ignoring the edges of the scene.
'matrix'	The camera measures several points within the scene.

When set, the property adjusts the camera's metering mode. The property can be set while recordings or previews are in progress. The default value is `'average'`. All possible values for the attribute can be obtained from the `PiCamera.METER_MODES` attribute.

preview_alpha
Retrieves or sets the opacity of the preview window.

When queried, the `preview_alpha` property returns a value between 0 and 255 indicating the opacity of the preview window, where 0 is completely transparent and 255 is completely opaque. The default value is 255. The property can be set while recordings or previews are in progress.

Note: If the preview is not running, the property will not reflect changes to it, but they will be in effect next time the preview is started. In other words, you can set preview_alpha to 128, but querying it will still return 255 (the default) until you call `start_preview()` at which point the preview will appear semi-transparent and `preview_alpha` will suddenly return 128. This appears to be a firmware issue.

---

preview_fullscreen
Retrieves or sets full-screen for the preview window.

The `preview_fullscreen` property is a bool which controls whether the preview window takes up the entire display or not. When set to `False`, the `preview_window` property can be used to control the precise size of the preview display. The property can be set while recordings or previews are active.

Note: The `preview_fullscreen` attribute is afflicted by the same issue as `preview_alpha` with regards to changes while the preview is not running.

---

preview_layer
Retrieves of sets the layer of the preview window.

The `preview_layer` property is an integer which controls the layer that the preview window occupies. It defaults to 2 which results in the preview appearing above all other output.

Warning: Operation of this attribute is not yet fully understood. The documentation above is incomplete and may be incorrect!

preview_window
Retrieves or sets the size of the preview window.

When the `preview_fullscreen` property is set to `False`, the `preview_window` property specifies the size and position of the preview window on the display. The property is a 4-tuple consisting of `(x, y, width, height)`. The property can be set while recordings or previews are active.

Note: The `preview_window` attribute is afflicted by the same issue as `preview_alpha` with regards to changes while the preview is not running.

---

previewing
Returns `True` if the `start_preview()` method has been called, and no `stop_preview()` call has been made yet.

raw_format
Retrieves or sets the raw format of the camera's ports.

Deprecated since version 1.0: Please use `'yuv'` or `'rgb'` directly as a format in the various capture methods instead.

recording
Returns `True` if the `start_recording()` method has been called, and no `stop_recording()` call has been made yet.

resolution
Retrieves or sets the resolution at which image captures, video recordings, and previews will be captured.

When queried, the `resolution` property returns the resolution at which the camera will operate as a tuple of `(width, height)` measured in pixels. This is the resolution that the `capture()` method will produce images at, and the resolution that `start_recording()` will produce videos at.

When set, the property reconfigures the camera so that the next call to these methods will use the new resolution. The resolution must be specified as a `(width, height)` tuple, the camera must not be closed, and no recording must be active when the property is set.

The property defaults to the Pi's currently configured display resolution.

Note: This attribute, in combination with `framerate`, determines the mode that the camera operates in. The actual sensor framerate and resolution used by the camera is influenced, but not directly set, by this property. See *Camera Modes* for more information.

---

rotation
Retrieves or sets the current rotation of the camera's image.

When queried, the `rotation` property returns the rotation applied to the image. Valid values are 0, 90, 180, and 270.

When set, the property changes the color effect applied by the camera. The property can be set while recordings or previews are in progress. The default value is `0`.

saturation
Retrieves or sets the saturation setting of the camera.

When queried, the `saturation` property returns the color saturation of the camera as an integer between -100 and 100. When set, the property adjusts the saturation of the camera. Saturation can be adjusted while previews or recordings are in progress. The default value is 0.

sharpness
Retrieves or sets the sharpness setting of the camera.

When queried, the `sharpness` property returns the sharpness level of the camera (a measure of the amount of post-processing to reduce or increase image sharpness) as an integer between -100 and 100. When set, the property adjusts the sharpness of the camera. Sharpness can be adjusted while previews or recordings are in progress. The default value is 0.

shutter_speed
Retrieves or sets the shutter speed of the camera in microseconds.

When queried, the `shutter_speed` property returns the shutter speed of the camera in microseconds, or 0 which indicates that the speed will be automatically determined according to lighting conditions. Faster shutter times naturally require greater amounts of illumination and vice versa.

When set, the property adjusts the shutter speed of the camera, which most obviously affects the illumination of subsequently captured images. Shutter speed can be adjusted while previews or recordings are running. The default value is 0 (auto).

vflip
Retrieves or sets whether the camera's output is vertically flipped.

When queried, the `vflip` property returns a boolean indicating whether or not the camera's output is vertically flipped. The property can be set while recordings or previews are in progress. The default value is `False`.

video_stabilization
Retrieves or sets the video stabilization mode of the camera.

When queried, the `video_stabilization` property returns a boolean value indicating whether or not the camera attempts to compensate for motion.

When set, the property activates or deactivates video stabilization. The property can be set while recordings or previews are in progress. The default value is `False`.

Note: The built-in video stabilization only accounts for vertical and horizontal motion, not rotation.

1.7.2 PiCameraCircularIO

class `picamera.`**`PiCameraCircularIO`**(*camera*, *size=None*, *seconds=None*, *bitrate=17000000*)

A derivative of `CircularIO` which tracks camera frames.

PiCameraCircularIO provides an in-memory stream based on a ring buffer. It is a specialization of `CircularIO` which associates video frame meta-data with the recorded stream, accessible from the `frames` property.

Warning: The class makes a couple of assumptions which will cause the frame meta-data tracking to break if they are not adhered to:

- the stream is only ever appended to - no writes ever start from the middle of the stream
- the stream is never truncated (from the right; being ring buffer based, left truncation will occur automatically)

The *camera* parameter specifies the `PiCamera` instance that will be recording video to the stream. If specified, the *size* parameter determines the maximum size of the stream in bytes. If *size* is not specified (or `None`), then *seconds* must be specified instead. This provides the maximum length of the stream in seconds, assuming a data rate in bits-per-second given by the *bitrate* parameter (which defaults to `17000000`, or 17Mbps, which is also the default bitrate used for video recording by `PiCamera`). You cannot specify both *size* and *seconds*.

`frames`

Returns an iterator over the frame meta-data.

As the camera records video to the stream, the class captures the meta-data associated with each frame (in the form of a `PiVideoFrame` tuple), discarding meta-data for frames which are no longer fully stored within the underlying ring buffer. You can use the frame meta-data to locate, for example, the first keyframe present in the stream in order to determine an appropriate range to extract.

1.7.3 CircularIO

class `picamera.`**`CircularIO`**(*size*)

A thread-safe stream which uses a ring buffer for storage.

CircularIO provides an in-memory stream similar to the `io.BytesIO` class. However, unlike BytesIO its underlying storage is a ring buffer with a fixed maximum size. Once the maximum size is reached, writing effectively loops round to the beginning to the ring and starts overwriting the oldest content.

The *size* parameter specifies the maximum size of the stream in bytes. The `read()`, `tell()`, and `seek()` methods all operate equivalently to those in `io.BytesIO` whilst `write()` only differs in the wrapping behaviour described above. A `read1()` method is also provided for efficient reading of the underlying ring buffer in write-sized chunks (or less).

A re-entrant threading lock guards all operations, and is accessible for external use via the `lock` attribute.

The performance of the class is geared toward faster writing than reading on the assumption that writing will be the common operation and reading the rare operation (a reasonable assumption for the camera use-case, but not necessarily for more general usage).

`getvalue`()

Return `bytes` containing the entire contents of the buffer.

`read`(*n=-1*)

Read up to *n* bytes from the stream and return them. As a convenience, if *n* is unspecified or -1,

readall() is called. Fewer than *n* bytes may be returned if there are fewer than *n* bytes from the current stream position to the end of the stream.

If 0 bytes are returned, and *n* was not 0, this indicates end of the stream.

read1(*n=-1*)
Read up to *n* bytes from the stream using only a single call to the underlying object.

In the case of CircularIO this roughly corresponds to returning the content from the current position up to the end of the write that added that content to the stream (assuming no subsequent writes overwrote the content). read1() is particularly useful for efficient copying of the stream's content.

readable()
Returns True, indicating that the stream supports read().

seek(*offset*, *whence=0*)
Change the stream position to the given byte *offset*. *offset* is interpreted relative to the position indicated by *whence*. Values for *whence* are:

- SEEK_SET or 0 – start of the stream (the default); *offset* should be zero or positive
- SEEK_CUR or 1 – current stream position; *offset* may be negative
- SEEK_END or 2 – end of the stream; *offset* is usually negative

Return the new absolute position.

seekable()
Returns True, indicating the stream supports seek() and tell().

tell()
Return the current stream position.

truncate(*size=None*)
Resize the stream to the given *size* in bytes (or the current position if *size* is not specified). This resizing can extend or reduce the current stream size. In case of extension, the contents of the new file area will be NUL (\x00) bytes. The new stream size is returned.

The current stream position isn't changed unless the resizing is expanding the stream, in which case it may be set to the maximum stream size if the expansion causes the ring buffer to loop around.

writable()
Returns True, indicating that the stream supports write().

write(*b*)
Write the given bytes or bytearray object, *b*, to the underlying stream and return the number of bytes written.

lock
A re-entrant threading lock which is used to guard all operations.

size
Return the maximum size of the buffer in bytes.

1.7.4 PiVideoFrame

class picamera.**PiVideoFrame**(*index*, *key*, *frame_size*, *video_size*, *split_size*, *timestamp*)

index
Returns the zero-based number of the frame. This is a monotonic counter that is simply incremented every time the camera returns a frame-end buffer. As a consequence, this attribute cannot be used to detect dropped frames.

position
Returns the zero-based position of the frame in the stream containing it.

keyframe
Returns a bool indicating whether the current frame is a keyframe (an intra-frame, or I-frame in MPEG parlance).

frame_size
Returns the size in bytes of the current frame.

video_size
Returns the size in bytes of the entire video up to the current frame. Note that this is unlikely to match the size of the actual file/stream written so far. Firstly this is because the frame attribute is only updated when the encoder outputs the *end* of a frame, which will cause the reported size to be smaller than the actual amount written. Secondly this is because a stream may utilize buffering which will cause the actual amount written (e.g. to disk) to lag behind the value reported by this attribute.

split_size
Returns the size in bytes of the video recorded since the last call to either `start_recording()` or `split_recording()`. For the reasons explained above, this may differ from the size of the actual file/stream written so far.

timestamp
Returns the presentation timestamp (PTS) of the current frame as reported by the encoder. This is represented by the number of microseconds (millionths of a second) since video recording started. As the frame attribute is only updated when the encoder outputs the end of a frame, this value may lag behind the actual time since `start_recording()` was called.

> **Warning:** Currently, the video encoder occasionally returns "time unknown" values in this field which picamera represents as `None`. If you are querying this property you will need to check the value is not `None` before using it.

header
Contains a bool indicating whether the current frame is actually an SPS/PPS header. Typically it is best to split an H.264 stream so that it starts with an SPS/PPS header.

1.7.5 Exceptions

exception `picamera.`**PiCameraError**
Base class for PiCamera errors

exception `picamera.`**PiCameraValueError**
Raised when an invalid value is fed to a PiCamera object

exception `picamera.`**PiCameraRuntimeError**
Raised when an invalid sequence of operations is attempted with a PiCamera object

1.8 Change log

1.8.1 Release 1.3 (2014-03-22)

1.3 was partly new functionality:

- The *bayer* parameter was added to the `'jpeg'` format in the capture methods to permit output of the camera's raw sensor data (#52)
- The `record_sequence()` method was added to provide a cleaner interface for recording multiple consecutive video clips (#53)
- The *splitter_port* parameter was added to all capture methods and `start_recording()` to permit recording multiple simultaneous video streams (presumably with different options, primarily *resize*) (#56)

- The limits on the `framerate` attribute were increased after firmware #656 introduced numerous new camera modes including 90fps recording (at lower resolutions) (#65)

And partly bug fixes:

- It was reported that Exif metadata (including thumbnails) wasn't fully recorded in JPEG output (#59)
- Raw captures with `capture_continuous()` and `capture_sequence()` were broken (#55)

1.8.2 Release 1.2 (2014-02-02)

1.2 was mostly a bug fix release:

- A bug introduced in 1.1 caused `split_recording()` to fail if it was preceded by a video-port-based image capture (#49)
- The documentation was enhanced to try and full explain the discrepancy between preview and capture resolution, and to provide some insight into the underlying workings of the camera (#23)
- A new property was introduced for configuring the preview's layer at runtime although this probably won't find use until OpenGL overlays are explored (#48)

1.8.3 Release 1.1 (2014-01-25)

1.1 was mostly a bug fix release:

- A nasty race condition was discovered which led to crashes with long-running processes (#40)
- An assertion error raised when performing raw captures with an active resize parameter was fixed (#46)
- A couple of documentation enhancements made it in (#41 and #47)

1.8.4 Release 1.0 (2014-01-11)

In 1.0 the major features added were:

- Debian packaging! (#12)
- The new `frame` attribute permits querying information about the frame last written to the output stream (number, timestamp, size, keyframe, etc.) (#34, #36)
- All capture methods (`capture()` et al), and the `start_recording()` method now accept a `resize` parameter which invokes a resizer prior to the encoding step (#21)
- A new `PiCameraCircularIO` stream class is provided to permit holding the last *n* seconds of video in memory, ready for writing out to disk (or whatever you like) (#39)
- There's a new way to specify raw captures - simply use the format you require with the capture method of your choice. As a result of this, the `raw_format` attribute is now deprecated (#32)

Some bugs were also fixed:

- GPIO.cleanup is no longer called on `close()` (#35), and GPIO set up is only done on first use of the `led` attribute which should resolve issues that users have been having with using picamera in conjunction with GPIO
- Raw RGB video-port based image captures are now working again too (#32)

As this is a new major-version, all deprecated elements were removed:

- The continuous method was removed; this was replaced by `capture_continuous()` in 0.5 (#7)

1.8.5 Release 0.8 (2013-12-09)

In 0.8 the major features added were:

- Capture of images whilst recording without frame-drop. Previously, images could be captured whilst recording but only from the still port which resulted in dropped frames in the recorded video due to the mode switch. In 0.8, `use_video_port=True` can be specified on capture methods whilst recording video to avoid this.
- Splitting of video recordings into multiple files. This is done via the new `split_recording()` method, and requires that the `start_recording()` method was called with *inline_headers* set to True. The latter has now been made the default (technically this is a backwards incompatible change, but it's relatively trivial and I don't anticipate anyone's code breaking because of this change).

In addition a few bugs were fixed:

- Documentation updates that were missing from 0.7 (specifically the new video recording parameters)
- The ability to perform raw captures through the video port
- Missing exception imports in the encoders module (which caused very confusing errors in the case that an exception was raised within an encoder thread)

1.8.6 Release 0.7 (2013-11-14)

0.7 is mostly a bug fix release, with a few new video recording features:

- Added `quantisation` and `inline_headers` options to `start_recording()` method
- Fixed bugs in the `crop` property
- The issue of captures fading to black over time when the preview is not running has been resolved. This solution was to permanently activate the preview, but pipe it to a null-sink when not required. Note that this means rapid capture gets even slower when not using the video port
- LED support is via RPi.GPIO only; the RPIO library simply doesn't support it at this time
- Numerous documentation fixes

1.8.7 Release 0.6 (2013-10-30)

In 0.6, the major features added were:

- New `'raw'` format added to all capture methods (`capture()`, `capture_continuous()`, and `capture_sequence()`) to permit capturing of raw sensor data
- New `raw_format` attribute to permit control of raw format (defaults to `'yuv'`, only other setting currently is `'rgb'`)
- New `shutter_speed` attribute to permit manual control of shutter speed (defaults to 0 for automatic shutter speed, and requires latest firmware to operate - use `sudo rpi-update` to upgrade)
- New "Recipes" chapter in the documentation which demonstrates a wide variety of capture techniques ranging from trivial to complex

1.8.8 Release 0.5 (2013-10-21)

In 0.5, the major features added were:

- New `capture_sequence()` method
- `continuous()` method renamed to `capture_continuous()`. Old method name retained for compatiblity until 1.0.

- *use_video_port* option for `capture_sequence()` and `capture_continuous()` to allow rapid capture of JPEGs via video port
- New `framerate` attribute to control video and rapid-image capture frame rates
- Default value for `ISO` changed from 400 to 0 (auto) which fixes `exposure_mode` not working by default
- *intraperiod* and *profile* options for `start_recording()`

In addition a few bugs were fixed:

- Byte strings not being accepted by `continuous()`
- Erroneous docs for `ISO`

Many thanks to the community for the bug reports!

1.8.9 Release 0.4 (2013-10-11)

In 0.4, several new attributes were introduced for configuration of the preview window:

- `preview_alpha`
- `preview_fullscreen`
- `preview_window`

Also, a new method for rapid continual capture of still images was introduced: `continuous()`.

1.8.10 Release 0.3 (2013-10-04)

The major change in 0.3 was the introduction of custom Exif tagging for captured images, and fixing a silly bug which prevented more than one image being captured during the lifetime of a PiCamera instance.

1.8.11 Release 0.2

The major change in 0.2 was support for video recording, along with the new `resolution` property which replaced the separate `preview_resolution` and `stills_resolution` properties.

1.9 License

LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

p

picamera, **??**