Picamera Documentation

Release 1.12

Dave Jones

January 30, 2017

Contents

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

CHAPTER 1

Links

- The code is licensed under the BSD license
- 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
- Packages can be downloaded from PyPI, but reading the installation instructions is more likely to be useful

Table of Contents

2.1 Installation

2.1.1 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 update
$ sudo apt-get install python-picamera 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 python-picamera python3-picamera
```

Note: If you are using a recent installation of Raspbian, you may find that the python-picamera package is already installed (it is included by default in recent versions).

2.1.2 Alternate distro installation

On distributions other than Raspbian, it is probably simplest to install system wide using Python's `pip` tool:

```
$ sudo pip install picamera
```

If you wish to use the classes in the *picamera.array* module then specify the "array" option which will pull in numpy as a dependency (be warned that building numpy takes a *long* time on a Pi):

```
$ sudo pip install "picamera[array]"
```

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
```

2.1.3 Firmware upgrades

The behaviour of the Pi's camera module is dictated by the Pi's firmware. Over time, considerable work has gone into fixing bugs and extending the functionality of the Pi's camera module through new firmware releases. Whilst the picamera library attempts to maintain backward compatibility with older Pi firmwares, it is only tested against the latest firmware at the time of release, and not all functionality may be available if you are running an older firmware. As an example, the *annotate_text* attribute relies on a recent firmware; older firmwares lacked the functionality.

You can determine the revision of your current firmware with the following command:

```
$ uname -a
```

The firmware revision is the number after the #:

```
Linux kermit 3.12.26+ #707 PREEMPT Sat Aug 30 17:39:19 BST 2014 armv6l GNU/Linux
                        /
                       /
  firmware revision --+
```

On Raspbian, the standard upgrade procedure should keep your firmware up to date:

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

Warning: Previously, these documents have suggested using the `rpi-update` utility to update the Pi's firmware; this is now discouraged. If you have previously used the `rpi-update` utility to update your firmware, you can switch back to using `apt` to manage it with the following commands:

```
$ sudo apt-get update
$ sudo apt-get install --reinstall libraspberrypi0 libraspberrypi-{bin,dev,doc} raspberrypi-boot
$ sudo rm /boot/.firmware_revision
```

You will need to reboot after doing so.

Note: Please note that the PiTFT screen (and similar GPIO-driven screens) requires a custom firmware for operation. This firmware lags behind the official firmware and at the time of writing lacks several features including long exposures and text overlays.

2.1.4 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 lsb-release build-essential git git-core \
    exuberant-ctags python-virtualenv python3-virtualenv python-dev \
    python3-dev libjpeg8-dev zlib1g-dev libav-tools \
    texlive-latex-recommended texlive-latex-extra texlive-fonts-recommended
$ virtualenv -p /usr/bin/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 clone:

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

For anybody wishing to hack on the project, I would strongly recommend reading through the *PiCamera* class' source, to get a handle on using the mmalobj layer. This is a layer introduced in picamera 1.11 to ease the usage of libmmal (the underlying library that picamera, raspistill, and raspivid all rely upon).

Beneath mmalobj is a ctypes translation of the libmmal headers but my hope is that most developers will never need to deal with this directly (thus, a working knowledge of C is hopefully no longer necessary to hack on picamera).

Various classes for specialized applications also exist (*PiCameraCircularIO*, *PiBayerArray*, etc.)

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)!

2.1.5 Test suite

If you wish to run the picamera test suite, follow the instructions in *Development installation* above and then make the "test" target within the sandbox:

```
$ source sandbox/bin/activate
(sandbox) $ cd picamera
(sandbox) $ make test
```

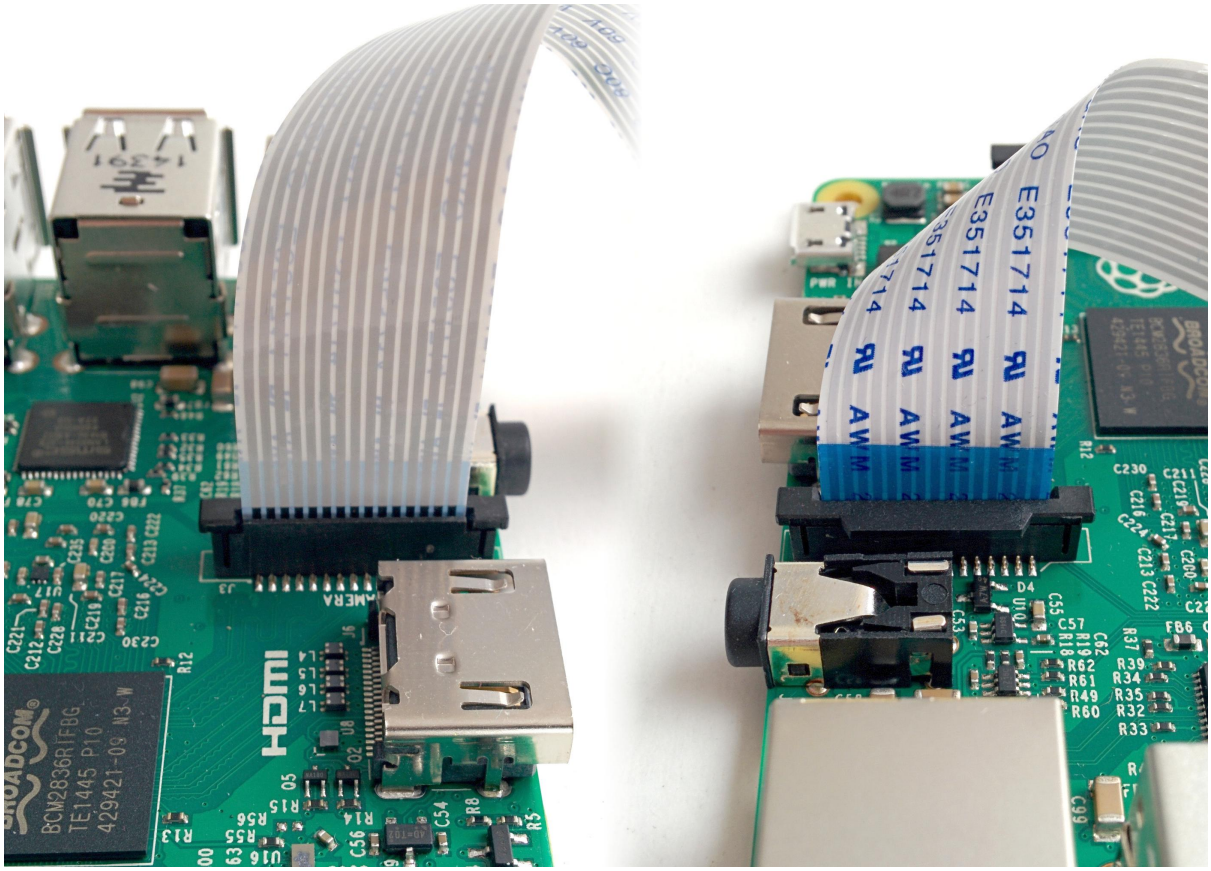
Warning: The test suite takes a *very* long time to execute (at least 1 hour on an overclocked Pi 3). Depending on configuration, it can also lockup the camera requiring a reboot to reset, so ensure you are familiar with SSH or using alternate TTYs to access a command line in the event you need to reboot.

2.2 Getting Started

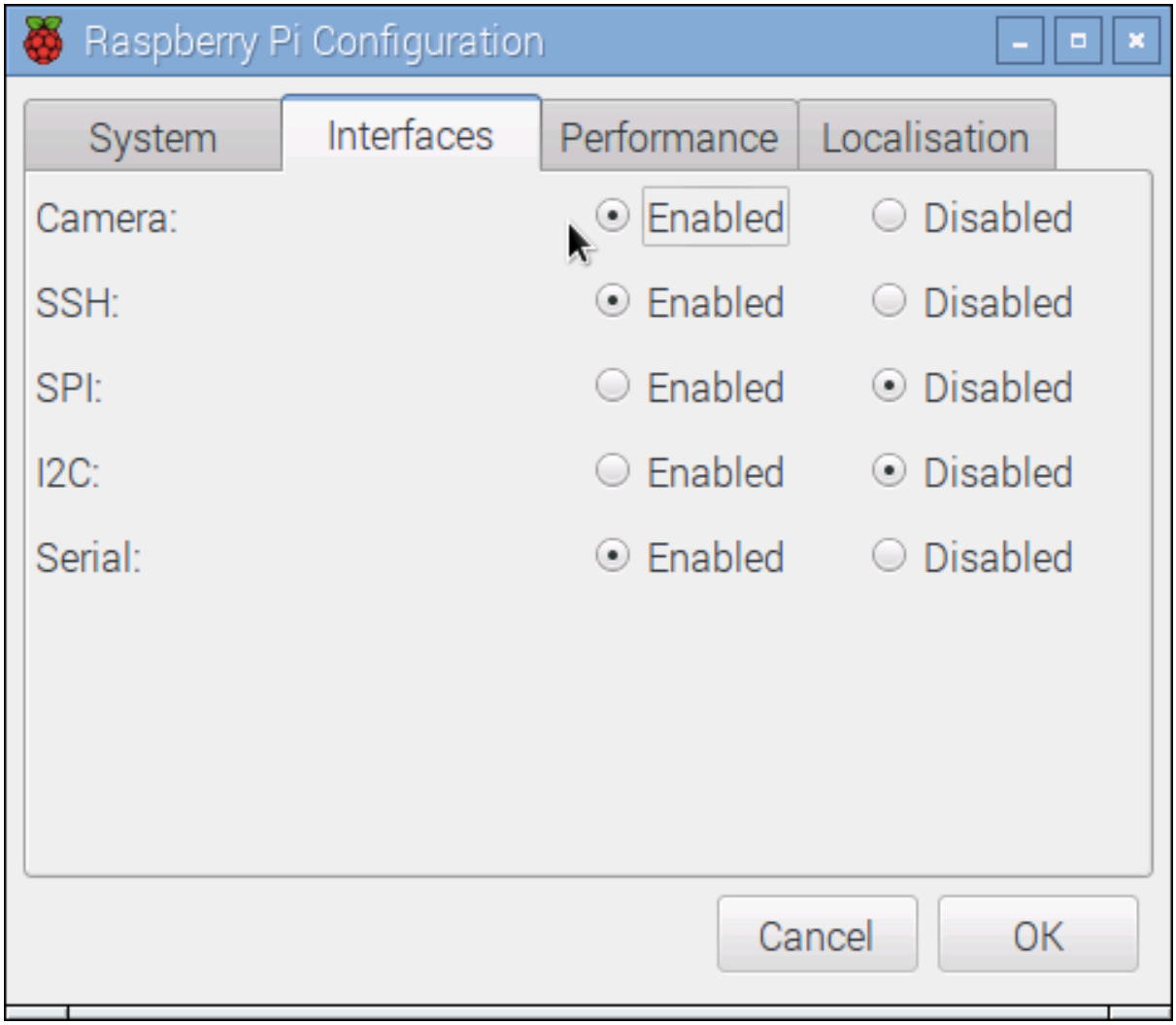
Warning: Make sure your Pi is off while installing the camera module. Although it is possible to install the camera while the Pi is on, this isn't good practice (if the camera is active when removed, it's possible to damage it).

Connect your camera module to the CSI port on your Raspberry Pi; this is the long thin port adjacent to the HDMI socket. Gently lift the collar on top of the CSI port (if it comes off, don't worry, you can push it back in but try to be more gentle in future!). Slide the ribbon cable of the camera module into the port with the blue side facing the Ethernet port (or where the Ethernet port would be if you've got a model A/A+).

Once the cable is seated in the port, press the collar back down to lock the cable in place. If done properly you should be able to easily lift the Pi by the camera's cable without it falling out. The following illustrations show a well-seated camera cable with the correct orientation:



Make sure the camera module isn't sat on anything conductive (e.g. the Pi's USB ports or its GPIO pins). Now, apply power to your Pi. Once booted, start the Raspberry Pi Configuration utility and enable the camera module:



You will need to reboot after doing this (but this is one-time setup so you won't need to do it again unless you re-install your operating system or switch SD cards). Once rebooted, start a terminal and try the following command:

```
raspistill -o image.jpg
```

If everything is working correctly, the camera should start, a preview from the camera should appear on the display and, after a 5 second delay it should capture an image (storing it as `image.jpg`) before shutting down the camera. Proceed to the *Basic Recipes*.

If something else happens, read any error message displayed and try any recommendations suggested by such messages. If your Pi reboots as soon as you run this command, your power supply is insufficient for running your Pi plus the camera module (and whatever other peripherals you have attached).

2.3 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.

2.3.1 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:

```
from time import sleep
from picamera import PiCamera

camera = PiCamera()
camera.resolution = (1024, 768)
camera.start_preview()
# Camera warm-up 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.

2.3.2 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:

```
from io import BytesIO
from time import sleep
from picamera import PiCamera

# Create an in-memory stream
my_stream = BytesIO()
camera = PiCamera()
camera.start_preview()
# Camera warm-up 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. However, if the object has a `flush` method, this will be called prior to capture returning. This should ensure that once capture returns the data is accessible to other processes although the object still needs to be closed:

```
from time import sleep
from picamera import PiCamera

# Explicitly open a new file called my_image.jpg
my_file = open('my_image.jpg', 'wb')
camera = PiCamera()
camera.start_preview()
sleep(2)
camera.capture(my_file)
# At this point my_file.flush() has been called, but the file has
# not yet been closed
my_file.close()
```

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). As well as using stream classes built into Python (like `BytesIO`) you can also construct your own *custom outputs*.

2.3.3 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:

```
from io import BytesIO
from time import sleep
from picamera import PiCamera
from PIL import Image

# Create the in-memory stream
stream = BytesIO()
camera = PiCamera()
camera.start_preview()
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)
```

2.3.4 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:

```
from time import sleep
from picamera import PiCamera

camera = PiCamera()
camera.resolution = (1024, 768)
camera.start_preview()
# Camera warm-up 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.

2.3.5 Capturing consistent images

You may wish to capture a sequence of images all of which look the same in terms of brightness, color, and contrast (this can be useful in timelapse photography, for example). Various attributes need to be used in order to ensure consistency across multiple shots. Specifically, you need to ensure that the camera's exposure time, white balance, and gains are all fixed:

- To fix exposure time, set the *shutter_speed* attribute to a reasonable value.
- Optionally, set *iso* to a fixed value.
- To fix exposure gains, let *analog_gain* and *digital_gain* settle on reasonable values, then set *exposure_mode* to `'off'`.
- To fix white balance, set the *awb_mode* to `'off'`, then set *awb_gains* to a (red, blue) tuple of gains.

It can be difficult to know what appropriate values might be for these attributes. For *iso*, a simple rule of thumb is that 100 and 200 are reasonable values for daytime, while 400 and 800 are better for low light. To determine a reasonable value for *shutter_speed* you can query the *exposure_speed* attribute. For exposure gains, it's usually enough to wait until *analog_gain* is greater than 1 (the default, which will produce entirely black frames) before *exposure_mode* is set to `'off'`. Finally, to determine reasonable values for *awb_gains* simply query the property while *awb_mode* is set to something other than `'off'`. Again, this will tell you the camera's white balance gains as determined by the auto-white-balance algorithm.

The following script provides a brief example of configuring these settings:

```
from time import sleep
from picamera import PiCamera

camera = PiCamera(resolution=(1280, 720), framerate=30)
# Set ISO to the desired value
camera.iso = 100
# Wait for the automatic gain control to settle
sleep(2)
# Now fix the values
camera.shutter_speed = camera.exposure_speed
camera.exposure_mode = 'off'
g = camera.awb_gains
camera.awb_mode = 'off'
camera.awb_gains = g
# Finally, take several photos with the fixed settings
camera.capture_sequence(['image%02d.jpg' % i for i in range(10)])
```

2.3.6 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:

```
from time import sleep
from picamera import PiCamera

camera = PiCamera()
camera.start_preview()
sleep(2)
for filename in camera.capture_continuous('img{counter:03d}.jpg'):
    print('Captured %s' % filename)
    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):

```
from time import sleep
from picamera 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
    sleep(delay)

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

2.3.7 Capturing in low light

Using similar tricks to those in *Capturing consistent images*, the Pi's camera can capture images in low light conditions. The primary objective is to set a high gain, and a long exposure time to allow the camera to gather as

much light as possible. However, the *shutter_speed* attribute is constrained by the camera's *framerate* so the first thing we need to do is set a very slow framerate. The following script captures an image with a 6 second exposure time (the maximum the Pi's V1 camera module is capable of; the V2 camera module can manage 10 second exposures):

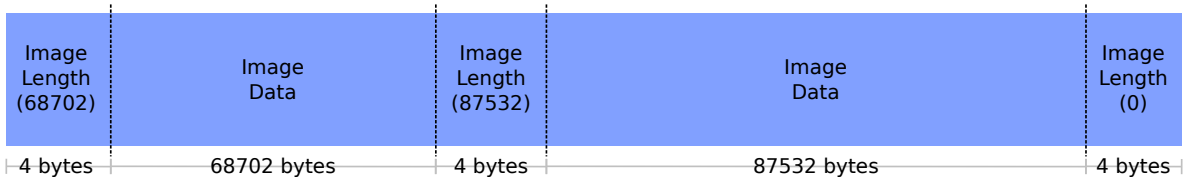
```
from picamera import PiCamera
from time import sleep
from fractions import Fraction

# Set a framerate of 1/6fps, then set shutter
# speed to 6s and ISO to 800
camera = PiCamera(resolution=(1280, 720), framerate=Fraction(1, 6))
camera.shutter_speed = 6000000
camera.iso = 800
# Give the camera a good long time to set gains and
# measure AWB (you may wish to use fixed AWB instead)
sleep(30)
camera.exposure_mode = 'off'
# Finally, capture an image with a 6s exposure. Due
# to mode switching on the still port, this will take
# longer than 6 seconds
camera.capture('dark.jpg')
```

In anything other than dark conditions, the image produced by this script will most likely be completely white or at least heavily over-exposed.

Note: The Pi's camera module uses a rolling shutter. This means that moving subjects may appear distorted if they move relative to the camera. This effect will be exaggerated by using longer exposure times.

2.3.8 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. We'll use a very simple protocol for communication: first the length of the image will be sent as a 32-bit integer (in Little Endian format), then this will be followed by the bytes of image data. If the length is 0, this indicates that the connection should be closed as no more images will be forthcoming. This protocol is illustrated below:



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(struct.calcsize('<L')))[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:
    camera = picamera.PiCamera()
    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.

2.3.9 Recording video to a file

Recording a video to a file is simple:

```
import picamera

camera = picamera.PiCamera()
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).

2.3.10 Recording video to a stream

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

```
from io import BytesIO
from picamera import PiCamera

stream = BytesIO()
camera = PiCamera()
camera.resolution = (640, 480)
camera.start_recording(stream, format='h264', quality=23)
camera.wait_recording(15)
camera.stop_recording()
```

Here, we've set the *quality* parameter to indicate to the encoder the level of image quality that we'd like it to try and maintain. The camera's H.264 encoder is primarily constrained by two parameters:

- *bitrate* limits the encoder's output to a certain number of bits per second. The default is 17000000 (17Mbps), and the maximum value is 25000000 (25Mbps). Higher values give the encoder more "freedom" to encode at higher qualities. You will likely find that the default doesn't constrain the encoder at all except at higher recording resolutions.
- *quality* tells the encoder what level of image quality to maintain. Values can be between 1 (highest quality) and 40 (lowest quality), with typical values providing a reasonable trade-off between bandwidth and quality being between 20 and 25.

As well as using stream classes built into Python (like BytesIO) you can also construct your own *custom outputs*. This is particularly useful for video recording, as discussed in the linked recipe.

2.3.11 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

camera = picamera.PiCamera(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

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

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

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

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

```
finally:
    camera.stop_recording()
```

In the above script we use the special *copy_to()* method to copy the stream to a disk file. This automatically handles details like finding the start of the first key-frame in the circular buffer, and also provides facilities like writing a specific number of bytes or seconds.

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.

Changed in version 1.11: Added use of the *copy_to()*

2.3.13 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', '25', '-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.

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:
    camera = picamera.PiCamera()
    camera.resolution = (640, 480)
    camera.framerate = 24
    # 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. It's also possible to reverse the direction of this recipe relatively easily. In this scenario, the Pi acts as the server, waiting for a connection from the client. When it accepts a connection, it starts streaming video over it for 60 seconds. Another variation (just for the purposes of demonstration) is that we initialize the camera straight away instead of waiting for a connection to allow the streaming to start faster on connection:

```
import socket
import time
import picamera

camera = picamera.PiCamera()
camera.resolution = (640, 480)
camera.framerate = 24

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('wb')
try:
    camera.start_recording(connection, format='h264')
```

```
    camera.wait_recording(60)
    camera.stop_recording()
finally:
    connection.close()
    server_socket.close()
```

One advantage of this setup is that no script is needed on the client side - we can simply use VLC with a network URL:

```
vlc tcp/h264://my_pi_address:8000/
```

Note: VLC (or mplayer) will *not* work for playback on a Pi. Neither is (currently) capable of using the GPU for decoding, and thus they attempt to perform video decoding on the Pi's CPU (which is not powerful enough for the task). You will need to run these applications on a faster machine (though "faster" is a relative term here: even an Atom powered netbook should be quick enough for the task at non-HD resolutions).

2.3.14 Overlaying images on the preview

The camera preview system can operate multiple layered renderers simultaneously. While the picamera library only permits a single renderer to be connected to the camera's preview port, it does permit additional renderers to be created which display a static image. These overlaid renderers can be used to create simple user interfaces.

Note: Overlay images will *not* appear in image captures or video recordings. If you need to embed additional information in the output of the camera, please refer to *Overlaying text on the output*.

One difficulty of working with overlay renderers is that they expect unencoded RGB input which is padded up to the camera's block size. The camera's block size is 32x16 so any image data provided to a renderer must have a width which is a multiple of 32, and a height which is a multiple of 16. The specific RGB format expected is interleaved unsigned bytes. If all this sounds complicated, don't worry; it's quite simple to produce in practice.

The following example demonstrates loading an arbitrary size image with PIL, padding it to the required size, and producing the unencoded RGB data for the call to *add_overlay()*:

```
import picamera
from PIL import Image
from time import sleep

camera = picamera.PiCamera()
camera.resolution = (1280, 720)
camera.framerate = 24
camera.start_preview()

# Load the arbitrarily sized image
img = Image.open('overlay.png')
# Create an image padded to the required size with
# mode 'RGB'
pad = Image.new('RGB', (
    ((img.size[0] + 31) // 32) * 32,
    ((img.size[1] + 15) // 16) * 16,
    ))
# Paste the original image into the padded one
pad.paste(img, (0, 0))

# Add the overlay with the padded image as the source,
# but the original image's dimensions
o = camera.add_overlay(pad.tostring(), size=img.size)
# By default, the overlay is in layer 0, beneath the
# preview (which defaults to layer 2). Here we make
```

```
# the new overlay semi-transparent, then move it above
# the preview
o.alpha = 128
o.layer = 3

# Wait indefinitely until the user terminates the script
while True:
    sleep(1)
```

Alternatively, instead of using an image file as the source, you can produce an overlay directly from a numpy array. In the following example, we construct a numpy array with the same resolution as the screen, then draw a white cross through the center and overlay it on the preview as a simple cross-hair:

```
import time
import picamera
import numpy as np

# Create an array representing a 1280x720 image of
# a cross through the center of the display. The shape of
# the array must be of the form (height, width, color)
a = np.zeros((720, 1280, 3), dtype=np.uint8)
a[360, :, :] = 0xff
a[:, 640, :] = 0xff

camera = picamera.PiCamera()
camera.resolution = (1280, 720)
camera.framerate = 24
camera.start_preview()
# Add the overlay directly into layer 3 with transparency;
# we can omit the size parameter of add_overlay as the
# size is the same as the camera's resolution
o = camera.add_overlay(np.getbuffer(a), layer=3, alpha=64)
try:
    # Wait indefinitely until the user terminates the script
    while True:
        time.sleep(1)
finally:
    camera.remove_overlay(o)
```

Given that overlaid renderers can be hidden (by moving them below the preview's *layer* which defaults to 2), made semi-transparent (with the *alpha* property), and resized so that they don't *fill the screen*, they can be used to construct simple user interfaces.

New in version 1.8.

2.3.15 Overlaying text on the output

The camera includes a rudimentary annotation facility which permits up to 255 characters of ASCII text to be overlaid on all output (including the preview, image captures and video recordings). To achieve this, simply assign a string to the *annotate_text* attribute:

```
import picamera
import time

camera = picamera.PiCamera()
camera.resolution = (640, 480)
camera.framerate = 24
camera.start_preview()
camera.annotate_text = 'Hello world!'
time.sleep(2)
# Take a picture including the annotation
camera.capture('foo.jpg')
```

With a little ingenuity, it's possible to display longer strings:

```
import picamera
import time
import itertools

s = "This message would be far too long to display normally..."

camera = picamera.PiCamera()
camera.resolution = (640, 480)
camera.framerate = 24
camera.start_preview()
camera.annotate_text = ' ' * 31
for c in itertools.cycle(s):
    camera.annotate_text = camera.annotate_text[1:31] + c
    time.sleep(0.1)
```

And of course, it can be used to display (and embed) a timestamp in recordings (this recipe also demonstrates drawing a background behind the timestamp for contrast with the *annotate_background* attribute):

```
import picamera
import datetime as dt

camera = picamera.PiCamera(resolution=(1280, 720), framerate=24)
camera.start_preview()
camera.annotate_background = picamera.Color('black')
camera.annotate_text = dt.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
camera.start_recording('timestamped.h264')
start = dt.datetime.now()
while (dt.datetime.now() - start).seconds < 30:
    camera.annotate_text = dt.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    camera.wait_recording(0.2)
camera.stop_recording()
```

New in version 1.7.

2.3.16 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). Another method is to use the `disable_camera_led` option in the boot configuration.

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

camera = picamera.PiCamera()
# 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.

2.4 Advanced Recipes

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

2.4.1 Capturing to a numpy array

Since 1.11, picamera can capture directly to any object which supports Python's buffer protocol (including numpy's `ndarray`). Simply pass the object as the destination of the capture and the image data will be written directly to the object. The target object must fulfil various requirements (some of which are dependent on the version of Python you are using):

1. The buffer object must be writeable (e.g. you cannot capture to a `bytes` object as it is immutable).
2. The buffer object must be large enough to receive all the image data.
3. (Python 2.x only) The buffer object must be 1-dimensional.
4. (Python 2.x only) The buffer object must have byte-sized items.

For example, to capture directly to a three-dimensional numpy `ndarray` (Python 3.x only):

```
import time
import picamera
import numpy as np

with picamera.PiCamera() as camera:
    camera.resolution = (320, 240)
    camera.framerate = 24
    time.sleep(2)
    output = np.empty((240, 320, 3), dtype=np.uint8)
    camera.capture(output, 'rgb')
```

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.

So, to capture a 100x100 image we first need to provide a 128x112 array, then strip off the uninitialized pixels afterward. The following example demonstrates this along with the re-shaping necessary under Python 2.x:

```
import time
import picamera
import numpy as np

with picamera.PiCamera() as camera:
    camera.resolution = (100, 100)
    camera.framerate = 24
    time.sleep(2)
    output = np.empty((112 * 128 * 3,), dtype=np.uint8)
    camera.capture(output, 'rgb')
    output = output.reshape((112, 128, 3))
    output = output[:100, :100, :]
```

Warning: Under certain circumstances (non-resized, non-YUV, video-port captures), the resolution is rounded to 16x16 blocks instead of 32x16. Adjust your resolution rounding accordingly.

New in version 1.11.

2.4.2 Capturing to an OpenCV object

This is a variation on *Capturing to a numpy array*. OpenCV uses numpy arrays as images and defaults to colors in planar BGR. Hence, the following is all that's required to capture an OpenCV compatible image (under Python 3.x):

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

with picamera.PiCamera() as camera:
    camera.resolution = (320, 240)
    camera.framerate = 24
    time.sleep(2)
    image = np.empty((240, 320, 3), dtype=np.uint8)
    camera.capture(image, 'bgr')
```

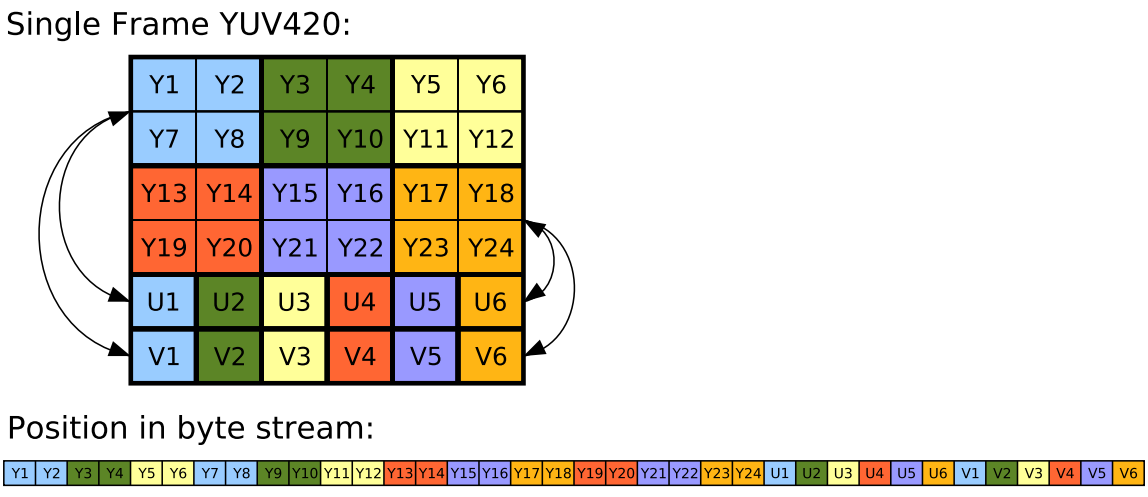
Changed in version 1.11: Replaced recipe with direct array capture example.

2.4.3 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). This is illustrated in the diagram below:



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:

```
   128.0  100 rounded up to nearest multiple of 32
×  112.0  100 rounded up to nearest multiple of 16
×    1.5  bytes of data per pixel in YUV4:2:0 format
= 21504.0  bytes
```

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', 'w+b')
# 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)
```

Note: You may note that we are using `open()` in the code above instead of `io.open()` as in the other examples. This is because numpy's `numpy.fromfile()` method annoyingly only accepts "real" file objects.

This recipe is now encapsulated in the *PiYUVArray* class in the *picamera.array* module, which means the same can be achieved as follows:

```
import time
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    with picamera.array.PiYUVArray(camera) as stream:
        camera.resolution = (100, 100)
        camera.start_preview()
        time.sleep(2)
        camera.capture(stream, 'yuv')
        # Show size of YUV data
        print(stream.array.shape)
        # Show size of RGB converted data
        print(stream.rgb_array.shape)
```

As of 1.11 you can also capture directly to numpy arrays (see *Capturing to a numpy array*). Due to the difference in resolution of the Y and UV components, this isn't directly useful (if you need all three components, you're better off using *PiYUVArray* as this rescales the UV components for convenience). However, if you only require the Y plane you can provide a buffer just large enough for this plane and ignore the error that occurs when writing to the buffer (picamera will deliberately write as much as it can to the buffer before raising an exception to support this use-case):

```
import time
import picamera
import picamera.array
import numpy as np

with picamera.PiCamera() as camera:
    camera.resolution = (100, 100)
    time.sleep(2)
    y_data = np.empty((112, 128), dtype=np.uint8)
    try:
        camera.capture(y_data, 'yuv')
    except IOError:
        pass
    y_data = y_data[:100, :100]
    # y_data now contains the Y-plane only
```

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. See *Raw Bayer data captures* for more information.

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.

Changed in version 1.5: Added note about new *picamera.array* module.

Changed in version 1.11: Added instructions for direct array capture.

2.4.4 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:

```
  128.0  100 rounded up to nearest multiple of 32
× 112.0  100 rounded up to nearest multiple of 16
×   3.0  bytes of data per pixel in RGB format
= 43008.0  bytes
```

Warning: Under certain circumstances (non-resized, non-YUV, video-port captures), the resolution is rounded to 16x16 blocks instead of 32x16. Adjust your resolution rounding accordingly.

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.

As the planes in RGB data are all equally sized (in contrast to YUV420) it is trivial to capture directly into a numpy array (Python 3.x only; see *Capturing to a numpy array* for Python 2.x instructions):

```
import time
import picamera
import picamera.array
import numpy as np

with picamera.PiCamera() as camera:
    camera.resolution = (100, 100)
    time.sleep(2)
    image = np.empty((128, 112, 3), dtype=np.uint8)
    camera.capture(image, 'rgb')
    image = image[:100, :100]
```

Note: RGB captures from the still port do not work at the full resolution of the camera (they result in an out of memory error). Either use YUV captures, or capture from the video port if you require full resolution.

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.

Changed in version 1.5: Added note about new *picamera.array* module.

Changed in version 1.11: Added instructions for direct array capture.

2.4.5 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 discussion in *Camera Modes*).
- No Exif information is embedded in JPEG images captured through the video-port.
- Captures typically appear "grainier" with this technique. Captures from the still port use a slower, more intensive denoise algorithm.

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 particularly 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 firstly that the Raspberry Pi's IO bandwidth is extremely limited and secondly that, 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).

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. Alternatively, you may wish to investigate sending the data over the network (which typically has more bandwidth available than the SD card interface) and having another machine perform any required processing. However, if you can perform your processing fast enough, you may not need to involve the disk or network 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:

```
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:
            # Wait for an image to be written to the stream
            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:
            if pool:
                processor = pool.pop()
            else:
                processor = None
        if processor:
            yield processor.stream
            processor.event.set()
        else:
            # When the pool is starved, wait a while for it to refill
            time.sleep(0.1)

with picamera.PiCamera() as camera:
    pool = [ImageProcessor() for i in range(4)]
    camera.resolution = (640, 480)
    camera.framerate = 30
    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()
```

2.4.6 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 set *use_video_port* to `True` in 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)
        camera.framerate = 30
        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 10fps of streaming at 640x480 on firmware #685. 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*:

```
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_lock = threading.Lock()
    pool = []

    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:
                # Wait for the image to be written to the stream
                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:
                if pool:
                    streamer = pool.pop()
                else:
                    streamer = None
            if streamer:
                yield streamer.stream
                streamer.event.set()
                count += 1
            else:
                # When the pool is starved, wait a while for it to refill
                time.sleep(0.1)
            finish = time.time()

    with picamera.PiCamera() as camera:
        pool = [ImageStreamer() for i in range(4)]
        camera.resolution = (640, 480)
        camera.framerate = 30
        time.sleep(2)
        start = time.time()
        camera.capture_sequence(streams(), 'jpeg', use_video_port=True)

    # Shut down the streamers in an orderly fashion
    while pool:
        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 %d seconds at %.2ffps' % (
    count, finish-start, count / (finish-start)))
```

On the same firmware, the above script achieves about 15fps. It is possible the new high framerate modes may achieve more (the fact that 15fps is half of the specified 30fps framerate suggests some stall on every other frame).

2.4.7 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 images captured via the still port 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 should not have dropped frames, assuming the image can be produced before the next video frame is due:

```
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. Higher resolutions or non-JPEG image formats may still cause dropped frames (only JPEG encoding is hardware accelerated).

2.4.8 Recording at multiple resolutions

The camera is capable of recording multiple streams at different resolutions simultaneously by use of the video splitter. This is probably most useful for performing analysis on a low-resolution stream, while simultaneously recording a high resolution stream for storage or viewing.

The following simple recipe demonstrates using the *splitter_port* parameter of the *start_recording()* method to begin two simultaneous recordings, each with a different resolution:

```
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (1024, 768)
    camera.framerate = 30
    camera.start_recording('highres.h264')
    camera.start_recording('lowres.h264', splitter_port=2, resize=(320, 240))
    camera.wait_recording(30)
    camera.stop_recording(splitter_port=2)
    camera.stop_recording()
```

There are 4 splitter ports in total that can be used (numbered 0, 1, 2, and 3). The video recording methods default to using splitter port 1, while the image capture methods default to splitter port 0 (when the *use_video_port* parameter is also True). A splitter port cannot be simultaneously used for video recording and image capture so you are advised to avoid splitter port 0 for video recordings unless you never intend to capture images whilst recording.

New in version 1.3.

2.4.9 Recording motion vector data

The Pi's camera is capable of outputting the motion vector estimates that the camera's H.264 encoder calculates while generating compressed video. These can be directed to a separate output file (or file-like object) with the *motion_output* parameter of the *start_recording()* method. Like the normal *output* parameter this accepts a string representing a filename, or a file-like object:

```
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (640, 480)
    camera.framerate = 30
    camera.start_recording('motion.h264', motion_output='motion.data')
    camera.wait_recording(10)
    camera.stop_recording()
```

Motion data is calculated at the macro-block level (an MPEG macro-block represents a 16x16 pixel region of the frame), and includes one extra column of data. Hence, if the camera's resolution is 640x480 (as in the example above) there will be 41 columns of motion data ((640 / 16) + 1), in 30 rows (480 / 16).

Motion data values are 4-bytes long, consisting of a signed 1-byte x vector, a signed 1-byte y vector, and an unsigned 2-byte SAD (Sum of Absolute Differences) value for each macro-block. Hence in the example above, each frame will generate 4920 bytes of motion data (41 * 30 * 4). Assuming the data contains 300 frames (in practice it may contain a few more) the motion data should be 1,476,000 bytes in total.

The following code demonstrates loading the motion data into a three-dimensional numpy array. The first dimension represents the frame, with the latter two representing rows and finally columns. A structured data-type is used for the array permitting easy access to x, y, and SAD values:

```
from __future__ import division

import numpy as np

width = 640
height = 480
cols = (width + 15) // 16
cols += 1 # there's always an extra column
rows = (height + 15) // 16

motion_data = np.fromfile(
    'motion.data', dtype=[
        ('x', 'i1'),
        ('y', 'i1'),
        ('sad', 'u2'),
        ])
frames = motion_data.shape[0] // (cols * rows)
motion_data = motion_data.reshape((frames, rows, cols))

# Access the data for the first frame
motion_data[0]

# Access just the x-vectors from the fifth frame
motion_data[4]['x']

# Access SAD values for the tenth frame
motion_data[9]['sad']
```

You can calculate the amount of motion the vector represents simply by calculating the magnitude of the vector with Pythagoras' theorem. The SAD (Sum of Absolute Differences) value can be used to determine how well the encoder thinks the vector represents the original reference frame.

The following code extends the example above to use PIL to produce a PNG image from the magnitude of each frame's motion vectors:

```
from __future__ import division

import numpy as np
from PIL import Image

width = 640
```

```
height = 480
cols = (width + 15) // 16
cols += 1
rows = (height + 15) // 16

m = np.fromfile(
    'motion.data', dtype=[
        ('x', 'i1'),
        ('y', 'i1'),
        ('sad', 'u2'),
        ])
frames = m.shape[0] // (cols * rows)
m = m.reshape((frames, rows, cols))

for frame in range(frames):
    data = np.sqrt(
        np.square(m[frame]['x'].astype(np.float)) +
        np.square(m[frame]['y'].astype(np.float))
        ).clip(0, 255).astype(np.uint8)
    img = Image.fromarray(data)
    filename = 'frame%03d.png' % frame
    print('Writing %s' % filename)
    img.save(filename)
```

You may wish to investigate the *PiMotionArray* class in the *picamera.array* module which simplifies the above recipes to the following:

```
import numpy as np
import picamera
import picamera.array
from PIL import Image

with picamera.PiCamera() as camera:
    with picamera.array.PiMotionArray(camera) as stream:
        camera.resolution = (640, 480)
        camera.framerate = 30
        camera.start_recording('/dev/null', format='h264', motion_output=stream)
        camera.wait_recording(10)
        camera.stop_recording()
        for frame in range(stream.array.shape[0]):
            data = np.sqrt(
                np.square(stream.array[frame]['x'].astype(np.float)) +
                np.square(stream.array[frame]['y'].astype(np.float))
                ).clip(0, 255).astype(np.uint8)
            img = Image.fromarray(data)
            filename = 'frame%03d.png' % frame
            print('Writing %s' % filename)
            img.save(filename)
```

Finally, the following command line can be used to generate an animation from the generated PNGs with ffmpeg (this will take a *very* long time on the Pi so you may wish to transfer the images to a faster machine for this step):

```
avconv -r 30 -i frame%03d.png -filter:v scale=640:480 -c:v libx264 motion.mp4
```

New in version 1.5.

2.4.10 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

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
                stream.copy_to('before.h264', seconds=10)
                stream.clear()
                # 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 using the *seconds* parameter of the *copy_to()* method to limit the before file to 10 seconds of data (given that the circular buffer may contain considerably more than this).

New in version 1.0.

Changed in version 1.11: Added use of *copy_to()*

2.4.11 Custom outputs

All methods in the picamera library which accept a filename also accept file-like objects. Typically, this is only used with actual file objects, or with memory streams (like `io.BytesIO`). However, building a custom output object is extremely easy and in certain cases very useful. A file-like object (as far as picamera is concerned) is simply an object with a `write` method which must accept a single parameter consisting of a byte-string, and which can optionally return the number of bytes written. The object can optionally implement a `flush` method (which has no parameters), which will be called at the end of output.

Custom outputs are particularly useful with video recording as the custom output's `write` method will be called (at least) once for every frame that is output, allowing you to implement code that reacts to each and every frame without going to the bother of a full *custom encoder*. However, one should bear in mind that because the `write` method is called so frequently, its implementation must be sufficiently rapid that it doesn't stall the encoder (it must perform its processing and return before the next write is due to arrive).

The following trivial example demonstrates an incredibly simple custom output which simply throws away the output while counting the number of bytes that would have been written and prints this at the end of the output:

```
from __future__ import print_function

import picamera

class MyOutput(object):
    def __init__(self):
        self.size = 0

    def write(self, s):
        self.size += len(s)

    def flush(self):
        print('%d bytes would have been written' % self.size)

with picamera.PiCamera() as camera:
    camera.resolution = (640, 480)
    camera.framerate = 60
    camera.start_recording(MyOutput(), format='h264')
    camera.wait_recording(10)
    camera.stop_recording()
```

The following example shows how to use a custom output to construct a crude motion detection system. We construct a custom output object which is used as the destination for motion vector data (this is particularly simple as motion vector data always arrives as single chunks; frame data by contrast sometimes arrives in several separate chunks). The output object doesn't actually write the motion data anywhere; instead it loads it into a numpy array and analyses whether there are any significantly large vectors in the data, printing a message to the console if there are. As we are not concerned with keeping the actual video output in this example, we use `/dev/null` as the destination for the video data:

```
from __future__ import division

import picamera
import numpy as np

motion_dtype = np.dtype([
    ('x', 'i1'),
    ('y', 'i1'),
    ('sad', 'u2'),
    ])

class MyMotionDetector(object):
    def __init__(self, camera):
        width, height = camera.resolution
        self.cols = (width + 15) // 16
        self.cols += 1 # there's always an extra column
```

```
        self.rows = (height + 15) // 16

    def write(self, s):
        # Load the motion data from the string to a numpy array
        data = np.fromstring(s, dtype=motion_dtype)
        # Re-shape it and calculate the magnitude of each vector
        data = data.reshape((self.rows, self.cols))
        data = np.sqrt(
            np.square(data['x'].astype(np.float)) +
            np.square(data['y'].astype(np.float))
            ).clip(0, 255).astype(np.uint8)
        # If there're more than 10 vectors with a magnitude greater
        # than 60, then say we've detected motion
        if (data > 60).sum() > 10:
            print('Motion detected!')
        # Pretend we wrote all the bytes of s
        return len(s)

with picamera.PiCamera() as camera:
    camera.resolution = (640, 480)
    camera.framerate = 30
    camera.start_recording(
        # Throw away the video data, but make sure we're using H.264
        '/dev/null', format='h264',
        # Record motion data to our custom output object
        motion_output=MyMotionDetector(camera)
        )
    camera.wait_recording(30)
    camera.stop_recording()
```

You may wish to investigate the classes in the *picamera.array* module which implement several custom outputs for analysis of data with numpy. In particular, the *PiMotionAnalysis* class can be used to remove much of the boiler plate code from the recipe above:

```
import picamera
import picamera.array
import numpy as np

class MyMotionDetector(picamera.array.PiMotionAnalysis):
    def analyse(self, a):
        a = np.sqrt(
            np.square(a['x'].astype(np.float)) +
            np.square(a['y'].astype(np.float))
            ).clip(0, 255).astype(np.uint8)
        # If there're more than 10 vectors with a magnitude greater
        # than 60, then say we've detected motion
        if (a > 60).sum() > 10:
            print('Motion detected!')

with picamera.PiCamera() as camera:
    camera.resolution = (640, 480)
    camera.framerate = 30
    camera.start_recording(
        '/dev/null', format='h264',
        motion_output=MyMotionDetector(camera)
        )
    camera.wait_recording(30)
    camera.stop_recording()
```

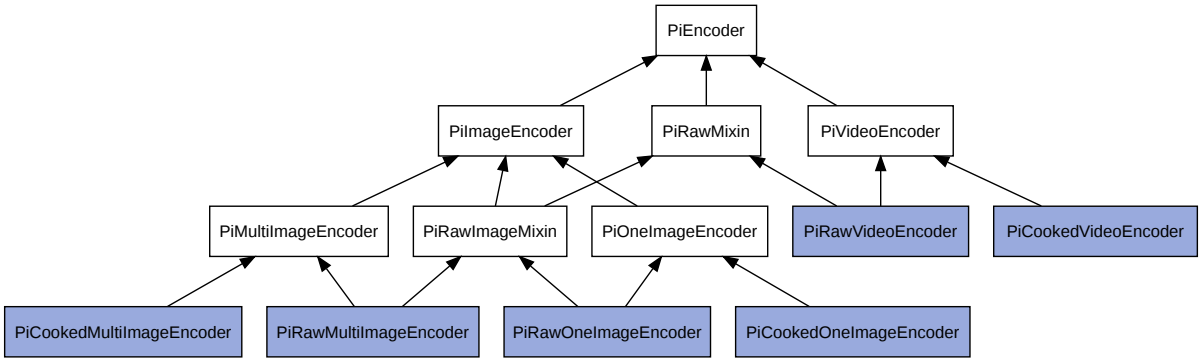
New in version 1.5.

2.4.12 Custom encoders

You can override and/or extend the encoder classes used during image or video capture. This is particularly useful with video capture as it allows you to run your own code in response to every frame, although naturally whatever code runs within the encoder's callback has to be reasonably quick to avoid stalling the encoder pipeline.

Writing a custom encoder is quite a bit harder than writing a *custom output* and in most cases there's little benefit. The only thing a custom encoder gives you that a custom output doesn't is access to the buffer header flags. For many output formats (MJPEG and YUV for example), these won't tell you anything interesting (i.e. they'll simply indicate that the buffer contains a full frame and nothing else). Currently, the only format where the buffer header flags contain useful information is H.264. Even then, most of the information (I-frame, P-frame, motion information, etc.) would be accessible from the *frame* attribute which you could access from your custom output's `write` method.

The encoder classes defined by picamera form the following hierarchy (shaded classes are actually instantiated by the implementation in picamera, white classes implement base functionality but aren't technically "abstract"):



The following table details which *PiCamera* methods use which encoder classes, and which method they call to construct these encoders:

Method(s)	Call	Returns
capture() *capture_continuous()* *capture_sequence()*	_get_image_encoder()	*PiCookedOneImageEncoder* *PiRawOneImageEncoder*
capture_sequence()	_get_images_encoder()	*PiCookedMultiImageEncoder* *PiRawMultiImageEncoder*
start_recording() *record_sequence()*	_get_video_encoder()	*PiCookedVideoEncoder* *PiRawVideoEncoder*

It is recommended, particularly in the case of the image encoder classes, that you familiarize yourself with the specific function of these classes so that you can determine the best class to extend for your particular needs. You may find that one of the intermediate classes is a better basis for your own modifications.

In the following example recipe we will extend the *PiCookedVideoEncoder* class to store how many I-frames and P-frames are captured (the camera's encoder doesn't use B-frames):

```
import picamera
import picamera.mmal as mmal


# Override PiVideoEncoder to keep track of the number of each type of frame
class MyEncoder(picamera.PiCookedVideoEncoder):
    def start(self, output, motion_output=None):
        self.parent.i_frames = 0
        self.parent.p_frames = 0
        super(MyEncoder, self).start(output, motion_output)

    def _callback_write(self, buf):
        # Only count when buffer indicates it's the end of a frame, and
        # it's not an SPS/PPS header (..._CONFIG)
```

```
        if (
                (buf[0].flags & mmal.MMAL_BUFFER_HEADER_FLAG_FRAME_END) and
                not (buf[0].flags & mmal.MMAL_BUFFER_HEADER_FLAG_CONFIG)
            ):
            if buf[0].flags & mmal.MMAL_BUFFER_HEADER_FLAG_KEYFRAME:
                self.parent.i_frames += 1
            else:
                self.parent.p_frames += 1
        # Remember to return the result of the parent method!
        return super(MyEncoder, self)._callback_write(buf)


# Override PiCamera to use our custom encoder for video recording
class MyCamera(picamera.PiCamera):
    def __init__(self):
        super(MyCamera, self).__init__()
        self.i_frames = 0
        self.p_frames = 0

    def _get_video_encoder(
            self, camera_port, output_port, format, resize, **options):
        return MyEncoder(
                self, camera_port, output_port, format, resize, **options)


with MyCamera() as camera:
    camera.start_recording('foo.h264')
    camera.wait_recording(10)
    camera.stop_recording()
    print('Recording contains %d I-frames and %d P-frames' % (
            camera.i_frames, camera.p_frames))
```

Please note that the above recipe is flawed: PiCamera is capable of initiating *multiple simultaneous recordings*. If this were used with the above recipe, then each encoder would wind up incrementing the `i_frames` and `p_frames` attributes on the `MyCamera` instance leading to incorrect results.

New in version 1.5.

2.4.13 Raw Bayer data captures

The `bayer` parameter of the *`capture()`* method causes the raw Bayer data recorded by the camera's sensor to be output as part of the image meta-data.

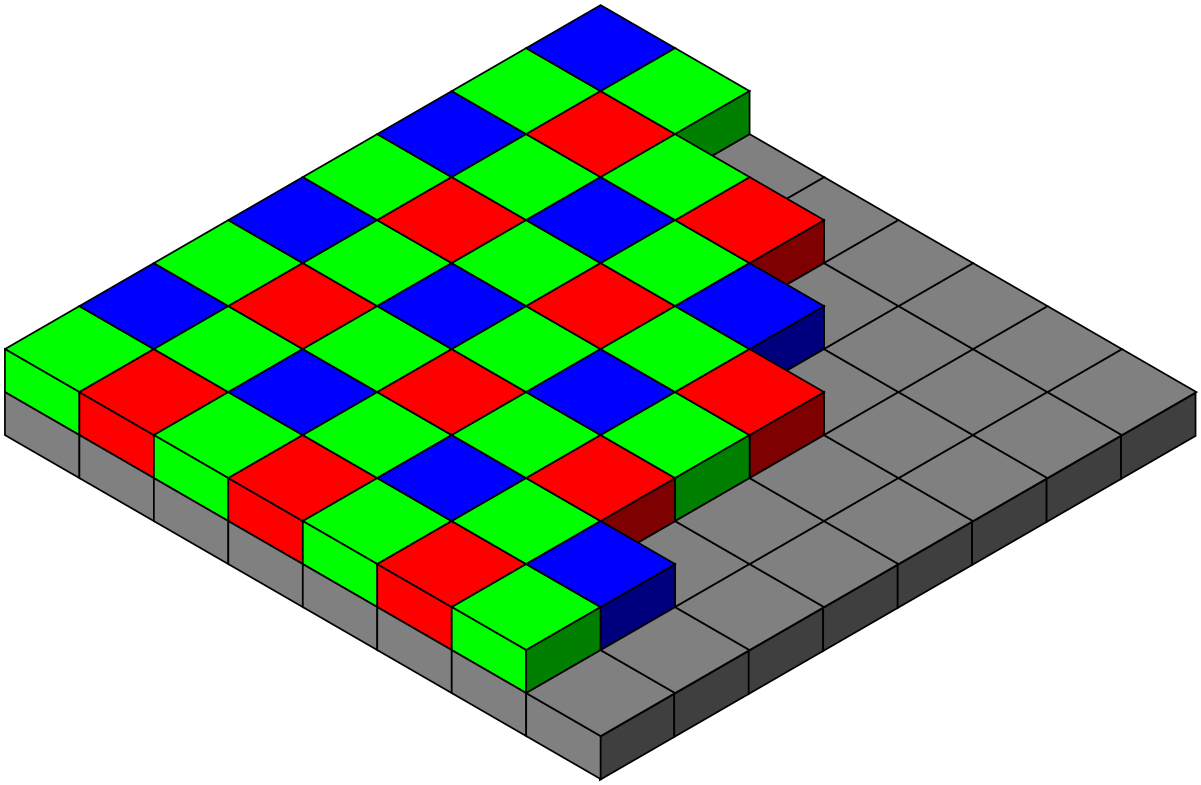
Note: The `bayer` parameter only operates with the JPEG format, and only for captures from the still port (i.e. when `use_video_port` is False, as it is by default).

Raw Bayer data differs considerably from simple unencoded captures; it is the data recorded by the camera's sensor prior to *any* GPU processing including auto white balance, vignette compensation, smoothing, down-scaling, etc. This also means:

- Bayer data is *always* full resolution, regardless of the camera's output *`resolution`* and any `resize` parameter.
- Bayer data occupies the last 6,404,096 bytes of the output file for the V1 module, or the last 10,270,208 bytes for the V2 module. The first 32,768 bytes of this is header data which starts with the string `'BRCM'`.
- Bayer data consists of 10-bit values, because this is the sensitivity of the OV5647 and IMX219 sensors used in the Pi's camera modules. The 10-bit values are organized as 4 8-bit values, followed by the low-order 2-bits of the 4 values packed into a fifth byte.

Bytes	Bits: MSB 8	7	6	5	4	3	2	LSB 1
1	10	9	8	7	6	5	4	3
2	10	9	8	7	6	5	4	3
3	10	9	8	7	6	5	4	3
4	10	9	8	7	6	5	4	3
5	2	1	2	1	2	1	2	1

- Bayer data is organized in a BGGR pattern (a minor variation of the common Bayer CFA). The raw data therefore has twice as many green pixels as red or blue and if viewed "raw" will look distinctly strange (too dark, too green, and with zippering effects along any straight edges).



- To make a "normal" looking image from raw Bayer data you will need to perform de-mosaicing at the very least, and probably some form of color balance.

This (heavily commented) example script causes the camera to capture an image including the raw Bayer data. It then proceeds to unpack the Bayer data into a 3-dimensional numpy array representing the raw RGB data and finally performs a rudimentary de-mosaic step with weighted averages. A couple of numpy tricks are used to improve performance but bear in mind that all processing is happening on the CPU and will be considerably slower than normal image captures:

```
from __future__ import (
    unicode_literals,
    absolute_import,
    print_function,
    division,
    )


import io
import time
import picamera
```

```
import numpy as np
from numpy.lib.stride_tricks import as_strided

stream = io.BytesIO()
with picamera.PiCamera() as camera:
    # Let the camera warm up for a couple of seconds
    time.sleep(2)
    # Capture the image, including the Bayer data
    camera.capture(stream, format='jpeg', bayer=True)
    ver = {
        'RP_ov5647': 1,
        'RP_imx219': 2,
        }[camera.exif_tags['IFD0.Model']]

# Extract the raw Bayer data from the end of the stream, check the
# header and strip if off before converting the data into a numpy array

offset = {
    1: 6404096,
    2: 10270208,
    }[ver]
data = stream.getvalue()[-offset:]
assert data[:4] == 'BRCM'
data = data[32768:]
data = np.fromstring(data, dtype=np.uint8)

# For the V1 module, the data consists of 1952 rows of 3264 bytes of data.
# The last 8 rows of data are unused (they only exist because the maximum
# resolution of 1944 rows is rounded up to the nearest 16).
#
# For the V2 module, the data consists of 2480 rows of 4128 bytes of data.
# There's actually 2464 rows of data, but the sensor's raw size is 2466
# rows, rounded up to the nearest multiple of 16: 2480.
#
# Likewise, the last few bytes of each row are unused (why?). Here we
# reshape the data and strip off the unused bytes.

reshape, crop = {
    1: ((1952, 3264), (1944, 3240)),
    2: ((2480, 4128), (2464, 4100)),
    }[ver]
data = data.reshape(reshape)[:crop[0], :crop[1]]

# Horizontally, each row consists of 10-bit values. Every four bytes are
# the high 8-bits of four values, and the 5th byte contains the packed low
# 2-bits of the preceding four values. In other words, the bits of the
# values A, B, C, D and arranged like so:
#
#  byte 1   byte 2   byte 3   byte 4   byte 5
# AAAAAAAA BBBBBBBB CCCCCCCC DDDDDDDD AABBCCDD
#
# Here, we convert our data into a 16-bit array, shift all values left by
# 2-bits and unpack the low-order bits from every 5th byte in each row,
# then remove the columns containing the packed bits

data = data.astype(np.uint16) << 2
for byte in range(4):
    data[:, byte::5] |= ((data[:, 4::5] >> ((4 - byte) * 2)) & 0b11)
data = np.delete(data, np.s_[4::5], 1)

# Now to split the data up into its red, green, and blue components. The
# Bayer pattern of the OV5647 sensor is BGGR. In other words the first
# row contains alternating green/blue elements, the second row contains
```

```
# alternating red/green elements, and so on as illustrated below:
#
# GBGBGBGBGBGBGB
# RGRGRGRGRGRGRG
# GBGBGBGBGBGBGB
# RGRGRGRGRGRGRG
#
# Please note that if you use vflip or hflip to change the orientation
# of the capture, you must flip the Bayer pattern accordingly

rgb = np.zeros(data.shape + (3,), dtype=data.dtype)
rgb[1::2, 0::2, 0] = data[1::2, 0::2] # Red
rgb[0::2, 0::2, 1] = data[0::2, 0::2] # Green
rgb[1::2, 1::2, 1] = data[1::2, 1::2] # Green
rgb[0::2, 1::2, 2] = data[0::2, 1::2] # Blue

# At this point we now have the raw Bayer data with the correct values
# and colors but the data still requires de-mosaicing and
# post-processing. If you wish to do this yourself, end the script here!
#
# Below we present a fairly naive de-mosaic method that simply
# calculates the weighted average of a pixel based on the pixels
# surrounding it. The weighting is provided by a byte representation of
# the Bayer filter which we construct first:

bayer = np.zeros(rgb.shape, dtype=np.uint8)
bayer[1::2, 0::2, 0] = 1 # Red
bayer[0::2, 0::2, 1] = 1 # Green
bayer[1::2, 1::2, 1] = 1 # Green
bayer[0::2, 1::2, 2] = 1 # Blue

# Allocate an array to hold our output with the same shape as the input
# data. After this we define the size of window that will be used to
# calculate each weighted average (3x3). Then we pad out the rgb and
# bayer arrays, adding blank pixels at their edges to compensate for the
# size of the window when calculating averages for edge pixels.

output = np.empty(rgb.shape, dtype=rgb.dtype)
window = (3, 3)
borders = (window[0] - 1, window[1] - 1)
border = (borders[0] // 2, borders[1] // 2)

rgb_pad = np.zeros((
    rgb.shape[0] + borders[0],
    rgb.shape[1] + borders[1],
    rgb.shape[2]), dtype=rgb.dtype)
rgb_pad[
    border[0]:rgb_pad.shape[0] - border[0],
    border[1]:rgb_pad.shape[1] - border[1],
    :] = rgb
rgb = rgb_pad

bayer_pad = np.zeros((
    bayer.shape[0] + borders[0],
    bayer.shape[1] + borders[1],
    bayer.shape[2]), dtype=bayer.dtype)
bayer_pad[
    border[0]:bayer_pad.shape[0] - border[0],
    border[1]:bayer_pad.shape[1] - border[1],
    :] = bayer
bayer = bayer_pad

# In numpy >=1.7.0 just use np.pad (version in Raspbian is 1.6.2 at the
```

```
# time of writing...)
#
#rgb = np.pad(rgb, [
#    (border[0], border[0]),
#    (border[1], border[1]),
#    (0, 0),
#    ], 'constant')
#bayer = np.pad(bayer, [
#    (border[0], border[0]),
#    (border[1], border[1]),
#    (0, 0),
#    ], 'constant')

# For each plane in the RGB data, we use a nifty numpy trick
# (as_strided) to construct a view over the plane of 3x3 matrices. We do
# the same for the bayer array, then use Einstein summation on each
# (np.sum is simpler, but copies the data so it's slower), and divide
# the results to get our weighted average:

for plane in range(3):
    p = rgb[..., plane]
    b = bayer[..., plane]
    pview = as_strided(p, shape=(
        p.shape[0] - borders[0],
        p.shape[1] - borders[1]) + window, strides=p.strides * 2)
    bview = as_strided(b, shape=(
        b.shape[0] - borders[0],
        b.shape[1] - borders[1]) + window, strides=b.strides * 2)
    psum = np.einsum('ijkl->ij', pview)
    bsum = np.einsum('ijkl->ij', bview)
    output[..., plane] = psum // bsum

# At this point output should contain a reasonably "normal" looking
# image, although it still won't look as good as the camera's normal
# output (as it lacks vignette compensation, AWB, etc).
#
# If you want to view this in most packages (like GIMP) you'll need to
# convert it to 8-bit RGB data. The simplest way to do this is by
# right-shifting everything by 2-bits (yes, this makes all that
# unpacking work at the start rather redundant...)

output = (output >> 2).astype(np.uint8)
with open('image.data', 'wb') as f:
    output.tofile(f)
```

This recipe is also encapsulated in the `PiBayerArray` class in the *picamera.array* module, which means the same can be achieved as follows:

```
import time
import picamera
import picamera.array
import numpy as np

with picamera.PiCamera() as camera:
    with picamera.array.PiBayerArray(camera) as stream:
        camera.capture(stream, 'jpeg', bayer=True)
        # Demosaic data and write to output (just use stream.array if you
        # want to skip the demosaic step)
        output = (stream.demosaic() >> 2).astype(np.uint8)
        with open('image.data', 'wb') as f:
            output.tofile(f)
```

New in version 1.3.

Changed in version 1.5: Added note about new *picamera.array* module.

2.4.14 Using a flash with the camera

The Pi's camera module includes an LED flash driver which can be used to illuminate a scene upon capture. The flash driver has two configurable GPIO pins:

- one for connection to an LED based flash (xenon flashes won't work with the camera module due to it having a rolling shutter). This will fire before (flash metering) and during capture
- one for an optional privacy indicator (a requirement for cameras in some jurisdictions). This will fire after taking a picture to indicate that the camera has been used

These pins are configured by updating the VideoCore device tree blob. Firstly, install the device tree compiler, then grab a copy of the default device tree source:

```
$ sudo apt-get install device-tree-compiler
$ wget http://www.raspberrypi.org/documentation/configuration/images/dt-blob.dts
```

The device tree source contains a number of sections enclosed in curly braces, which form a hierarchy of definitions. The section to edit will depend on which revision of Raspberry Pi you have:

Model	Section
Raspberry Pi Model B revision 1	`/videocore/pins_rev1`
Raspberry Pi Model A Raspberry Pi Model B revision 2	`/videocore/pins_rev2`
Raspberry Pi Model A+ Raspberry Pi Model B+ Raspberry Pi 2 Model B	`/videocore/pins_bplus`

Under the section for your particular model of Pi you will find `pin_config` and `pin_defines` sections. Under the `pin_config` section you need to configure the GPIO pins you want to use for the flash and privacy indicator as using pull down termination. Then, under the `pin_defines` section you need to associate those pins with the `FLASH_0_ENABLE` and `FLASH_0_INDICATOR` pins.

For example, to configure GPIO 17 as the flash pin, leaving the privacy indicator pin absent, on a Raspberry Pi Model B revision 2 you would add the following line under the `/videocore/pins_rev2/pin_config` section:

```
pin@p17 { function = "output"; termination = "pull_down"; };
```

Please note that GPIO pins will be numbered according to the Broadcom pin numbers (BCM mode in the RPi.GPIO library, *not* BOARD mode). Then change the following section under `/videocore/pins_rev2/pin_defines`. Specifically, change the type from "absent" to "internal", and add a number property defining the flash pin as GPIO 17:

```
pin-define@FLASH_0_ENABLE {
    type = "internal";
    number = <17>;
};
```

With the device tree source updated, you now need to compile it into a binary blob for the firmware to read. This is done with the following command line:

```
$ dtc -I dts -O dtb dt-blob.dts -o dt-blob.bin
```

Dissecting this command line, the following components are present:

- `dtc` - Execute the device tree compiler
- `-I dts` - The input file is in device tree source format
- `-O dtb` - The output file should be produced in device tree binary format
- `dt-blob.dts` - The first anonymous parameter is the input filename

- `-o dt-blob.bin` - The output filename

This should output the following:

```
DTC: dts->dtb  on file "dt-blob.dts"
```

If anything else is output, it will most likely be an error message indicating you have made a mistake in the device tree source. In this case, review your edits carefully (note that sections and properties *must* be semi-colon terminated for example), and try again.

Now the device tree binary blob has been produced, it needs to be placed on the first partition of the SD card. In the case of non-NOOBS Raspbian installs, this is generally the partition mounted as `/boot`:

```
$ sudo cp dt-blob.bin /boot/
```

However, in the case of NOOBS Raspbian installs, this is the recovery partition, which is not mounted by default:

```
$ sudo mkdir /mnt/recovery
$ sudo mount /dev/mmcblk0p1 /mnt/recovery
$ sudo cp dt-blob.bin /mnt/recovery
$ sudo umount /mnt/recovery
$ sudo rmdir /mnt/recovery
```

Please note that the filename and location are important. The binary blob must be named `dt-blob.bin` (all lowercase), and it must be placed in the root directory of the first partition on the SD card. Once you have rebooted the Pi (to activate the new device tree configuration) you can test the flash with the following simple script:

```
import picamera

with picamera.PiCamera() as camera:
    camera.flash_mode = 'on'
    camera.capture('foo.jpg')
```

You should see your flash LED blink twice during the execution of the script.

Warning: The GPIOs only have a limited current drive which is insufficient for powering the sort of LEDs typically used as flashes in mobile phones. You will require a suitable drive circuit to power such devices, or risk damaging your Pi. One developer on the Pi forums notes:

> For reference, the flash driver chips we have used on mobile phones will often drive up to 500mA into the LED. If you're aiming for that, then please think about your power supply too.

If you wish to experiment with the flash driver without attaching anything to the GPIO pins, you can also reconfigure the camera's own LED to act as the flash LED. Obviously this is no good for actual flash photography but it can demonstrate whether your configuration is good. In this case you need not add anything to the `pin_config` section (the camera's LED pin is already defined to use pull down termination), but you do need to set `CAMERA_0_LED` to absent, and `FLASH_0_ENABLE` to the old `CAMERA_0_LED` definition (this will be pin 5 in the case of `pins_rev1` and `pins_rev2`, and pin 32 in the case of `pins_bplus`). For example, change:

```
pin_define@CAMERA_0_LED {
    type = "internal";
    number = <5>;
};
pin_define@FLASH_0_ENABLE {
    type = "absent";
};
```

into this:

```
pin_define@CAMERA_0_LED {
    type = "absent";
};
pin_define@FLASH_0_ENABLE {
    type = "internal";
```

```
    number = <5>;
};
```

After compiling and installing the device tree blob according to the instructions above, and rebooting the Pi, you should find the camera LED now acts as a flash LED with the Python script above.

New in version 1.10.

2.5 Frequently Asked Questions (FAQ)

2.5.1 AttributeError: 'module' object has no attribute 'PiCamera'

You've named your script `picamera.py` (or you've named some other script `picamera.py`. If you name a script after a system or third-party package you will break imports for that system or third-party package. Delete or rename that script (and any associated `.pyc` files), and try again.

2.5.2 Can I put the preview in a window?

No. The camera module's preview system is quite crude: it simply tells the GPU to overlay the preview on the Pi's video output. The preview has no knowledge (or interaction with) the X-Windows environment (incidentally, this is why the preview works quite happily from the command line, even without anyone logged in).

That said, the preview area can be resized and repositioned via the *window* attribute of the *preview* object. If your program can respond to window repositioning and sizing events you can "cheat" and position the preview within the borders of the target window. However, there's currently no way to allow anything to appear on top of the preview so this is an imperfect solution at best.

2.5.3 Help! I started a preview and can't see my console!

As mentioned above, the preview is simply an overlay over the Pi's video output. If you start a preview you may therefore discover you can't see your console anymore and there's no obvious way of getting it back. If you're confident in your typing skills you can try calling *stop_preview()* by typing "blindly" into your hidden console. However, the simplest way of getting your display back is usually to hit `Ctrl+D` to terminate the Python process (which should also shut down the camera).

When starting a preview, you may want to set the *alpha* parameter of the *start_preview()* method to something like 128. This should ensure that when the preview is displayed, it is partially transparent so you can still see your console.

2.5.4 The preview doesn't work on my PiTFT screen

The camera's preview system directly overlays the Pi's output on the HDMI or composite video ports. At this time, it will not operate with GPIO-driven displays like the PiTFT. Some projects, like the Adafruit Touchscreen Camera project, have approximated a preview by rapidly capturing unencoded images and displaying them on the PiTFT instead.

2.5.5 How much power does the camera require?

The camera requires 250mA when running. Note that simply creating a *PiCamera* object means the camera is running (due to the hidden preview that is started to allow the auto-exposure algorithm to run). If you are running your Pi from batteries, you should *close()* (or destroy) the instance when the camera is not required in order to conserve power. For example, the following code captures 60 images over an hour, but leaves the camera running all the time:

```
import picamera
import time

with picamera.PiCamera() as camera:
    camera.resolution = (1280, 720)
    time.sleep(1) # Camera warm-up time
    for i, filename in enumerate(camera.capture_continuous('image{counter:02d}.jpg')):
        print('Captured %s' % filename)
        # Capture one image a minute
        time.sleep(60)
        if i == 59:
            break
```

By contrast, this code closes the camera between shots (but can't use the convenient *capture_continuous()* method as a result):

```
import picamera
import time

for i in range(60):
    with picamera.PiCamera() as camera:
        camera.resolution = (1280, 720)
        time.sleep(1) # Camera warm-up time
        filename = 'image%02d.jpg' % i
        camera.capture(filename)
        print('Captured %s' % filename)
    # Capture one image a minute
    time.sleep(59)
```

Note: Please note the timings in the scripts above are approximate. A more precise example of timing is given in *Capturing timelapse sequences*.

If you are experiencing lockups or reboots when the camera is active, your power supply may be insufficient. A practical minimum is 1A for running a Pi with an active camera module; more may be required if additional peripherals are attached.

2.5.6 How can I take two consecutive pictures with equivalent settings?

See the *Capturing consistent images* recipe.

2.5.7 Can I use picamera with a USB webcam?

No. The picamera library relies on libmmal which is specific to the Pi's camera module.

2.5.8 How can I tell what version of picamera I have installed?

The picamera library relies on the setuptools package for installation services. You can use the setuptools `pkg_resources` API to query which version of picamera is available in your Python environment like so:

```
>>> from pkg_resources import require
>>> require('picamera')
[picamera 1.2 (/usr/local/lib/python2.7/dist-packages)]
>>> require('picamera')[0].version
'1.2'
```

If you have multiple versions installed (e.g. from `pip` and `apt-get`) they will not show up in the list returned by the `require` method. However, the first entry in the list will be the version that `import picamera` will import.

If you receive the error "No module named pkg_resources", you need to install the `pip` utility. This can be done with the following command in Raspbian:

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

2.5.9 How come I can't upgrade to the latest version?

If you are using Raspbian, firstly check that you haven't got both a PyPI (`pip`) and an apt (`apt-get`) installation of picamera installed simultaneously. If you have, one will be taking precedence and it may not be the most up to date version.

Secondly, please understand that while the PyPI release process is entirely automated (so as soon as a new picamera release is announced, it will be available on PyPI), the release process for Raspbian packages is semi-manual. There is typically a delay of a few days after a release before updated picamera packages become accessible in the Raspbian repository.

Users desperate to try the latest version may choose to uninstall their `apt` based copy (uninstall instructions are provided in the *installation instructions*, and install using *pip instead*. However, be aware that keeping a PyPI based installation up to date is a more manual process (sticking with `apt` ensures everything gets upgraded with a simple `sudo apt-get upgrade` command).

2.5.10 Why is there so much latency when streaming video?

The first thing to understand is that streaming latency has little to do with the encoding or sending end of things (i.e. the Pi), and much more to do with the playing or receiving end. If the Pi weren't capable of encoding a frame before the next frame arrived, it wouldn't be capable of recording video at all (because its internal buffers would rapidly become filled with unencoded frames).

So, why do players typically introduce several seconds worth of latency? The primary reason is that most players (e.g. VLC) are optimized for playing streams over a network. Such players allocate a large (multi-second) buffer and only start playing once this is filled to guard against possible future packet loss.

A secondary reason that all such players allocate at least a couple of frames worth of buffering is that the MPEG standard includes certain frame types that require it:

- I-frames (intra-frames, also known as "key frames"). These frames contain a complete picture and thus are the largest sort of frames. They occur at the start of playback and at periodic points during the stream.
- P-frames (predicted frames). These frames describe the changes from the prior frame to the current frame, therefore one must have successfully decoded the prior frame in order to decode a P-frame.
- B-frames (bi-directional predicted frames). These frames describe the changes from the next frame to the current frame, therefore one must have successfully decoded the *next* frame in order to decode the current B-frame.

B-frames aren't produced by the Pi's camera (or, as I understand it, by most real-time recording cameras) as it would require buffering yet-to-be-recorded frames before encoding the current one. However, most recorded media (DVDs, Blu-rays, and hence network video streams) do use them, so players must support them. It is simplest to write such a player by assuming that any source may contain B-frames, and buffering at least 2 frames worth of data at all times to make decoding them simpler.

As for the network in between, a slow wifi network may introduce a frame's worth of latency, but not much more than that. Check the ping time across your network; it's likely to be less than 30ms in which case your network cannot account for more than a frame's worth of latency.

TL;DR: the reason you've got lots of latency when streaming video is nothing to do with the Pi. You need to persuade your video player to reduce or forgo its buffering.

2.5.11 Why are there more than 20 seconds of video in the circular buffer?

Read the note at the bottom of the *Recording to a circular stream* recipe. When you set the number of seconds for the circular stream you are setting a *lower bound* for a given bitrate (which defaults to 17Mbps - the same as the video recording default). If the recorded scene has low motion or complexity the stream can store considerably more than the number of seconds specified.

If you need to copy a specific number of seconds from the stream, see the *seconds* parameter of the `copy_to()` method (which was introduced in release 1.11).

Finally, if you specify a different bitrate limit for the stream and the recording, the seconds limit will be inaccurate.

2.5.12 Can I move the annotation text?

No: the firmware provides no means of moving the annotation text. The only configurable attributes of the annotation are currently color and font size.

2.5.13 Why is playback too fast/too slow in VLC/omxplayer/etc.?

The camera's H264 encoder doesn't output a full MP4 file (which would contain frames-per-second meta-data). Instead it outputs an H264 NAL stream which just has frame-size and a few other details (but not FPS).

Most players (like VLC) default to 24, 25, or 30 fps. Hence, recordings at 12fps will appear "fast", while recordings as 60fps will appear "slow". Your playback client needs to be told what fps to use when playing back (assuming it supports such an option).

For those wondering why the camera doesn't output a full MP4 file, consider that the Pi camera's heritage is mobile phone cameras. In these devices you only want the camera to output the H264 stream so you can mux it with, say, an AAC stream recorded from the microphone input and wrap the result into a full MP4 file.

2.5.14 Out of resources at full resolution on a V2 module

See *Hardware Limits*.

2.5.15 Preview flickers at full resolution on a V2 module

Use the new `resolution` property to select a lower resolution for the preview, or specify one when starting the preview. For example:

```
from picamera import PiCamera

camera = PiCamera()
camera.resolution = camera.MAX_RESOLUTION
camera.start_preview(resolution=(1024, 768))
```

2.6 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.

2.6.1 Camera Modes

The Pi's camera has a discrete set of input modes. On the V1 camera these are as follows:

#	Resolution	Aspect Ratio	Framerates	Video	Image	FoV	Binning
1	1920x1080	16:9	1-30fps	x		Partial	None
2	2592x1944	4:3	1-15fps	x	x	Full	None
3	2592x1944	4:3	0.1666-1fps	x	x	Full	None
4	1296x972	4:3	1-42fps	x		Full	2x2
5	1296x730	16:9	1-49fps	x		Full	2x2
6	640x480	4:3	42.1-60fps	x		Full	4x4
7	640x480	4:3	60.1-90fps	x		Full	4x4

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 apt-get dist-upgrade` to upgrade to the latest firmware.

On the V2 camera, these are:

#	Resolution	Aspect Ratio	Framerates	Video	Image	FoV	Binning
1	1920x1080	16:9	0.1-30fps	x		Partial	None
2	3280x2464	4:3	0.1-15fps	x	x	Full	None
3	3280x2464	4:3	0.1-15fps	x	x	Full	None
4	1640x1232	4:3	0.1-40fps	x		Full	2x2
5	1640x922	16:9	0.1-40fps	x		Full	2x2
6	1280x720	16:9	40-90fps	x		Partial	2x2
7	640x480	4:3	40-90fps	x		Partial	2x2

Modes with full field of view (FoV) capture from the whole area of the camera's sensor (2592x1944 pixels for the V1 camera, 3280x2464 for the V2 camera). Modes with partial FoV capture from the center of the sensor. The combination of FoV limiting, and binning is used to achieve the requested resolution.

The image below illustrates the difference between full and partial FoV for the V1 camera:

While the various FoVs for the V2 camera are illustrated in the following image:

The input mode can be manually specified with the *sensor_mode* parameter in the `PiCamera` constructor (using one of the values from the # column in the tables above). This defaults to 0 indicating that the mode should be selected automatically 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 two exist, both with the maximum resolution).
- 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.
- The closer the aspect ratio of the requested `resolution` 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 (note these examples assume the V1 camera module):

- If you set the `resolution` to 1024x768 (a 4:3 aspect ratio), and `framerate` to anything less than 42fps, the 1296x972 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 1296x972 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).

2.6.2 Hardware Limits

The are additional limits imposed by the GPU hardware that performs all image and video processing:

- The maximum resolution for MJPEG recording depends partially on GPU memory. If you get "Out of resource" errors with MJPEG recording at high resolutions, try increasing `gpu_mem` in `/boot/config.txt`.
- The maximum horizontal resolution for default H264 recording is 1920. Any attempt to recording H264 video at higher horizontal resolutions will fail.
- However, H264 high profile level 4.2 has slightly higher limits and may succeed with higher resolutions.
- The maximum resolution of the V2 camera can cause issues with previews. Currently, picamera runs previews at the same resolution as captures (equivalent to `-fp` in `raspistill`). You may need to increase `gpu_mem` in `/boot/config.txt` to achieve full resolution operation with the V2 camera module.
- The maximum framerate of the camera depends on several factors. With overclocking, 120fps has been achieved on a V2 module but 90fps is the maximum supported framerate.
- The maximum exposure time is currently 6 seconds on the V1 camera module, and 10 seconds on the V2 camera module. Remember that exposure time is limited by framerate, so you need to set an extremely slow *framerate* before setting *shutter_speed*.

2.6.3 Under the Hood

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

The MMAL layer below picamera presents the camera with 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 one of the two supported still modes (see *Camera Modes*) so that images are captured using the full area of the sensor. It also uses a strong de-noise algorithm 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 (this is due to the still port using a slower, more aggressive denoise algorithm).

The Preview Port

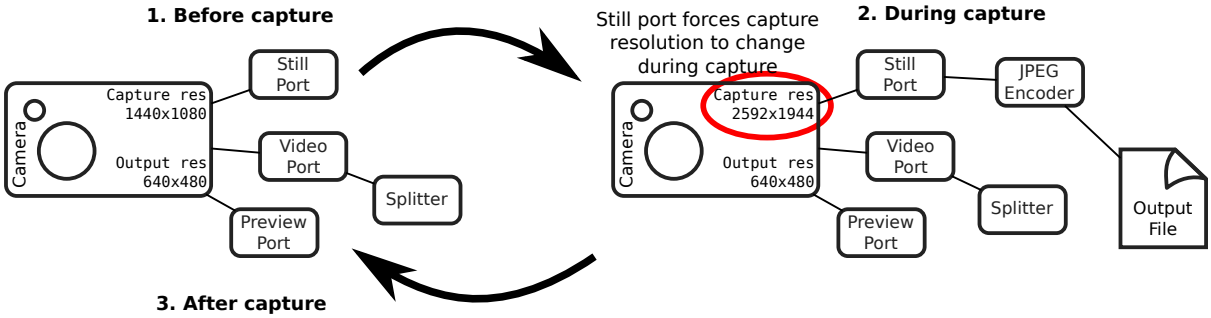
The preview port operates more or less identically to the video port. The preview port is always connected to some form of output to ensure that the auto-gain algorithm can run. When an instance of *PiCamera* is constructed, the

preview port is initially connected to an instance of *PiNullSink*. When *start_preview()* is called, this null sink is destroyed and the preview port is connected to an instance of *PiPreviewRenderer*. The reverse occurs when *stop_preview()* is called.

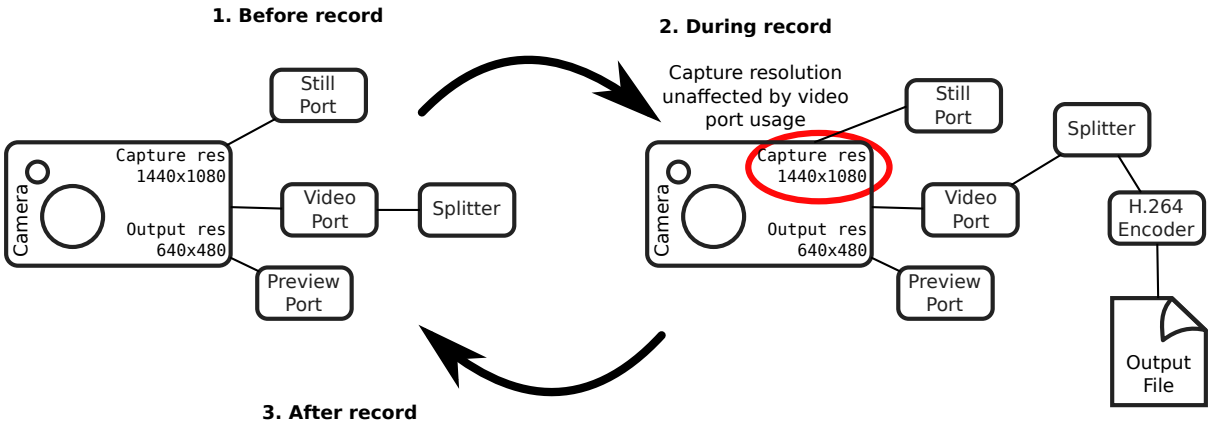
Encoders

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).

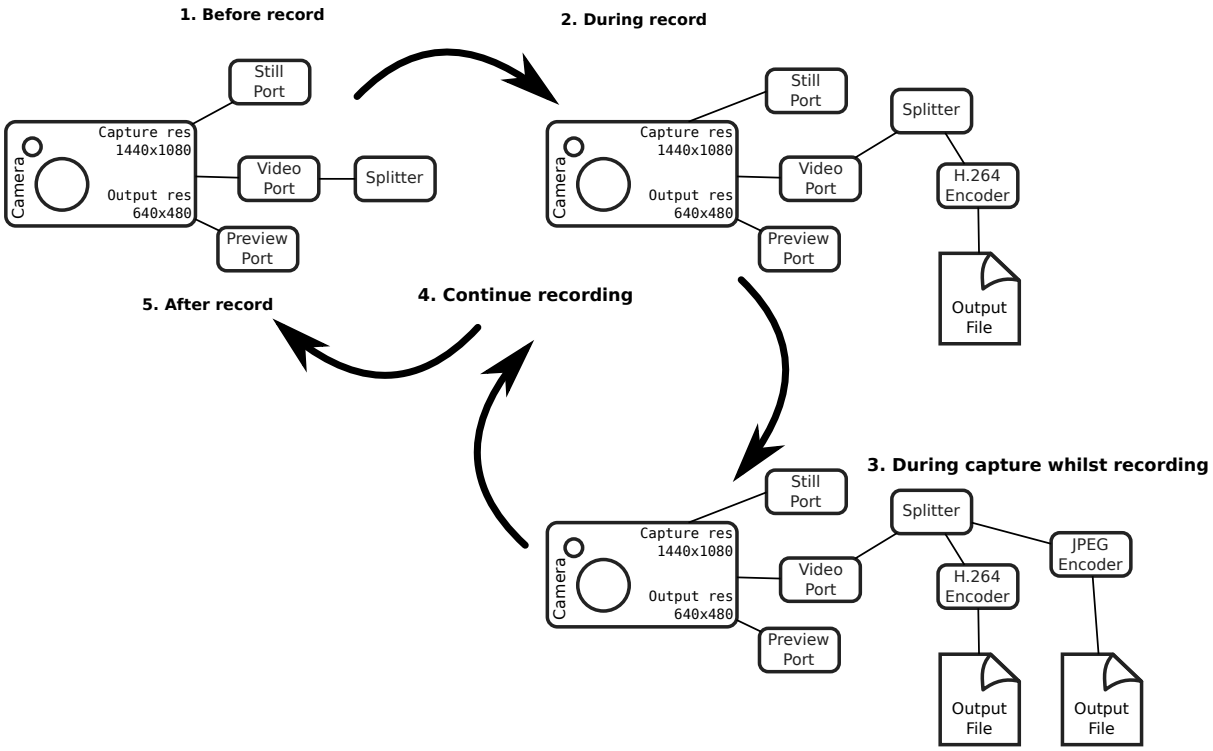
Encoders are connected directly to the still port. For example, when capturing a picture using the still port, the camera's state conceptually moves through these states:



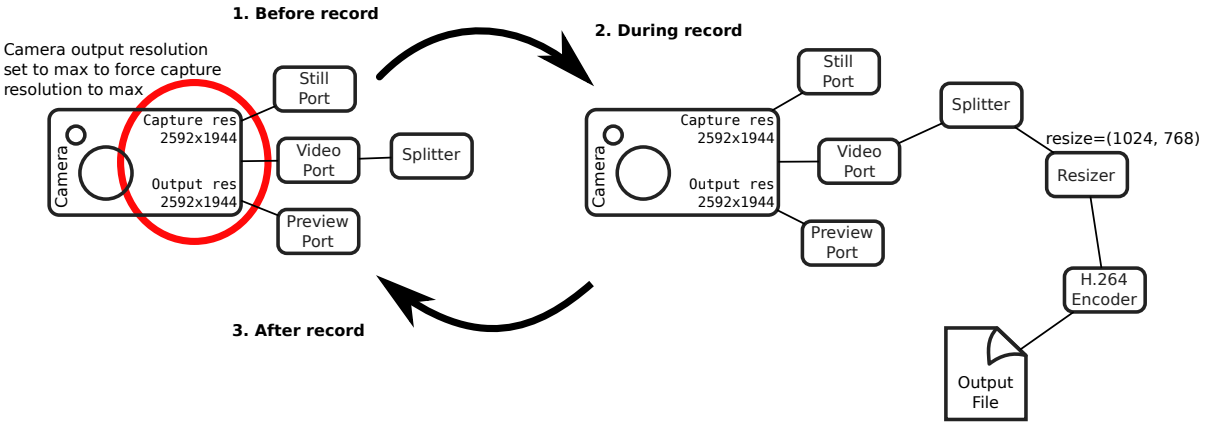
As you have probably noticed in the diagram above, the video port is a little more complex. In order to permit simultaneous video recording and image capture via the video port, a "splitter" component is permanently connected to the video port by picamera, and encoders are in turn attached to one of 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:



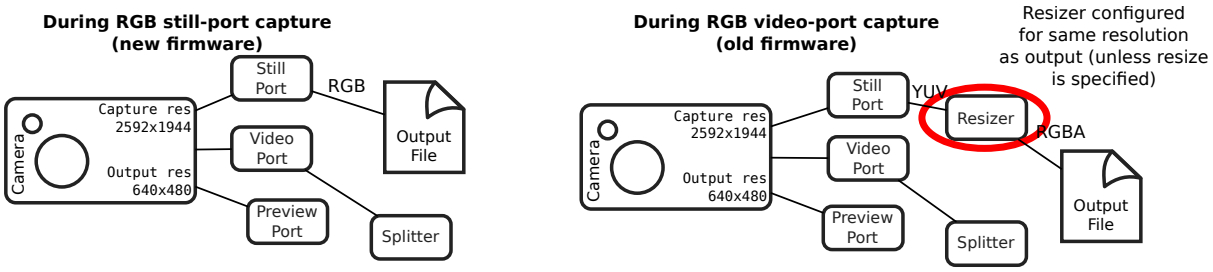
When the `resize` parameter is passed to one of the aforementioned methods, a resizer component is placed between the camera's ports and the encoder, causing the output to be resized before it reaches the encoder. This is particularly useful for video recording, as the H.264 encoder cannot cope with full resolution input (the GPU hardware can only handle frame widths up to 1920 pixels). Hence, when performing full frame video recording, the camera's setup looks like this:



Finally, when performing unencoded captures an encoder is (naturally) not required. Instead data is taken directly from the camera's ports. However, various firmware limitations require acrobatics in the pipeline to achieve requested encodings.

For example, in older firmwares the camera's still port cannot be configured for RGB output (due to a faulty buffer size check). However, they can be configured for YUV output so in this case picamera configures the still port for YUV output, attaches as resizer (configured with the same input and output resolution), then configures the resizer's output for RGBA (the resizer doesn't support RGB for some reason). It then runs the capture and strips the redundant alpha bytes off the data.

Recent firmwares fix the buffer size check, so with these picamera will simply configure the still port for RGB output (since 1.11):



Encodings

A further complication is the "OPAQUE" encoding. This is the most efficient encoding to use when connecting MMAL components as it simply passes pointers around under the hood rather than full frame data. However, not all OPAQUE encodings are equivalent:

- The preview port's OPAQUE encoding contains a single image.
- The video port's OPAQUE encoding contains two images (used for motion estimation by various encoders).
- The still port's OPAQUE encoding contains strips of a single image.
- The JPEG image encoder accepts the still port's OPAQUE strips format.
- The MJPEG video encoder does *not* accept the OPAQUE strips format, only the single and dual image variants provided by the preview or video ports.
- The H264 video encoder in older firmwares only accepts the dual image OPAQUE format (it will accept full-frame YUV input instead though). In newer firmwares it now accepts the single image OPAQUE format too (presumably constructing the second image itself for motion estimation).
- The splitter accepts single or dual image OPAQUE input, but only outputs single image OPAQUE input (or YUV; in later firmwares it also supports RGB or BGR output).
- The resizer theoretically accepts OPAQUE input (though the author hasn't managed to get this working at the time of writing) but will only produce YUV/RGBA/BGRA output.

The new `mmalobj` layer introduced in picamera 1.11 is aware of these OPAQUE encoding differences and attempts to configure connections between components with the most efficient formats possible. However, it is not aware of firmware revisions so if you're playing with MMAL components via this layer be prepared to do some tinkering to get your pipeline working.

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.

2.7 Deprecated Functionality

The picamera library is (at the time of writing) nearly a year old and has grown quite rapidly in this time. Occasionally, when adding new functionality to the library, the API is obvious and natural (e.g. *`start_recording()`* and *`stop_recording()`*). At other times, it's been less obvious (e.g. unencoded captures) and my initial attempts have proven to be less than ideal. In such situations I've endeavoured to improve the API without breaking backward compatibility by introducing new methods or attributes and deprecating the old ones.

This means that, as of release 1.8, there's quite a lot of deprecated functionality floating around the library which it would be nice to tidy up, partly to simplify the library for debugging, and partly to simplify it for new users. To assuage any fears that I'm imminently going to break backward compatibility: I intend to leave a gap of at least

a year between deprecating functionality and removing it, hopefully providing ample time for people to migrate their scripts.

Furthermore, to distinguish any release which is backwards incompatible, I would increment the major version number in accordance with semantic versioning. In other words, the first release in which currently deprecated functionality would be removed would be version 2.0, and as of the release of 1.8 it's at least a year away. Any future 1.x releases will include all currently deprecated functions.

Of course, that still means people need a way of determining whether their scripts use any deprecated functionality in the picamera library. All deprecated functionality is documented, and the documentation includes pointers to the intended replacement functionality (see *raw_format* for example). However, Python also provides excellent methods for determining automatically whether any deprecated functionality is being used via the warnings module.

2.7.1 Finding and fixing deprecated usage

As of release 1.8, all deprecated functionality will raise DeprecationWarning when used. By default, the Python interpreter suppresses these warnings (as they're only of interest to developers, not users) but you can easily configure different behaviour.

The following example script uses a number of deprecated functions:

```
import io
import time
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (1280, 720)
    camera.framerate = (24, 1)
    camera.start_preview()
    camera.preview_fullscreen = True
    camera.preview_alpha = 128
    time.sleep(2)
    camera.raw_format = 'yuv'
    stream = io.BytesIO()
    camera.capture(stream, 'raw', use_video_port=True)
```

Despite using deprecated functionality the script runs happily (and silently) with picamera 1.8. To discover what deprecated functions are being used, we add a couple of lines to tell the warnings module that we want "default" handling of DeprecationWarning; "default" handling means that the first time an attempt is made to raise this warning at a particular location, the warning's details will be printed to the console. All future invocations from the same location will be ignored. This saves flooding the console with warning details from tight loops. With this change, the script looks like this:

```
import io
import time
import picamera

import warnings
warnings.filterwarnings('default', category=DeprecationWarning)

with picamera.PiCamera() as camera:
    camera.resolution = (1280, 720)
    camera.framerate = (24, 1)
    camera.start_preview()
    camera.preview_fullscreen = True
    camera.preview_alpha = 128
    time.sleep(2)
    camera.raw_format = 'yuv'
    stream = io.BytesIO()
    camera.capture(stream, 'raw', use_video_port=True)
```

And produces the following output on the console when run:

```
/usr/share/pyshared/picamera/camera.py:149: DeprecationWarning: Setting framerate or gains as a tu
  "Setting framerate or gains as a tuple is deprecated; "
/usr/share/pyshared/picamera/camera.py:3125: DeprecationWarning: PiCamera.preview_fullscreen is de
  'PiCamera.preview_fullscreen is deprecated; '
/usr/share/pyshared/picamera/camera.py:3068: DeprecationWarning: PiCamera.preview_alpha is deprecа
  'PiCamera.preview_alpha is deprecated; use '
/usr/share/pyshared/picamera/camera.py:1833: DeprecationWarning: PiCamera.raw_format is deprecated
  'PiCamera.raw_format is deprecated; use required format '
/usr/share/pyshared/picamera/camera.py:1359: DeprecationWarning: The "raw" format option is depred
  'The "raw" format option is deprecated; specify the '
/usr/share/pyshared/picamera/camera.py:1827: DeprecationWarning: PiCamera.raw_format is deprecated
  'PiCamera.raw_format is deprecated; use required format '
```

This tells us which pieces of deprecated functionality are being used in our script, but it doesn't tell us where in the script they were used. For this, it is more useful to have warnings converted into full blown exceptions. With this change, each time a `DeprecationWarning` would have been printed, it will instead cause the script to terminate with an unhandled exception and a full stack trace:

```
import io
import time
import picamera

import warnings
warnings.filterwarnings('error', category=DeprecationWarning)

with picamera.PiCamera() as camera:
    camera.resolution = (1280, 720)
    camera.framerate = (24, 1)
    camera.start_preview()
    camera.preview_fullscreen = True
    camera.preview_alpha = 128
    time.sleep(2)
    camera.raw_format = 'yuv'
    stream = io.BytesIO()
    camera.capture(stream, 'raw', use_video_port=True)
```

Now when we run the script it produces the following:

```
Traceback (most recent call last):
  File "test_deprecated.py", line 10, in <module>
    camera.framerate = (24, 1)
  File "/usr/share/pyshared/picamera/camera.py", line 1888, in _set_framerate
    n, d = to_rational(value)
  File "/usr/share/pyshared/picamera/camera.py", line 149, in to_rational
    "Setting framerate or gains as a tuple is deprecated; "
DeprecationWarning: Setting framerate or gains as a tuple is deprecated; please use one of Python
```

This tells us that line 10 of our script is using deprecated functionality, and provides a hint of how to fix it. We change line 10 to use an int instead of a tuple for the framerate. Now we run again, and this time get the following:

```
Traceback (most recent call last):
  File "test_deprecated.py", line 12, in <module>
    camera.preview_fullscreen = True
  File "/usr/share/pyshared/picamera/camera.py", line 3125, in _set_preview_fullscreen
    'PiCamera.preview_fullscreen is deprecated; '
DeprecationWarning: PiCamera.preview_fullscreen is deprecated; use PiCamera.preview.fullscreen ins
```

Now we can tell line 12 has a problem, and once again the exception tells us how to fix it. We continue in this fashion until the script looks like this:

```
import io
import time
```

```
import picamera

import warnings
warnings.filterwarnings('error', category=DeprecationWarning)

with picamera.PiCamera() as camera:
    camera.resolution = (1280, 720)
    camera.framerate = 24
    camera.start_preview()
    camera.preview.fullscreen = True
    camera.preview.alpha = 128
    time.sleep(2)
    stream = io.BytesIO()
    camera.capture(stream, 'yuv', use_video_port=True)
```

The script now runs to completion, so we can be confident it's no longer using any deprecated functionality and will run happily even when this functionality is removed in release 2.0. At this point, you may wish to remove the `filterwarnings` line as well (or at least comment it out).

2.7.2 List of deprecated functionality

For convenience, all currently deprecated functionality is detailed below. You may wish to skim this list to check whether you're currently using deprecated functions, but I would urge users to take advantage of the warnings system documented in the prior section as well.

Unencoded captures

In very early versions of picamera, unencoded captures were created by specifying the `'raw'` format with the *capture()* method, with the *raw_format* attribute providing the actual encoding. The attribute is deprecated, as is usage of the value `'raw'` with the *format* parameter of all the capture methods.

The deprecated method of taking unencoded captures looks like this:

```
camera.raw_format = 'rgb'
camera.capture('output.data', format='raw')
```

In such cases, simply remove references to *raw_format* and place the required format directly within the *capture()* call:

```
camera.capture('output.data', format='rgb')
```

Recording quality

The *quantization* parameter for *start_recording()* and *record_sequence()* is deprecated in favor of the *quality* parameter; this change was made to keep the recording methods consistent with the capture methods, and to make the meaning of the parameter more obvious to newcomers. The values of the parameter remain the same (i.e. 1-100 for MJPEG recordings with higher values indicating higher quality, and 1-40 for H.264 recordings with lower values indicating higher quality).

The deprecated method of setting recording quality looks like this:

```
camera.start_recording('foo.h264', quantization=25)
```

Simply replace the `quantization` parameter with the `quality` parameter like so:

```
camera.start_recording('foo.h264', quality=25)
```

Fractions as tuples

Several attributes in picamera expect rational (fractional) values. In early versions of picamera, these values could only be specified as a tuple expressed as `(numerator, denominator)`. In later versions, support was expanded to accept any of Python's numeric types.

The following code illustrates the deprecated usage of a tuple representing a rational value:

```
camera.framerate = (24, 1)
```

Such cases can be replaced with any of Python's numeric types, including int, float, `Decimal`, and `Fraction`. All the following examples are functionally equivalent to the deprecated example above:

```
from decimal import Decimal
from fractions import Fraction

camera.framerate = 24
camera.framerate = 24.0
camera.framerate = Fraction(72, 3)
camera.framerate = Decimal('24')
camera.framerate = Fraction('48/2')
```

These attributes return a `Fraction` instance as well, but one modified to permit access as a tuple in order to maintain backward compatibility. This is also deprecated behaviour. The following example demonstrates accessing the *framerate* attribute as a tuple:

```
n, d = camera.framerate
print('The framerate is %d/%d fps' % (n, d))
```

In such cases, use the standard `numerator` and `denominator` attributes of the returned fraction instead:

```
f = camera.framerate
print('The framerate is %d/%d fps' % (f.numerator, f.denominator))
```

Alternatively, you may wish to simply convert the `Fraction` instance to a float for greater convenience:

```
f = float(camera.framerate)
print('The framerate is %0.2f fps' % f)
```

Preview functions

Release 1.8 introduced rather sweeping changes to the preview system to incorporate the ability to create multiple static overlays on top of the preview. As a result, the preview system is no longer incorporated into the *PiCamera* class. Instead, it is represented by the *preview* attribute which is a separate *PiPreviewRenderer* instance when the preview is active.

This change meant that *preview_alpha* was deprecated in favor of the *alpha* property of the new *preview* attribute. Similar changes were made to *preview_layer*, *preview_fullscreen*, and *preview_window*. The following snippet illustrates the deprecated method of setting preview related attributes:

```
camera.start_preview()
camera.preview_alpha = 128
camera.preview_fullscreen = False
camera.preview_window = (0, 0, 640, 480)
```

In this case, where preview attributes are altered *after* the preview has been activated, simply modify the corresponding attributes on the preview object:

```
camera.start_preview()
camera.preview.alpha = 128
camera.preview.fullscreen = False
camera.preview.window = (0, 0, 640, 480)
```

Unfortuantely, this simple change is not possible when preview attributes are altered *before* the preview has been activated, as the *preview* attribute is None when the preview is not active. To accomodate this use-case, optional parameters were added to *start_preview()* to provide initial settings for the preview renderer. The following example illustrates the deprecated method of setting preview related attribtues prior to activating the preview:

```
camera.preview_alpha = 128
camera.preview_fullscreen = False
camera.preview_window = (0, 0, 640, 480)
camera.start_preview()
```

Remove the lines setting the attributes, and use the corresponding keyword parameters of the *start_preview()* method instead:

```
camera.start_preview(
    alpha=128, fullscreen=False, window=(0, 0, 640, 480))
```

Finally, the *previewing* attribute is now obsolete (and thus deprecated) as its functionality can be trivially obtained by checking the *preview* attribute. The following example illustrates the deprecated method of checking whether the preview is activate:

```
if camera.previewing:
    print('The camera preview is running')
else:
    print('The camera preview is not running')
```

Simply replace *previewing* with *preview* to bring this code up to date:

```
if camera.preview:
    print('The camera preview is running')
else:
    print('The camera preview is not running')
```

Array stream truncation

In release 1.8, the base *PiArrayOutput* class was changed to derive from io.BytesIO in order to add support for seeking, and to improve performance. The prior implementation had been non-seekable, and therefore to accommodate re-use of the stream between captures the *truncate()* method had an unusual side-effect not seen with regular Python streams: after truncation, the position of the stream was set to the new length of the stream. In all other Python streams, the truncate method doesn't affect the stream position. The method is overridden in 1.8 to maintain its unusual behaviour, but this behaviour is nonetheless deprecated.

The following snippet illustrates the method of truncating an array stream in picamera versions 1.7 and older:

```
with picamera.array.PiYUVArray(camera) as stream:
    for i in range(3):
        camera.capture(stream, 'yuv')
        print(stream.array.shape)
        stream.truncate(0)
```

If you only need your script to work with picamera versions 1.8 and newer, such code should be updated to use seek and truncate as you would with any regular Python stream instance:

```
with picamera.array.PiYUVArray(camera) as stream:
    for i in range(3):
        camera.capture(stream, 'yuv')
        print(stream.array.shape)
        stream.seek(0)
        stream.truncate()
```

Unfortunately, this will not work if your script needs to work with prior versions of picamera as well (since such streams were non-seekable in prior versions). In this case, call seekable() to determine the correct course of action:

```
with picamera.array.PiYUVArray(camera) as stream:
    for i in range(3):
        camera.capture(stream, 'yuv')
        print(stream.array.shape)
        if stream.seekable():
            stream.seek(0)
            stream.truncate()
        else:
            stream.truncate(0)
```

Confusing crop

In release 1.8, the *crop* attribute was renamed to *zoom*; the old name was retained as a deprecated alias for backward compatibility. This change was made as `crop` was a thoroughly misleading name for the attribute (which actually sets the "region of interest" for the sensor), leading to numerous support questions.

The following example illustrates the deprecated attribute name:

```
camera.crop = (0.25, 0.25, 0.5, 0.5)
```

Simply replace *crop* with *zoom* in such cases:

```
camera.zoom = (0.25, 0.25, 0.5, 0.5)
```

Incorrect ISO capitalisation

In release 1.8, the *ISO* attribute was renamed to *iso* for compliance with PEP-8 (even though it's an acronym this is still more consistent with the existing API; consider *led*, *awb_mode*, and so on).

The following example illustrates the deprecated attribute case:

```
camera.ISO = 100
```

Simply replace references to *ISO* with *iso*:

```
camera.iso = 100
```

Frame types

Over time, several capabilities were added to the H.264 encoder in the GPU firmware. This expanded the number of possible frame types from a simple key-frame / non-key-frame affair, to a multitude of possibilities (P-frame, I-frame, SPS/PPS header, motion vector data, and who knows in future). Rather than keep adding more and more boolean fields to the *PiVideoFrame* named tuple, release 1.5 introduced the *PiVideoFrameType* enumeration used by the *frame_type* attribute and deprecated the *keyframe* and *header* attributes.

The following code illustrates usage of the deprecated boolean fields:

```
if camera.frame.keyframe:
    handle_keyframe()
elif camera.frame.header:
    handle_header()
else:
    handle_frame()
```

In such cases, test the *frame_type* attribute against the corresponding value of the *PiVideoFrameType* enumeration:

```
if camera.frame.frame_type == picamera.PiVideoFrameType.key_frame:
    handle_keyframe()
elif camera.frame.frame_type == picamera.PiVideoFrameType.sps_header:
    handle_header()
```

```
else:
    handle_frame()
```

Alternatively, you may find something like this more elegant (and more future proof as it'll throw a `KeyError` in the event of an unrecognized frame type):

```
handler = {
    picamera.PiVideoFrameType.key_frame:  handle_keyframe,
    picamera.PiVideoFrameType.sps_header: handle_header,
    picamera.PiVideoFrameType.frame:      handle_frame,
    }[camera.frame.frame_type]
handler()
```

Annotation background color

In release 1.10, the *annotate_background* attribute was enhanced to support setting the background color of annotation text. Older versions of picamera treated this attribute as a bool (`False` for no background, `True` to draw a black background).

In order to provide the new functionality while maintaining a certain amount of backward compatibility, the new attribute accepts `None` for no background and a *Color* instance for a custom background color. It is worth noting that the truth values of `None` and `False` are equivalent, as are the truth values of a *Color* instance and `True`. Hence, naive tests against the attribute value will continue to work.

The following example illustrates the deprecated behaviour of setting the attribute as a boolean:

```
camera.annotate_background = False
camera.annotate_background = True
```

In such cases, replace `False` with `None`, and `True` with a *Color* instance of your choosing. Bear in mind that older Pi firmwares can only produce a black background, so you may wish to stick with black to ensure equivalent behaviour:

```
camera.annotate_background = None
camera.annotate_background = picamera.Color('black')
```

Naive tests against the attribute should work as normal, but specific tests (which are considered bad practice anyway), should be re-written. The following example illustrates specific boolean tests:

```
if camera.annotate_background == False:
    pass
if camera.annotate_background is True:
    pass
```

Such cases should be re-written to remove the specific boolean value mentioned in the test (this is a general rule, not limited to this deprecation case):

```
if not camera.annotate_background:
    pass
if camera.annotate_background:
    pass
```

Analysis classes use analyze

The various analysis classes in *picamera.array* were adjusted in 1.11 to use *analyze()* (US English spelling) instead of `analyse` (UK English spelling). The following example illustrates the old usage:

```
import picamera.array

class MyAnalyzer(picamera.array.PiRGBAnalysis):
    def analyse(self, array):
        print('Array shape:', array.shape)
```

This should simply be re-written as:

```
import picamera.array

class MyAnalyzer(picamera.array.PiRGBAnalysis):
    def analyze(self, array):
        print('Array shape:', array.shape)
```

2.8 API - The PiCamera Class

The picamera library contains numerous classes, but the primary one that all users are likely to interact with is *PiCamera*, documented below. With the exception of the contents of the *picamera.array* module, all classes in picamera are accessible from the package's top level namespace. In other words, the following import is sufficient to import everything in the library (excepting the contents of *picamera.array*):

```
import picamera
```

2.8.1 PiCamera

class picamera.**PiCamera**(*camera_num=0*, *stereo_mode='none'*, *stereo_decimate=False*, *resolution=None*, *framerate=None*, *sensor_mode=0*, *led_pin=None*, *clock_mode='reset'*)

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

Upon construction, this class initializes the camera. The *camera_num* parameter (which defaults to 0) selects the camera module that the instance will represent. Only the Raspberry Pi compute module currently supports more than one camera.

The *sensor_mode*, *resolution*, *framerate*, and *clock_mode* parameters provide initial values for the *sensor_mode*, *resolution*, *framerate*, and *clock_mode* attributes of the class (these attributes are all relatively expensive to set individually, hence setting them all upon construction is a speed optimization). Please refer to the attribute documentation for more information and default values.

The *stereo_mode* and *stereo_decimate* parameters configure dual cameras on a compute module for sterescopic mode. These parameters can only be set at construction time; they cannot be altered later without closing the *PiCamera* instance and recreating it. The *stereo_mode* parameter defaults to `'none'` (no stereoscopic mode) but can be set to `'side-by-side'` or `'top-bottom'` to activate a stereoscopic mode. If the *stereo_decimate* parameter is `True`, the resolution of the two cameras will be halved so that the resulting image has the same dimensions as if stereoscopic mode were not being used.

The *led_pin* parameter can be used to specify the GPIO pin which should be used to control the camera's LED via the *led* attribute. If this is not specified, it should default to the correct value for your Pi platform. You should only need to specify this parameter if you are using a custom DeviceTree blob (this is only typical on the Compute Module platform).

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:

```
    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
```

Changed in version 1.8: Added *stereo_mode* and *stereo_decimate* parameters.

Changed in version 1.9: Added *resolution*, *framerate*, and *sensor_mode* parameters.

Changed in version 1.10: Added *led_pin* parameter.

Changed in version 1.11: Added *clock_mode* parameter, and permitted setting of resolution as appropriately formatted string.

add_overlay(*source*, *size=None*, ***options*)

Adds a static overlay to the preview output.

This method creates a new static overlay using the same rendering mechanism as the preview. Overlays will appear on the Pi's video output, but will not appear in captures or video recordings. Multiple overlays can exist; each call to *`add_overlay()`* returns a new *`PiOverlayRenderer`* instance representing the overlay.

The optional *size* parameter specifies the size of the source image as a `(width, height)` tuple. If this is omitted or `None` then the size is assumed to be the same as the camera's current *`resolution`*.

The *source* must be an object that supports the buffer protocol which has the same length as an image in RGB format (colors represented as interleaved unsigned bytes) with the specified *size* after the width has been rounded up to the nearest multiple of 32, and the height has been rounded up to the nearest multiple of 16.

For example, if *size* is `(1280, 720)`, then *source* must be a buffer with length 1280 × 720 × 3 bytes, or 2,764,800 bytes (because 1280 is a multiple of 32, and 720 is a multiple of 16 no extra rounding is required). However, if *size* is `(97, 57)`, then *source* must be a buffer with length 128 × 64 × 3 bytes, or 24,576 bytes (pixels beyond column 97 and row 57 in the source will be ignored).

New overlays default to *layer* 0, whilst the preview defaults to layer 2. Higher numbered layers obscure lower numbered layers, hence new overlays will be invisible (if the preview is running) by default. You can make the new overlay visible either by making any existing preview transparent (with the *`alpha`* property) or by moving the overlay into a layer higher than the preview (with the *`layer`* property).

All keyword arguments captured in *options* are passed onto the *`PiRenderer`* constructor. All camera properties except *`resolution`* and *`framerate`* can be modified while overlays exist. The reason for these exceptions is that the overlay has a static resolution and changing the camera's mode would require resizing of the source.

> **Warning:** If too many overlays are added, the display output will be disabled and a reboot will generally be required to restore the display. Overlays are composited "on the fly". Hence, a real-time constraint exists wherein for each horizontal line of HDMI output, the content of all source layers must be fetched, resized, converted, and blended to produce the output pixels.
> If enough overlays exist (where "enough" is a number dependent on overlay size, display resolution, bus frequency, and several other factors making it unrealistic to calculate in advance), this process breaks down and video output fails. One solution is to add `dispmanx_offline=1` to `/boot/config.txt` to force the use of an off-screen buffer. Be aware that this requires more GPU memory and may reduce the update rate.

New in version 1.8.

capture(*output*, *format=None*, *use_video_port=False*, *resize=None*, *splitter_port=0*, *bayer=False*, ***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. If *output* is not a string, but is an object with a `write` method, it is assumed to 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 are required but `flush` will be called at the end of capture if it is present). If *output* is not a string, and has no `write` method it is assumed to be a writeable object implementing the buffer protocol. In this case, the image data will be written directly to the underlying buffer (which must be large enough to accept the image data).

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 output in. 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. Please note that JPEG quality is not a percentage and definitions of quality vary widely.
- *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

Changed in version 1.11: Support for buffer outputs was added.

capture_continuous(*output*, *format=None*, *use_video_port=False*, *resize=None*, *splitter_port=0*, *burst=False*, *bayer=False*, ***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, but has a `write` method, 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. If *output* is not a string, and has no `write` method, it is assumed to be a writeable object supporting the buffer protocol; each image is simply written to the buffer sequentially.

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

If *use_video_port* is `False` (the default), the *burst* parameter can be used to make still port captures faster. Specifically, this prevents the preview from switching resolutions between captures which significantly speeds up consecutive captures from the still port. The downside is that this mode is

currently has several bugs; the major issue is that if captures are performed too quickly some frames will come back severely underexposed. It is recommended that users avoid the *burst* parameter unless they absolutely require it and are prepared to work around such issues.

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

Changed in version 1.11: Support for buffer outputs was added.

capture_sequence(*outputs*, *format='jpeg'*, *use_video_port=False*, *resize=None*, *splitter_port=0*, *burst=False*, *bayer=False*, ***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, or a writeable buffer object. 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.

If *use_video_port* is `False` (the default), the *burst* parameter can be used to make still port captures faster. Specifically, this prevents the preview from switching resolutions between captures which significantly speeds up consecutive captures from the still port. The downside is that this mode is currently has several bugs; the major issue is that if captures are performed too quickly some frames will come back severely underexposed. It is recommended that users avoid the *burst* parameter unless they absolutely require it and are prepared to work around such issues.

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

Changed in version 1.11: Support for buffer outputs 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.

remove_overlay(*overlay*)
Removes a static overlay from the preview output.

This method removes an overlay which was previously created by *add_overlay()*. The *overlay* parameter specifies the *PiRenderer* instance that was returned by *add_overlay()*.

New in version 1.8.

request_key_frame(*splitter_port=1*)
Request the encoder generate a key-frame as soon as possible.

When called, the video encoder running on the specified *splitter_port* will attempt to produce a key-frame (full-image frame) as soon as possible. The *splitter_port* defaults to `1`. Valid values are between `0` and `3` inclusive.

Note: This method is only meaningful for recordings encoded in the H264 format as MJPEG produces full frames for every frame recorded. Furthermore, there's no guarantee that the *next* frame will be a key-frame; this is simply a request to produce one as soon as possible after the call.

New in version 1.11.

split_recording(*output*, *splitter_port=1*, ***options*)
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*.

The *output* parameter is treated as in the *start_recording()* method (it can be a string, a file-like object, or a writeable buffer object).

The *motion_output* parameter can be used to redirect the output of the motion vector data in the same fashion as *output*. If *motion_output* is None (the default) then motion vector data will not be redirected and will continue being written to the output specified by the *motion_output* parameter given to *start_recording()*. Alternatively, if you only wish to redirect motion vector data, you can set *output* to None and given a new value for *motion_output*.

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 other options as these cannot be changed in the middle of recording. Only the new *output* (and *motion_output*) can be specified. Furthermore, the format of the recording is currently limited to H264, and *inline_headers* must be True when *start_recording()* is called (this is the default).

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

Changed in version 1.5: The *motion_output* parameter was added

Changed in version 1.11: Support for buffer outputs was added.

start_preview(***options*)
Displays the preview overlay.

This method starts a camera preview as an overlay on the Pi's primary display (HDMI or composite). A *PiRenderer* instance (more specifically, a *PiPreviewRenderer*) is constructed with the keyword arguments captured in *options*, and is returned from the method (this instance is also accessible from the *preview* attribute for as long as the renderer remains active). By default, the renderer will be opaque and fullscreen.

This means the default 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.

All other camera properties can be modified "live" while the preview is running (e.g. *brightness*).

Note: Because the default 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. If *output* is not a string, but is an object with a write method, it 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 are required but flush will be called at the end of recording if it is present). If *output* is not a string, and has no write method it is assumed to be a writeable object implementing the buffer protocol. In this case, the video frames will be written sequentially to the underlying buffer (which must be large enough to accept all frame data).

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 video output in. 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
- 'yuv' - Write the raw video data to a file in YUV420 format
- 'rgb' - Write the raw video data to a file in 24-bit RGB format
- 'rgba' - Write the raw video data to a file in 32-bit RGBA format
- 'bgr' - Write the raw video data to a file in 24-bit BGR format
- 'bgra' - Write the raw video data to a file in 32-bit BGRA format

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 resolution of the camera sensor (which is not possible in H.264 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'.
- *level* - The H.264 level to use for encoding. Defaults to '4', but can be one of '4', '4.1', or '4.2'.
- *intra_period* - The key frame rate (the rate at which I-frames are inserted in the output). Defaults to None, but can be any 32-bit integer value representing the number of frames between successive I-frames. The special value 0 causes the encoder to produce a single initial I-frame, and then only P-frames subsequently. Note that *split_recording()* will fail in this mode.
- *intra_refresh* - The key frame format (the way in which I-frames will be inserted into the output stream). Defaults to None, but can be one of 'cyclic', 'adaptive', 'both', or 'cyclicrows'.
- *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.
- *sei* - When True, specifies the encoder should include "Supplemental Enhancement Information" within the output stream. Defaults to False if not specified.
- *motion_output* - Indicates the output destination for motion vector estimation data. When None (the default), motion data is not output. Otherwise, this can be a filename string, a file-like object, or a writeable buffer object (as with the *output* parameter).

All encoded formats accept the following additional options:

- *bitrate* - The bitrate at which video will be encoded. Defaults to 17000000 (17Mbps) if not specified. The maximum value is 25000000 (25Mbps), except for H.264 level 4.2 for which the maximum is 62500000 (62.5Mbps). Bitrate 0 indicates the encoder should not use bitrate control (the encoder is limited by the quality only).
- *quality* - 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). For the mjpeg format, use JPEG quality values between 1 and 100 (where higher values are higher quality). Quality 0 is special and seems to be a "reasonable quality" default.
- *quantization* - Deprecated alias for *quality*.

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

Changed in version 1.5: The *quantization* parameter was deprecated in favor of *quality*, and the *motion_output* parameter was added.

Changed in version 1.11: Support for buffer outputs was added.

stop_preview()
Hides the preview overlay.

If *start_preview()* has previously been called, this method shuts down the preview display which generally results in the underlying display 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) the recording will stop, but 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.

Deprecated since version 1.8: Please use the *iso* attribute instead.

analog_gain
Retrieves the current analog gain of the camera.

When queried, this property returns the analog gain currently being used by the camera. The value represents the analog gain of the sensor prior to digital conversion. The value is returned as a `Fraction` instance.

New in version 1.6.

annotate_background
Controls what background is drawn behind the annotation.

The *annotate_background* attribute specifies if a background will be drawn behind the *annotation text* and, if so, what color it will be. The value is specified as a *Color* or `None` if no background should be drawn. The default is `None`.

Note: For backward compatibility purposes, the value `False` will be treated as `None`, and the value `True` will be treated as the color black. The "truthiness" of the values returned by the attribute are backward compatible although the values themselves are not.

New in version 1.8.

Changed in version 1.10: In prior versions this was a bool value with `True` representing a black background.

annotate_foreground
Controls the color of the annotation text.

The *annotate_foreground* attribute specifies, partially, the color of the annotation text. The value is specified as a *Color*. The default is white.

Note: The underlying firmware does not directly support setting all components of the text color, only the Y' component of a Y'UV tuple. This is roughly (but not precisely) analogous to the "brightness" of a color, so you may choose to think of this as setting how bright the annotation text will be relative to its background. In order to specify just the Y' component when setting this attribute, you may choose to construct the *Color* instance as follows:

```
camera.annotate_foreground = picamera.Color(y=0.2, u=0, v=0)
```

New in version 1.10.

annotate_frame_num
Controls whether the current frame number is drawn as an annotation.

The *annotate_frame_num* attribute is a bool indicating whether or not the current frame number is rendered as an annotation, similar to *annotate_text*. The default is `False`.

New in version 1.8.

annotate_text
Retrieves or sets a text annotation for all output.

When queried, the *annotate_text* property returns the current annotation (if no annotation has been set, this is simply a blank string).

When set, the property immediately applies the annotation to the preview (if it is running) and to any future captures or video recording. Strings longer than 255 characters, or strings containing non-ASCII characters will raise a *PiCameraValueError*. The default value is `''`.

Changed in version 1.8: Text annotations can now be 255 characters long. The prior limit was 32 characters.

annotate_text_size
Controls the size of the annotation text.

The *annotate_text_size* attribute is an int which determines how large the annotation text will appear on the display. Valid values are in the range 6 to 160, inclusive. The default is 32.

New in version 1.10.

awb_gains
Gets or sets the auto-white-balance gains of the camera.

When queried, this attribute returns a tuple of values representing the *(red, blue)* balance of the camera. The *red* and *blue* values are returned `Fraction` instances. The values will be between 0.0 and 8.0.

When set, this attribute adjusts the camera's auto-white-balance gains. The property can be specified as a single value in which case both red and blue gains will be adjusted equally, or as a *(red, blue)* tuple. Values can be specified as an int, float or `Fraction` and each gain must be between 0.0 and 8.0. Typical values for the gains are between 0.9 and 1.9. The property can be set while recordings or previews are in progress.

Note: This attribute only has an effect when *awb_mode* is set to `'off'`.

Changed in version 1.6: Prior to version 1.6, this attribute was write-only.

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, and are as follows:

- `'off'`
- `'auto'`
- `'sunlight'`
- `'cloudy'`
- `'shade'`
- `'tungsten'`
- `'fluorescent'`
- `'incandescent'`
- `'flash'`
- `'horizon'`

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'`.

Note: AWB mode `'off'` is special: this disables the camera's automatic white balance permitting manual control of the white balance via the *awb_gains* property.

---

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.

clock_mode
Retrieves or sets the mode of the camera's clock.

This is an advanced property which can be used to control the nature of the frame timestamps available from the *frame* property. When this is "reset" (the default) each frame's timestamp will be relative to the start of the recording. When this is "raw", each frame's timestamp will be relative to the last initialization of the camera.

The initial value of this property can be specified with the *clock_mode* parameter in the *PiCamera* constructor, and will default to "reset" if not specified.

New in version 1.11.

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 zoom applied to the camera's input.

Deprecated since version 1.8: Please use the *zoom* attribute instead.

digital_gain
Retrieves the current digital gain of the camera.

When queried, this property returns the digital gain currently being used by the camera. The value represents the digital gain the camera applies after conversion of the sensor's analog output. The value is returned as a `Fraction` instance.

New in version 1.6.

drc_strength
Retrieves or sets the dynamic range compression strength of the camera.

When queried, the *drc_strength* property returns a string indicating the amount of dynamic range compression the camera applies to images.

When set, the attributes adjusts the strength of the dynamic range compression applied to the camera's output. Valid values are given in the list below:

- `'off'`
- `'low'`
- `'medium'`
- `'high'`

The default value is `'off'`. All possible values for the attribute can be obtained from the `PiCamera.DRC_STRENGTHS` attribute.

New in version 1.6.

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. Each increment represents 1/6th of a stop. Hence setting the attribute to 6 increases exposure by 1 stop. 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, and are as follows:

- `'off'`
- `'auto'`
- `'night'`

- 'nightpreview'
- 'backlight'
- 'spotlight'
- 'sports'
- 'snow'
- 'beach'
- 'verylong'
- 'fixedfps'
- 'antishake'
- 'fireworks'

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'.

Note: Exposure mode 'off' is special: this disables the camera's automatic gain control, fixing the values of *digital_gain* and *analog_gain*. Please note that these properties are not directly settable, and default to low values when the camera is first initialized. Therefore it is important to let them settle on higher values before disabling automatic gain control otherwise all frames captured will appear black.

---

exposure_speed

Retrieves the current shutter speed of the camera.

When queried, this property returns the shutter speed currently being used by the camera. If you have set *shutter_speed* to a non-zero value, then *exposure_speed* and *shutter_speed* should be equal. However, if *shutter_speed* is set to 0 (auto), then you can read the actual shutter speed being used from this attribute. The value is returned as an integer representing a number of microseconds. This is a read-only property.

New in version 1.6.

flash_mode

Retrieves or sets the flash mode of the camera.

When queried, the *flash_mode* property returns a string representing the flash setting of the camera. The possible values can be obtained from the PiCamera.FLASH_MODES attribute, and are as follows:

- 'off'
- 'auto'
- 'on'
- 'redeye'
- 'fillin'
- 'torch'

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

Note: You must define which GPIO pins the camera is to use for flash and privacy indicators. This is done within the Device Tree configuration which is considered an advanced topic. Specifically, you need to define pins FLASH_0_ENABLE and optionally FLASH_0_INDICATOR (for the privacy indicator). More information can be found in this *recipe*.

New in version 1.10.

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), which encoder's frame information is returned is arbitrary. If you require information from a specific encoder, you will need to extract it from _encoders explicitly.

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).

Setting and retrieving framerate as a (numerator, denominator) tuple is deprecated and will be removed in 2.0. Please use a Fraction instance instead (which is just as accurate and also permits direct use with math operators).

When set, the property configures 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. For example, the following definitions are all equivalent:

```
from fractions import Fraction

camera.framerate = 30
camera.framerate = 30 / 1
camera.framerate = Fraction(30, 1)
camera.framerate = (30, 1) # deprecated
```

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 *sensor_mode* for more information.

The initial value of this property can be specified with the *framerate* parameter in the *PiCamera* constructor, and will default to 30 if not specified.

framerate_delta
Retrieves or sets a fractional amount that is added to the camera's framerate for the purpose of minor framerate adjustments.

When queried, the *framerate_delta* property returns the amount that the camera's *framerate* has been adjusted. This defaults to 0 (so the camera's framerate is the actual framerate used).

When set, the property adjusts the camera's framerate on the fly. The property can be set while recordings or previews are in progress. Thus the framerate used by the camera is actually *framerate* + *framerate_delta*.

Note: Framerates deltas can be fractional with adjustments as small as 1/256th of an fps possible (finer adjustments will be rounded). With an appropriately tuned PID controller, this can be used to achieve synchronization between the camera framerate and other devices.

If the new framerate demands a mode switch (such as moving between a low framerate and a high framerate mode), currently active recordings may drop a frame. This should only happen when specifying quite large deltas, or when framerate is at the boundary of a sensor mode (e.g. 49fps).

The framerate delta can be specified as an int, float, `Fraction` or a `(numerator, denominator)` tuple. For example, the following definitions are all equivalent:

```
from fractions import Fraction

camera.framerate_delta = 0.5
camera.framerate_delta = 1 / 2 # in python 3
camera.framerate_delta = Fraction(1, 2)
camera.framerate_delta = (1, 2) # deprecated
```

Note: This property is reset to 0 when *framerate* is set.

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_denoise

Retrieves or sets whether denoise will be applied to image captures.

When queried, the *image_denoise* property returns a boolean value indicating whether or not the camera software will apply a denoise algorithm to image captures.

When set, the property activates or deactivates the denoise algorithm for image captures. The property can be set while recordings or previews are in progress. The default value is `True`.

New in version 1.7.

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, and are as follows:

- 'none'
- 'negative'
- 'solarize'
- 'sketch'
- 'denoise'
- 'emboss'
- 'oilpaint'
- 'hatch'

- 'gpen'
- 'pastel'
- 'watercolor'
- 'film'
- 'blur'
- 'saturation'
- 'colorswap'
- 'washedout'
- 'posterise'
- 'colorpoint'
- 'colorbalance'
- 'cartoon'
- 'deinterlace1'
- 'deinterlace2'

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'.

image_effect_params

Retrieves or sets the parameters for the current *effect*.

When queried, the *image_effect_params* property either returns None (for effects which have no configurable parameters, or if no parameters have been configured), or a tuple of numeric values up to six elements long.

When set, the property changes the parameters of the current *effect* as a sequence of numbers, or a single number. Attempting to set parameters on an effect which does not support parameters, or providing an incompatible set of parameters for an effect will raise a *PiCameraValueError* exception.

The effects which have parameters, and what combinations those parameters can take is as follows:

Effect	Parameters	Description
'solarize'	*yuv*, *x0*, *y1*, *y2*, *y3*	*yuv* controls whether data is processed as RGB (0) or YUV(1). Input values from 0 to *x0* - 1 are remapped linearly onto the range 0 to *y0*. Values from *x0* to 255 are remapped linearly onto the range *y1* to *y2*.
	x0, *y0*, *y1*, *y2*	Same as above, but *yuv* defaults to 0 (process as RGB).
	yuv	Same as above, but *x0*, *y0*, *y1*, *y2* default to 128, 128, 128, 0 respectively.
'colorpoint'	*quadrant*	*quadrant* specifies which quadrant of the U/V space to retain chroma from: 0=green, 1=red/yellow, 2=blue, 3=purple. There is no default; this effect does nothing until parameters are set.
'colorbalance'	*lens*, *r*, *g*, *b*, *u*, *v*	*lens* specifies the lens shading strength (0.0 to 256.0, where 0.0 indicates lens shading has no effect). *r*, *g*, *b* are multipliers for their respective color channels (0.0 to 256.0). *u* and *v* are offsets added to the U/V plane (0 to 255).
	lens, *r*, *g*, *b*	Same as above but *u* are defaulted to 0.
	lens, *r*, *b*	Same as above but *g* also defaults to to 1.0.
'colorswap'	*dir*	If *dir* is 0, swap RGB to BGR. If *dir* is 1, swap RGB to BRG.
'posterise'	*steps*	Control the quantization steps for the image. Valid values are 2 to 32, and the default is 4.
'blur'	*size*	Specifies the size of the kernel. Valid values are 1 or 2.
'film'	*strength*, *u*, *v*	*strength* specifies the strength of effect. *u* and *v* are offsets added to the U/V plane (0 to 255).
'watercolor'	*u*, *v*	*u* and *v* specify offsets to add to the U/V plane (0 to 255).
		No parameters indicates no U/V effect.

New in version 1.8.

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 values (e.g. 100) imply less sensitivity than higher values (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 1600. 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.

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

Note: You can query the *analog_gain* and *digital_gain* attributes to determine the actual gains being used by the camera. If both are 1.0 this equates to ISO 100. Please note that this capability requires an up to date firmware (#692 or later).

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

Note: Some users on the Pi camera forum have noted that higher ISO values than 800 (specifically up to 1600) can be achieved in certain conditions with *exposure_mode* set to 'sports' and *iso* set to 0. It doesn't appear to be possible to manually request an ISO setting higher than 800, but the picamera library will permit settings up to 1600 in case the underlying firmware permits such settings

in particular circumstances.

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.

Note: At present, the camera's LED cannot be controlled on the Pi 3 (the GPIOs used to control the camera LED were re-routed to GPIO expander on the Pi 3).

Warning: There are circumstances in which the camera firmware may override an existing LED setting. For example, in the case that the firmware resets the camera (as can happen with a CSI-2 timeout), the LED may also be reset. If you wish to guarantee that the LED remain off at all times, you may prefer to use the `disable_camera_led` option in config.txt (this has the added advantage that sudo privileges and GPIO access are not required, at least for LED control).

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:

- `'average'`
- `'spot'`
- `'backlit'`
- `'matrix'`

When set, the property adjusts the camera's metering mode. All modes set up two regions: a center region, and an outer region. The major difference between each mode is the size of the center region. The `'backlit'` mode has the largest central region (30% of the width), while `'spot'` has the smallest (10% of the width).

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.

overlays
Retrieves all active *PiRenderer* overlays.

If no overlays are current active, *overlays* will return an empty iterable. Otherwise, it will return an iterable of *PiRenderer* instances which are currently acting as overlays. Note that the preview renderer is an exception to this: it is *not* included as an overlay despite being derived from *PiRenderer*.

New in version 1.8.

preview
Retrieves the *PiRenderer* displaying the camera preview.

If no preview is currently active, *preview* will return `None`. Otherwise, it will return the instance of *PiRenderer* which is currently connected to the camera's preview port for rendering what the camera sees. You can use the attributes of the *PiRenderer* class to configure the appearance of the preview. For example, to make the preview semi-transparent:

```
import picamera

with picamera.PiCamera() as camera:
    camera.start_preview()
    camera.preview.alpha = 128
```

New in version 1.8.

preview_alpha
Retrieves or sets the opacity of the preview window.

Deprecated since version 1.8: Please use the *alpha* attribute of the *preview* object instead.

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

Deprecated since version 1.8: Please use the *fullscreen* attribute of the *preview* object instead.

preview_layer
Retrieves or sets the layer of the preview window.

Deprecated since version 1.8: Please use the *layer* attribute of the *preview* object instead.

preview_window
Retrieves or sets the size of the preview window.

Deprecated since version 1.8: Please use the *window* attribute of the *preview* object instead.

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

Deprecated since version 1.8: Test whether *preview* is `None` instead.

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 configures the camera so that the next call to these methods will use the new resolution. The resolution can be specified as a `(width, height)` tuple, as a string formatted `'WIDTHxHEIGHT'`, or as a string containing a commonly recognized display resolution name (e.g. "VGA", "HD", "1080p", etc). For example, the following definitions are all equivalent:

```
camera.resolution = (1280, 720)
camera.resolution = '1280x720'
camera.resolution = '1280 x 720'
camera.resolution = 'HD'
camera.resolution = '720p'
```

The camera must not be closed, and no recording must be active when the property is set.

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 *sensor_mode* for more information.

The initial value of this property can be specified with the *resolution* parameter in the *PiCamera* constructor, and will default to the display's resolution or 1280x720 if the display has been disabled (with `tvservice -o`).

Changed in version 1.11: Resolution permitted to be set as a string. Preview resolution added as separate property.

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 rotation applied to the camera's input. 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.

sensor_mode
Retrieves or sets the input mode of the camera's sensor.

This is an advanced property which can be used to control the camera's sensor mode. By default, mode 0 is used which allows the camera to automatically select an input mode based on the requested *resolution* and *framerate*. Valid values are currently between 0 and 7. The set of valid sensor modes (along with the heuristic used to select one automatically) are detailed in the *Camera Modes* section of the documentation.

Note: At the time of writing, setting this property does nothing unless the camera has been initialized with a sensor mode other than 0. Furthermore, some mode transitions appear to require setting the property twice (in a row). This appears to be a firmware limitation.

The initial value of this property can be specified with the *sensor_mode* parameter in the *PiCamera* constructor, and will default to 0 if not specified.

New in version 1.9.

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 by the auto-exposure algorithm. 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).

Note: You can query the *exposure_speed* attribute to determine the actual shutter speed being used when this attribute is set to 0. Please note that this capability requires an up to date firmware (#692 or later).

Note: In later firmwares, this attribute is limited by the value of the *framerate* attribute. For example, if framerate is set to 30fps, the shutter speed cannot be slower than 33,333µs (1/fps).

still_stats
Retrieves or sets whether statistics will be calculated from still frames or the prior preview frame.

When queried, the *still_stats* property returns a boolean value indicating when scene statistics will be calculated for still captures (that is, captures where the *use_video_port* parameter of *capture()* is `False`). When this property is `False` (the default), statistics will be calculated from the preceding preview frame (this also applies when the preview is not visible). When *True*, statistics will be calculated from the captured image itself.

When set, the propetry controls when scene statistics will be calculated for still captures. The property can be set while recordings or previews are in progress. The default value is `False`.

The advantages to calculating scene statistics from the captured image are that time between startup and capture is reduced as only the AGC (automatic gain control) has to converge. The downside is that processing time for captures increases and that white balance and gain won't necessarily match the preview.

New in version 1.9.

timestamp
Retrieves the system time according to the camera firmware.

The camera's timestamp is a 64-bit integer representing the number of microseconds since the last system boot. When the camera's *clock_mode* is `'raw'` the values returned by this attribute are comparable to those from the *frame timestamp* attribute.

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_denoise
Retrieves or sets whether denoise will be applied to video recordings.

When queried, the *video_denoise* property returns a boolean value indicating whether or not the camera software will apply a denoise algorithm to video recordings.

When set, the property activates or deactivates the denoise algorithm for video recordings. The property can be set while recordings or previews are in progress. The default value is `True`.

New in version 1.7.

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.

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

When queried, the *zoom* 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 (this is also known as 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.

2.8.2 PiVideoFrameType

class picamera.**PiVideoFrameType**
This class simply defines constants used to represent the type of a frame in *PiVideoFrame.frame_type*. Effectively it is a namespace for an enum.

frame
Indicates a predicted frame (P-frame). This is the most common frame type.

key_frame
Indicates an intra-frame (I-frame) also known as a key frame.

sps_header
Indicates an inline SPS/PPS header (rather than picture data) which is typically used as a split point.

motion_data
Indicates the frame is inline motion vector data, rather than picture data.

New in version 1.5.

2.8.3 PiVideoFrame

class picamera.**PiVideoFrame**(*index*, *frame_type*, *frame_size*, *video_size*, *split_size*, *timestamp*)
This class is a namedtuple derivative used to store information about a video frame. It is recommended that you access the information stored by this class by attribute name rather than position (for example: `frame.index` rather than `frame[0]`).

index
Returns the zero-based number of the frame. This is a monotonic counter that is simply incremented every time the camera starts outputting a new frame. As a consequence, this attribute cannot be used to detect dropped frames. Nor does it necessarily represent actual frames; it will be incremented for SPS headers and motion data buffers too.

frame_type
Returns a constant indicating the kind of data that the frame contains (see *PiVideoFrameType*). Please note that certain frame types contain no image data at all.

frame_size
Returns the size in bytes of the current frame. If a frame is written in multiple chunks, this value will increment while *index* remains static. Query *complete* to determine whether the frame has been completely output yet.

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. 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. When the

camera's clock mode is `'reset'` (the default), this is the number of microseconds (millionths of a second) since video recording started. When the camera's *clock_mode* is `'raw'`, this is the number of microseconds since the last system reboot. See *timestamp* for more information.

> **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.

complete
Returns a bool indicating whether the current frame is complete or not. If the frame is complete then *frame_size* will not increment any further, and will reset for the next frame.

Changed in version 1.5: Deprecated *header* and *keyframe* attributes and added the new *frame_type* attribute instead.

Changed in version 1.9: Added the *complete* attribute.

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.

Deprecated since version 1.5: Please compare *frame_type* to *PiVideoFrameType.sps_header* instead.

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

Deprecated since version 1.5: Please compare *frame_type* to *PiVideoFrameType.key_frame* instead.

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

2.9 API - Streams

The picamera library defines a few custom stream implementations useful for implementing certain common use cases (like security cameras which only record video upon some external trigger like a motion sensor).

2.9.1 PiCameraCircularIO

class picamera.**PiCameraCircularIO**(*camera*, *size=None*, *seconds=None*, *bitrate=17000000*, *splitter_port=1*)
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 exception to this is the *clear()* method.

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*.

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 do specify something else, ensure it is equal to the *splitter_port* parameter of the corresponding call to *start_recording()*. For example:

```
import picamera

with picamera.PiCamera() as camera:
    with picamera.PiCameraCircularIO(camera, splitter_port=2) as stream:
        camera.start_recording(stream, format='h264', splitter_port=2)
        camera.wait_recording(10, splitter_port=2)
        camera.stop_recording(splitter_port=2)
```

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.

clear()
Resets the stream to empty safely.

This method truncates the stream to empty, and clears the associated frame meta-data too, ensuring that subsequent writes operate correctly (see the warning in the *PiCameraCircularIO* class documentation).

copy_to(*output*, *size=None*, *seconds=None*, *first_frame=2*)
Copies content from the stream to *output*.

By default, this method copies all complete frames from the circular stream to the filename or file-like object given by *output*.

If *size* is specified then the copy will be limited to the whole number of frames that fit within the specified number of bytes. If *seconds* if specified, then the copy will be limited to that number of seconds worth of frames. Only one of *size* or *seconds* can be specified. If neither is specified, all frames are copied.

If *first_frame* is specified, it defines the frame type of the first frame to be copied. By default this is *sps_header* as this must usually be the first frame in an H264 stream. If *first_frame* is `None`, not such limit will be applied.

> **Warning:** Note that if a frame of the specified type (e.g. SPS header) cannot be found within the specified number of seconds or bytes then this method will simply copy nothing (but no error will be raised).

The stream's position is not affected by this method.

2.9.2 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 `io.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()*.

readall()
Read and return all bytes from the stream until EOF, using multiple calls to the stream if necessary.

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.

2.9.3 BufferIO

class picamera.**BufferIO**(*obj*)
A stream which uses a writeable memoryview for storage.

This is used internally by picamera for capturing directly to an existing object which supports the buffer protocol (like a numpy array). Because the underlying storage is fixed in size, the stream also has a fixed size and will raise an IOError exception if an attempt is made to write beyond the end of the buffer (though seek beyond the end is supported).

Users should never need this class directly.

getvalue()
Return bytes containing the entire contents of the buffer.

read(*n=-1*)
Read up to *n* bytes from the buffer 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 buffer position to the end of the buffer.

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

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

readall()
Read and return all bytes from the buffer until EOF.

seek(*offset*, *whence=0*)
Change the buffer 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 buffer (the default); *offset* should be zero or positive
- SEEK_CUR or 1 – current buffer position; *offset* may be negative
- SEEK_END or 2 – end of the buffer; *offset* is usually negative

Return the new absolute position.

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

tell()
Return the current buffer position.

truncate(*size=None*)
Raises NotImplementedError as the underlying buffer cannot be resized.

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

write(*b*)
Write the given bytes or bytearray object, *b*, to the underlying buffer and return the number of bytes written. If the underlying buffer isn't large enough to contain all the bytes of *b*, as many bytes as possible will be written before raising IOError.

size
Return the maximum size of the buffer in bytes.

2.10 API - Renderers

Renderers are used by the camera to provide preview and overlay functionality on the Pi's display. Users will rarely need to construct instances of these classes directly (*start_preview()* and *add_overlay()* are generally used instead) but may find the attribute references for them useful.

2.10.1 PiRenderer

class picamera.**PiRenderer**(*parent*, *layer=0*, *alpha=255*, *fullscreen=True*, *window=None*, *crop=None*, *rotation=0*, *vflip=False*, *hflip=False*)

Wraps *MMALRenderer* for use by PiCamera.

The *parent* parameter specifies the *PiCamera* instance that has constructed this renderer. The *layer* parameter specifies the layer that the renderer will inhabit. Higher numbered layers obscure lower numbered layers (unless they are partially transparent). The initial opacity of the renderer is specified by the *alpha* parameter (which defaults to 255, meaning completely opaque). The *fullscreen* parameter which defaults to `True` indicates whether the renderer should occupy the entire display. Finally, the *window* parameter (which only has meaning when *fullscreen* is `False`) is a four-tuple of `(x, y, width, height)` which gives the screen coordinates that the renderer should occupy when it isn't full-screen.

This base class isn't directly used by *PiCamera*, but the two derivatives defined below, *PiOverlayRenderer* and *PiPreviewRenderer*, are used to produce overlays and the camera preview respectively.

close()

Finalizes the renderer and deallocates all structures.

This method is called by the camera prior to destroying the renderer (or more precisely, letting it go out of scope to permit the garbage collector to destroy it at some future time).

alpha

Retrieves or sets the opacity of the renderer.

When queried, the *alpha* property returns a value between 0 and 255 indicating the opacity of the renderer, 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.

crop

Retrieves or sets the area to read from the source.

The *crop* property specifies the rectangular area that the renderer will read from the source as a 4-tuple of `(x, y, width, height)`. The special value `(0, 0, 0, 0)` (which is also the default) means to read entire area of the source. The property can be set while recordings or previews are active.

For example, if the camera's resolution is currently configured as 1280x720, setting this attribute to `(160, 160, 640, 400)` will crop the preview to the center 640x400 pixels of the input. Note that this property does not affect the size of the output rectangle, which is controlled with *fullscreen* and *window*.

Note: This property only affects the renderer; it has no bearing on image captures or recordings (unlike the *zoom* property of the *PiCamera* class).

---

fullscreen

Retrieves or sets whether the renderer appears full-screen.

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

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

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

Note: This property only affects the renderer; it has no bearing on image captures or recordings (unlike the *hflip* property of the *PiCamera* class).

---

layer
Retrieves or sets the layer of the renderer.

The *layer* property is an integer which controls the layer that the renderer occupies. Higher valued layers obscure lower valued layers (with 0 being the "bottom" layer). The default value is 2. The property can be set while recordings or previews are in progress.

rotation
Retrieves or sets the current rotation of the renderer.

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

When set, the property changes the rotation applied to the renderer's output. The property can be set while recordings or previews are active. The default is 0.

Note: This property only affects the renderer; it has no bearing on image captures or recordings (unlike the *rotation* property of the *PiCamera* class).

---

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

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

Note: This property only affects the renderer; it has no bearing on image captures or recordings (unlike the *vflip* property of the *PiCamera* class).

---

window
Retrieves or sets the size of the renderer.

When the *fullscreen* property is set to `False`, the *window* property specifies the size and position of the renderer 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.

2.10.2 PiOverlayRenderer

class picamera.**PiOverlayRenderer**(*parent*, *source*, *resolution=None*, *layer=0*, *alpha=255*, *fullscreen=True*, *window=None*, *crop=None*, *rotation=0*, *vflip=False*, *hflip=False*)
Represents an *MMALRenderer* with a static source for overlays.

This class descends from *PiRenderer* and adds a static source for the *MMALRenderer*. The optional *resolution* parameter specifies the size of the source image as a `(width, height)` tuple. If this is omitted or `None` then the resolution is assumed to be the same as the parent camera's current *resolution*.

The *source* must be an object that supports the buffer protocol which has the same length as an image in RGB format (colors represented as interleaved unsigned bytes) with the specified *resolution* after the width

has been rounded up to the nearest multiple of 32, and the height has been rounded up to the nearest multiple of 16.

For example, if *resolution* is `(1280, 720)`, then *source* must be a buffer with length 1280 x 720 x 3 bytes, or 2,764,800 bytes (because 1280 is a multiple of 32, and 720 is a multiple of 16 no extra rounding is required). However, if *resolution* is `(97, 57)`, then *source* must be a buffer with length 128 x 64 x 3 bytes, or 24,576 bytes (pixels beyond column 97 and row 57 in the source will be ignored).

The *layer*, *alpha*, *fullscreen*, and *window* parameters are the same as in `PiRenderer`.

update(*source*)
Update the overlay with a new source of data.

The new *source* buffer must have the same size as the original buffer used to create the overlay. There is currently no method for changing the size of an existing overlay (remove and recreate the overlay if you require this).

2.10.3 PiPreviewRenderer

class picamera.**PiPreviewRenderer**(*parent*, *source*, *resolution=None*, *layer=2*, *alpha=255*, *fullscreen=True*, *window=None*, *crop=None*, *rotation=0*, *vflip=False*, *hflip=False*)
Represents an `MMALRenderer` which uses the camera's preview as a source.

This class descends from `PiRenderer` and adds an `MMALConnection` to connect the renderer to an MMAL port. The *source* parameter specifies the `MMALPort` to connect to the renderer.

The *layer*, *alpha*, *fullscreen*, and *window* parameters are the same as in `PiRenderer`.

resolution
Retrieves or sets the resolution of the preview renderer.

By default, the preview's resolution matches the camera's resolution. However, particularly high resolutions (such as the maximum resolution of the V2 camera module) can cause issues. In this case, you may wish to set a lower resolution for the preview that the camera's resolution.

When queried, the `resolution` property returns `None` if the preview's resolution is derived from the camera's. In this case, changing the camera's resolution will also cause the preview's resolution to change. Otherwise, it returns the current preview resolution as a tuple.

Note: The preview resolution cannot be greater than the camera's resolution (in either access). If you set a preview resolution, then change the camera's resolution below the preview's resolution, this property will silently revert to `None`, meaning the preview's resolution will follow the camera's resolution.

When set, the property reconfigures the preview renderer with the new resolution. As a special case, setting the property to `None` will cause the preview to follow the camera's resolution once more. The property can be set while recordings are in progress. The default is `None`.

Note: This property only affects the renderer; it has no bearing on image captures or recordings (unlike the `resolution` property of the `PiCamera` class).

New in version 1.11.

2.10.4 PiNullSink

class picamera.**PiNullSink**(*parent*, *source*)
Implements an `MMALNullSink` which can be used in place of a renderer.

The *parent* parameter specifies the *PiCamera* instance which constructed this *MMALNullSink*. The *source* parameter specifies the *MMALPort* which the null-sink should connect to its input.

The null-sink can act as a drop-in replacement for *PiRenderer* in most cases, but obviously doesn't implement attributes like `alpha`, `layer`, etc. as it simply dumps any incoming frames. This is also the reason that this class doesn't derive from *PiRenderer* like all other classes in this module.

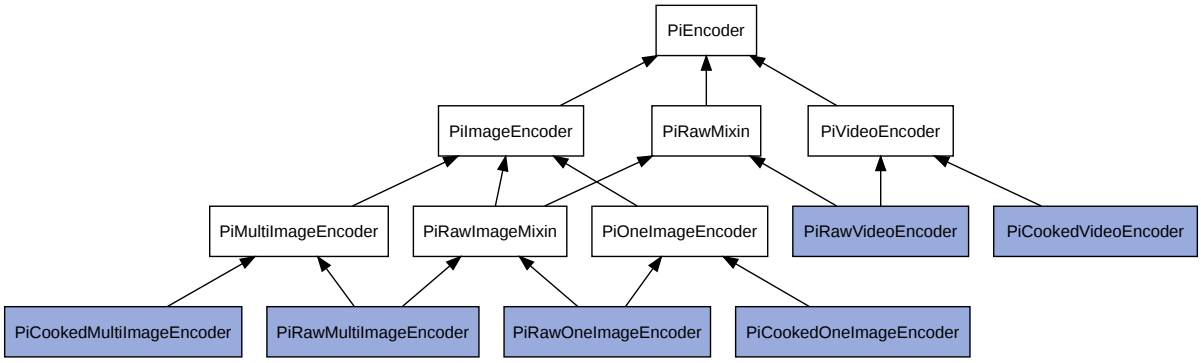
close()
Finalizes the null-sink and deallocates all structures.

This method is called by the camera prior to destroying the null-sink (or more precisely, letting it go out of scope to permit the garbage collector to destroy it at some future time).

2.11 API - Encoders

Encoders are typically used by the camera to compress captured images or video frames for output to disk. However, picamera also has classes representing "unencoded" output (raw RGB, etc). Most users will have no direct need to use these classes directly, but advanced users may find them useful as base classes for *Custom encoders*.

The inheritance diagram for the following classes is displayed below:



2.11.1 PiEncoder

class picamera.**PiEncoder**(*parent*, *camera_port*, *input_port*, *format*, *resize*, ***options*)
Base implementation of an MMAL encoder for use by PiCamera.

The *parent* parameter specifies the *PiCamera* instance that has constructed the encoder. The *camera_port* parameter provides the MMAL camera port that the encoder should enable for capture (this will be the still or video port of the camera component). The *input_port* parameter specifies the MMAL port that the encoder should connect to its input. Sometimes this will be the same as the camera port, but if other components are present in the pipeline (e.g. a splitter), it may be different.

The *format* parameter specifies the format that the encoder should produce in its output. This is specified as a string and will be one of the following for image encoders:

- 'jpeg'
- 'png'
- 'gif'
- 'bmp'
- 'yuv'
- 'rgb'
- 'rgba'
- 'bgr'

- 'bgra'

And one of the following for video encoders:

- 'h264'
- 'mjpeg'

The *resize* parameter is either None (indicating no resizing should take place), or a (width, height) tuple specifying the resolution that the output of the encoder should be resized to.

Finally, the *options* parameter specifies additional keyword arguments that can be used to configure the encoder (e.g. bitrate for videos, or quality for images).

camera_port
The *MMALVideoPort* that needs to be activated and deactivated in order to start/stop capture. This is not necessarily the port that the encoder component's input port is connected to (for example, in the case of video-port based captures, this will be the camera video port behind the splitter).

encoder
The *MMALComponent* representing the encoder, or None if no encoder component has been created (some encoder classes don't use an actual encoder component, for example *PiRawImageMixin*).

event
A threading.Event instance used to synchronize operations (like start, stop, and split) between the control thread and the callback thread.

exception
If an exception occurs during the encoder callback, this attribute is used to store the exception until it can be re-raised in the control thread.

format
The image or video format that the encoder is expected to produce. This is equal to the value of the *format* parameter.

input_port
The *MMALVideoPort* that the encoder should be connected to.

output_port
The *MMALVideoPort* that produces the encoder's output. In the case no encoder component is created, this should be the camera/component output port responsible for producing data. In other words, this attribute **must** be set on initialization.

outputs
A mapping of key to (output, opened) tuples where output is a file-like object, and opened is a bool indicating whether or not we opened the output object (and thus whether we are responsible for eventually closing it).

outputs_lock
A threading.Lock() instance used to protect access to *outputs*.

parent
The *PiCamera* instance that created this PiEncoder instance.

pool
A pointer to a pool of MMAL buffers.

resizer
The *MMALResizer* component, or None if no resizer component has been created.

_callback(*port*, *buf*)
The encoder's main callback function.

When the encoder is active, this method is periodically called in a background thread. The *port* parameter specifies the MMALPort providing the output (typically this is the encoder's output port, but in the case of unencoded captures may simply be a camera port), while the *buf* parameter is an *MMALBuffer* which can be used to obtain the data to write, along with meta-data about the current frame.

This method must set *event* when the encoder has finished (and should set *exception* if an exception occurred during encoding).

Developers wishing to write a custom encoder class may find it simpler to override the *_callback_write()* method, rather than deal with these complexities.

_callback_write(*buf*, *key=0*)
Writes output on behalf of the encoder callback function.

This method is called by *_callback()* to handle writing to an object in *outputs* identified by *key*. The *buf* parameter is an *MMALBuffer* which can be used to obtain the data. The method is expected to return a boolean to indicate whether output is complete (`True`) or whether more data is expected (`False`).

The default implementation simply writes the contents of the buffer to the output identified by *key*, and returns `True` if the buffer flags indicate end of stream. Image encoders will typically override the return value to indicate `True` on end of frame (as they only wish to output a single image). Video encoders will typically override this method to determine where key-frames and SPS headers occur.

_close_output(*key=0*)
Closes the output associated with *key* in *outputs*.

Closes the output object associated with the specified *key*, and removes it from the *outputs* dictionary (if we didn't open the object then we attempt to flush it instead).

_create_encoder(*format*)
Creates and configures the *MMALEncoder* component.

This method only constructs the encoder; it does not connect it to the input port. The method sets the *encoder* attribute to the constructed encoder component, and the *output_port* attribute to the encoder's output port (or the previously constructed resizer's output port if one has been requested). Descendent classes extend this method to finalize encoder configuration.

Note: It should be noted that this method is called with the initializer's `option` keyword arguments. This base implementation expects no additional arguments, but descendent classes extend the parameter list to include options relevant to them.

_create_resizer(*width*, *height*)
Creates and configures an *MMALResizer* component.

This is called when the initializer's *resize* parameter is something other than `None`. The *width* and *height* parameters are passed to the constructed resizer. Note that this method only constructs the resizer - it does not connect it to the encoder. The method sets the *resizer* attribute to the constructed resizer component.

_open_output(*output*, *key=0*)
Opens *output* and associates it with *key* in *outputs*.

If *output* is a string, this method opens it as a filename and keeps track of the fact that the encoder was the one to open it (which implies that *_close_output()* should eventually close it). Otherwise, if *output* has a `write` method it is assumed to be a file-like object and it is used verbatim. If *output* is neither a string, nor an object with a `write` method it is assumed to be a writeable object supporting the buffer protocol (this is wrapped in a *BufferIO* stream to simplify writing).

The opened output is added to the *outputs* dictionary with the specified *key*.

close()
Finalizes the encoder and deallocates all structures.

This method is called by the camera prior to destroying the encoder (or more precisely, letting it go out of scope to permit the garbage collector to destroy it at some future time). The method destroys all components that the various create methods constructed and resets their attributes.

start(*output*)
Starts the encoder object writing to the specified output.

This method is called by the camera to start the encoder capturing data from the camera to the specified output. The *output* parameter is either a filename, or a file-like object (for image and video encoders), or an iterable of filenames or file-like objects (for multi-image encoders).

stop()
Stops the encoder, regardless of whether it's finished.

This method is called by the camera to terminate the execution of the encoder. Typically, this is used with video to stop the recording, but can potentially be called in the middle of image capture to terminate the capture.

wait(*timeout=None*)
Waits for the encoder to finish (successfully or otherwise).

This method is called by the owning camera object to block execution until the encoder has completed its task. If the *timeout* parameter is None, the method will block indefinitely. Otherwise, the *timeout* parameter specifies the (potentially fractional) number of seconds to block for. If the encoder finishes successfully within the timeout, the method returns `True`. Otherwise, it returns `False`.

active
Returns `True` if the MMAL encoder exists and is enabled.

2.11.2 PiVideoEncoder

class picamera.**PiVideoEncoder**(*parent*, *camera_port*, *input_port*, *format*, *resize*, ***options*)
Encoder for video recording.

This derivative of *PiEncoder* configures itself for H.264 or MJPEG encoding. It also introduces a *split()* method which is used by *split_recording()* and *record_sequence()* to redirect future output to a new filename or object. Finally, it also extends *PiEncoder.start()* and *PiEncoder._callback_write()* to track video frame meta-data, and to permit recording motion data to a separate output object.

encoder_type
alias of `MMALVideoEncoder`

_callback_write(*buf*, *key=0*)
Extended to implement video frame meta-data tracking, and to handle splitting video recording to the next output when *split()* is called.

_create_encoder(*format*, *bitrate=17000000*, *intra_period=None*, *profile='high'*, *quantization=0*, *quality=0*, *inline_headers=True*, *sei=False*, *motion_output=None*, *intra_refresh=None*, *level='4'*)
Extends the base *_create_encoder()* implementation to configure the video encoder for H.264 or MJPEG output.

request_key_frame()
Called to request an I-frame from the encoder.

This method is called by *request_key_frame()* and *split()* to force the encoder to output an I-frame as soon as possible.

split(*output*, *motion_output=None*)
Called to switch the encoder's output.

This method is called by *split_recording()* and *record_sequence()* to switch the encoder's `output` object to the *output* parameter (which can be a filename or a file-like object, as with *start()*).

start(*output*, *motion_output=None*)
Extended to initialize video frame meta-data tracking.

2.11.3 PiImageEncoder

class picamera.**PiImageEncoder**(*parent*, *camera_port*, *input_port*, *format*, *resize*, ***options*)
Encoder for image capture.

This derivative of *PiEncoder* extends the *_create_encoder()* method to configure the encoder for a variety of encoded image outputs (JPEG, PNG, etc.).

encoder_type
alias of MMALImageEncoder

_create_encoder(*format*, *quality=85*, *thumbnail=(64*, *48*, *35)*)
Extends the base *_create_encoder()* implementation to configure the image encoder for JPEG, PNG, etc.

2.11.4 PiRawMixin

class picamera.**PiRawMixin**(*parent*, *camera_port*, *input_port*, *format*, *resize*, ***options*)
Mixin class for "raw" (unencoded) output.

This mixin class overrides the initializer of *PiEncoder*, along with _create_resizer() and *_create_encoder()* to configure the pipeline for unencoded output. Specifically, it disables the construction of an encoder, and sets the output port to the input port passed to the initializer, unless resizing is required (either for actual resizing, or for format conversion) in which case the resizer's output is used.

_callback_write(*buf*, *key=0*)
Overridden to strip alpha bytes when required.

_create_encoder(*format*)
Overridden to skip creating an encoder. Instead, this class simply uses the resizer's port as the output port (if a resizer has been configured) or the specified input port otherwise.

2.11.5 PiCookedVideoEncoder

class picamera.**PiCookedVideoEncoder**(*parent*, *camera_port*, *input_port*, *format*, *resize*, ***options*)
Video encoder for encoded recordings.

This class is a derivative of *PiVideoEncoder* and only exists to provide naming symmetry with the image encoder classes.

2.11.6 PiRawVideoEncoder

class picamera.**PiRawVideoEncoder**(*parent*, *camera_port*, *input_port*, *format*, *resize*, ***options*)
Video encoder for unencoded recordings.

This class is a derivative of *PiVideoEncoder* and the *PiRawMixin* class intended for use with *start_recording()* when it is called with an unencoded format.

Warning: This class creates an inheritance diamond. Take care to determine the MRO of super-class calls.

2.11.7 PiOneImageEncoder

class picamera.**PiOneImageEncoder**(*parent*, *camera_port*, *input_port*, *format*, *resize*, ***options*)
Encoder for single image capture.

This class simply extends *_callback_write()* to terminate capture at frame end (i.e. after a single frame has been received).

2.11.8 PiMultiImageEncoder

class picamera.**PiMultiImageEncoder**(*parent*, *camera_port*, *input_port*, *format*, *resize*, ***options*)

Encoder for multiple image capture.

This class extends *PiImageEncoder* to handle an iterable of outputs instead of a single output. The *_callback_write()* method is extended to terminate capture when the iterable is exhausted, while *PiEncoder._open_output()* is overridden to begin iteration and rely on the new *_next_output()* method to advance output to the next item in the iterable.

_next_output(*key=0*)

This method moves output to the next item from the iterable passed to *start()*.

2.11.9 PiRawImageMixin

class picamera.**PiRawImageMixin**(*parent*, *camera_port*, *input_port*, *format*, *resize*, ***options*)

Mixin class for "raw" (unencoded) image capture.

The *_callback_write()* method is overridden to manually calculate when to terminate output.

_callback_write(*buf*, *key=0*)

Overridden to manually calculate when to terminate capture (see comments in __init__()).

2.11.10 PiCookedOneImageEncoder

class picamera.**PiCookedOneImageEncoder**(*parent*, *camera_port*, *input_port*, *format*, *resize*, ***options*)

Encoder for "cooked" (encoded) single image output.

This encoder extends *PiOneImageEncoder* to include Exif tags in the output.

2.11.11 PiRawOneImageEncoder

class picamera.**PiRawOneImageEncoder**(*parent*, *camera_port*, *input_port*, *format*, *resize*, ***options*)

Single image encoder for unencoded capture.

This class is a derivative of *PiOneImageEncoder* and the *PiRawImageMixin* class intended for use with *capture()* (et al) when it is called with an unencoded image format.

Warning: This class creates an inheritance diamond. Take care to determine the MRO of super-class calls.

2.11.12 PiCookedMultiImageEncoder

class picamera.**PiCookedMultiImageEncoder**(*parent*, *camera_port*, *input_port*, *format*, *resize*, ***options*)

Encoder for "cooked" (encoded) multiple image output.

This encoder descends from *PiMultiImageEncoder* but includes no new functionality as video-port based encodes (which is all this class is used for) don't support Exif tag output.

2.11.13 PiRawMultiImageEncoder

class picamera.**PiRawMultiImageEncoder**(*parent*, *camera_port*, *input_port*, *format*, *resize*, ***options*)

Multiple image encoder for unencoded capture.

This class is a derivative of *PiMultiImageEncoder* and the *PiRawImageMixin* class intended for use with *capture_sequence()* when it is called with an unencoded image format.

Warning: This class creates an inheritance diamond. Take care to determine the MRO of super-class calls.

2.12 API - Exceptions

All exceptions defined by picamera are listed in this section. All exception classes utilize multiple inheritance in order to make testing for exception types more intuitive. For example, *PiCameraValueError* derives from both *PiCameraError* and ValueError. Hence it will be caught by blocks intended to catch any error specific to the picamera library:

```
try:
    camera.brightness = int(some_user_value)
except PiCameraError:
    print('Something went wrong with the camera')
```

Or by blocks intended to catch value errors:

```
try:
    camera.contrast = int(some_user_value)
except ValueError:
    print('Invalid value')
```

2.12.1 Warnings

exception picamera.**PiCameraWarning**
Base class for PiCamera warnings.

exception picamera.**PiCameraDeprecated**
Raised when deprecated functionality in picamera is used.

exception picamera.**PiCameraFallback**
Raised when picamera has to fallback on old functionality.

exception picamera.**PiCameraResizerEncoding**
Raised when picamera uses a resizer purely for encoding purposes.

exception picamera.**PiCameraAlphaStripping**
Raised when picamera does alpha-byte stripping.

2.12.2 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.

exception picamera.**PiCameraClosed**
Raised when a method is called on a camera which has already been closed.

exception picamera.**PiCameraNotRecording**
Raised when *stop_recording()* or *split_recording()* are called against a port which has no recording active.

exception picamera.**PiCameraAlreadyRecording**
Raised when *start_recording()* or *record_sequence()* are called against a port which already has an active recording.

exception picamera.**PiCameraMMALError**(*status*, *prefix=''*)
Raised when an MMAL operation fails for whatever reason.

2.12.3 Functions

picamera.**mmal_check**(*status*, *prefix=''*)
Checks the return status of an mmal call and raises an exception on failure.

The *status* parameter is the result of an MMAL call. If *status* is anything other than MMAL_SUCCESS, a *PiCameraMMALError* exception is raised. The optional *prefix* parameter specifies a prefix message to place at the start of the exception's message to provide some context.

2.13 API - Colors and Color Matching

The picamera library includes a comprehensive *Color* class which is capable of converting between numerous color representations and calculating color differences. Various ancillary classes can be used to manipulate aspects of a color.

2.13.1 Color

class picamera.**Color**
The Color class is a tuple which represents a color as red, green, and blue components.

The class has a flexible constructor which allows you to create an instance from a variety of color systems including RGB, Y'UV, Y'IQ, HLS, and HSV. There are also explicit constructors for each of these systems to allow you to force the use of a system in your code. For example, an instance of *Color* can be constructed in any of the following ways:

```
>>> Color('#f00')
<Color "#ff0000">
>>> Color('green')
<Color "#008000">
>>> Color(0, 0, 1)
<Color "#0000ff">
>>> Color(hue=0, saturation=1, value=0.5)
<Color "#7f0000">
>>> Color(y=0.4, u=-0.05, v=0.615)
<Color "#ff0f4c">
```

The specific forms that the default constructor will accept are enumerated below:

Style	Description
Single positional parameter	Equivalent to calling *Color.from_string()*.
Three positional parameters	Equivalent to calling *Color.from_rgb()* if all three parameters are between 0.0 and 1.0, or *Color.from_rgb_bytes()* otherwise.
Three named parameters, "r", "g", "b"	
Three named parameters, "red", "green", "blue"	
Three named parameters, "y", "u", "v"	Equivalent to calling *Color.from_yuv()* if "y" is between 0.0 and 1.0, "u" is between -0.436 and 0.436 and "v" is

If the constructor parameters do not conform to any of the variants in the table above, a `ValueError` will be thrown.

Internally, the color is *always* represented as 3 float values corresponding to the red, green, and blue components of the color. These values take a value from 0.0 to 1.0 (least to full intensity). The class provides several attributes which can be used to convert one color system into another:

```
>>> Color('#f00').hls
(0.0, 0.5, 1.0)
>>> Color.from_string('green').hue
Hue(deg=120.0)
>>> Color.from_rgb_bytes(0, 0, 255).yuv
(0.114, 0.435912, -0.099978)
```

As *Color* derives from tuple, instances are immutable. While this provides the advantage that they can be used as keys in a dict, it does mean that colors themselves cannot be directly manipulated (e.g. by reducing the red component).

However, several auxilliary classes in the module provide the ability to perform simple transformations of colors via operators which produce a new *Color* instance. For example:

```
>>> Color('red') - Red(0.5)
<Color "#7f0000">
>>> Color('green') + Red(0.5)
<Color "#7f8000">
>>> Color.from_hls(0.5, 0.5, 1.0)
<Color "#00feff">
>>> Color.from_hls(0.5, 0.5, 1.0) * Lightness(0.8)
<Color "#00cbcc">
>>> (Color.from_hls(0.5, 0.5, 1.0) * Lightness(0.8)).hls
(0.5, 0.4, 1.0)
```

From the last example above one can see that even attributes not directly stored by the color (such as lightness) can be manipulated in this fashion. In this case a *Color* instance is constructed from HLS (hue, lightness, saturation) values with a lightness of 0.5. This is multiplied by a *Lightness* instance with a value of 0.8 which constructs a new *Color* with the same hue and saturation, but a lightness of 0.5 * 0.8 = 0.4.

If an instance is converted to a string (with `str()`) it will return a string containing the 7-character HTML code for the color (e.g. "#ff0000" for red). As can be seen in the examples above, a similar representation is returned for `repr()`.

difference(*other*, *method='euclid'*)

Determines the difference between this color and *other* using the specified *method*. The *method* is specified as a string, and the following methods are valid:

- 'euclid' - This is the default method. Calculate the Euclidian distance. This is by far the fastest method, but also the least accurate in terms of human perception.
- 'cie1976' - Use the CIE 1976 formula for calculating the difference between two colors in CIE Lab space.
- 'cie1994g' - Use the CIE 1994 formula with the "graphic arts" bias for calculating the difference.
- 'cie1994t' - Use the CIE 1994 forumula with the "textiles" bias for calculating the difference.
- 'cie2000' - Use the CIEDE 2000 formula for calculating the difference.

Note that the Euclidian distance will be significantly different to the other calculations; effectively this just measures the distance between the two colors by treating them as coordinates in a three dimensional Euclidian space. All other methods are means of calculating a Delta E value in which 2.3 is considered a just-noticeable difference (JND).

> **Warning:** This implementation has yet to receive any significant testing (constructor methods for CIELab need to be added before this can be done).

classmethod from_cie_lab(*l*, *a*, *b*)
Construct a *Color* from (L*, a*, b*) float values representing a color in the CIE Lab color space. The conversion assumes the sRGB working space with reference white D65.

classmethod from_cie_luv(*l*, *u*, *v*)
Construct a *Color* from (L*, u*, v*) float values representing a color in the CIE Luv color space. The conversion assumes the sRGB working space with reference white D65.

classmethod from_cie_xyz(*x*, *y*, *z*)
Construct a *Color* from (X, Y, Z) float values representing a color in the CIE 1931 color space. The conversion assumes the sRGB working space with reference white D65.

classmethod from_hls(*h*, *l*, *s*)
Construct a *Color* from HLS (hue, lightness, saturation) floats between 0.0 and 1.0.

classmethod from_hsv(*h*, *s*, *v*)
Construct a *Color* from HSV (hue, saturation, value) floats between 0.0 and 1.0.

classmethod from_rgb(*r*, *g*, *b*)
Construct a *Color* from three RGB float values between 0.0 and 1.0.

classmethod from_rgb_565(*n*)
Construct a *Color* from an unsigned 16-bit integer number in RGB565 format.

classmethod from_rgb_bytes(*r*, *g*, *b*)
Construct a *Color* from three RGB byte values between 0 and 255.

classmethod from_string(*s*)
Construct a *Color* from a 4 or 7 character CSS-like representation (e.g. "#f00" or "#ff0000" for red), or from one of the named colors (e.g. "green" or "wheat") from the CSS standard. Any other string format will result in a `ValueError`.

classmethod from_yiq(*y*, *i*, *q*)
Construct a *Color* from three Y'IQ float values. Y' can be between 0.0 and 1.0, while I and Q can be between -1.0 and 1.0.

classmethod from_yuv(*y*, *u*, *v*)
Construct a *Color* from three Y'UV float values. The Y value may be between 0.0 and 1.0. U may be between -0.436 and 0.436, while V may be between -0.615 and 0.615.

classmethod from_yuv_bytes(*y*, *u*, *v*)
Construct a *Color* from three Y'UV byte values between 0 and 255. The U and V values are biased by 128 to prevent negative values as is typical in video applications. The Y value is biased by 16 for the same purpose.

blue
Returns the blue component of the color as a *Blue* instance which can be used in operations with other *Color* instances.

cie_lab
Returns a 3-tuple of (L*, a*, b*) float values representing the color in the CIE Lab color space with the D65 standard illuminant.

cie_luv
Returns a 3-tuple of (L*, u*, v*) float values representing the color in the CIE Luv color space with the D65 standard illuminant.

cie_xyz
Returns a 3-tuple of (X, Y, Z) float values representing the color in the CIE 1931 color space. The conversion assumes the sRGB working space, with reference white D65.

green
Returns the green component of the color as a *Green* instance which can be used in operations with other *Color* instances.

hls
Returns a 3-tuple of (hue, lightness, saturation) float values (between 0.0 and 1.0).

hsv
Returns a 3-tuple of (hue, saturation, value) float values (between 0.0 and 1.0).

hue
Returns the hue of the color as a *Hue* instance which can be used in operations with other *Color* instances.

lightness
Returns the lightness of the color as a *Lightness* instance which can be used in operations with other *Color* instances.

red
Returns the red component of the color as a *Red* instance which can be used in operations with other *Color* instances.

rgb
Returns a 3-tuple of (red, green, blue) float values (between 0.0 and 1.0).

rgb_565
Returns an unsigned 16-bit integer number representing the color in the RGB565 encoding.

rgb_bytes
Returns a 3-tuple of (red, green, blue) byte values.

saturation
Returns the saturation of the color as a *Saturation* instance which can be used in operations with other *Color* instances.

yiq
Returns a 3-tuple of (y, i, q) float values; y values can be between 0.0 and 1.0, whilst i and q values can be between -1.0 and 1.0.

yuv
Returns a 3-tuple of (y, u, v) float values; y values can be between 0.0 and 1.0, u values are between -0.436 and 0.436, and v values are between -0.615 and 0.615.

yuv_bytes
Returns a 3-tuple of (y, u, v) byte values. Y values are biased by 16 in the result to prevent negatives. U and V values are biased by 128 for the same purpose.

2.13.2 Manipulation Classes

class picamera.**Red**
Represents the red component of a *Color* for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the *Color.red* attribute. Addition, subtraction, and multiplication are supported with *Color* instances. For example:

```
>>> Color.from_rgb(0, 0, 0) + Red(0.5)
<Color "#7f0000">
>>> Color('#f00') - Color('#900').red
<Color "#660000">
>>> (Red(0.1) * Color('red')).red
Red(0.1)
```

class picamera.**Green**
Represents the green component of a *Color* for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the *Color.green* attribute. Addition, subtraction, and multiplication are supported with *Color* instances. For example:

```
>>> Color(0, 0, 0) + Green(0.1)
<Color "#001900">
>>> Color.from_yuv(1, -0.4, -0.6) - Green(1)
<Color "#50002f">
```

```
>>> (Green(0.5) * Color('white')).rgb
(Red(1.0), Green(0.5), Blue(1.0))
```

class picamera.**Blue**

Represents the blue component of a *Color* for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the *Color.blue* attribute. Addition, subtraction, and multiplication are supported with *Color* instances. For example:

```
>>> Color(0, 0, 0) + Blue(0.2)
<Color "#000033">
>>> Color.from_hls(0.5, 0.5, 1.0) - Blue(1)
<Color "#00fe00">
>>> Blue(0.9) * Color('white')
<Color "#fffff5">
```

class picamera.**Hue**

Represents the hue of a *Color* for use in transformations. Instances of this class can be constructed directly with a float value in the range [0.0, 1.0) representing an angle around the HSL hue wheel. As this is a circular mapping, 0.0 and 1.0 effectively mean the same thing, i.e. out of range values will be normalized into the range [0.0, 1.0).

The class can also be constructed with the keyword arguments `deg` or `rad` if you wish to specify the hue value in degrees or radians instead, respectively. Instances can also be constructed by querying the *Color.hue* attribute.

Addition, subtraction, and multiplication are supported with *Color* instances. For example:

```
>>> Color(1, 0, 0).hls
(0.0, 0.5, 1.0)
>>> (Color(1, 0, 0) + Hue(deg=180)).hls
(0.5, 0.5, 1.0)
```

Note that whilst multiplication by a *Hue* doesn't make much sense, it is still supported. However, the circular nature of a hue value can lead to suprising effects. In particular, since 1.0 is equivalent to 0.0 the following may be observed:

```
>>> (Hue(1.0) * Color.from_hls(0.5, 0.5, 1.0)).hls
(0.0, 0.5, 1.0)
```

class picamera.**Saturation**

Represents the saturation of a *Color* for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the *Color.saturation* attribute. Addition, subtraction, and multiplication are supported with *Color* instances. For example:

```
>>> Color(0.9, 0.9, 0.6) + Saturation(0.1)
<Color "#ebeb92">
>>> Color('red') - Saturation(1)
<Color "#7f7f7f">
>>> Saturation(0.5) * Color('wheat')
<Color "#e4d9c3">
```

class picamera.**Lightness**

Represents the lightness of a *Color* for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the *Color.lightness* attribute. Addition, subtraction, and multiplication are supported with *Color* instances. For example:

```
>>> Color(0, 0, 0) + Lightness(0.1)
<Color "#191919">
>>> Color.from_rgb_bytes(0x80, 0x80, 0) - Lightness(0.2)
<Color "#191900">
>>> Lightness(0.9) * Color('wheat')
<Color "#f0cd8d">
```

2.14 API - Arrays

The picamera library provides a set of classes designed to aid in construction of n-dimensional numpy arrays from camera output. In order to avoid adding a hard dependency on numpy to picamera, this module (*picamera.array*) is not automatically imported by the main picamera package and must be explicitly imported, e.g.:

```
import picamera
import picamera.array
```

2.14.1 PiArrayOutput

class picamera.array.**PiArrayOutput**(*camera*, *size=None*)

Base class for capture arrays.

This class extends io.BytesIO with a numpy array which is intended to be filled when flush() is called (i.e. at the end of capture).

array

After flush() is called, this attribute contains the frame's data as a multi-dimensional numpy array. This is typically organized with the dimensions (rows, columns, plane). Hence, an RGB image with dimensions *x* and *y* would produce an array with shape (y, x, 3).

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 file size. The new file size is returned.

In prior versions of picamera, truncation also changed the position of the stream (because prior versions of these stream classes were non-seekable). This functionality is now deprecated; scripts should use seek() and *truncate()* as one would with regular BytesIO instances.

2.14.2 PiRGBArray

class picamera.array.**PiRGBArray**(*camera*, *size=None*)

Produces a 3-dimensional RGB array from an RGB capture.

This custom output class can be used to easily obtain a 3-dimensional numpy array, organized (rows, columns, colors), from an unencoded RGB capture. The array is accessed via the *array* attribute. For example:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    with picamera.array.PiRGBArray(camera) as output:
        camera.capture(output, 'rgb')
        print('Captured %dx%d image' % (
                output.array.shape[1], output.array.shape[0]))
```

You can re-use the output to produce multiple arrays by emptying it with truncate(0) between captures:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    with picamera.array.PiRGBArray(camera) as output:
        camera.resolution = (1280, 720)
        camera.capture(output, 'rgb')
        print('Captured %dx%d image' % (
                output.array.shape[1], output.array.shape[0]))
        output.truncate(0)
```

```
            camera.resolution = (640, 480)
            camera.capture(output, 'rgb')
            print('Captured %dx%d image' % (
                    output.array.shape[1], output.array.shape[0]))
```

If you are using the GPU resizer when capturing (with the *resize* parameter of the various `capture()` methods), specify the resized resolution as the optional *size* parameter when constructing the array output:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    camera.resolution = (1280, 720)
    with picamera.array.PiRGBArray(camera, size=(640, 360)) as output:
        camera.capture(output, 'rgb', resize=(640, 360))
        print('Captured %dx%d image' % (
                output.array.shape[1], output.array.shape[0]))
```

2.14.3 PiYUVArray

class `picamera.array.`**`PiYUVArray`**(*camera*, *size=None*)

Produces 3-dimensional YUV & RGB arrays from a YUV capture.

This custom output class can be used to easily obtain a 3-dimensional numpy array, organized (rows, columns, channel), from an unencoded YUV capture. The array is accessed via the *array* attribute. For example:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    with picamera.array.PiYUVArray(camera) as output:
        camera.capture(output, 'yuv')
        print('Captured %dx%d image' % (
                output.array.shape[1], output.array.shape[0]))
```

The `rgb_array` attribute can be queried for the equivalent RGB array (conversion is performed using the ITU-R BT.601 matrix):

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    with picamera.array.PiYUVArray(camera) as output:
        camera.resolution = (1280, 720)
        camera.capture(output, 'yuv')
        print(output.array.shape)
        print(output.rgb_array.shape)
```

If you are using the GPU resizer when capturing (with the *resize* parameter of the various *capture()* methods), specify the resized resolution as the optional *size* parameter when constructing the array output:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    camera.resolution = (1280, 720)
    with picamera.array.PiYUVArray(camera, size=(640, 360)) as output:
        camera.capture(output, 'yuv', resize=(640, 360))
        print('Captured %dx%d image' % (
                output.array.shape[1], output.array.shape[0]))
```

2.14.4 PiBayerArray

class picamera.array.**PiBayerArray**(*camera*)
Produces a 3-dimensional RGB array from raw Bayer data.

This custom output class is intended to be used with the *capture()* method, with the *bayer* parameter set to True, to include raw Bayer data in the JPEG output. The class strips out the raw data, constructing a 3-dimensional numpy array organized as (rows, columns, colors). The resulting data is accessed via the *array* attribute:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    with picamera.array.PiBayerArray(camera) as output:
        camera.capture(output, 'jpeg', bayer=True)
        print(output.array.shape)
```

Note that Bayer data is *always* full resolution, so the resulting array always has the shape (1944, 2592, 3) with the V1 module, or (2464, 3280, 3) with the V2 module; this also implies that the optional *size* parameter (for specifying a resizer resolution) is not available with this array class. As the sensor records 10-bit values, the array uses the unsigned 16-bit integer data type.

By default, de-mosaicing is **not** performed; if the resulting array is viewed it will therefore appear dark and too green (due to the green bias in the Bayer pattern). A trivial weighted-average demosaicing algorithm is provided in the demosaic() method:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    with picamera.array.PiBayerArray(camera) as output:
        camera.capture(output, 'jpeg', bayer=True)
        print(output.demosaic().shape)
```

Viewing the result of the de-mosaiced data will look more normal but still considerably worse quality than the regular camera output (as none of the other usual post-processing steps like auto-exposure, white-balance, vignette compensation, and smoothing have been performed).

2.14.5 PiMotionArray

class picamera.array.**PiMotionArray**(*camera*, *size=None*)
Produces a 3-dimensional array of motion vectors from the H.264 encoder.

This custom output class is intended to be used with the *motion_output* parameter of the *start_recording()* method. Once recording has finished, the class generates a 3-dimensional numpy array organized as (frames, rows, columns) where rows and columns are the number of rows and columns of macro-blocks (16x16 pixel blocks) in the original frames. There is always one extra column of macro-blocks present in motion vector data.

The data-type of the *array* is an (x, y, sad) structure where x and y are signed 1-byte values, and sad is an unsigned 2-byte value representing the sum of absolute differences of the block. For example:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    with picamera.array.PiMotionArray(camera) as output:
        camera.resolution = (640, 480)
        camera.start_recording(
              '/dev/null', format='h264', motion_output=output)
        camera.wait_recording(30)
```

```
            camera.stop_recording()
            print('Captured %d frames' % output.array.shape[0])
            print('Frames are %dx%d blocks big' % (
                output.array.shape[2], output.array.shape[1]))
```

If you are using the GPU resizer with your recording, use the optional *size* parameter to specify the resizer's output resolution when constructing the array:

```
import picamera
import picamera.array

with picamera.PiCamera() as camera:
    camera.resolution = (640, 480)
    with picamera.array.PiMotionArray(camera, size=(320, 240)) as output:
        camera.start_recording(
            '/dev/null', format='h264', motion_output=output,
            resize=(320, 240))
        camera.wait_recording(30)
        camera.stop_recording()
        print('Captured %d frames' % output.array.shape[0])
        print('Frames are %dx%d blocks big' % (
            output.array.shape[2], output.array.shape[1]))
```

Note: This class is not suitable for real-time analysis of motion vector data. See the *PiMotionAnalysis* class instead.

2.14.6 PiAnalysisOutput

class picamera.array.**PiAnalysisOutput**(*camera*, *size=None*)

Base class for analysis outputs.

This class extends io.IOBase with a stub *analyze()* method which will be called for each frame output. In this base implementation the method simply raises NotImplementedError.

analyse(*array*)

Deprecated alias of *analyze()*.

analyze(*array*)

Stub method for users to override.

2.14.7 PiRGBAnalysis

class picamera.array.**PiRGBAnalysis**(*camera*, *size=None*)

Provides a basis for per-frame RGB analysis classes.

This custom output class is intended to be used with the *start_recording()* method when it is called with *format* set to 'rgb' or 'bgr'. While recording is in progress, the write() method converts incoming frame data into a numpy array and calls the stub *analyze()* method with the resulting array (this deliberately raises NotImplementedError in this class; you must override it in your descendent class).

Note: If your overridden *analyze()* method runs slower than the required framerate (e.g. 33.333ms when framerate is 30fps) then the camera's effective framerate will be reduced. Furthermore, this doesn't take into account the overhead of picamera itself so in practice your method needs to be a bit faster still.

The array passed to *analyze()* is organized as (rows, columns, channel) where the channels 0, 1, and 2 are R, G, and B respectively (or B, G, R if *format* is 'bgr').

2.14.8 PiYUVAnalysis

class picamera.array.**PiYUVAnalysis**(*camera*, *size=None*)
Provides a basis for per-frame YUV analysis classes.

This custom output class is intended to be used with the *start_recording()* method when it is called with *format* set to 'yuv'. While recording is in progress, the write() method converts incoming frame data into a numpy array and calls the stub *analyze()* method with the resulting array (this deliberately raises NotImplementedError in this class; you must override it in your descendent class).

Note: If your overridden *analyze()* method runs slower than the required framerate (e.g. 33.333ms when framerate is 30fps) then the camera's effective framerate will be reduced. Furthermore, this doesn't take into account the overhead of picamera itself so in practice your method needs to be a bit faster still.

The array passed to *analyze()* is organized as (rows, columns, channel) where the channel 0 is Y (luminance), while 1 and 2 are U and V (chrominance) respectively. The chrominance values normally have quarter resolution of the luminance values but this class makes all channels equal resolution for ease of use.

2.14.9 PiMotionAnalysis

class picamera.array.**PiMotionAnalysis**(*camera*, *size=None*)
Provides a basis for real-time motion analysis classes.

This custom output class is intended to be used with the *motion_output* parameter of the *start_recording()* method. While recording is in progress, the write method converts incoming motion data into numpy arrays and calls the stub *analyze()* method with the resulting array (which deliberately raises NotImplementedError in this class).

Note: If your overridden *analyze()* method runs slower than the required framerate (e.g. 33.333ms when framerate is 30fps) then the camera's effective framerate will be reduced. Furthermore, this doesn't take into account the overhead of picamera itself so in practice your method needs to be a bit faster still.

The array passed to *analyze()* is organized as (rows, columns) where rows and columns are the number of rows and columns of macro-blocks (16x16 pixel blocks) in the original frames. There is always one extra column of macro-blocks present in motion vector data.

The data-type of the array is an (x, y, sad) structure where x and y are signed 1-byte values, and sad is an unsigned 2-byte value representing the sum of absolute differences of the block.

An example of a crude motion detector is given below:

```
import numpy as np
import picamera
import picamera.array

class DetectMotion(picamera.array.PiMotionAnalysis):
    def analyze(self, a):
        a = np.sqrt(
            np.square(a['x'].astype(np.float)) +
            np.square(a['y'].astype(np.float))
            ).clip(0, 255).astype(np.uint8)
        # If there're more than 10 vectors with a magnitude greater
        # than 60, then say we've detected motion
        if (a > 60).sum() > 10:
            print('Motion detected!')

with picamera.PiCamera() as camera:
    with DetectMotion(camera) as output:
        camera.resolution = (640, 480)
```

```
            camera.start_recording(
                  '/dev/null', format='h264', motion_output=output)
            camera.wait_recording(30)
            camera.stop_recording()
```

You can use the optional *size* parameter to specify the output resolution of the GPU resizer, if you are using the *resize* parameter of *start_recording()*.

2.15 API - mmalobj

This module provides an object-oriented interface to `libmmal` which is the library underlying picamera, `raspistill`, and `raspivid`. It is provided to ease the usage of `libmmal` to Python coders unfamiliar with C and also works around some of the idiosyncracies in `libmmal`.

2.15.1 Components

class `picamera.mmalobj.`**MMALComponent**

Represents a generic MMAL component. Class attributes are read to determine the component type, and the OPAQUE sub-formats of each connectable port.

close()

Close the component and release all its resources. After this is called, most methods will raise exceptions if called.

control

The *MMALControlPort* control port of the component which can be used to configure most aspects of the component's behaviour.

enabled

Retrieves or sets whether the component is currently enabled. When a component is disabled it does not produce or consume data. Components may be implicitly enabled by downstream components.

inputs

A sequence of *MMALPort* objects representing the inputs of the component.

outputs

A sequence of *MMALPort* objects representing the outputs of the component.

class `picamera.mmalobj.`**MMALCamera**

Bases: *picamera.mmalobj.MMALComponent*

Represents the MMAL camera component.

The intended use of the output ports (which in turn determines the behaviour of those ports) is as follows:

- Port 0 is intended for preview renderers
- Port 1 is intended for video recording
- Port 2 is intended for still image capture

annotate_rev

The annotation capabilities of the firmware have evolved over time and several structures are available for querying and setting video annotations. By default the *MMALCamera* class will pick the latest annotation structure supported by the current firmware but you can select older revisions with *annotate_rev* for other purposes (e.g. testing).

class `picamera.mmalobj.`**MMALCameraInfo**

Bases: *picamera.mmalobj.MMALComponent*

Represents the MMAL camera-info component.

info_rev
The camera information capabilities of the firmware have evolved over time and several structures are available for querying camera information. When initialized, *MMALCameraInfo* will attempt to discover which structure is in use by the extant firmware. This property can be used to discover the structure version and to modify the version in use for other purposes (e.g. testing).

class picamera.mmalobj.**MMALDownstreamComponent**
Bases: *picamera.mmalobj.MMALComponent*

Represents an MMAL component that acts as a filter of some sort, with a single input that connects to an upstream source port. This is an asbtract base class.

class picamera.mmalobj.**MMALSplitter**
Bases: *picamera.mmalobj.MMALDownstreamComponent*

Represents the MMAL splitter component.

class picamera.mmalobj.**MMALResizer**
Bases: *picamera.mmalobj.MMALDownstreamComponent*

Represents the MMAL resizer component.

class picamera.mmalobj.**MMALEncoder**
Bases: *picamera.mmalobj.MMALDownstreamComponent*

Represents a generic MMAL encoder. This is an abstract base class.

class picamera.mmalobj.**MMALVideoEncoder**
Bases: *picamera.mmalobj.MMALEncoder*

Represents the MMAL video encoder component.

class picamera.mmalobj.**MMALImageEncoder**
Bases: *picamera.mmalobj.MMALEncoder*

Represents the MMAL image encoder component.

class picamera.mmalobj.**MMALRenderer**
Bases: *picamera.mmalobj.MMALDownstreamComponent*

Represents the MMAL preview renderer component.

class picamera.mmalobj.**MMALNullSink**
Bases: *picamera.mmalobj.MMALDownstreamComponent*

Represents the MMAL null-sink component.

2.15.2 Ports

class picamera.mmalobj.**MMALControlPort**(*port*)
Represents an MMAL port with properties to configure the port's parameters.

disable()
Disable the port.

enable(*callback=None*)
Enable the port with the specified callback function (this must be `None` for connected ports, and a callable for disconnected ports).

The callback function must accept two parameters which will be this *MMALControlPort* (or descendent) and an *MMALBuffer* instance. Any return value will be ignored.

flush()
Flush the port.

send_buffer(*buf*)
Send *MMALBuffer* *buf* to the port.

enabled
Returns a `bool` indicating whether the port is currently enabled. Unlike other classes, this is a read-only property. Use *`enable()`* and *`disable()`* to modify the value.

index
Returns an integer indicating the port's position within its owning list (inputs, outputs, etc.)

params
The configurable parameters for the port. This is presented as a mutable mapping of parameter numbers to values, implemented by the *`MMALPortParams`* class.

class `picamera.mmalobj.`**`MMALPort`**(*port*, *opaque_subformat='OPQV'*)
Bases: *`picamera.mmalobj.MMALControlPort`*

Represents an MMAL port with properties to configure and update the port's format. This is the base class of *`MMALVideoPort`*, *`MMALAudioPort`*, and *`MMALSubPicturePort`*.

commit()
Commits the port's configuration and automatically updates the number and size of associated buffers. This is typically called after adjusting the port's format and/or associated settings (like width and height for video ports).

copy_from(*source*)
Copies the port's *`format`* from the *source* *`MMALControlPort`*.

disable()
Disable the port.

enable(*callback=None*)
Enable the port with the specified callback function (this must be `None` for connected ports, and a callable for disconnected ports).

The callback function must accept two parameters which will be this *`MMALControlPort`* (or descendent) and an *`MMALBuffer`* instance. The callback should return `True` when processing is complete and no further calls are expected (e.g. at frame-end for an image encoder), and `False` otherwise.

bitrate
Retrieves or sets the bitrate limit for the port's format.

format
Retrieves or sets the encoding format of the port. Setting this attribute implicitly sets the encoding variant to a sensible value (I420 in the case of OPAQUE).

After setting this attribute, call *`commit()`* to make the changes effective.

opaque_subformat
Retrieves or sets the opaque sub-format that the port speaks. While most formats (I420, RGBA, etc.) mean one thing, the opaque format is special; different ports produce different sorts of data when configured for OPQV format. This property stores a string which uniquely identifies what the associated port means for OPQV format.

If the port does not support opaque format at all, set this property to `None`.

`MMALConnection` uses this information when negotiating formats for a connection between two ports.

pool
Returns the *`MMALPool`* associated with the buffer, if any.

supported_formats
Retrieves a sequence of supported encodings on this port.

> **Warning:** This property does not work on the camera's still port (`MMALCamera.outputs[2]`) due to an underlying firmware bug.

class picamera.mmalobj.**MMALVideoPort**(*port*, *opaque_subformat='OPQV'*)
Bases: *picamera.mmalobj.MMALPort*

Represents an MMAL port used to pass video data.

framerate
Retrieves or sets the framerate of the port's video frames in fps.

After setting this attribute, call *commit()* to make the changes effective.

framesize
Retrieves or sets the size of the port's video frames as a (width, height) tuple. This attribute implicitly handles scaling the given size up to the block size of the camera (32x16).

After setting this attribute, call *commit()* to make the changes effective.

class picamera.mmalobj.**MMALSubPicturePort**(*port*, *opaque_subformat='OPQV'*)
Bases: *picamera.mmalobj.MMALPort*

Represents an MMAL port used to pass sub-picture (caption) data.

class picamera.mmalobj.**MMALAudioPort**(*port*, *opaque_subformat='OPQV'*)
Bases: *picamera.mmalobj.MMALPort*

Represents an MMAL port used to pass audio data.

class picamera.mmalobj.**MMALPortParams**(*port*)
Represents the parameters of an MMAL port. This class implements the *MMALControlPort.params* attribute.

Internally, the class understands how to convert certain structures to more common Python data-types. For example, parameters that expect an MMAL_RATIONAL_T type will return and accept Python's Fraction class (or any other numeric types), while parameters that expect an MMAL_BOOL_T type will treat anything as a truthy value. Parameters that expect the MMAL_PARAMETER_STRING_T structure will be treated as plain strings, and likewise MMAL_PARAMETER_INT32_T and similar structures will be treated as plain ints.

Parameters that expect more complex structures will return and expect those structures verbatim.

2.15.3 Connections

class picamera.mmalobj.**MMALConnection**(*source*, *target*)
Represents an MMAL internal connection between two components. The constructor accepts arguments providing the *source* *MMALPort* and *target* *MMALPort*.

The connection will automatically negotiate the most efficient format supported by both ports (implicitly handling the incompatibility of some OPAQUE sub-formats). See *Under the Hood* for more information.

2.15.4 Buffers

class picamera.mmalobj.**MMALBuffer**(*buf*)
Represents an MMAL buffer header. This is usually constructed from the buffer header pointer and is largely supplied simply to make working with the buffer's data a bit simpler; accessing the *data* attribute implicitly locks the buffer's memory and returns the data as a bytes string.

copy(*data=None*)
Return a copy of this buffer header, optionally replacing the *data* attribute with *data* which must be an object supporting the buffer protocol.

update(*data*)
Overwrites the *data* in the buffer. The *data* parameter is an object supporting the buffer protocol which contains up to *size* bytes.

> **Warning:** Some buffer objects *cannot* be modified without consequence (for example, buffers returned by an encoder's output port).

command
Returns the command set in the buffer's meta-data. This is usually 0 for buffers returned by an encoder; typically this is only used by buffers sent to the callback of a control port.

data
Returns the data held in the buffer as a `bytes` string.

dts
Returns the decoding timestamp (DTS) of the buffer.

flags
Returns the flags set in the buffer's meta-data.

length
Returns the length of data held in the buffer. This is equal to calling `len()` on *data* but faster (as retrieving the buffer's data requires memory locks in certain cases).

pts
Returns the presentation timestamp (PTS) of the buffer.

size
Returns the length of the buffer's data area in bytes. This will be greater than or equal to *length* and is fixed in value.

class picamera.mmalobj.**MMALPool**(*pool*)
Construct an MMAL pool containing *num* buffer headers of *size* bytes.

get_buffer()
Get the next buffer from the pool.

send_all_buffers(*port*)
Send all buffers from the pool to *port*.

send_buffer(*port*)
Get a buffer from the pool and send it to *port*.

class picamera.mmalobj.**MMALPortPool**(*port*)
Bases: *picamera.mmalobj.MMALPool*

Construct an MMAL pool for the number and size of buffers required by the *MMALPort* *port*.

send_all_buffers(*port*)
Send all buffers from the pool to the port the pool is associated with.

send_buffer()
Get a buffer from the pool and send it to the port the pool is associated with.

2.15.5 Debugging

The following functions are useful for quickly dumping the state of a given MMAL pipeline:

picamera.mmalobj.**debug_pipeline**(*port*)
Given an *MMALVideoPort* *port*, this traces all objects in the pipeline feeding it (including components and connections) and yields each object in turn. Hence the generator typically yields something like:

- *MMALVideoPort* (the specified output port)
- *MMALEncoder* (the encoder which owns the output port)
- *MMALVideoPort* (the encoder's input port)
- *MMALConnection* (the connection between the splitter and encoder)
- *MMALVideoPort* (the splitter's output port)

- *MMALSplitter* (the splitter on the camera's video port)
- *MMALVideoPort* (the splitter's input port)
- *MMALConnection* (the connection between the splitter and camera)
- *MMALVideoPort* (the camera's video port)
- *MMALCamera* (the camera component)

picamera.mmalobj.**print_pipeline**(*port*)
Prints a human readable representation of the pipeline feeding the specified *MMALVideoPort* *port*.

Note: It is also worth noting that most classes, in particular *MMALVideoPort* and *MMALBuffer* have useful `repr()` outputs which can be extremely useful with simple `print()` calls for debugging.

2.16 Change log

2.16.1 Release 1.12 (2016-07-03)

1.12 is almost entirely a bug fix release:

- Fixed issue with unencoded captures in Python 3 (#297)
- Fixed several Python 3 bytes/unicode issues that were related to #297 (I'd erroneously run the picamera test suite twice against Python 2 instead of 2 and 3 when releasing 1.11, which is how these snuck in)
- Fixed multi-dimensional arrays for overlays under Python 3
- Finished alternate CIE constructors for the *Color* class

2.16.2 Release 1.11 (2016-06-19)

1.11 on the surface consists mostly of enhancements, but underneath includes a major re-write of picamera's core:

- Direct capture to buffer-protocol objects, such as numpy arrays (#241)
- Add *request_key_frame()* method to permit manual request of an I-frame during H264 recording; this is now used implicitly by *split_recording()* (#257)
- Added *timestamp* attribute to query camera's clock (#212)
- Added *framerate_delta* to permit small adjustments to the camera's framerate to be performed "live" (#279)
- Added *clear()* and *copy_to()* methods to *PiCameraCircularIO* (#216)
- Prevent setting attributes on the main *PiCamera* class to ease debugging in educational settings (#240)
- Due to the core re-writes in this version, you may require cutting edge firmware (`sudo rpi-update`) if you are performing unencoded captures, unencoded video recording, motion estimation vector sampling, or manual sensor mode setting.
- Added property to control preview's *resolution* separately from the camera's *resolution* (required for maximum resolution previews on the V2 module - #296).

There are also several bug fixes:

- Fixed basic stereoscopic operation on compute module (#218)
- Fixed accessing framerate as a tuple (#228)
- Fixed hang when invalid file format is specified (#236)
- Fixed multiple bayer captures with *capture_sequence()* and *capture_continuous()* (#264)

- Fixed usage of "falsy" custom outputs with `motion_output` (#281)

Many thanks to the community, and especially thanks to 6by9 (one of the firmware developers) who's fielded seemingly endless questions and requests from me in the last couple of months!

2.16.3 Release 1.10 (2015-03-31)

1.10 consists mostly of minor enhancements:

- The major enhancement is the addition of support for the camera's flash driver. This is relatively complex to configure, but a full recipe has been included in the documentation (#184)
- A new *intra_refresh* attribute is added to the *start_recording()* method permitting control of the intra-frame refresh method (#193)
- The GPIO pins controlling the camera's LED are now configurable. This is mainly for any compute module users, but also for anyone who wishes to use the device tree blob to reconfigure the pins used (#198)
- The new annotate V3 struct is now supported, providing custom background colors for annotations, and configurable text size. As part of this work a new *Color* class was introduced for representation and manipulation of colors (#203)
- Reverse enumeration of frames in *PiCameraCircularIO* is now supported efficiently (without having to convert frames to a list first) (#204)
- Finally, the API documentation has been re-worked as it was getting too large to comfortably load on all platforms (no ticket)

2.16.4 Release 1.9 (2015-01-01)

1.9 consists mostly of bug fixes with a couple of minor new features:

- The camera's sensor mode can now be forced to a particular setting upon camera initialization with the new `sensor_mode` parameter to *PiCamera* (#165)
- The camera's initial framerate and resolution can also be specified as keyword arguments to the *PiCamera* initializer. This is primarily intended to reduce initialization time (#180)
- Added the *still_stats* attribute which controls whether an extra statistics pass is made when capturing images from the still port (#166)
- Fixed the *led* attribute so it should now work on the Raspberry Pi model B+ (#170)
- Fixed a nasty memory leak in overlay renderers which caused the camera to run out of memory when overlays were repeatedly created and destroyed (#174) * Fixed a long standing issue with MJPEG recording which caused camera lockups when resolutions greater than VGA were used (#47 and #179)
- Fixed a bug with incorrect frame metadata in *PiCameraCircularIO*. Unfortunately this required breaking backwards compatibility to some extent. If you use this class and rely on the frame metadata, please familiarize yourself with the new *complete* attribute (#177)
- Fixed a bug which caused *PiCameraCircularIO* to ignore the splitter port it was recording against (#176)
- Several documentation issues got fixed too (#167, #168, #171, #172, #182)

Many thanks to the community for providing several of these fixes as pull requests, and thanks for all the great bug reports. Happy new year everyone!

2.16.5 Release 1.8 (2014-09-05)

1.8 consists of several new features and the usual bug fixes:

- A new chapter on detecting and correcting deprecated functionality was added to the docs (#149)

- Stereoscopic cameras are now tentatively supported on the Pi compute module. Please note I have no hardware for testing this, so the implementation is possibly (probably!) wrong; bug reports welcome! (#153)
- Text annotation functionality has been extended; up to 255 characters are now possible, and the new *annotate_frame_num* attribute adds rendering of the current frame number. In addition, the new *annotate_background* flag permits a dark background to be rendered behind all annotations for contrast (#160)
- Arbitrary image overlays can now be drawn on the preview using the new *add_overlay()* method. A new recipe has been included demonstrating overlays from PIL images and numpy arrays. As part of this work the preview system was substantially changed; all older scripts should continue to work but please be aware that most preview attributes are now deprecated; the new *preview* attribute replaces them (#144)
- Image effect parameters can now be controlled via the new *image_effect_params* attribute (#143)
- A bug in the handling of framerates meant that long exposures (>1s) weren't operating correctly. This *should* be fixed, but I'd be grateful if users could test this and let me know for certain (Exif metadata reports the configured exposure speed so it can't be used to determine if things are actually working) (#135)
- A bug in 1.7 broke compatibility with older firmwares (resulting in an error message mentioning "mmal_queue_timedwait"). The library should now on older firmwares (#154)
- Finally, the confusingly named *crop* attribute was changed to a deprecated alias for the new *zoom* attribute (#146)

2.16.6 Release 1.7 (2014-08-08)

1.7 consists once more of new features, and more bug fixes:

- Text overlay on preview, image, and video output is now possible (#16)
- Support for more than one camera on the compute module has been added, but hasn't been tested yet (#84)
- The *exposure_mode* `'off'` has been added to allow locking down the exposure time, along with some new recipes demonstrating this capability (#116)
- The valid values for various attributes including *awb_mode*, *meter_mode*, and *exposure_mode* are now automatically included in the documentation (#130)
- Support for unencoded formats (YUV, RGB, etc.) has been added to the *start_recording()* method (#132)
- A couple of analysis classes have been added to *picamera.array* to support the new unencoded recording formats (#139)
- Several issues in the `PiBayerArray` class were fixed; this should now work correctly with Python 3, and the `demosaic()` method should operate correctly (#133, #134)
- A major issue with multi-resolution recordings which caused all recordings to stop prematurely was fixed (#136)
- Finally, an issue with the example in the documentation for custom encoders was fixed (#128)

Once again, many thanks to the community for another round of excellent bug reports - and many thanks to 6by9 and jamesh for their excellent work on the firmware and official utilities!

2.16.7 Release 1.6 (2014-07-21)

1.6 is half bug fixes, half new features:

- The *awb_gains* attribute is no longer write-only; you can now read it to determine the red/blue balance that the camera is using (#98)

- The new read-only *exposure_speed* attribute will tell you the shutter speed the camera's auto-exposure has determined, or the shutter speed you've forced with a non-zero value of *shutter_speed* (#98)
- The new read-only *analog_gain* and *digital_gain* attributes can be used to determine the amount of gain the camera is applying at a couple of crucial points of the image processing pipeline (#98)
- The new *drc_strength* attribute can be used to query and set the amount of dynamic range compression the camera will apply to its output (#110)
- The *intra_period* parameter for *start_recording()* can now be set to *0* (which means "produce one initial I-frame, then just P-frames") (#117)
- The *burst* parameter was added to the various *capture()* methods; users are strongly advised to read the cautions in the docs before relying on this parameter (#115)
- One of the advanced recipes in the manual ("splitting to/from a circular stream") failed under 1.5 due to a lack of splitter-port support in the circular I/O stream class. This has now been rectified by adding a *splitter_port* parameter to the constructor of *PiCameraCircularIO* (#109)
- Similarly, the *array extensions* introduced in 1.5 failed to work when resizers were present in the pipeline. This has been fixed by adding a *size* parameter to the constructor of all the custom output classes defined in that module (#121)
- A bug that caused picamera to fail when the display was disabled has been squashed (#120)

As always, many thanks to the community for another great set of bug reports!

2.16.8 Release 1.5 (2014-06-11)

1.5 fixed several bugs and introduced a couple of major new pieces of functionality:

- The new *picamera.array* module provides a series of custom output classes which can be used to easily obtain numpy arrays from a variety of sources (#107)
- The *motion_output* parameter was added to *start_recording()* to enable output of motion vector data generated by the H.264 encoder. A couple of new recipes were added to the documentation to demonstrate this (#94)
- The ability to construct custom encoders was added, including some examples in the documentation. Many thanks to user Oleksandr Sviridenko (d2rk) for helping with the design of this feature! (#97)
- An example recipe was added to the documentation covering loading and conversion of raw Bayer data (#95)
- Speed of unencoded RGB and BGR captures was substantially improved in both Python 2 and 3 with a little optimization work. The warning about using alpha-inclusive modes like RGBA has been removed as a result (#103)
- An issue with out-of-order calls to *stop_recording()* when multiple recordings were active was resolved (#105)
- Finally, picamera caught up with raspistill and raspivid by offering a friendly error message when used with a disabled camera - thanks to Andrew Scheller (lurch) for the suggestion! (#89)

2.16.9 Release 1.4 (2014-05-06)

1.4 mostly involved bug fixes with a couple of new bits of functionality:

- The *sei* parameter was added to *start_recording()* to permit inclusion of "Supplemental Enhancement Information" in the output stream (#77)
- The *awb_gains* attribute was added to permit manual control of the auto-white-balance red/blue gains (#74)
- A bug which cause *split_recording()* to fail when low framerates were configured was fixed (#87)

- A bug which caused picamera to fail when used in UNIX-style daemons, unless the module was imported *after* the double-fork to background was fixed (#85)
- A bug which caused the `frame` attribute to fail when queried in Python 3 was fixed (#80)
- A bug which caused raw captures with "odd" resolutions (like 100x100) to fail was fixed (#83)

Known issues:

- Added a workaround for full-resolution YUV captures failing. This isn't a complete fix, and attempting to capture a JPEG before attempting to capture full-resolution YUV data will still fail, unless the GPU memory split is set to something huge like 256Mb (#73)

Many thanks to the community for yet more excellent quality bug reports!

2.16.10 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)

2.16.11 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)

2.16.12 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)

2.16.13 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)

2.16.14 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)

2.16.15 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

2.16.16 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

2.16.17 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!

2.16.18 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()`.

2.16.19 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.

2.16.20 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.

2.17 License

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

p

Symbols

A

B

C

D

O

P

R

S

T

U

V

W

Y

Z