Picamera 1.13 Documentation

Release 1.13

Dave Jones

Feb 25, 2017

Contents

CHAPTER 1

Installation

Raspbian installation

If you are using the Raspbian[1] distro, you probably have picamera installed by default. You can find out simply by starting Python and trying to import picamera:

```
$ python -c "import picamera"
$ python3 -c "import picamera"
```

If you get no error, you've already got picamera installed! Just continue to *Getting Started* (page 5). If you don't have picamera installed you'll see something like the following:

```
$ python -c "import picamera"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named picamera
$ python3 -c "import picamera"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named 'picamera'
```

To install picamera on Raspbian, it is best to use 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 run:

```
$ 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
```

[1] https://www.raspberrypi.org/downloads/raspbian/

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` (page 151) module then specify the "array" option which will pull in numpy as a dependency:

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

> **Warning:** Be warned that older versions of pip will attempt to build numpy from source. This will take a *very* long time on a Pi (several hours on slower models). Modern versions of pip will download and install a pre-built numpy "wheel" instead which is much faster.

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
```

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` (page 106) 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} z
> >   raspberrypi-bootloader
> $ sudo rm /boot/.firmware_revision
> ```

You will need to reboot after doing so.

Note: Please note that the PiTFT[2] 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] https://www.adafruit.com/product/1601

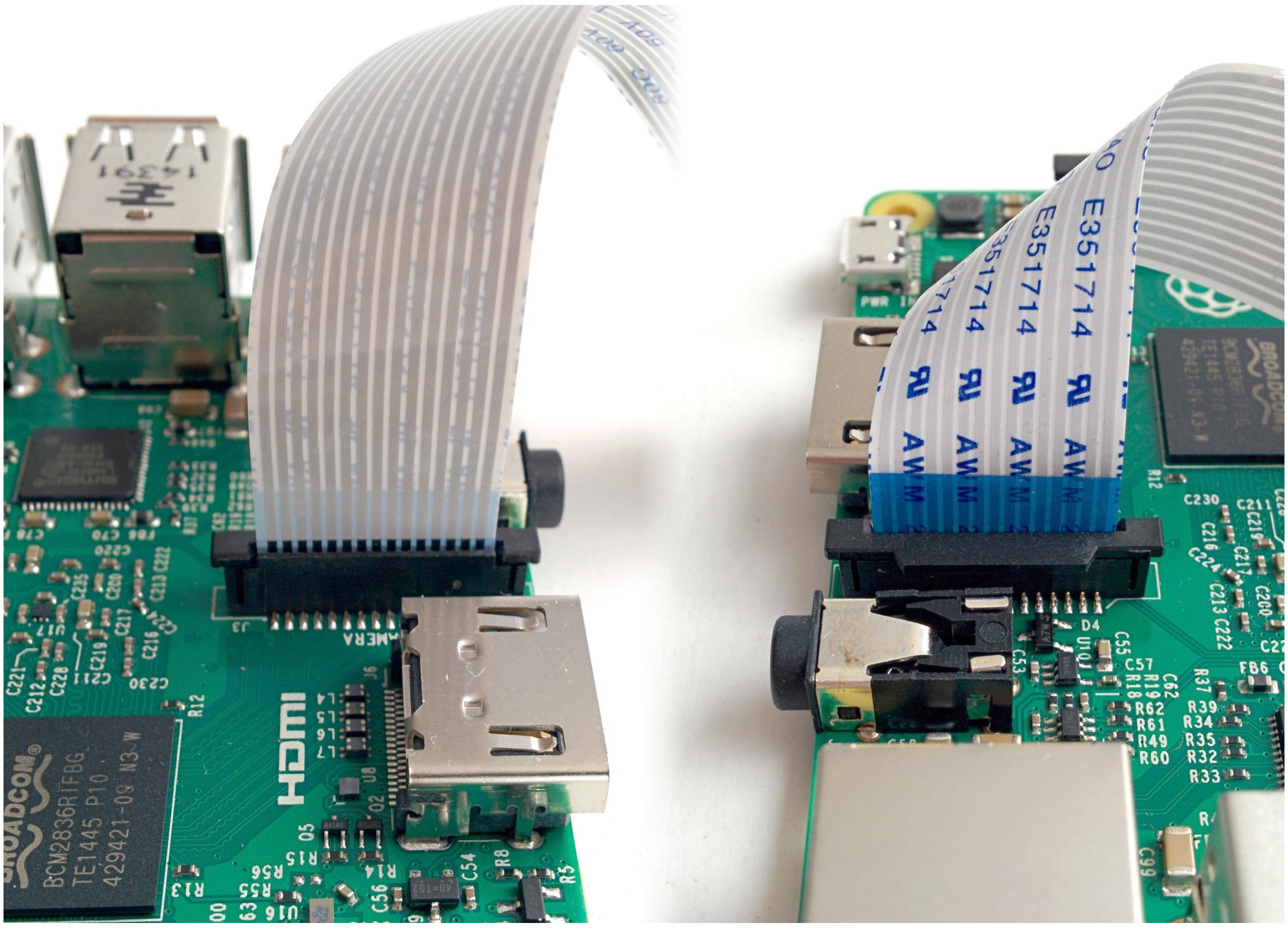
CHAPTER 2

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:

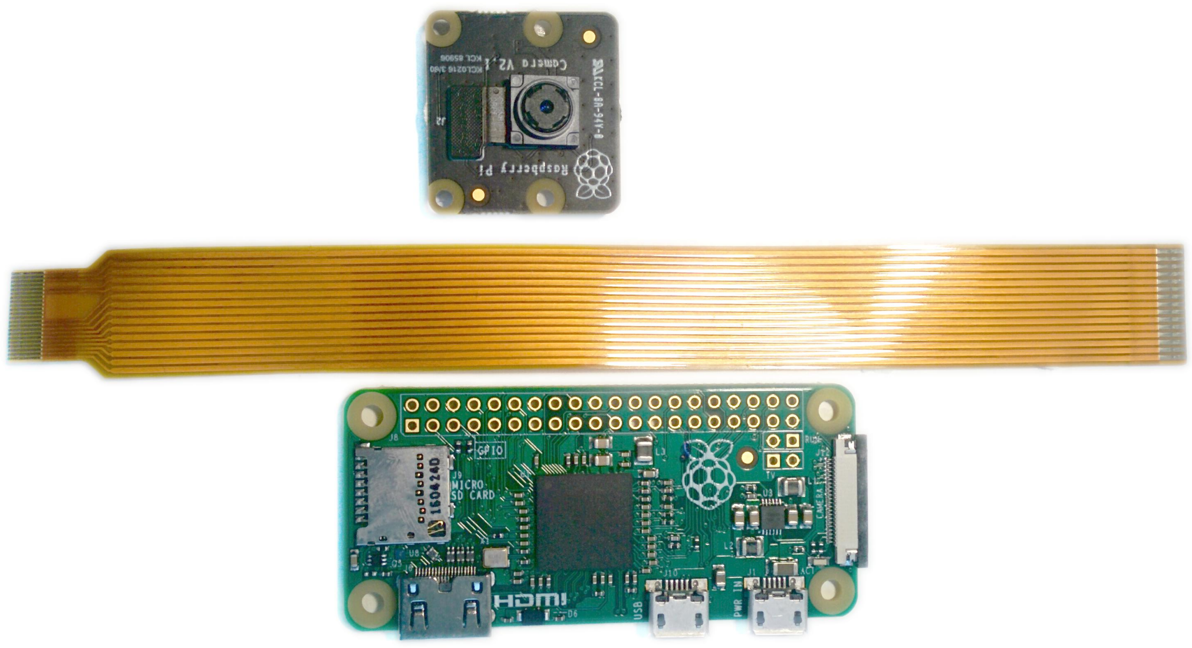


Make sure the camera module isn't sat on anything conductive (e.g. the Pi's USB ports or its GPIO pins).

Pi Zero

The 1.2 model of the Raspberry Pi Zero[3] includes a small form-factor CSI port which requires a camera adapter cable[4].

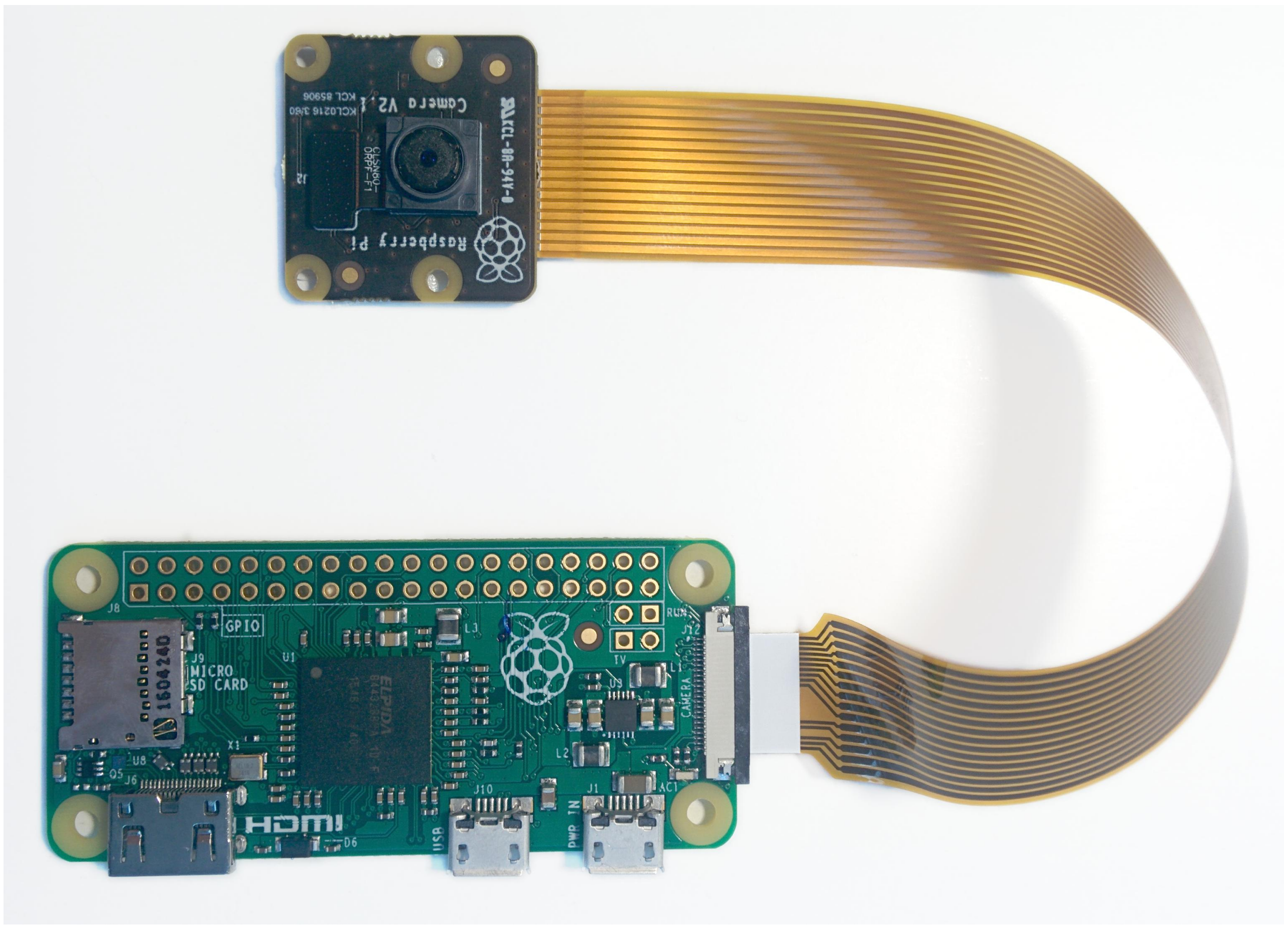
[3] https://www.raspberrypi.org/products/pi-zero/
[4] https://shop.pimoroni.com/products/camera-cable-raspberry-pi-zero-edition



To attach a camera module to a Pi Zero:

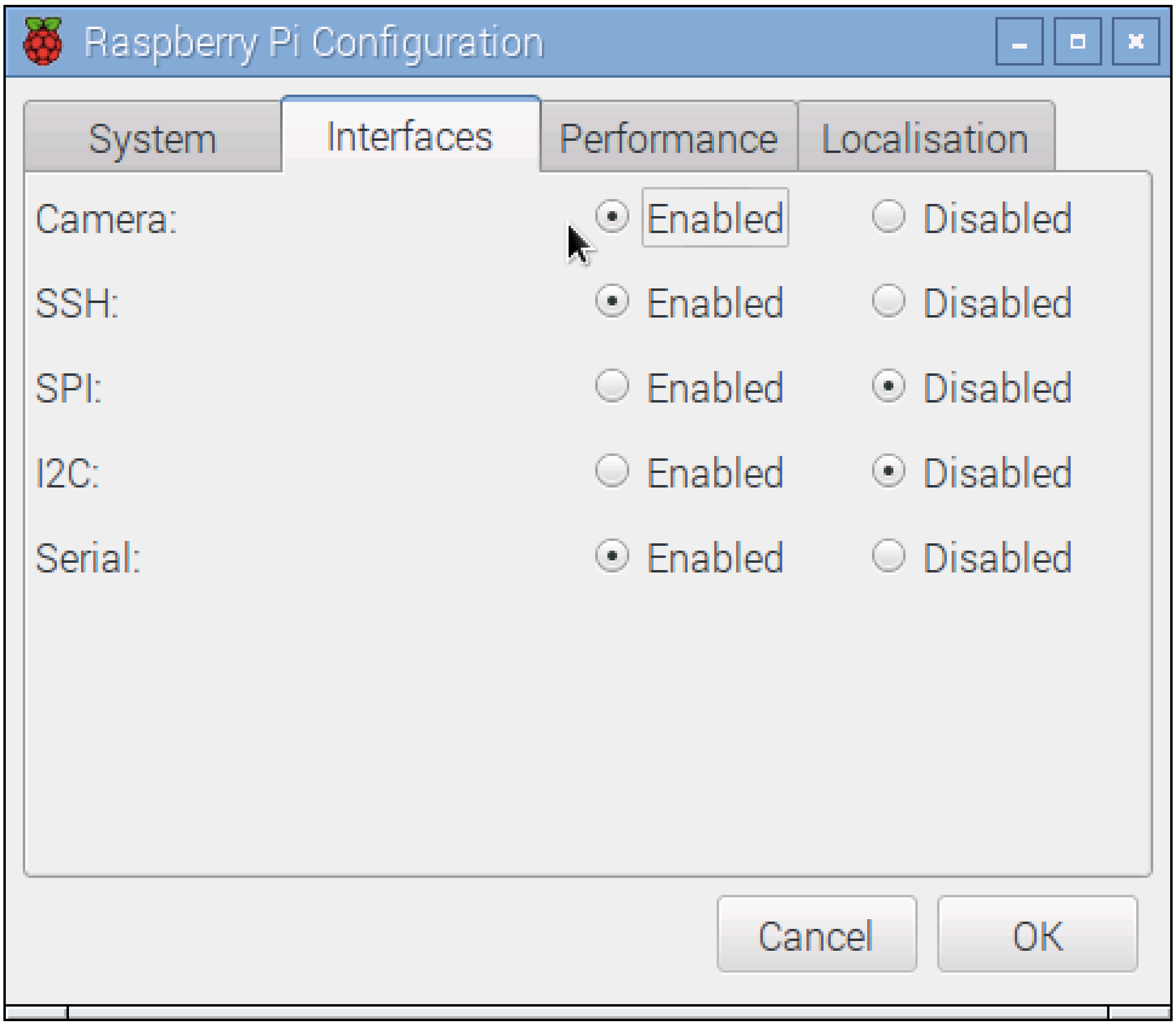
1. Remove the existing camera module's cable by gently lifting the collar on the camera module and pulling the cable out.
2. Next, insert the wider end of the adapter cable with the conductors facing in the same direction as the camera's lens.
3. Finally, attach the adapter to the Pi Zero by gently lifting the collar at the edge of the board (be careful with this as they are more delicate than the collars on the regular CSI ports) and inserting the smaller end of the adapter with the conductors facing the back of the Pi Zero.

Your setup should look something like this:



Testing

Now, apply power to your Pi. Once booted, start the Raspberry Pi Configuration utility and enable the camera module:


Raspberry Pi Configuration

Camera: [✓ Enabled] Disabled

SSH: [✓ Enabled] Disabled

SPI: Enabled [✓ Disabled]

I2C: Enabled [✓ Disabled]

Serial: [✓ Enabled] Disabled

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* (page 11).

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).

CHAPTER 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.

Warning: When trying out these scripts do *not* name your file `picamera.py`. Naming scripts after existing Python modules will cause errors when you try and import those modules (because Python checks the current directory before checking other paths).

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()` (page 97) 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()` (page 97) method returns, the data should be accessible to other processes.

Capturing to a stream

Capturing an image to a file-like object (a `socket()`[5], a `io.BytesIO`[6] stream, an existing open file object, etc.) is as simple as specifying that object as the output of whatever `capture()` (page 97) method you're using:

[5] https://docs.python.org/3.4/library/socket.html#socket.socket
[6] https://docs.python.org/3.4/library/io.html#io.BytesIO

```
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`[7] 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`[8]) you can also construct your own *custom outputs* (page 30).

Capturing to a PIL Image

This is a variation on *Capturing to a stream* (page 11). First we'll capture an image to a `BytesIO`[9] 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[10] 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
```

[7] https://docs.python.org/3.4/library/io.html#io.BytesIO
[8] https://docs.python.org/3.4/library/io.html#io.BytesIO
[9] https://docs.python.org/3.4/library/io.html#io.BytesIO
[10] http://effbot.org/imagingbook/pil-index.htm

```
stream.seek(0)
image = Image.open(stream)
```

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()` (page 97) 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()` (page 103) method.

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` (page 119) attribute to a reasonable value.
- Optionally, set `iso` (page 115) to a fixed value.
- To fix exposure gains, let `analog_gain` (page 105) and `digital_gain` (page 108) settle on reasonable values, then set `exposure_mode` (page 110) to `'off'`.
- To fix white balance, set the `awb_mode` (page 107) to `'off'`, then set `awb_gains` (page 106) to a (red, blue) tuple of gains.

It can be difficult to know what appropriate values might be for these attributes. For `iso` (page 115), 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` (page 119) you can query the `exposure_speed` (page 110) attribute. For exposure gains, it's usually enough to wait until `analog_gain` (page 105) is greater than 1 before `exposure_mode` (page 110) is set to `'off'`. Finally, to determine reasonable values for `awb_gains` (page 106) simply query the property while `awb_mode` (page 107) 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)])
```

Capturing timelapse sequences

The simplest way to capture long time-lapse sequences is with the `capture_continuous()` (page 98) 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`[11] 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()
```

Capturing in low light

Using similar tricks to those in *Capturing consistent images* (page 13), 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` (page 119) attribute is constrained by the camera's `framerate` (page 111) 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):

[11] https://docs.python.org/3.4/library/datetime.html#module-datetime

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

# Force sensor mode 3 (the long exposure mode), set
# the framerate to 1/6fps, the shutter speed to 6s,
# and ISO to 800 (for maximum gain)
camera = PiCamera(
    resolution=(1280, 720),
    framerate=Fraction(1, 6),
    sensor_mode=3)
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')
```

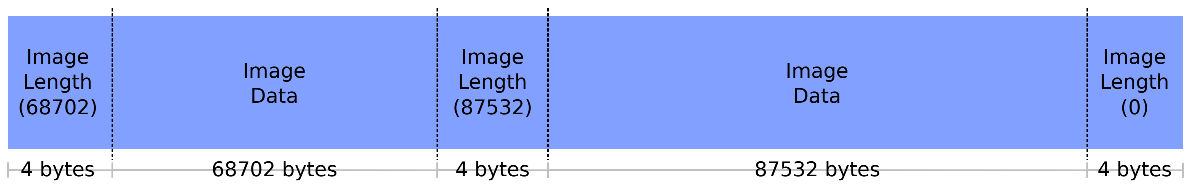
In anything other than dark conditions, the image produced by this script will most likely be completely white or at least heavily over-exposed.

Note: The Pi's camera module uses a rolling shutter[12]. 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.

When using long exposures, it is often preferable to use *framerate_range* (page 113) instead of *framerate* (page 111). This allows the camera to vary the framerate on the fly and use shorter framerates where possible (leading to shorter capture delays). This hasn't been used in the script above as the shutter speed is forced to 6 seconds (the maximum possible on the V1 camera module) which would make a framerate range pointless.

Capturing to a network stream

This is a variation of *Capturing timelapse sequences* (page 14). 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[13] 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):

```python
import io
import socket
import struct
```

[12] https://en.wikipedia.org/wiki/Rolling_shutter
[13] https://en.wikipedia.org/wiki/Endianness

```
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.

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()`* (page 105) in the example above instead of `time.sleep()`[14] which we've been using in the image capture recipes above. The *`wait_recording()`* (page 105) method is similar in that it will pause for the number of seconds specified, but unlike `time.sleep()`[15] 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()`[16] instead, such errors would only be raised by the *`stop_recording()`* (page 104) call (which could be long after the error actually occurred).

Recording video to a stream

This is very similar to *Recording video to a file* (page 17):

```
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()
```

[14] https://docs.python.org/3.4/library/time.html#time.sleep
[15] https://docs.python.org/3.4/library/time.html#time.sleep
[16] https://docs.python.org/3.4/library/time.html#time.sleep

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`[17]) you can also construct your own *custom outputs* (page 30). This is particularly useful for video recording, as discussed in the linked recipe.

Recording over multiple files

If you wish split your recording over multiple files, you can use the `split_recording()` (page 102) 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()` (page 102) method will only split files at a key-frame).

The `record_sequence()` (page 101) 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()` (page 101) method was introduced in version 1.3

Recording to a circular stream

This is similar to *Recording video to a stream* (page 17) but uses a special kind of in-memory stream provided by the picamera library. The `PiCameraCircularIO` (page 125) class implements a ring buffer[18] 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

[17] https://docs.python.org/3.4/library/io.html#io.BytesIO
[18] https://en.wikipedia.org/wiki/Circular_buffer

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()* (page 126) 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()* (page 126)

Recording to a network stream

This is similar to *Recording video to a stream* (page 17) but instead of an in-memory stream like `BytesIO`[19], we will use a file-like object created from a `socket()`[20]. Unlike the example in *Capturing to a network stream* (page 15) 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
```

[19] https://docs.python.org/3.4/library/io.html#io.BytesIO
[20] https://docs.python.org/3.4/library/socket.html#socket.socket

```
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:

```
# on the server
$ nc -l 8000 | vlc --demux h264 -

# on the client
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).

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* (page 23).

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()` (page 96):

```
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.tobytes(), 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` (page 130) which defaults to 2), made semi-transparent (with the `alpha` (page 129) property), and resized so that they don't `fill the screen` (page 130), they can be used to construct simple user interfaces.

New in version 1.8.

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` (page 106) 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` (page 105) 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.

Controlling the LED

In certain circumstances, you may find the V1 camera module's red LED a hindrance (the V2 camera module lacks an LED). 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[21].

However, provided you have the RPi.GPIO[22] 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* (page 116) 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')
```

Note: The camera LED cannot currently be controlled when the module is attached to a Raspberry Pi 3 Model B as the GPIO that controls the LED has moved to a GPIO expander not directly accessible to the ARM processor.

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.

[21] https://www.raspberrypi.org/documentation/configuration/config-txt.md
[22] https://pypi.python.org/pypi/RPi.GPIO

CHAPTER 4

Advanced Recipes

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

Warning: When trying out these scripts do *not* name your file `picamera.py`. Naming scripts after existing Python modules will cause errors when you try and import those modules (because Python checks the current directory before checking other paths).

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`[23]). 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`[24] 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`[25] (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)
```

[23] https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray
[24] https://docs.python.org/3.4/library/functions.html#bytes
[25] https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html#numpy.ndarray

```
    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.

Capturing to an OpenCV object

This is a variation on *Capturing to a numpy array* (page 25). OpenCV[26] 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:

```
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')
    image = image.reshape((240, 320, 3))
```

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

Unencoded image capture (YUV format)

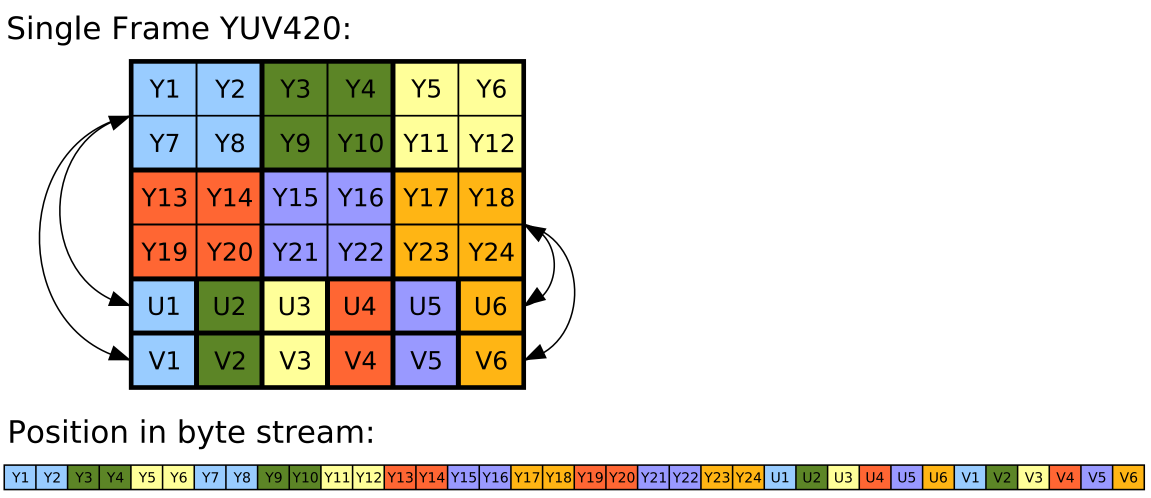
If you want images captured without loss of detail (due to JPEG's lossy compression), you are probably better off exploring PNG as an alternate image format (PNG uses lossless compression). However, some applications (particularly scientific ones) simply require the image data in numeric form. For this,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

[26] http://opencv.org/

```
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[27] format used is YUV420[28] (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[29] 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:

$$\begin{array}{rrl} & 128.0 & \text{100 rounded up to nearest multiple of 32} \\ \times & 112.0 & \text{100 rounded up to nearest multiple of 16} \\ \times & 1.5 & \text{bytes of data per pixel in YUV420 format} \\ \hline & 21504.0 & \text{bytes total} \end{array} \tag{4.1}$$

The first 14336 bytes of the data (128*112) will be Y values, the next 3584 bytes ($128 \times 112 \div 4$) 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[30] arrays, and converting the data to RGB format in an efficient manner:

```
from __future__ import division

import time
import picamera
```

[27] https://en.wikipedia.org/wiki/YUV
[28] https://en.wikipedia.org/wiki/YUV#Y.E2.80.B2UV420p_.28and_Y.E2.80.B2V12_or_YV12.29_to_RGB888_conversion
[29] https://en.wikipedia.org/wiki/YUV#Y.E2.80.B2UV420p_.28and_Y.E2.80.B2V12_or_YV12.29_to_RGB888_conversion
[30] http://www.numpy.org/

```python
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()`[31] in the code above instead of `io.open()`[32] as in the other examples. This is because numpy's `numpy.fromfile()`[33] method annoyingly only accepts "real" file objects.

This recipe is now encapsulated in the *PiYUVArray* (page 152) class in the *picamera.array* (page 151) module, which means the same can be achieved as follows:

```python
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')
```

[31] https://docs.python.org/3.4/library/functions.html#open
[32] https://docs.python.org/3.4/library/io.html#io.open
[33] https://docs.scipy.org/doc/numpy/reference/generated/numpy.fromfile.html#numpy.fromfile

```
        # 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* (page 25)). 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` (page 152) 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)* (page 29) 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()` (page 97) method. See *Raw Bayer data captures* (page 48) for more information.

Changed in version 1.0: The `raw_format` (page 117) attribute is now deprecated, as is the `'raw'` format specification for the `capture()` (page 97) method. Simply use the `'yuv'` format instead, as shown in the code above.

Changed in version 1.5: Added note about new `picamera.array` (page 151) module.

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

Unencoded image capture (RGB format)

The RGB format is rather larger than the YUV[34] format discussed in the section above, but is more useful for most analyses. To have the camera produce output in RGB[35] format, you simply need to specify `'rgb'` as the format for the `capture()` (page 97) method instead:

```
import time
import picamera

with picamera.PiCamera() as camera:
    camera.resolution = (100, 100)
    camera.start_preview()
```

[34] https://en.wikipedia.org/wiki/YUV
[35] https://en.wikipedia.org/wiki/RGB

```
    time.sleep(2)
    camera.capture('image.data', 'rgb')
```

The size of RGB[36] data can be calculated similarly to YUV[37] captures. Firstly round the resolution appropriately (see *Unencoded image capture (YUV format)* (page 26) 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:

$$\begin{array}{rrl} & 128.0 & \text{100 rounded up to nearest multiple of 32} \\ \times & 112.0 & \text{100 rounded up to nearest multiple of 16} \\ \times & 3.0 & \text{bytes of data per pixel in RGB format} \\ \hline & 43008.0 & \text{bytes total} \end{array} \tag{4.2}$$

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[38] 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[39] data are all equally sized (in contrast to YUV420[40]) it is trivial to capture directly into a numpy array (Python 3.x only; see *Capturing to a numpy array* (page 25) 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` (page 117) attribute is now deprecated, as is the `'raw'` format specification for the `capture()` (page 97) method. Simply use the `'rgb'` format instead, as shown in the code above.

Changed in version 1.5: Added note about new `picamera.array` (page 151) module.

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

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`[41]). However, building a custom output

[36] https://en.wikipedia.org/wiki/RGB
[37] https://en.wikipedia.org/wiki/YUV
[38] https://en.wikipedia.org/wiki/RGB
[39] https://en.wikipedia.org/wiki/RGB
[40] https://en.wikipedia.org/wiki/YUV#Y.E2.80.B2UV420p_.28and_Y.E2.80.B2V12_or_YV12.29_to_RGB888_conversion
[41] https://docs.python.org/3.4/library/io.html#io.BytesIO

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* (page 46). 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 if you wish to avoid dropping frames).

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:

```
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* (page 151) module which implement several custom outputs for analysis of data with numpy. In particular, the *PiMotionAnalysis* (page 156) 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.

Unconventional file outputs

As noted in prior sections, picamera accepts a wide variety of things as an output:

- A string, which will be treated as a filename.

- A file-like object, e.g. as returned by `open()`[42].
- A *custom output* (page 30).
- Any mutable object that implements the buffer interface.

The simplest of these, the filename, hides a certain amount of complexity. It can be important to understand exactly how picamera treats files, especially when dealing with "unconventional" files (e.g. pipes, FIFOs, etc.)

When given a filename, picamera does the following:

1. Opens the specified file with the `'wb'` mode, i.e. open for writing, truncating the file first, in binary mode.
2. The file is opened with a larger-than-normal buffer size, specifically 64Kb. A large buffer size is utilized because it improves performance and system load with the majority use-case, i.e. sequentially writing video to the disk.
3. The requested data (image captures, video recording, etc.) is written to the open file.
4. Finally, the file is flushed and closed. Note that this is the only circumstance in which picamera will presume to close the output for you, because picamera opened the output for you.

As noted above, this fits the majority use case (sequentially writing video to a file) very well. However, if you are piping data to another process via a FIFO (which picamera will simply treat as any other file), you may wish to avoid all the buffering. In this case, you can simply open the output yourself with no buffering. As noted above, you will then be responsible for closing the output when you are finished with it (you opened it, so the responsibility for closing it is yours as well).

For example:

```
import io
import os
import picamera

with picamera.PiCamera(resolution='VGA') as camera:
    os.mkfifo('video_fifo')
    f = io.open('video_fifo', 'wb', buffering=0)
    try:
        camera.start_recording(f, format='h264')
        camera.wait_recording(10)
        camera.stop_recording()
    finally:
        f.close()
        os.unlink('video_fifo')
```

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 *Sensor Modes* (page 75)).
- 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()` (page 97) and `capture_continuous()` (page 98) both support *use_video_port*, `capture_sequence()` (page 100) is by far the fastest method (because it does not re-initialize

[42] https://docs.python.org/3.4/library/functions.html#open

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()` (page 100) 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',
        ], use_video_port=True)
```

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()` (page 100) method:

```
import time
import picamera

frames = 60

def filenames():
    frame = 0
    while frame < frames:
        yield 'image%02d.jpg' % frame
        frame += 1

with picamera.PiCamera(resolution='720p', framerate=30) as camera:
    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. Thankfully this is relatively easy as the JPEG format has a simple magic number[43] (`FF D8`). This means we can use a *custom output* (page 30) to separate the frames out of an MJPEG video recording by inspecting the first two bytes of each buffer:

```
import io
import time
import picamera

class SplitFrames(object):
    def __init__(self):
        self.frame_num = 0
        self.output = None

    def write(self, buf):
        if buf.startswith(b'\xff\xd8'):
            # Start of new frame; close the old one (if any) and
            # open a new output
            if self.output:
                self.output.close()
            self.frame_num += 1
            self.output = io.open('image%02d.jpg' % self.frame_num, 'wb')
        self.output.write(buf)

with picamera.PiCamera(resolution='720p', framerate=30) as camera:
    camera.start_preview()
    # Give the camera some warm-up time
    time.sleep(2)
    output = SplitFrames()
    start = time.time()
    camera.start_recording(output, format='mjpeg')
    camera.wait_recording(2)
    camera.stop_recording()
    finish = time.time()
print('Captured %d frames at %.2ffps' % (
    output.frame_num,
    output.frame_num / (finish - start)))
```

So far, we've just saved the captured frames to disk. This is fine if you're intending to process later with another script, but what if we want to perform all processing within the current script? In this case, we may not need to involve the disk (or network) at all. We can set up a pool of parallel threads to accept and process image streams as captures come in:

[43] https://en.wikipedia.org/wiki/Magic_number_(programming)#Magic_numbers_in_files

```python
import io
import time
import threading
import picamera

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

    def run(self):
        # This method runs in a separate thread
        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
                    #self.owner.done=True
                finally:
                    # Reset the stream and event
                    self.stream.seek(0)
                    self.stream.truncate()
                    self.event.clear()
                    # Return ourselves to the available pool
                    with self.owner.lock:
                        self.owner.pool.append(self)

class ProcessOutput(object):
    def __init__(self):
        self.done = False
        # Construct a pool of 4 image processors along with a lock
        # to control access between threads
        self.lock = threading.Lock()
        self.pool = [ImageProcessor(self) for i in range(4)]
        self.processor = None

    def write(self, buf):
        if buf.startswith(b'\xff\xd8'):
            # New frame; set the current processor going and grab
            # a spare one
            if self.processor:
                self.processor.event.set()
            with self.lock:
                if self.pool:
                    self.processor = self.pool.pop()
                else:
                    # No processor's available, we'll have to skip
                    # this frame; you may want to print a warning
                    # here to see whether you hit this case
                    self.processor = None
        if self.processor:
            self.processor.stream.write(buf)
    def flush(self):
        # When told to flush (this indicates end of recording), shut
        # down in an orderly fashion. First, add the current processor
        # back to the pool
        if self.processor:
            with self.lock:
                self.pool.append(self.processor)
                self.processor = None
        # Now, empty the pool, joining each thread as we go
        while True:
            with self.lock:
                try:
                    proc = self.pool.pop()
                except IndexError:
                    pass # pool is empty
            proc.terminated = True
            proc.join()

with picamera.PiCamera(resolution='VGA') as camera:
    camera.start_preview()
    time.sleep(2)
    output = ProcessOutput()
    camera.start_recording(output, format='mjpeg')
    while not output.done:
        camera.wait_recording(1)
    camera.stop_recording()
```

Unencoded video capture

Just as unencoded RGB data can be captured as images, the Pi's camera module can also capture an unencoded stream of RGB (or YUV) video data. Combining this with the methods presented in *Custom outputs* (page 30) (via the classes from `picamera.array` (page 151)), we can produce a fairly rapid color detection script:

```
import picamera
import numpy as np
from picamera.array import PiRGBAnalysis
from picamera.color import Color

class MyColorAnalyzer(PiRGBAnalysis):
    def __init__(self, camera):
        super(MyColorAnalyzer, self).__init__(camera)
        self.last_color = ''

    def analyze(self, a):
        # Convert the average color of the pixels in the middle box
        c = Color(
            r=int(np.mean(a[30:60, 60:120, 0])),
            g=int(np.mean(a[30:60, 60:120, 1])),
            b=int(np.mean(a[30:60, 60:120, 2]))
            )
        # Convert the color to hue, saturation, lightness
        h, l, s = c.hls
        c = 'none'
        if s > 1/3:
            if h > 8/9 or h < 1/36:
                c = 'red'
            elif 5/9 < h < 2/3:
                c = 'blue'
            elif 5/36 < h < 4/9:
                c = 'green'
        # If the color has changed, update the display
        if c != self.last_color:
            self.camera.annotate_text = c
            self.last_color = c

with picamera.PiCamera(resolution='160x90', framerate=24) as camera:
    # Fix the camera's white-balance gains
    camera.awb_mode = 'off'
    camera.awb_gains = (1.4, 1.5)
    # Draw a box over the area we're going to watch
    camera.start_preview(alpha=128)
    box = np.zeros((96, 160, 3), dtype=np.uint8)
    box[30:60, 60:120, :] = 0x80
    camera.add_overlay(memoryview(box), size=(160, 90), layer=3, alpha=64)
    # Construct the analysis output and start recording data to it
    with MyColorAnalyzer(camera) as analyzer:
        camera.start_recording(analyzer, 'rgb')
        try:
            while True:
                camera.wait_recording(1)
        finally:
            camera.stop_recording()
```

Rapid capture and streaming

Following on from *Rapid capture and processing* (page 33), we can combine the video capture technique with *Capturing to a network stream* (page 15). 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()` (page 98) 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()
        count = 0
        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())
            count += 1
            if time.time() - start > 30:
                break
            stream.seek(0)
            stream.truncate()
    connection.write(struct.pack('<L', 0))
```

```
finally:
    connection.close()
    client_socket.close()
    finish = time.time()
print('Sent %d images in %d seconds at %.2ffps' % (
    count, finish-start, count / (finish-start)))
```

Using this technique, the author can manage about 19fps of streaming at 640x480. However, utilizing the MJPEG splitting demonstrated in *Rapid capture and processing* (page 33) we can manage much faster:

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

class SplitFrames(object):
    def __init__(self, connection):
        self.connection = connection
        self.stream = io.BytesIO()
        self.count = 0

    def write(self, buf):
        if buf.startswith(b'\xff\xd8'):
            # Start of new frame; send the old one's length
            # then the data
            size = self.stream.tell()
            if size > 0:
                self.connection.write(struct.pack('<L', size))
                self.connection.flush()
                self.stream.seek(0)
                self.connection.write(self.stream.read(size))
                self.count += 1
                self.stream.seek(0)
        self.stream.write(buf)

client_socket = socket.socket()
client_socket.connect(('my_server', 8000))
connection = client_socket.makefile('wb')
try:
    output = SplitFrames(connection)
    with picamera.PiCamera(resolution='VGA', framerate=30) as camera:
        time.sleep(2)
        start = time.time()
        camera.start_recording(output, format='mjpeg')
        camera.wait_recording(30)
        camera.stop_recording()
        # Write the terminating 0-length to the connection to let the
        # server know we're done
        connection.write(struct.pack('<L', 0))
finally:
    connection.close()
    client_socket.close()
    finish = time.time()
print('Sent %d images in %d seconds at %.2ffps' % (
    output.count, finish-start, output.count / (finish-start)))
```

The above script achieves 30fps with ease.

Web streaming

Streaming video over the web is surprisingly complicated. At the time of writing, there are still no video standards that are universally supported by all web browsers on all platforms. Furthermore, HTTP was originally designed as a one-shot protocol for serving web-pages. Since its invention, various additions have been bolted on to cater for its ever increasing use cases (file downloads, resumption, streaming, etc.) but the fact remains there's no "simple" solution for video streaming at the moment.

If you want to have a play with streaming a "real" video format (specifically, MPEG1) you may want to have a look at the pistreaming[44] demo. However, for the purposes of this recipe we'll be using a much simpler format: MJPEG. The following script uses Python's built-in `http.server`[45] module to make a simple video streaming server:

```
import io
import picamera
import logging
import socketserver
from threading import Condition
from http import server

PAGE="""\
<html>
<head>
<title>picamera MJPEG streaming demo</title>
</head>
<body>
<h1>PiCamera MJPEG Streaming Demo</h1>
<img src="stream.mjpg" width="640" height="480" />
</body>
</html>
"""

class StreamingOutput(object):
    def __init__(self):
        self.frame = None
        self.buffer = io.BytesIO()
        self.condition = Condition()

    def write(self, buf):
        if buf.startswith(b'\xff\xd8'):
            # New frame, copy the existing buffer's content and notify all
            # clients it's available
            self.buffer.truncate()
            with self.condition:
                self.frame = self.buffer.getvalue()
                self.condition.notify_all()
            self.buffer.seek(0)
        return self.buffer.write(buf)

class StreamingHandler(server.BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path == '/':
            self.send_response(301)
            self.send_header('Location', '/index.html')
            self.end_headers()
        elif self.path == '/index.html':
            content = PAGE.encode('utf-8')
            self.send_response(200)
            self.send_header('Content-Type', 'text/html')
            self.send_header('Content-Length', len(content))
```

[44] https://github.com/waveform80/pistreaming/
[45] https://docs.python.org/3.4/library/http.server.html#module-http.server

```
            self.end_headers()
            self.wfile.write(content)
        elif self.path == '/stream.mjpg':
            self.send_response(200)
            self.send_header('Age', 0)
            self.send_header('Cache-Control', 'no-cache, private')
            self.send_header('Pragma', 'no-cache')
            self.send_header('Content-Type', 'multipart/x-mixed-replace;␣
↪boundary=FRAME')
            self.end_headers()
            try:
                while True:
                    with output.condition:
                        output.condition.wait()
                        frame = output.frame
                    self.wfile.write(b'--FRAME\r\n')
                    self.send_header('Content-Type', 'image/jpeg')
                    self.send_header('Content-Length', len(frame))
                    self.end_headers()
                    self.wfile.write(frame)
                    self.wfile.write(b'\r\n')
            except Exception as e:
                logging.warning(
                    'Removed streaming client %s: %s',
                    self.client_address, str(e))
        else:
            self.send_error(404)
            self.end_headers()

class StreamingServer(socketserver.ThreadingMixIn, server.HTTPServer):
    allow_reuse_address = True
    daemon_threads = True

with picamera.PiCamera(resolution='640x480', framerate=24) as camera:
    output = StreamingOutput()
    camera.start_recording(output, format='mjpeg')
    try:
        address = ('', 8000)
        server = StreamingServer(address, StreamingHandler)
        server.serve_forever()
    finally:
        camera.stop_recording()
```

Once the script is running, visit `http://your-pi-address:8000/` with your web-browser to view the video stream.

Note: This recipe assumes Python 3.x (the `http.server` module was named `SimpleHTTPServer` in Python 2.x)

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* (page 33)) 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).

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()` (page 103) 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.

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()` (page 103) 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[46] 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 ($(640 \div 16) + 1$), in 30 rows ($480 \div 16$).

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[47]) value for each macro-block. Hence in the example above, each frame will generate 4920 bytes of motion data ($41 \times 30 \times 4$). 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[48] with Pythagoras' theorem. The SAD (Sum of Absolute Differences[49]) 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
```

[46] https://en.wikipedia.org/wiki/Macroblock
[47] https://en.wikipedia.org/wiki/Sum_of_absolute_differences
[48] https://en.wikipedia.org/wiki/Magnitude_%28mathematics%29#Euclidean_vector_space
[49] https://en.wikipedia.org/wiki/Sum_of_absolute_differences

```
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` (page 154) and `PiMotionAnalysis` (page 156) classes in the `picamera.array` (page 151) 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)
```

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
```

Finally, as a demonstration of what can be accomplished with motion vectors, here's a gesture detection system:

```
import os
import numpy as np
import picamera
from picamera.array import PiMotionAnalysis

class GestureDetector(PiMotionAnalysis):
    QUEUE_SIZE = 10 # the number of consecutive frames to analyze
    THRESHOLD = 4.0 # the minimum average motion required in either axis

    def __init__(self, camera):
        super(GestureDetector, self).__init__(camera)
        self.x_queue = np.zeros(self.QUEUE_SIZE, dtype=np.float)
        self.y_queue = np.zeros(self.QUEUE_SIZE, dtype=np.float)
        self.last_move = ''

    def analyze(self, a):
        # Roll the queues and overwrite the first element with a new
        # mean (equivalent to pop and append, but faster)
        self.x_queue[1:] = self.x_queue[:-1]
        self.y_queue[1:] = self.y_queue[:-1]
        self.x_queue[0] = a['x'].mean()
        self.y_queue[0] = a['y'].mean()
        # Calculate the mean of both queues
        x_mean = self.x_queue.mean()
        y_mean = self.y_queue.mean()
        # Convert left/up to -1, right/down to 1, and movement below
        # the threshold to 0
        x_move = (
            '' if abs(x_mean) < self.THRESHOLD else
            'left' if x_mean < 0.0 else
            'right')
        y_move = (
            '' if abs(y_mean) < self.THRESHOLD else
            'down' if y_mean < 0.0 else
            'up')
        # Update the display
        movement = ('%s %s' % (x_move, y_move)).strip()
        if movement != self.last_move:
            self.last_move = movement
            if movement:
                print(movement)

with picamera.PiCamera(resolution='VGA', framerate=24) as camera:
    with GestureDetector(camera) as detector:
        camera.start_recording(
            os.devnull, format='h264', motion_output=detector)
        try:
            while True:
                camera.wait_recording(1)
        finally:
            camera.stop_recording()
```

Within a few inches of the camera, move your hand up, down, left, and right, parallel to the camera and you should see the direction displayed on the console.

New in version 1.5.

Splitting to/from a circular stream

This example builds on the one in *Recording to a circular stream* (page 18) and the one in *Capturing images whilst recording* (page 41) to demonstrate the beginnings of a security application. As before, a `PiCameraCircularIO` (page 125) 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:

```python
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()` (page 126) 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.

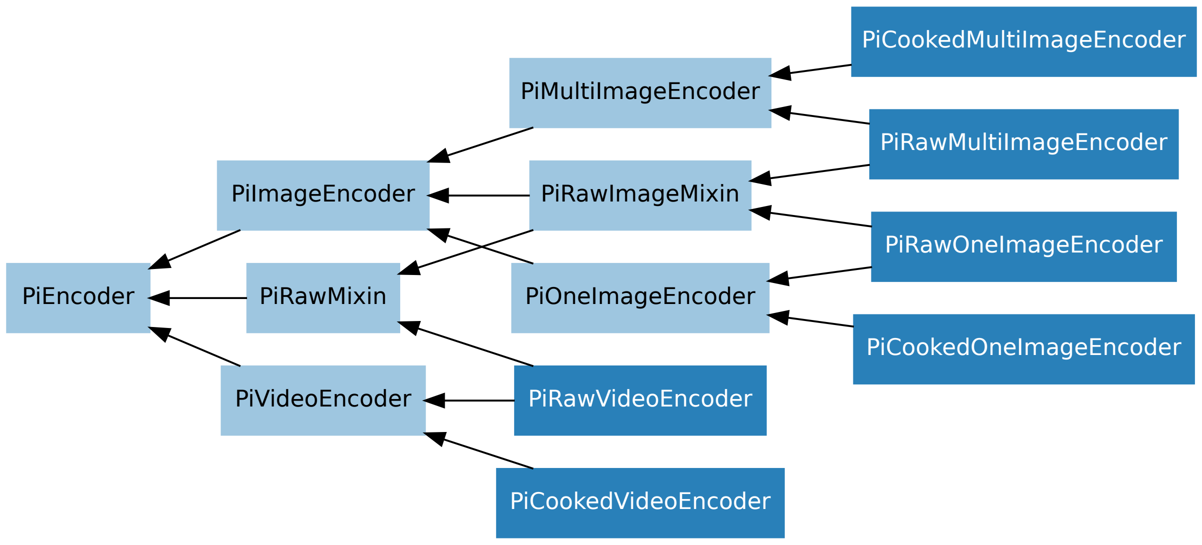
Changed in version 1.11: Added use of `copy_to()` (page 126)

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* (page 30) 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` (page 111) attribute which you could access from your custom output's `write` method.

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



The following table details which `PiCamera` (page 95) methods use which encoder classes, and which method they call to construct these encoders:

Method(s)	Calls	Returns
`capture()` (page 97) `capture_continuous()` (page 98) `capture_sequence()` (page 100)	`_get_image_encoder()`	`PiCookedOneImageEncoder` (page 139) `PiRawOneImageEncoder` (page 139)
`capture_sequence()` (page 100)	`_get_images_encoder()`	`PiCookedMultiImageEncoder` (page 139) `PiRawMultiImageEncoder` (page 139)
`start_recording()` (page 103) `record_sequence()` (page 101)	`_get_video_encoder()`	`PiCookedVideoEncoder` (page 138) `PiRawVideoEncoder` (page 138)

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` (page 138) class to store how many I-frames and P-frames are captured (the camera's encoder doesn't use B-frames):

```python
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.flags & mmal.MMAL_BUFFER_HEADER_FLAG_FRAME_END) and
                not (buf.flags & mmal.MMAL_BUFFER_HEADER_FLAG_CONFIG)
            ):
            if buf.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* (page 42). 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.

Raw Bayer data captures

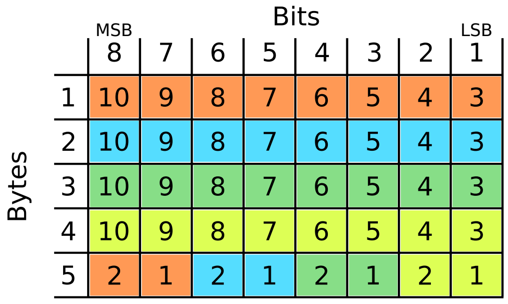
The `bayer` parameter of the `capture()` (page 97) method causes the raw Bayer data recorded by the camera's sensor to be output as part of the image meta-data.

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:

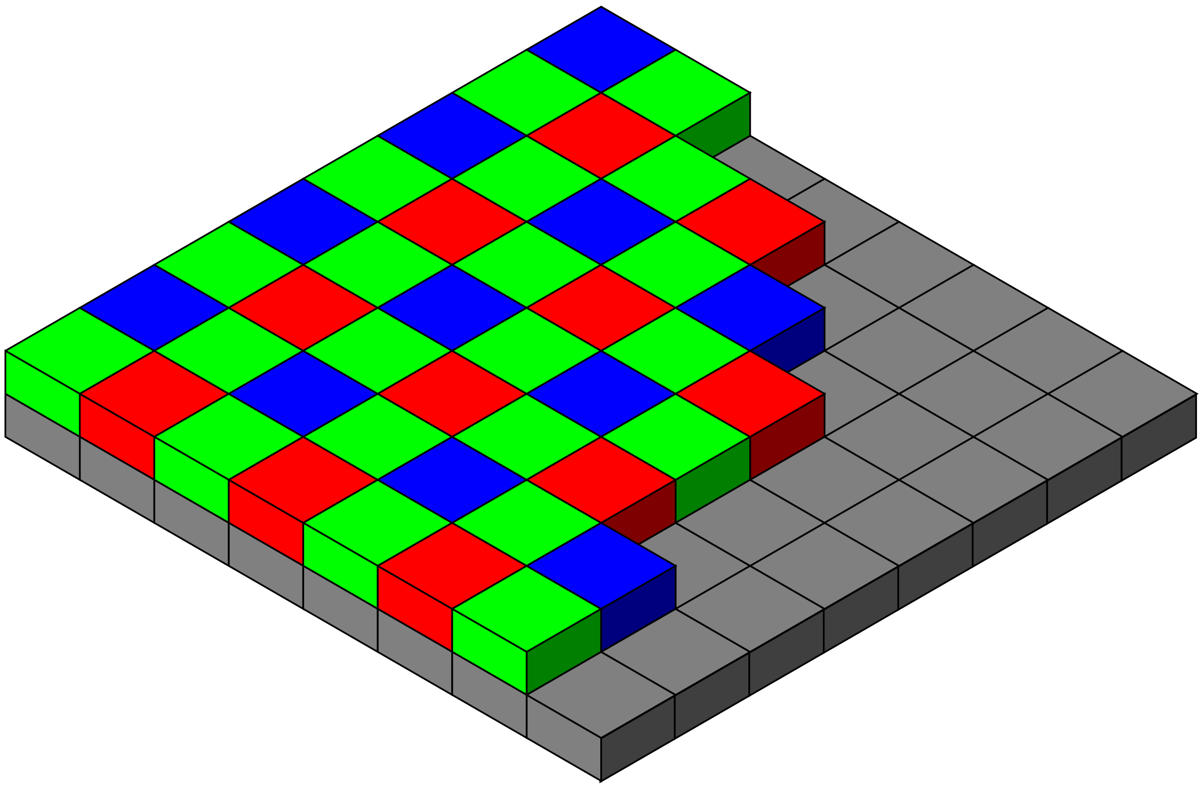
- Bayer data is *always* full resolution, regardless of the camera's output `resolution` (page 117) and any `resize` parameter.
- Bayer data occupies the last 6,404,096 bytes of the output file for the V1 module, or the last 10,270,208 bytes for the V2 module. The first 32,768 bytes of this is header data which starts with the string `'BRCM'`.

- Bayer data consists of 10-bit values, because this is the sensitivity of the OV5647[50] and IMX219[51] 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	8 (MSB)	7	6	5	4	3	2	1 (LSB)
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[52]). 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[53] at the very least, and probably some form of color balance[54].

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[55] 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:

[50] http://www.ovt.com/products/sensor.php?id=66
[51] http://www.sony.net/Products/SC-HP/new_pro/april_2014/imx219_e.html
[52] https://en.wikipedia.org/wiki/Bayer_filter
[53] https://en.wikipedia.org/wiki/Demosaicing
[54] https://en.wikipedia.org/wiki/Color_balance
[55] http://www.numpy.org/

```
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 = 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)
```

An enhanced version of this recipe (which also handles different bayer orders caused by flips and rotations) is also encapsulated in the *PiBayerArray* (page 153) class in the *picamera.array* (page 151) 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* (page 151) module.

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[56]). This will fire before (flash metering[57]) 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[58]. Firstly, install the device tree compiler, then grab a copy of the default device tree source:

```
$ sudo apt-get install device-tree-compiler
$ wget https://github.com/raspberrypi/firmware/raw/master/extra/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 (check the silk-screen writing on the board for the revision number if you are unsure):

Model	Section
Raspberry Pi Model B rev 1	`/videocore/pins_rev1`
Raspberry Pi Model A and Model B rev 2	`/videocore/pins_rev2`
Raspberry Pi Model A+	`/videocore/pins_aplus`
Raspberry Pi Model B+ rev 1.1	`/videocore/pins_bplus1`
Raspberry Pi Model B+ rev 1.2	`/videocore/pins_bplus2`
Raspberry Pi 2 Model B rev 1.0	`/videocore/pins_2b1`
Raspberry Pi 2 Model B rev 1.1 and rev 1.2	`/videocore/pins_2b2`
Raspberry Pi 3 Model B rev 1.0	`/videocore/pins_3b1`
Raspberry Pi 3 Model B rev 1.2	`/videocore/pins_3b2`
Raspberry Pi Zero rev 1.2 and rev 1.3	`/videocore/pins_pi0`

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 2 Model B rev 1.1 you would add the following line under the `/videocore/pins_2b2/pin_config` section:

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

Please note that GPIO pins will be numbered according to the Broadcom pin numbers[59] (BCM mode in the RPi.GPIO library, *not* BOARD mode). Then change the following section under `/videocore/pins_2b2/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 -q -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

[56] https://en.wikipedia.org/wiki/Rolling_shutter
[57] https://en.wikipedia.org/wiki/Through-the-lens_metering#Through_the_lens_flash_metering
[58] https://www.raspberrypi.org/documentation/configuration/pin-configuration.md
[59] https://raspberrypi.stackexchange.com/questions/12966/what-is-the-difference-between-board-and-bcm-for-gpio-pin-numbering

- `-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 nothing. If you get lots of warnings, you've forgotten the `-q` switch; you can ignore the warnings. 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 everything else). 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.

CHAPTER 5

Frequently Asked Questions (FAQ)

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.

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`* (page 131) attribute of the *`preview`* (page 117) 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.

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()`* (page 104) 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()`* (page 103) 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.

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[60], have approximated a preview by rapidly capturing unencoded images and displaying them on the PiTFT instead.

How much power does the camera require?

The camera requires 250mA[61] when running. Note that simply creating a *PiCamera* (page 95) 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()* (page 101) (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()* (page 98) 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* (page 14).

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.

[60] https://learn.adafruit.com/diy-wifi-raspberry-pi-touch-cam/overview
[61] https://www.raspberrypi.org/help/faqs/#cameraPower

How can I take two consecutive pictures with equivalent settings?

See the *Capturing consistent images* (page 13) recipe.

Can I use picamera with a USB webcam?

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

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
```

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* (page 1), and install using *pip instead* (page 2). 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).

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.

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* (page 18) 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()` (page 126) 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.

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.

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.

To convert the H264 NAL stream to a full MP4 file, there are a couple of options. The simplest is to use the `MP4Box` utility from the `gpac` package on Raspbian. Unfortunately this only works with files; it cannot accept redirected streams:

```
$ sudo apt-get install gpac
...
$ MP4Box -add input.h264 output.mp4
```

Alternatively you can use the console version of VLC to handle the conversion. This is a more complex command line, but a lot more powerful (it'll handle redirected streams and can be used with a vast array of outputs including HTTP, RTP, etc.):

```
$ sudo apt-get install vlc
...
$ cvlc input.h264 --play-and-exit --sout \
> '#standard{access=file,mux=mp4,dst=output.mp4}' :demux=h264 \
```

Or to read from stdin:

```
$ raspivid -t 5000 -o - | cvlc stream:///dev/stdin \
> --play-and-exit --sout \
> '#standard{access=file,mux=mp4,dst=output.mp4}' :demux=h264 \
```

Out of resources at full resolution on a V2 module

See *Hardware Limits* (page 78).

Preview flickers at full resolution on a V2 module

Use the new `resolution` (page 132) 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))
```

Camera locks up with multiprocessing

The camera firmware is designed to be used by a *single* process at a time. Attempting to use the camera from multiple processes simultaneously will fail in a variety of ways (from simple errors to the process locking up).

Python's `multiprocessing`[62] module creates multiple copies of a Python process (usually via `os.fork()`[63]) for the purpose of parallel processing. Whilst you can use `multiprocessing`[64] with picamera, you must ensure that only a *single* process creates a `PiCamera` (page 95) instance at any given time.

The following script demonstrates an approach with one process that owns the camera, which handles disseminating captured frames to other processes via a `Queue`[65]:

[62] https://docs.python.org/3.4/library/multiprocessing.html#module-multiprocessing
[63] https://docs.python.org/3.4/library/os.html#os.fork
[64] https://docs.python.org/3.4/library/multiprocessing.html#module-multiprocessing
[65] https://docs.python.org/3.4/library/multiprocessing.html#multiprocessing.Queue

```
import os
import io
import time
import multiprocessing as mp
from queue import Empty
import picamera
from PIL import Image

class QueueOutput(object):
    def __init__(self, queue, finished):
        self.queue = queue
        self.finished = finished
        self.stream = io.BytesIO()

    def write(self, buf):
        if buf.startswith(b'\xff\xd8'):
            # New frame, put the last frame's data in the queue
            size = self.stream.tell()
            if size:
                self.stream.seek(0)
                self.queue.put(self.stream.read(size))
                self.stream.seek(0)
        self.stream.write(buf)

    def flush(self):
        self.queue.close()
        self.queue.join_thread()
        self.finished.set()

def do_capture(queue, finished):
    with picamera.PiCamera(resolution='VGA', framerate=30) as camera:
        output = QueueOutput(queue, finished)
        camera.start_recording(output, format='mjpeg')
        camera.wait_recording(10)
        camera.stop_recording()

def do_processing(queue, finished):
    while not finished.wait(0.1):
        try:
            stream = io.BytesIO(queue.get(False))
        except Empty:
            pass
        else:
            stream.seek(0)
            image = Image.open(stream)
            # Pretend it takes 0.1 seconds to process the frame; on a quad-core
            # Pi this gives a maximum processing throughput of 40fps
            time.sleep(0.1)
            print('%d: Processing image with size %dx%d' % (
                os.getpid(), image.size[0], image.size[1]))

if __name__ == '__main__':
    queue = mp.Queue()
    finished = mp.Event()
    capture_proc = mp.Process(target=do_capture, args=(queue, finished))
    processing_procs = [
        mp.Process(target=do_processing, args=(queue, finished))
        for i in range(4)
        ]
    for proc in processing_procs:
        proc.start()
    capture_proc.start()
    for proc in processing_procs:
        proc.join()
    capture_proc.join()
```

VLC won't play back MJPEG recordings

MJPEG[66] is a particularly ill-defined format (see "Disadvantages[67]") which results in compatibility issues between software that purports to produce MJPEG files, and software that purports to play MJPEG files. This is one such case: the Pi's camera firmware produces an MJPEG file which simply consists of concatenated JPEGs; this is reasonably common on other devices and webcams, and is a nice simple format which makes parsing particularly easy (see *Web streaming* (page 40) for an example).

Unfortunately, VLC doesn't recognize this as a valid MJPEG file: it thinks it's a single JPEG image and doesn't bother reading the rest of the file (which is also a reasonable interpretation in the absence of any other information). Thankfully, extra command line switches can be provided to give it a hint that there's more to read in the file:

```
$ vlc --demux=mjpeg --mjpeg-fps=30 my_recording.mjpeg
```

[66] https://en.wikipedia.org/wiki/Motion_JPEG
[67] https://en.wikipedia.org/wiki/Motion_JPEG#Disadvantages

CHAPTER 6

Camera Hardware

This chapter provides an overview of how the camera works under various conditions, as well as an introduction to the software interface that picamera uses.

Theory of Operation

Many questions I receive regarding picamera are based on misunderstandings of how the camera works. This chapter attempts to correct those misunderstandings and gives the reader a basic description of the operation of the camera. The chapter deliberately follows a lie-to-children[68] model, presenting first a technically inaccurate but useful model of the camera's operation, then refining it closer to the truth later on.

Misconception #1

The Pi's camera module is basically a mobile phone camera module. Mobile phone digital cameras differ from larger, more expensive, cameras (DSLRs[69]) in a few respects. The most important of these, for understanding the Pi's camera, is that many mobile cameras (including the Pi's camera module) use a rolling shutter[70] to capture images. When the camera needs to capture an image, it reads out pixels from the sensor a row at a time rather than capturing all pixel values at once.

In fact, the "global shutter" on DSLRs typically also reads out pixels a row at a time. The major difference is that a DSLR will have a physical shutter that covers the sensor. Hence in a DSLR the procedure for capturing an image is to open the shutter, letting the sensor "view" the scene, close the shutter, then read out each line from the sensor.

The notion of "capturing an image" is thus a bit misleading as what we actually mean is "reading each row from the sensor in turn and assembling them back into an image".

Misconception #2

The notion that the camera is effectively idle until we tell it to capture a frame is also misleading. Don't think of the camera as a still image camera. Think of it as a video camera. Specifically one that, as soon as it is initialized, is constantly streaming frames (or rather rows of frames) down the ribbon cable to the Pi for processing.

[68] https://en.wikipedia.org/wiki/Lie-to-children
[69] https://en.wikipedia.org/wiki/Digital_single-lens_reflex_camera
[70] https://en.wikipedia.org/wiki/Rolling_shutter

The camera may seem idle, and your script may be doing nothing with the camera, but still numerous tasks are going on in the background (automatic gain control, exposure time, white balance, and several other tasks which we'll cover later on).

This background processing is why most of the picamera example scripts seen in prior chapters include a `sleep(2)` line after initializing the camera. The `sleep(2)` statement pauses your script for a couple of seconds. During this pause, the camera's firmware continually receives rows of frames from the camera and adjusts the sensor's gain and exposure times to make the frame look "normal" (not over- or under-exposed, etc).

So when we request the camera to "capture a frame" what we're really requesting is that the camera give us the next complete frame it assembles, rather than using it for gain and exposure then discarding it (as happens constantly in the background otherwise).

Exposure time

What does the camera sensor *actually detect*? It detects photon counts; the more photons that hit the sensor elements, the more those elements increment their counters. As our camera has no physical shutter (unlike a DSLR) we can't prevent light falling on the elements and incrementing the counts. In fact we can only perform two operations on the sensor: reset a row of elements, or read a row of elements.

To understand a typical frame capture, let's walk through the capture of a couple of frames of data with a hypothetical camera sensor, with only 8x8 pixels and no Bayer filter[71]. The sensor is sat in bright light, but as it's just been initialized, all the elements start off with a count of 0. The sensor's elements are shown on the left, and the frame buffer, that we'll read values into, is on the right:

Sensor elements								–>	Frame 1							
0	0	0	0	0	0	0	0									
0	0	0	0	0	0	0	0									
0	0	0	0	0	0	0	0									
0	0	0	0	0	0	0	0									
0	0	0	0	0	0	0	0									
0	0	0	0	0	0	0	0									
0	0	0	0	0	0	0	0									
0	0	0	0	0	0	0	0									

The first line of data is reset (in this case that doesn't change the state of any of the sensor elements). Whilst resetting that line, light is still falling on all the other elements so they increment by 1:

Sensor elements								–>	Frame 1							
0	0	0	0	0	0	0	0	Rst								
1	1	1	1	1	1	1	1									
1	1	1	1	1	1	1	1									
1	1	1	1	1	1	1	1									
1	1	1	1	1	1	1	1									
1	1	1	1	1	1	1	1									
1	1	1	1	1	1	1	1									
1	1	1	1	1	1	1	1									

The second line of data is reset (this time some sensor element states change). All other elements increment by 1. We've not read anything yet, because we want to leave a delay for the first row to "see" enough light before we read it:

[71] https://en.wikipedia.org/wiki/Bayer_filter

Sensor elements								–>	Frame 1							
1	1	1	1	1	1	1	1									
0	0	0	0	0	0	0	0	Rst								
2	2	2	2	2	2	2	2									
2	2	2	2	2	2	2	2									
2	2	2	2	2	2	2	2									
2	2	2	2	2	2	2	2									
2	2	2	2	2	2	2	2									
2	2	2	2	2	2	2	2									

The third line of data is reset. Again, all other elements increment by 1:

Sensor elements								–>	Frame 1							
2	2	2	2	2	2	2	2									
1	1	1	1	1	1	1	1									
0	0	0	0	0	0	0	0	Rst								
3	3	3	3	3	3	3	3									
3	3	3	3	3	3	3	3									
3	3	3	3	3	3	3	3									
3	3	3	3	3	3	3	3									
3	3	3	3	3	3	3	3									

Now the camera starts reading and resetting. The first line is read and the fourth line is reset:

Sensor elements								–>	Frame 1							
3	3	3	3	3	3	3	3	–>	3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2									
1	1	1	1	1	1	1	1									
0	0	0	0	0	0	0	0	Rst								
4	4	4	4	4	4	4	4									
4	4	4	4	4	4	4	4									
4	4	4	4	4	4	4	4									
4	4	4	4	4	4	4	4									

The second line is read whilst the fifth line is reset:

Sensor elements								–>	Frame 1							
4	4	4	4	4	4	4	4		3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	–>	3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2									
1	1	1	1	1	1	1	1									
0	0	0	0	0	0	0	0	Rst								
5	5	5	5	5	5	5	5									
5	5	5	5	5	5	5	5									
5	5	5	5	5	5	5	5									

At this point it should be fairly clear what's going on, so let's fast-forward to the point where the final line is reset:

Sensor elements								–>	Frame 1							
7	7	7	7	7	7	7	7		3	3	3	3	3	3	3	3
6	6	6	6	6	6	6	6		3	3	3	3	3	3	3	3
5	5	5	5	5	5	5	5		3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4		3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	–>	3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2									
1	1	1	1	1	1	1	1									
0	0	0	0	0	0	0	0	Rst								

At this point, the camera can start resetting the first line again while continuing to read the remaining lines from the sensor:

Sensor elements								–>	Frame 1							
0	0	0	0	0	0	0	0	Rst	3	3	3	3	3	3	3	3
7	7	7	7	7	7	7	7		3	3	3	3	3	3	3	3
6	6	6	6	6	6	6	6		3	3	3	3	3	3	3	3
5	5	5	5	5	5	5	5		3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4		3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	–>	3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2									
1	1	1	1	1	1	1	1									

Let's fast-forward to the state where the last row has been read. Our first frame is now complete:

Sensor elements								–>	Frame 1							
2	2	2	2	2	2	2	2		3	3	3	3	3	3	3	3
1	1	1	1	1	1	1	1		3	3	3	3	3	3	3	3
0	0	0	0	0	0	0	0	Rst	3	3	3	3	3	3	3	3
7	7	7	7	7	7	7	7		3	3	3	3	3	3	3	3
6	6	6	6	6	6	6	6		3	3	3	3	3	3	3	3
5	5	5	5	5	5	5	5		3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4		3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	–>	3	3	3	3	3	3	3	3

At this stage, Frame 1 would be sent off for post-processing and Frame 2 would be read into a new buffer:

Sensor elements								–>	Frame 2							
3	3	3	3	3	3	3	3	–>	3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2									
1	1	1	1	1	1	1	1									
0	0	0	0	0	0	0	0	Rst								
7	7	7	7	7	7	7	7									
6	6	6	6	6	6	6	6									
5	5	5	5	5	5	5	5									
4	4	4	4	4	4	4	4									

From the example above it should be clear that we can control the exposure time of a frame by varying the delay between resetting a line and reading it (reset and read don't really happen simultaneously, but they are synchronized which is all that matters for this process).

Minimum exposure time

There are naturally limits to the minimum exposure time: reading out a line of elements must take a certain minimum time. For example, if there are 500 rows on our hypothetical sensor, and reading each row takes a minimum of 20ns then it will take a minimum of $500 \times 20\text{ns} = 10\text{ms}$ to read a full frame. This is the *minimum* exposure time of our hypothetical sensor.

Maximum framerate is determined by the minimum exposure time

The framerate is the number of frames the camera can capture per second. Depending on the time it takes to capture one frame, the exposure time, we can only capture so many frames in a specific amount of time. For example, if it takes 10ms to read a full frame, then we cannot capture more than $\frac{1\text{s}}{10\text{ms}} = \frac{1\text{s}}{0.01\text{s}} = 100$ frames in a second. Hence the maximum framerate of our hypothetical 500 row sensor is 100fps.

This can be expressed in the word equation: $\frac{1\text{s}}{\text{min exposure time in s}} = \text{max framerate in fps}$ from which we can see the inverse relationship. The lower the minimum exposure time, the larger the maximum framerate and vice versa.

Maximum exposure time is determined by the minimum framerate

To maximise the exposure time we need to capture as few frames as possible per second, i.e. we need a very low framerate. Therefore the *maximum* exposure time is determined by the camera's *minimum* framerate. The minimum framerate is largely determined by how slow the sensor can be made to read lines (at the hardware level this is down to the size of registers for holding things like line read-out times).

This can be expressed in the word equation: $\frac{1\text{s}}{\text{min framerate in fps}} = \text{max exposure time in s}$

If we imagine that the minimum framerate of our hypothetical sensor is ½fps then the maximum exposure time will be $\frac{1\text{s}}{1/2} = 2\text{s}$.

Exposure time is limited by current framerate

More generally, the `framerate` (page 111) setting of the camera limits the maximum exposure time of a given frame. For example, if we set the framerate to 30fps, then we cannot spend more than $\frac{1\text{s}}{30} = 33^1/_3\text{ms}$ capturing any given frame.

Therefore, the `exposure_speed` (page 110) attribute, which reports the exposure time of the last processed frame (which is really a multiple of the sensor's line read-out time) is limited by the camera's `framerate` (page 111).

Note: Tiny framerate adjustments, done with `framerate_delta` (page 112), are achieved by reading extra "dummy" lines at the end of a frame. I.e reading a line but then discarding it.

Sensor gain

The other important factor influencing sensor element counts, aside from line read-out time, is the sensor's gain[72]. Specifically, the gain given by the `analog_gain` (page 105) attribute (the corresponding `digital_gain` (page 108) is simply post-processing which we'll cover later). However, there's an obvious issue: how is this gain "analog" if we're dealing with digital photon counts?

Time to reveal the first lie: the sensor elements are not simple digital counters but are in fact analog components that build up charge as more photons hit them. The analog gain influences how this charge is built-up. An analog-to-digital converter[73] (ADC) is used to convert the analog charge to a digital value during line read-out (in fact the ADC's speed is a large portion of the minimum line read-out time).

Note: Camera sensors also tend to have a border of non-sensing pixels (elements that are covered from light). These are used to determine what level of charge represents "optically black".

The camera's elements are affected by heat (thermal radiation, after all, is just part of the electromagnetic spectrum[74] close to the visible portion). Without the non-sensing pixels you would get different black levels at different ambient temperatures.

The analog gain cannot be *directly* controlled in picamera, but various attributes can be used to "influence" it.

- Setting `exposure_mode` (page 110) to `'off'` locks the analog (and digital) gains at their current values and doesn't allow them to adjust at all, no matter what happens to the scene, and no matter what other camera attributes may be adjusted.
- Setting `exposure_mode` (page 110) to values other than `'off'` permits the gains to "float" (change) according to the auto-exposure mode selected. Where possible, the camera firmware prefers to adjust the analog gain rather than the digital gain, because increasing the digital gain produces more noise. Some examples of the adjustments made for different auto-exposure modes include:
  - `'sports'` reduces motion blur by preferentially increasing gain rather than exposure time (i.e. line read-out time).

[72] https://en.wikipedia.org/wiki/Gain_(electronics)
[73] https://en.wikipedia.org/wiki/Analog-to-digital_converter
[74] https://en.wikipedia.org/wiki/Electromagnetic_spectrum

  - `'night'` is intended as a stills mode, so it permits very long exposure times while attempting to keep gains low.

- The `iso` (page 115) attribute effectively represents another set of auto-exposure modes with specific gains:
  - With the V1 camera module, ISO 100 attempts to use an overall gain of 1.0. ISO 200 attempts to use an overall gain of 2.0, and so on.
  - With the V2 camera module, ISO 100 produces an overall gain of ~1.84. ISO 60 produces overall gain of 1.0, and ISO 800 of 14.72 (the V2 camera module was calibrated against the ISO film speed[75] standard).

  Hence, one might be tempted to think that `iso` (page 115) provides a means of fixing the gains, but this isn't entirely true: the `exposure_mode` (page 110) setting takes precedence (setting the exposure mode to `'off'` will fix the gains no matter what ISO is later set, and some exposure modes like `'spotlight'` also override ISO-adjusted gains).

Division of labor

At this point, a reader familiar with operating system theory may be questioning how a non real-time operating system[76] (non-RTOS) like Linux could possibly be reading lines from the sensor? After all, to ensure each line is read in exactly the same amount of time (to ensure a constant exposure over the whole frame) would require extremely precise timing, which cannot be achieved in a non-RTOS.

Time to reveal the second lie: lines are not actively "read" from the sensor. Rather, the sensor is configured (via its registers) with a time per line and number of lines to read. Once started, the sensor simply reads lines, pushing the data out to the Pi at the configured speed.

That takes care of how each line's read-out time is kept constant, but it still doesn't answer the question of how we can guarantee that Linux is actually listening and ready to accept each line of data? The answer is quite simply that Linux *doesn't*. The CPU doesn't talk to the camera directly. In fact, none of the camera processing occurs on the CPU (running Linux) at all. Instead, it is done on the Pi's GPU (VideoCore IV) which is running its own real-time OS (VCOS).

Note: This is another lie: VCOS is actually an abstraction layer on top of an RTOS running on the GPU (ThreadX at the time of writing). However, given that RTOS has changed in the past (hence the abstraction layer), and that the user doesn't directly interact with it anyway, it is perhaps simpler to think of the GPU as running something called VCOS (without thinking too much about what that actually is).

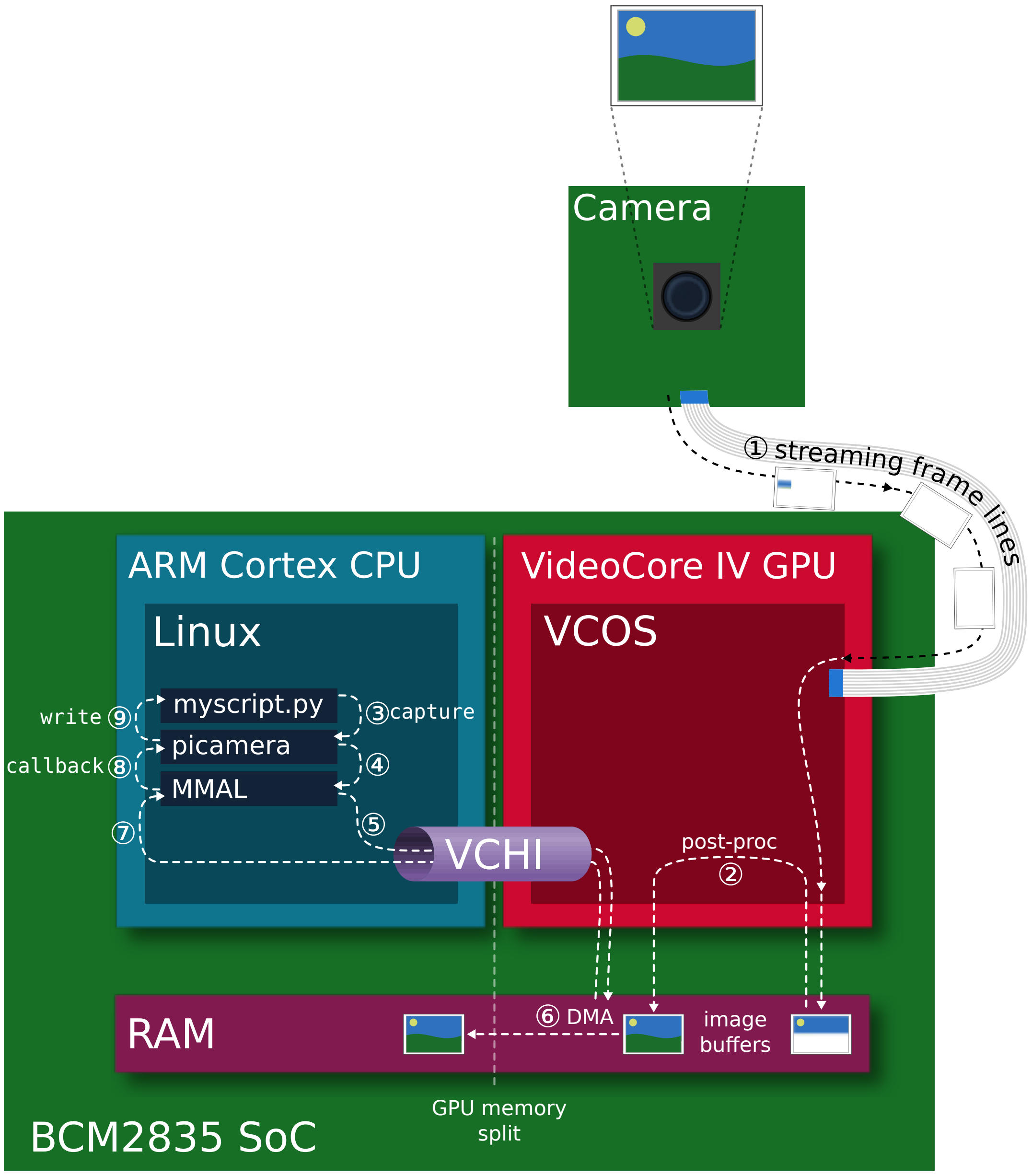
The following diagram illustrates that the BCM2835 system on a chip[77] (SoC) is comprised of an ARM Cortex CPU running Linux (under which is running `myscript.py` which is using picamera), and a VideoCore IV GPU running VCOS. The VideoCore Host Interface (VCHI) is a message passing system provided to permit communication between these two components. The available RAM is split between the two components (128Mb is a typical GPU memory split when using the camera). Finally, the camera module is shown above the SoC. It is connected to the SoC via a CSI-2 interface (providing 2Gbps of bandwidth).

The scenario depicted is as follows:

1. The camera's sensor has been configured and is continually streaming frame lines over the CSI-2 interface to the GPU.
2. The GPU is assembling complete frame buffers from these lines and performing post-processing on these buffers (we'll go into further detail about this part in the next section).
3. Meanwhile, over on the CPU, `myscript.py` makes a `capture` call using picamera.
4. The picamera library in turn uses the MMAL API to enact this request (actually there's quite a lot of MMAL calls that go on here but for the sake of simplicity we represent all this with a single arrow).

[75] https://en.wikipedia.org/wiki/Film_speed#Current_system:_ISO
[76] https://en.wikipedia.org/wiki/Real-time_operating_system
[77] https://en.wikipedia.org/wiki/System_on_a_chip

5. The MMAL API sends a message over VCHI requesting a frame capture (again, in reality there's a lot more activity than a single message).
6. In response, the GPU initiates a DMA[78] transfer of the next complete frame from its portion of RAM to the CPU's portion.
7. Finally, the GPU sends a message back over VCHI that the capture is complete.
8. This causes an MMAL thread to fire a callback in the picamera library, which in turn retrieves the frame (in reality, this requires more MMAL and VCHI activity).
9. Finally, picamera calls `write` on the output object provided by `myscript.py`.



Background processes

We've alluded briefly to some of the GPU processing going on in the sections above (gain control, exposure time, white balance, frame encoding, etc). Time to reveal the final lie: the GPU is not, as depicted in the prior section, one discrete component. Rather it is composed of numerous components each of which play a role in the camera's operation.

[78] https://en.wikipedia.org/wiki/Direct_memory_access

The diagram below depicts a more accurate representation of the GPU side of the BCM2835 SoC. From this we get our first glimpse of the frame processing "pipeline" and why it is called such. In the diagram, an H264 video is being recorded. The components that data passes through are as follows:

1. Starting at the camera module, some minor processing happens. Specifically, flips (horizontal and vertical), line skipping, and pixel binning[79] are configured on the sensor's registers. Pixel binning actually happens on the sensor itself, prior to the ADC to improve signal-to-noise ratios. See *hflip* (page 113), *vflip* (page 120), and *sensor_mode* (page 118).

2. As described previously, frame lines are streamed over the CSI-2 interface to the GPU. There, it is received by the Unicam component which writes the line data into RAM.

3. Next the GPU's image signal processor[80] (ISP) performs several post-processing steps on the frame data.

   These include (in order):

   - **Transposition**: If any rotation has been requested, the input is transposed to rotate the image (rotation is always implemented by some combination of transposition and flips).
   - **Black level compensation**: Use the non-light sensing elements (typically in a covered border) to determine what level of charge represents "optically black".
   - **Lens shading**: The camera firmware includes a table that corrects for chromatic distortion from the standard module's lens. This is one reason why third party modules incorporating different lenses may show non-uniform color across a frame.
   - **White balance**: The red and blue gains are applied to correct the color balance[81]. See *awb_gains* (page 106) and *awb_mode* (page 107).
   - **Digital gain**: As mentioned above, this is a straight-forward post-processing step that applies a gain to the Bayer values[82]. See *digital_gain* (page 108).
   - **Bayer de-noise**: This is a noise reduction algorithm run on the frame data while it is still in Bayer format.
   - **De-mosaic:** The frame data is converted from Bayer format to YUV420[83] which is the format used by the remainder of the pipeline.
   - **YUV de-noise**: Another noise reduction algorithm, this time with the frame in YUV420 format. See *image_denoise* (page 113) and *video_denoise* (page 120).
   - **Sharpening**: An algorithm to enhance edges in the image. See *sharpness* (page 119).
   - **Color processing**: The *brightness* (page 107), *contrast* (page 108), and *saturation* (page 118) adjustments are implemented.
   - **Distortion**: The distortion introduced by the camera's lens is corrected. At present this stage does nothing as the stock lens isn't a fish-eye lens[84]; it exists as an option should a future sensor require it.
   - **Resizing**: At this point, the frame is resized to the requested output resolution (all prior stages have been performed on "full" frame data at whatever resolution the sensor is configured to produce). See *resolution* (page 117).

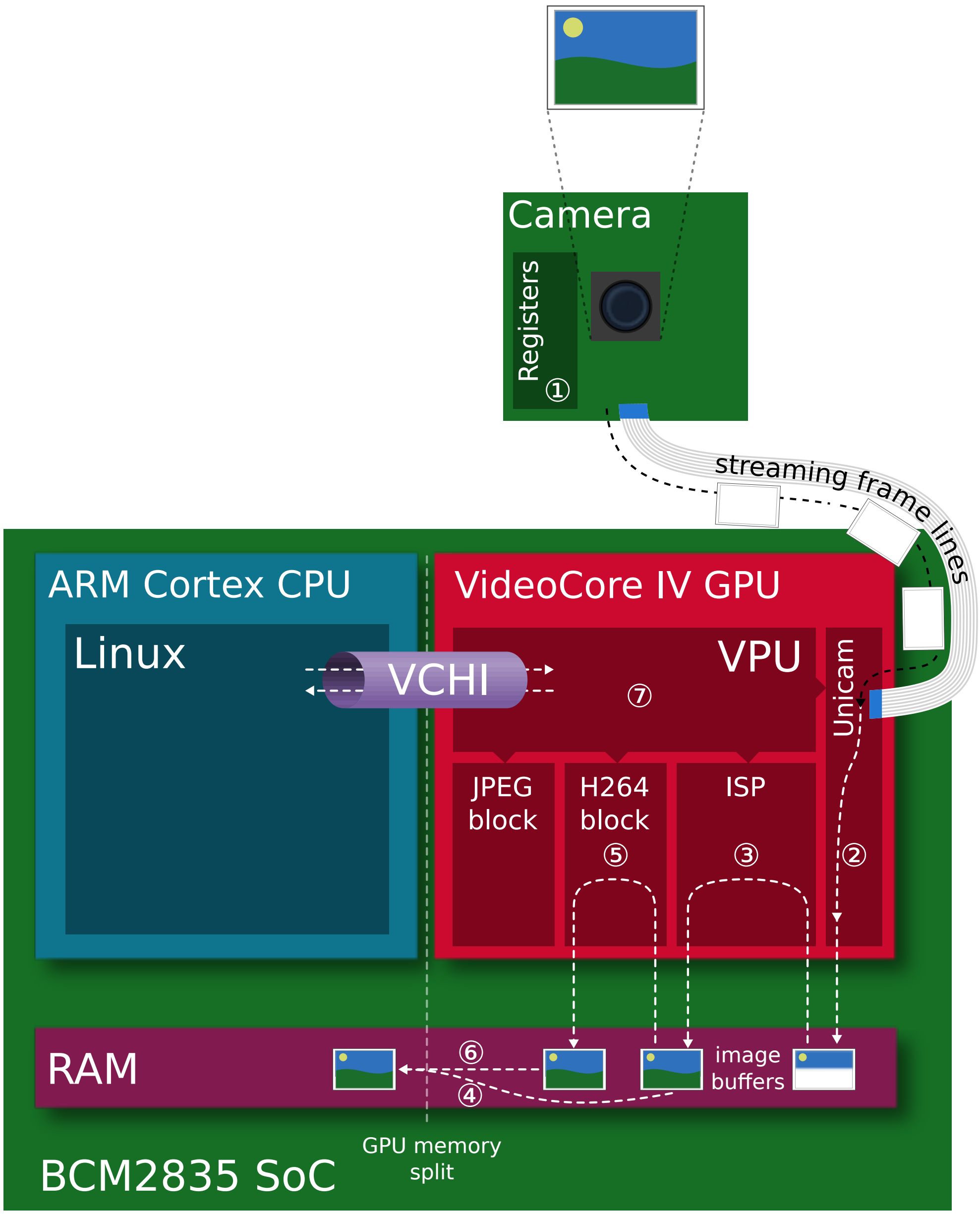
   Some of these steps can be controlled directly (e.g. brightness, noise reduction), others can only be influenced (e.g. analog and digital gain), and the remainder are not user-configurable at all (e.g. demosaic and lens shading).

   At this point the frame is effectively "complete".

4. If you are producing "unencoded" output (YUV, RGB, etc.) the pipeline ends at this point, with the frame data getting copied over to the CPU via DMA[85]. The ISP might be used to convert to RGB, but that's all.

[79] http://www.andor.com/learning-academy/ccd-binning-what-does-binning-mean
[80] https://en.wikipedia.org/wiki/Image_processor
[81] https://en.wikipedia.org/wiki/Color_balance
[82] https://en.wikipedia.org/wiki/Bayer_filter
[83] https://en.wikipedia.org/wiki/YUV#Y.E2.80.B2UV420p_.28and_Y.E2.80.B2V12_or_YV12.29_to_RGB888_conversion
[84] https://en.wikipedia.org/wiki/Fisheye_lens
[85] https://en.wikipedia.org/wiki/Direct_memory_access

5. If you are producing encoded output (H264, MJPEG, MPEG2, etc.) the next step is one of the encoding blocks, the H264 block in this case. The encoding blocks are specialized hardware designed specifically to produce particular encodings. For example, the JPEG block will include hardware for performing lots of parallel discrete cosine transforms[86] (DCTs), while the H264 block will include hardware for performing motion estimation[87].
6. Once encoded, the output is copied to the CPU via DMA[88].
7. Coordinating these components is the VPU, the general purpose component in the GPU running VCOS (ThreadX). The VPU configures and controls the other components in response to messages from VCHI. Currently the most complete documentation of the VPU is available from the videocoreiv repository[89].



Feedback loops

There are a couple of feedback loops running within the pipeline described above. When *exposure_mode* (page 110) is not `'off'`, automatic gain control (AGC) gathers statistics from each frame (prior to the de-mosaic

[86] https://en.wikipedia.org/wiki/Discrete_cosine_transform
[87] https://en.wikipedia.org/wiki/Motion_estimation
[88] https://en.wikipedia.org/wiki/Direct_memory_access
[89] https://github.com/hermanhermitage/videocoreiv

phase in the ISP). It tweaks the analog and digital gains, and the exposure time (line read-out time) attempting to nudge subsequent frames towards a target Y (luminance[90]) value.

Likewise, when `awb_mode` (page 107) is not `'off'`, automatic white balance (AWB) gathers statistics from frames (again, prior to de-mosaic). Typically AWB analysis only occurs on 1 out of every 3 streamed frames as it is computationally expensive. It adjusts the red and blue gains (`awb_gains` (page 106)) attempting to nudge subsequent frames towards the expected color balance[91].

You can observe the effect of the AGC loop quite easily during daylight. Ensure the camera module is pointed at something bright like the sky or the view through a window, and query the camera's analog gain and exposure time:

```
>>> camera = PiCamera()
>>> camera.start_preview(alpha=192)
>>> float(camera.analog_gain)
1.0
>>> camera.exposure_speed
3318
```

Force the camera to use a higher gain by setting `iso` (page 115) to 800. If you have the preview running, you'll see very little difference in the scene. However, if you subsequently query the exposure time you'll find the firmware has drastically reduced it to compensate for the higher sensor gain:

```
>>> camera.iso = 800
>>> camera.exposure_speed
198
```

You can force a longer exposure time with the `shutter_speed` (page 119) attribute at which point the scene will become quite washed out (because both the gain and exposure time are now fixed). If you let the gain float again by setting `iso` (page 115) back to automatic (0) you should find the gain reduces accordingly and the scene returns more or less to normal:

```
>>> camera.shutter_speed = 4000
>>> camera.exposure_speed
3998
>>> camera.iso = 0
>>> float(camera.analog_gain)
1.0
```

The camera's AGC loop attempts to produce a scene with a target Y (luminance[92]) value (or values) within the constraints set by things like ISO, shutter speed, and so forth. The target Y' value can be adjusted with the `exposure_compensation` (page 109) attribute which is measured in increments of 1/6th of an f-stop[93]. So if, whilst the exposure time is fixed, you increase the luminance that the camera is aiming for by a couple of stops, then wait a few seconds you should find that the gain has increased accordingly:

```
>>> camera.exposure_compensation = 12
>>> float(camera.analog_gain)
1.48046875
```

If you allow the exposure time to float once more (by setting `shutter_speed` (page 119) back to 0), then wait a few seconds, you should find the analog gain decreases back to 1.0, but the exposure time increases to maintain the deliberately over-exposed appearance of the scene:

```
>>> camera.shutter_speed = 0
>>> float(camera.analog_gain)
1.0
>>> camera.exposure_speed
4244
```

[90] https://en.wikipedia.org/wiki/Relative_luminance
[91] https://en.wikipedia.org/wiki/Color_balance
[92] https://en.wikipedia.org/wiki/Relative_luminance
[93] https://en.wikipedia.org/wiki/F-number

Sensor Modes

The Pi's camera modules have a discrete set of modes that they can use to output data to the GPU. On the V1 module these are as follows:

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

On the V2 module, these are:

#	Resolution	Aspect Ratio	Framerates	Video	Image	FoV	Binning
1	1920x1080	16:9	1/10 <= fps <= 30	x		Partial	None
2	3280x2464	4:3	1/10 <= fps <= 15	x	x	Full	None
3	3280x2464	4:3	1/10 <= fps <= 15	x	x	Full	None
4	1640x1232	4:3	1/10 <= fps <= 40	x		Full	2x2
5	1640x922	16:9	1/10 <= fps <= 40	x		Full	2x2
6	1280x720	16:9	40 < fps <= 90	x		Partial	2x2
7	640x480	4:3	40 < fps <= 90	x		Partial	2x2

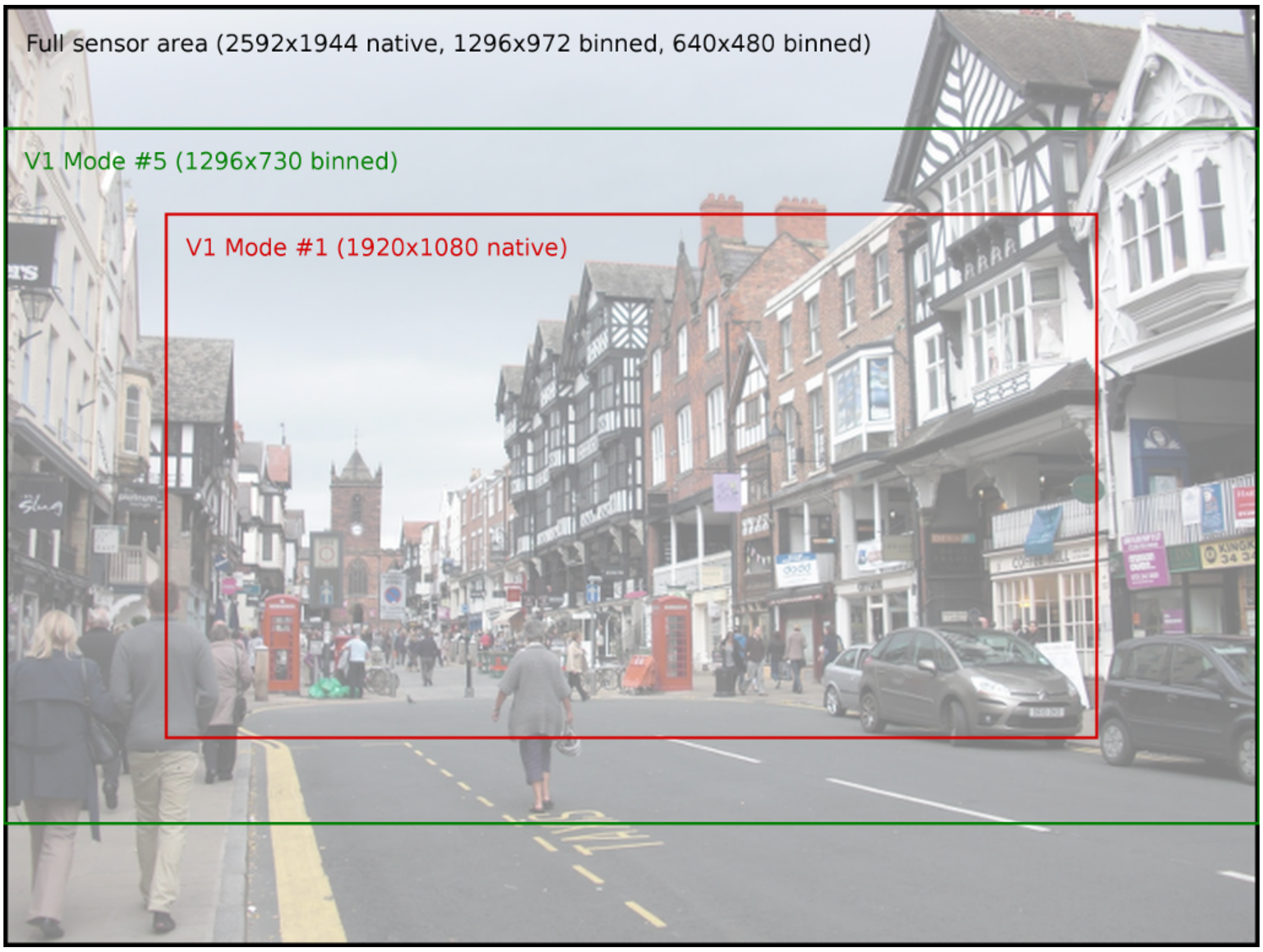
Note: These are *not* the set of possible output resolutions or framerates. These are merely the set of resolutions and framerates that the *sensor* can output directly to the GPU. The GPU's ISP block will resize to any requested resolution (within reason). Read on for details of mode selection.

Note: Sensor mode 3 on the V2 module appears to be a duplicate, but this is deliberate. The sensor modes of the V2 module were designed to mimic the closest equivalent sensor modes of the V1 module. Long exposures on the V1 module required a separate sensor mode; this wasn't required on the V2 module leading to the duplication of mode 2.

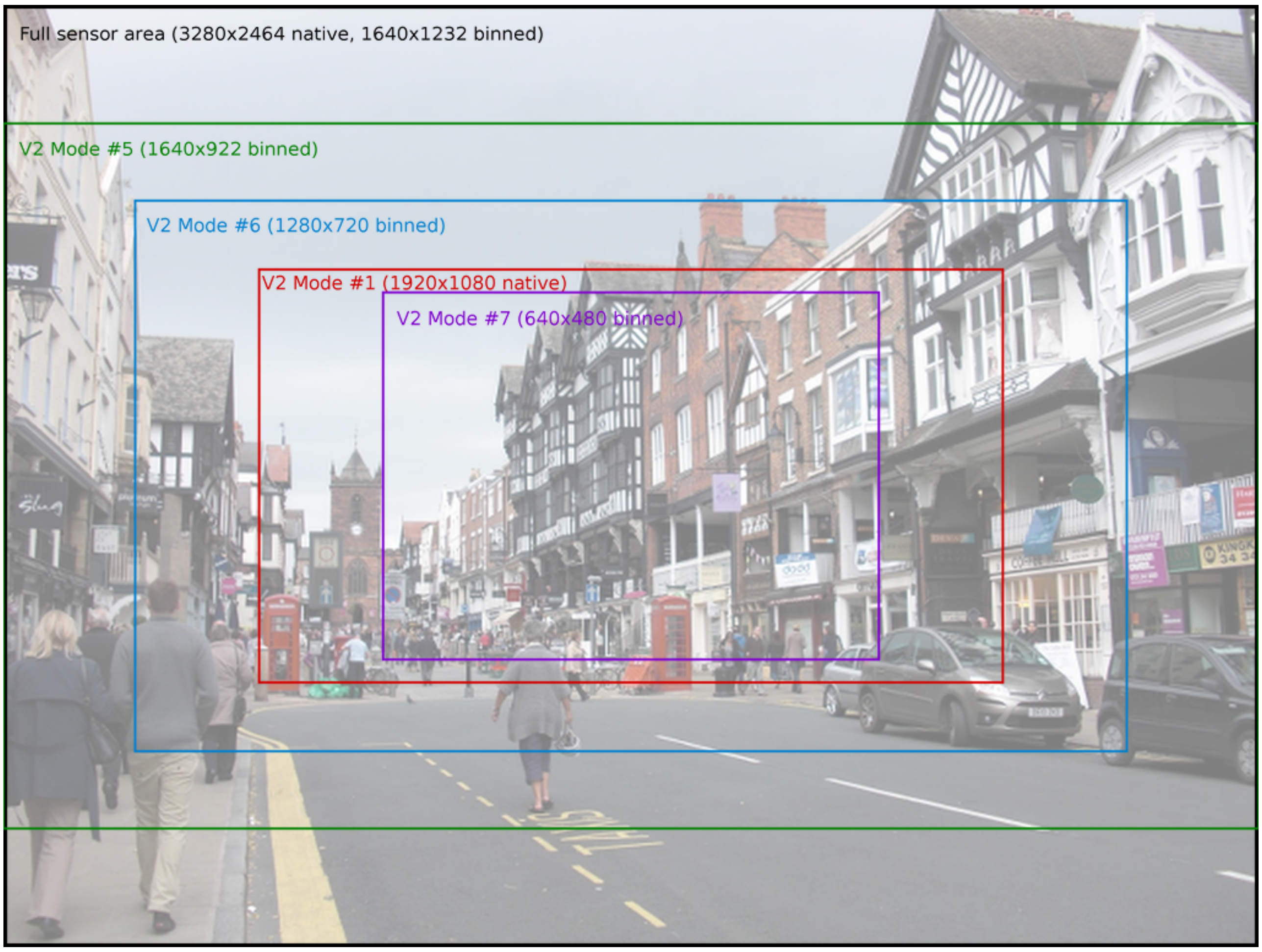
Modes with full field of view[94] (FoV) capture from the whole area of the camera's sensor (2592x1944 pixels for the V1 camera, 3280x2464 for the V2 camera). Modes with partial FoV capture from the center of the sensor. The combination of FoV limiting, and binning[95] is used to achieve the requested resolution.

The image below illustrates the difference between full and partial field of view for the V1 camera:

[94] https://en.wikipedia.org/wiki/Angle_of_view
[95] http://www.andor.com/learning-academy/ccd-binning-what-does-binning-mean


Full sensor area (2592x1944 native, 1296x972 binned, 640x480 binned)
V1 Mode #5 (1296x730 binned)
V1 Mode #1 (1920x1080 native)

While the various fields of view for the V2 camera are illustrated in the following image:


Full sensor area (3280x2464 native, 1640x1232 binned)
V2 Mode #5 (1640x922 binned)
V2 Mode #6 (1280x720 binned)
V2 Mode #1 (1920x1080 native)
V2 Mode #7 (640x480 binned)

The sensor's mode can be forced with the *sensor_mode* parameter in the `PiCamera` (page 95) constructor (using one of the values from the # column in the tables above). This parameter defaults to 0 indicating that the mode should be selected automatically based on the requested `resolution` (page 117) and `framerate` (page 111). The rules governing which sensor mode is selected are as follows:

- The capture mode must be acceptable. All 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` (page 117) is to the mode's resolution the better, but downscaling from a higher sensor resolution to a lower output resolution is preferable to upscaling from a lower sensor resolution.
- The requested `framerate` (page 111) should be within the range of the sensor mode.
- The closer the aspect ratio of the requested `resolution` (page 117) 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 tables above) will choose the mode which maximizes the resulting field of view[96] (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` (page 117) to 1024x768 (a 4:3 aspect ratio), and `framerate` (page 111) to anything less than 42fps, the 1296x972 mode (4) will be selected, and the GPU will downscale the result to 1024x768.
- If you set the `resolution` (page 117) to 1280x720 (a 16:9 wide-screen aspect ratio), and `framerate`

[96] https://en.wikipedia.org/wiki/Angle_of_view

(page 111) to anything less than 49fps, the 1296x730 mode (5) will be selected and downscaled appropriately.

- Setting `resolution` (page 117) to 1920x1080 and `framerate` (page 111) to 30fps exceeds the resolution of both the 1296x730 and 1296x972 modes (i.e. they would require upscaling), so the 1920x1080 mode (1) is selected instead, despite it having a reduced FoV.
- A `resolution` (page 117) of 800x600 and a `framerate` (page 111) 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).

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 (this is a limit of the H264 block in the GPU). Any attempt to record H264 video at higher horizontal resolutions will fail.
- The maximum resolution of the V2 camera may require additional GPU memory when operating at low framerates (<1fps). Increase `gpu_mem` in `/boot/config.txt` if you encounter "out of resources" errors when attempting long-exposure captures with a V2 module.
- The maximum resolution of the V2 camera can also 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, or configure the preview to use a lower `resolution` (page 132) than the camera itself.
- 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` (page 111) before setting `shutter_speed` (page 119).

MMAL

The MMAL layer below picamera provides a greatly simplified interface to the camera firmware running on the GPU. Conceptually, it 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 behaviour.

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 *Sensor Modes* (page 75)) so that images are captured using the full area of the sensor. It also uses a strong noise reduction algorithm on captured images so that they appear higher quality.

The still port is used by the various `capture()` (page 97) 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()` (page 103) method (for recording video), and is also used by the various `capture()` (page 97) 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 the stronger noise reduction algorithm).

The Preview Port

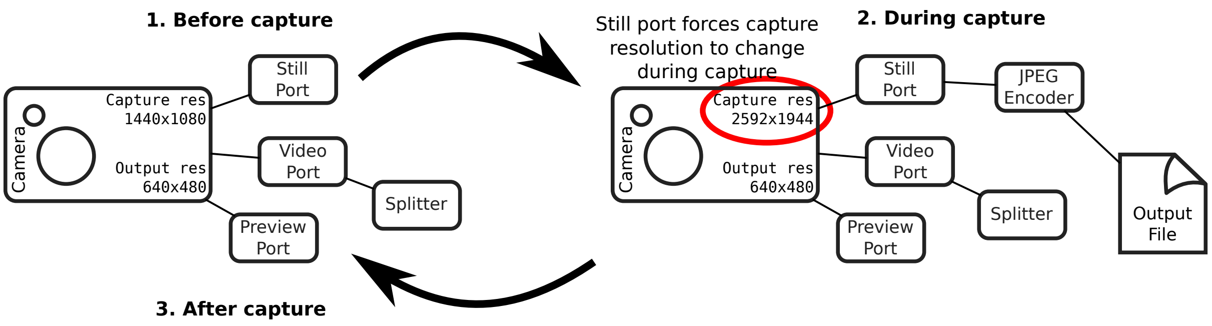
The preview port operates more or less identically to the video port. The preview port is always connected to some form of output to ensure that the auto-gain algorithm can run. When an instance of `PiCamera` (page 95) is constructed, the preview port is initially connected to an instance of `PiNullSink` (page 132). When `start_preview()` (page 103) is called, this null sink is destroyed and the preview port is connected to an instance of `PiPreviewRenderer` (page 132). The reverse occurs when `stop_preview()` (page 104) is called.

Pipelines

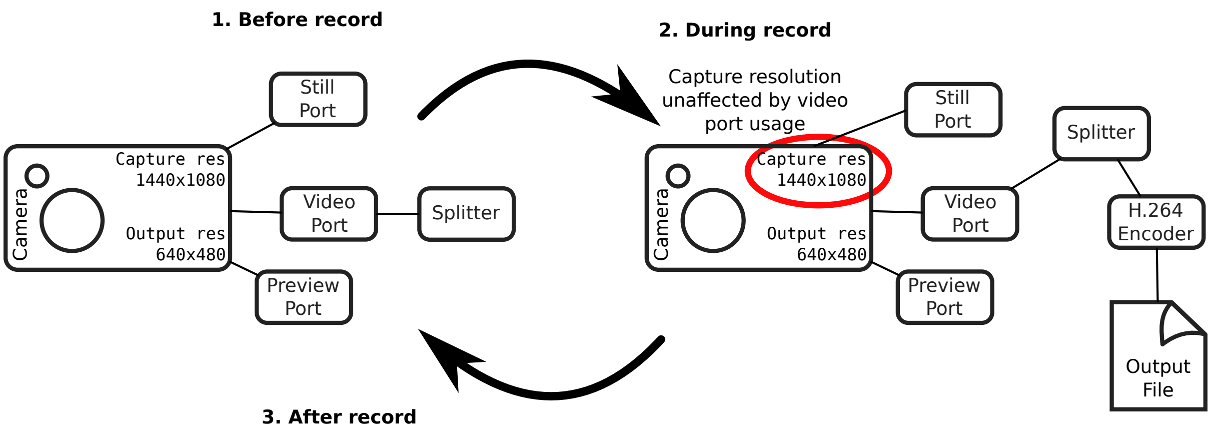
This section attempts to provide detail of what MMAL pipelines picamera constructs in response to various method calls.

The firmware 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).

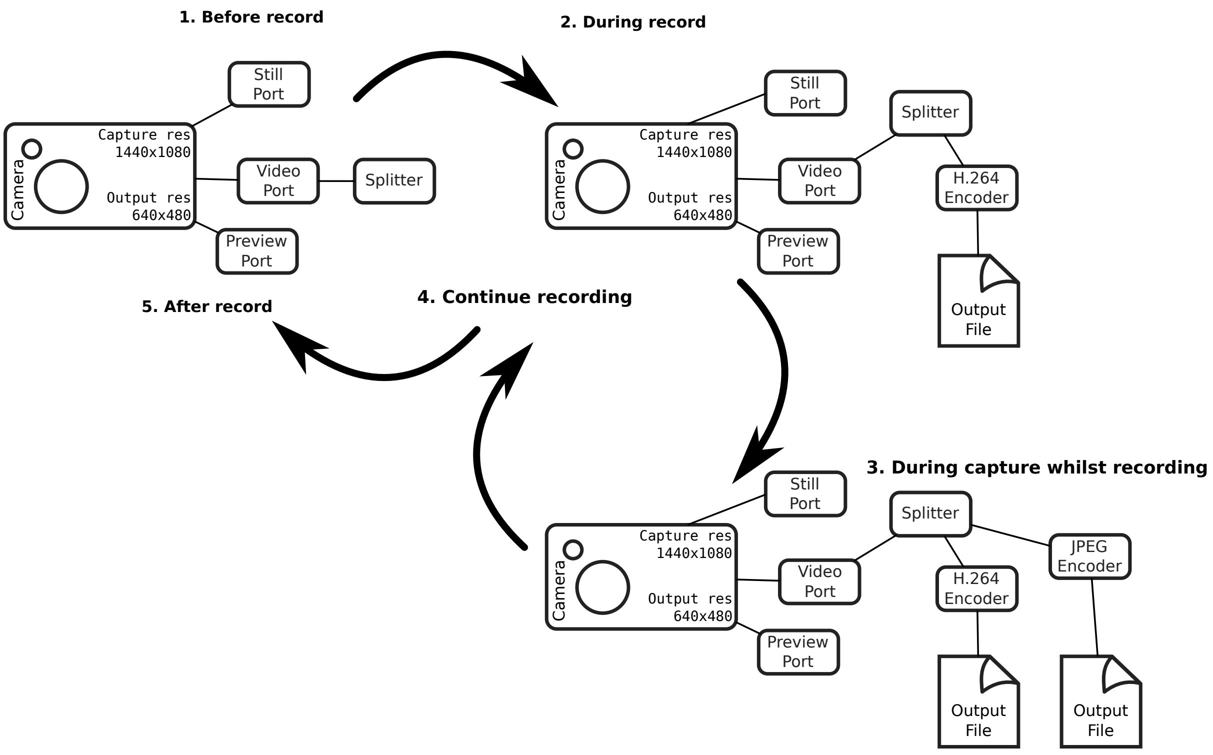
Encoders are connected directly to the still port. For example, when capturing a picture using the still port, the camera's state conceptually moves through these states:



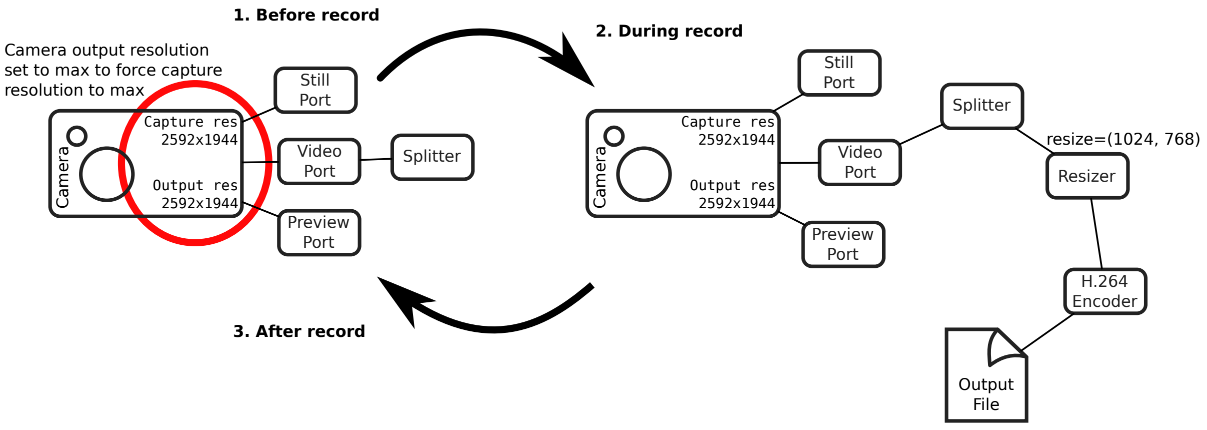
As you have probably noticed in the diagram above, the video port is a little more complex. In order to permit simultaneous video recording and image capture via the video port, a "splitter" component is permanently connected to the video port by picamera, and encoders are in turn attached to one of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



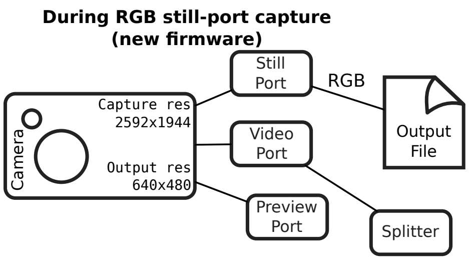
When the `resize` parameter is passed to one of the aforementioned methods, a resizer component is placed between the camera's ports and the encoder, causing the output to be resized before it reaches the encoder. This is particularly useful for video recording, as the H.264 encoder cannot cope with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

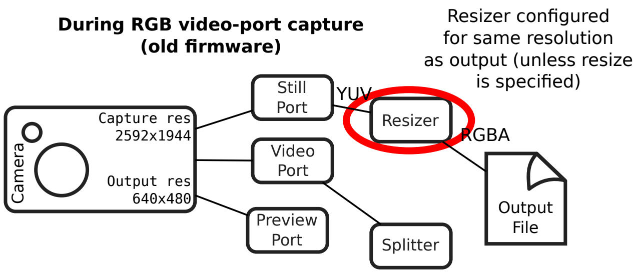


Finally, when performing unencoded captures an encoder is (naturally) not required. Instead data is taken directly from the camera's ports. However, various firmware limitations require acrobatics in the pipeline to achieve requested encodings.

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

The ports used to connect MMAL components together pass image data around in particular encodings. Often, this is the YUV420[97] encoding (this is the "preferred" internal format for the pipeline). On rare occasions, it is RGB[98] (RGB is a large and rather inefficient format). However, another format sometimes used is the "OPAQUE" encoding.

"OPAQUE" 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 (as such it's not really an encoding at all, but it's treated as such by the MMAL framework). 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 VPU resizer (`MMALResizer` (page 175)) 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, and BGRA output, not RGB or BGR.
- The ISP resizer (`MMALISPResizer` (page 175), not currently used by picamera's high level API, but available from the `mmalobj` (page 159) layer) accepts OPAQUE input, and will produce almost any unencoded output (including YUV, RGB, BGR, RGBA, and BGRA) but not OPAQUE.

The `mmalobj` (page 159) layer introduced in picamera 1.11 is aware of these OPAQUE encoding differences and attempts to configure connections between components using 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 the description above is MMAL's greatly simplified presentation of the imaging pipeline. This is far removed from what actually happens at the GPU's ISP level (described roughly in earlier sections). However, as MMAL is the API under-pinning the picamera library (along with the official `raspistill` and `raspivid` applications) it is worth understanding.

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.

[97] https://en.wikipedia.org/wiki/YUV#Y.E2.80.B2UV420p_.28and_Y.E2.80.B2V12_or_YV12.29_to_RGB888_conversion
[98] https://en.wikipedia.org/wiki/RGB

CHAPTER 7

Development

The main GitHub repository for the project can be found at:

https://github.com/waveform80/picamera

Anyone is more than welcome to open tickets to discuss bugs, new features, or just to ask usage questions (I find this useful for gauging what questions ought to feature in the FAQ, for example).

For anybody wishing to hack on the project, I would strongly recommend reading through the *PiCamera* (page 95) class' source, to get a handle on using the *mmalobj* (page 159) 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* (page 159) is a `ctypes`[99] 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* (page 125), *PiBayerArray* (page 153), 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)!

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 virtualenvwrapper python-virtualenv python3-virtualenv \
>   python-dev python3-dev libjpeg8-dev zlib1g-dev libav-tools
$ cd
$ mkvirtualenv -p /usr/bin/python3 picamera
$ workon picamera
(picamera) $ git clone https://github.com/waveform80/picamera.git
```

[99] https://docs.python.org/3.4/library/ctypes.html#module-ctypes

```
(picamera) $ cd picamera
(picamera) $ make develop
```

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

```
$ workon picamera
(picamera) $ cd ~/picamera
(picamera) $ git pull
(picamera) $ make develop
```

To remove your installation, destroy the sandbox and the clone:

```
(picamera) $ deactivate
$ rmvirtualenv picamera
$ rm -fr ~/picamera
```

Building the docs

If you wish to build the docs, you'll need a few more dependencies. Inkscape is used for conversion of SVGs to other formats, Graphviz is used for rendering certain charts, and TeX Live is required for building PDF output. The following command should install all required dependencies:

```
$ sudo apt-get install texlive-latex-recommended texlive-latex-extra \
    texlive-fonts-recommended graphviz inkscape
```

Once these are installed, you can use the "doc" target to build the documentation:

```
$ workon picamera
(picamera) $ cd ~/picamera
(picamera) $ make doc
```

The HTML output is written to `docs/_build/html` while the PDF output goes to `docs/_build/latex`.

Test suite

If you wish to run the picamera test suite, follow the instructions in *Development installation* (page 83) above and then make the "test" target within the sandbox:

```
$ workon picamera
(picamera) $ cd ~/picamera
(picamera) $ 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.

CHAPTER 8

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()` (page 103) and `stop_recording()` (page 104)). 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[100]. 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` (page 117) for example). However, Python also provides excellent methods for determining automatically whether any deprecated functionality is being used via the `warnings`[101] module.

Finding and fixing deprecated usage

As of release 1.8, all deprecated functionality will raise `DeprecationWarning`[102] 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:

[100] http://semver.org/
[101] https://docs.python.org/3.4/library/warnings.html#module-warnings
[102] https://docs.python.org/3.4/library/exceptions.html#DeprecationWarning

```
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`[103]; "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 tuple is deprecated; please use one of Python's many numeric␣
↪classes like int, float, Decimal, or Fraction instead
  "Setting framerate or gains as a tuple is deprecated; "
/usr/share/pyshared/picamera/camera.py:3125: DeprecationWarning: PiCamera.preview_
↪fullscreen is deprecated; use PiCamera.preview.fullscreen instead
  'PiCamera.preview_fullscreen is deprecated; '
/usr/share/pyshared/picamera/camera.py:3068: DeprecationWarning: PiCamera.preview_
↪alpha is deprecated; use PiCamera.preview.alpha instead
  'PiCamera.preview_alpha is deprecated; use '
/usr/share/pyshared/picamera/camera.py:1833: DeprecationWarning: PiCamera.raw_
↪format is deprecated; use required format directly with capture methods instead
  'PiCamera.raw_format is deprecated; use required format '
/usr/share/pyshared/picamera/camera.py:1359: DeprecationWarning: The "raw" format␣
↪option is deprecated; specify the required format directly instead ("yuv", "rgb",
↪ etc.)
  'The "raw" format option is deprecated; specify the '
/usr/share/pyshared/picamera/camera.py:1827: DeprecationWarning: PiCamera.raw_
↪format is deprecated; use required format directly with capture methods instead
```

[103] https://docs.python.org/3.4/library/exceptions.html#DeprecationWarning

```
'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`[104] 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's many numeric classes like int, float, Decimal, or Fraction␣
↪instead
```

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 instead
```

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)

```

[104] https://docs.python.org/3.4/library/exceptions.html#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).

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()`* (page 97) method, with the *`raw_format`* (page 117) 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`* (page 117) and place the required format directly within the *`capture()`* (page 97) call:

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

Recording quality

The *quantization* parameter for *`start_recording()`* (page 103) and *`record_sequence()`* (page 101) 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[105], float[106], `Decimal`[107], and `Fraction`[108]. 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`[109] 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`* (page 111) attribute as a tuple:

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

In such cases, use the standard `numerator`[110] and `denominator`[111] 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`[112] instance to a float[113] 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`* (page 95) class. Instead, it is represented by the *`preview`* (page 117) attribute which is a separate *`PiPreviewRenderer`* (page 132) instance when the preview is active.

This change meant that *`preview_alpha`* (page 117) was deprecated in favor of the *`alpha`* (page 129) property of the new *`preview`* (page 117) attribute. Similar changes were made to *`preview_layer`* (page 117), *`preview_fullscreen`* (page 117), and *`preview_window`* (page 117). The following snippet illustrates the deprecated method of setting preview related attributes:

[105] https://docs.python.org/3.4/library/stdtypes.html#typesnumeric
[106] https://docs.python.org/3.4/library/stdtypes.html#typesnumeric
[107] https://docs.python.org/3.4/library/decimal.html#decimal.Decimal
[108] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction
[109] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction
[110] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction.numerator
[111] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction.denominator
[112] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction
[113] https://docs.python.org/3.4/library/stdtypes.html#typesnumeric

```
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` (page 117) attribute is `None` when the preview is not active. To accomodate this use-case, optional parameters were added to `start_preview()` (page 103) 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()` (page 103) method instead:

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

Finally, the `previewing` (page 117) attribute is now obsolete (and thus deprecated) as its functionality can be trivially obtained by checking the `preview` (page 117) 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` (page 117) with `preview` (page 117) 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` (page 151) class was changed to derive from `io.BytesIO`[114] 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()` (page 151) 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:

[114] https://docs.python.org/3.4/library/io.html#io.BytesIO

```
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`* (page 108) attribute was renamed to *`zoom`* (page 120); 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`* (page 108) with *`zoom`* (page 120) in such cases:

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

Incorrect ISO capitalisation

In release 1.8, the *`ISO`* (page 105) attribute was renamed to *`iso`* (page 115) for compliance with PEP-8[115] (even though it's an acronym this is still more consistent with the existing API; consider *`led`* (page 116), *`awb_mode`* (page 107), and so on).

The following example illustrates the deprecated attribute case:

```
camera.ISO = 100
```

Simply replace references to *`ISO`* (page 105) with *`iso`* (page 115):

[115] http://legacy.python.org/dev/peps/pep-0008/

```
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* (page 121) named tuple, release 1.5 introduced the *PiVideoFrameType* (page 120) enumeration used by the *frame_type* (page 121) attribute and deprecated the *keyframe* (page 122) and *header* (page 121) 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* (page 121) attribute against the corresponding value of the *PiVideoFrameType* (page 120) 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`[116] 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* (page 105) 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* (page 145) 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* (page 145) 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
```

[116] https://docs.python.org/3.4/library/exceptions.html#KeyError

In such cases, replace `False` with `None`, and `True` with a *Color* (page 145) 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* (page 151) were adjusted in 1.11 to use *analyze()* (page 155) (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)
```

CHAPTER 9

API - The PiCamera Class

The picamera library contains numerous classes, but the primary one that all users are likely to interact with is `PiCamera` (page 95), documented below. With the exception of the contents of the `picamera.array` (page 151) 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` (page 151)):

```
import picamera
```

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', framerate_range=None*)

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*, *framerate_range*, and *clock_mode* parameters provide initial values for the `sensor_mode` (page 118), `resolution` (page 117), `framerate` (page 111), `framerate_range` (page 113), and `clock_mode` (page 107) 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` (page 95) 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` (page 116) 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[117] platform).

No preview or recording is started automatically upon construction. Use the `capture()` (page 97) method to capture images, the `start_recording()` (page 103) method to begin recording video, or the `start_preview()` (page 103) 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` (page 107). Others, like `resolution` (page 117), can only be adjusted when the camera is idle.

When you are finished with the camera, you should ensure you call the `close()` (page 101) 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`[118] statement, the `close()` (page 101) 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.

Changed in version 1.13: Added *framerate_range* parameter.

add_overlay(*source*, *size=None*, *format=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()` (page 96) returns a new `PiOverlayRenderer` (page 131) instance representing the overlay.

The *source* must be an object that supports the buffer protocol[119] in one of the supported unencoded formats: `'yuv'`, `'rgb'`, `'rgba'`, `'bgr'`, or `'bgra'`. The format can specified explicitly with the optional *format* parameter. If not specified, the method will attempt to guess the format based on the length of *source* and the *size* (assuming 3 bytes per pixel for RGB, and 4 bytes for RGBA).

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` (page 117).

The length of *source* must take into account that widths are rounded up to the nearest multiple of 32, and heights to the nearest multiple of 16. For example, if *size* is `(1280, 720)`, and *format* is `'rgb'`, 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)`, and *format* is `'rgb'` 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).

[117] https://www.raspberrypi.org/documentation/hardware/computemodule/cmio-camera.md
[118] https://docs.python.org/3.4/reference/compound_stmts.html#with
[119] https://docs.python.org/3.4/c-api/buffer.html#bufferobjects

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` (page 129) property) or by moving the overlay into a layer higher than the preview (with the `layer` (page 130) property).

All keyword arguments captured in *options* are passed onto the `PiRenderer` (page 129) constructor. All camera properties except `resolution` (page 117) and `framerate` (page 111) 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.

Changed in version 1.13: Added *format* parameter

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` (page 142) 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` (page 117) 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 *MMAL* (page 78) 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[120] vary widely.
- *restart* - Defines the restart interval for the JPEG encoder as a number of JPEG MCUs. The actual restart interval used will be a multiple of the number of MCUs per row in the resulting image.
- *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()`[121] method. Those two values are:

- `{counter}` - a simple incrementor that starts at 1 and increases by 1 for each image taken
- `{timestamp}` - a `datetime`[122] instance

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

[120] http://photo.net/learn/jpeg/#qual
[121] https://docs.python.org/3.4/library/stdtypes.html#str.format
[122] https://docs.python.org/3.4/library/datetime.html#datetime.datetime

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	
`'{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()` (page 97).

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()` (page 97), 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` (page 95) object, you should ensure you call the `close()` (page 101) 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()` (page 103), 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()`[123]. 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:
```

[123] https://docs.python.org/3.4/library/itertools.html#itertools.cycle

```
    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()` (page 96). The *overlay* parameter specifies the `PiRenderer` (page 129) instance that was returned by `add_overlay()` (page 96).

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()` (page 103) 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()` (page 103). 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()` (page 103), 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()` (page 103) 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` (page 129) instance (more specifically, a `PiPreviewRenderer` (page 132)) is constructed with the keyword arguments captured in *options*, and is returned from the method (this instance is also accessible from the `preview` (page 117) 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()` (page 104). The preview can be started and stopped multiple times during the lifetime of the `PiCamera` (page 95) object.

All other camera properties can be modified "live" while the preview is running (e.g. `brightness` (page 107)).

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` (page 142) 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', 'extended', 'high', or 'constrained'.
- *level* - The H.264 level[124] to use for encoding. Defaults to '4', but can be any H.264 level up to '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()` (page 102) 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()` (page 102). 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.
- *sps_timing* - When `True` the encoder includes the camera's framerate in the SPS header. 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 depends on the selected H.264 level[125] and profile. 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()` (page 103) 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.

[124] https://en.wikipedia.org/wiki/H.264/MPEG-4_AVC#Levels
[125] https://en.wikipedia.org/wiki/H.264/MPEG-4_AVC#Levels

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()` (page 103). If an error occurred during recording and `wait_recording()` (page 105) 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()` (page 105) or `stop_recording()` (page 104) 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` (page 115) 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`[126] instance.

New in version 1.6.

annotate_background

Controls what background is drawn behind the annotation.

The `annotate_background` (page 105) attribute specifies if a background will be drawn behind the `annotation text` (page 106) and, if so, what color it will be. The value is specified as a `Color` (page 145) 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.

[126] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction

The *annotate_foreground* (page 105) attribute specifies, partially, the color of the annotation text. The value is specified as a *Color* (page 145). 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[127] 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* (page 145) 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* (page 106) attribute is a bool indicating whether or not the current frame number is rendered as an annotation, similar to *annotate_text* (page 106). 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* (page 106) 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* (page 142). 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* (page 106) 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`[128] 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[129], float[130] or `Fraction`[131] 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.

[127] https://en.wikipedia.org/wiki/YUV
[128] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction
[129] https://docs.python.org/3.4/library/stdtypes.html#typesnumeric
[130] https://docs.python.org/3.4/library/stdtypes.html#typesnumeric
[131] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction

Note: This attribute only has an effect when *awb_mode* (page 107) is set to `'off'`. Also note that even with AWB disabled, some attributes (specifically *still_stats* (page 119) and *drc_strength* (page 108)) can cause AWB re-calculations.

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* (page 107) 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* (page 106) property. However, even with AWB disabled, some attributes (specifically *still_stats* (page 119) and *drc_strength* (page 108)) can cause AWB re-calculations.

brightness

Retrieves or sets the brightness setting of the camera.

When queried, the *brightness* (page 107) 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* (page 111) 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* (page 95) constructor, and will default to "reset" if not specified.

New in version 1.11.

closed

Returns `True` if the *close()* (page 101) method has been called.

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

When queried, the *color_effects* (page 107) 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* (page 108) 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* (page 120) 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`[132] instance.

New in version 1.6.

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

When queried, the *drc_strength* (page 108) property returns a string indicating the amount of dynamic range compression[133] 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.

Warning: Enabling DRC will override fixed white balance[134] gains (set via *awb_gains* (page 106) and *awb_mode* (page 107)).

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.

[132] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction
[133] https://en.wikipedia.org/wiki/Gain_compression
[134] https://www.raspberrypi.org/forums/viewtopic.php?p=875772&sid=92fa4ea70d1fe24590a4cdfb4a10c489#p875772

By default several Exif tags are automatically applied to any images taken with the *capture()* (page 97) 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()* (page 97). 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()* (page 97) 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* (page 109) 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* (page 110) 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* (page 108) and *analog_gain* (page 105).

Please note that these properties are not directly settable (although they can be influenced by setting *iso* (page 115) *prior* to fixing the gains), 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* (page 119) to a non-zero value, then *exposure_speed* (page 110) and *shutter_speed* (page 119) should be equal. However, if *shutter_speed* (page 119) 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* (page 110) 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[135] 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* (page 52).

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()`* (page 103)), this attribute will return a *`PiVideoFrame`* (page 121) 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()`* (page 103) 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()`* (page 103). 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`* (page 111) property returns the rate at which the camera's video and preview ports will operate as a `Fraction`[136] instance (which can be easily converted to an `int`[137] or `float`[138]). If *`framerate_range`* (page 113) has been set, then *`framerate`* (page 111) will be 0 which indicates that a dynamic range of framerates is being used.

Note: For backwards compatibility, a derivative of the `Fraction`[139] 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`[140] instance instead (which is just as accurate and also permits direct use with math operators).

[135] https://www.raspberrypi.org/documentation/configuration/pin-configuration.md
[136] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction
[137] https://docs.python.org/3.4/library/functions.html#int
[138] https://docs.python.org/3.4/library/functions.html#float
[139] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction
[140] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction

When set, the property configures the camera so that the next call to recording and previewing methods will use the new framerate. Setting this property implicitly sets `framerate_range` (page 113) so that the low and high values are equal to the new framerate. The framerate can be specified as an int[141], float[142], `Fraction`[143], 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` (page 117), 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` (page 118) for more information.

The initial value of this property can be specified with the *framerate* parameter in the `PiCamera` (page 95) 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` (page 112) property returns the amount that the camera's `framerate` (page 111) 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` (page 111) + `framerate_delta` (page 112).

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[144], float[145], `Fraction`[146] 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 implicitly reset to 0 when `framerate` (page 111) or `framerate_range`

[141] https://docs.python.org/3.4/library/stdtypes.html#typesnumeric
[142] https://docs.python.org/3.4/library/stdtypes.html#typesnumeric
[143] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction
[144] https://docs.python.org/3.4/library/stdtypes.html#typesnumeric
[145] https://docs.python.org/3.4/library/stdtypes.html#typesnumeric
[146] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction

(page 113) is set. When *framerate* (page 111) is 0 (indicating that *framerate_range* (page 113) is set), this property cannot be used. (there would be little point in making fractional adjustments to the framerate when the framerate itself is variable).

New in version 1.11.

framerate_range

Retrieves or sets a range between which the camera's framerate is allowed to float.

When queried, the *framerate_range* (page 113) property returns a `namedtuple()`[147] derivative with `low` and `high` components (index 0 and 1 respectively) which specify the limits of the permitted framerate range.

When set, the property configures the camera so that the next call to recording and previewing methods will use the new framerate range. Setting this property will implicitly set the *framerate* (page 111) property to 0 (indicating that a dynamic range of framerates is in use by the camera).

Note: Use of this property prevents use of *framerate_delta* (page 112) (there would be little point in making fractional adjustments to the framerate when the framerate itself is variable).

The low and high framerates can be specified as int[148], float[149], or `Fraction`[150] values. For example, the following definitions are all equivalent:

```
from fractions import Fraction

camera.framerate_range = (0.16666, 30)
camera.framerate_range = (Fraction(1, 6), 30 / 1)
camera.framerate_range = (Fraction(1, 6), Fraction(30, 1))
```

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

Note: This attribute, like *framerate* (page 111), 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* (page 118) for more information.

New in version 1.13.

hflip

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

When queried, the *hflip* (page 113) 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* (page 113) 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.

[147] https://docs.python.org/3.4/library/collections.html#collections.namedtuple
[148] https://docs.python.org/3.4/library/stdtypes.html#typesnumeric
[149] https://docs.python.org/3.4/library/stdtypes.html#typesnumeric
[150] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction

When queried, the `image_effect` (page 113) 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` (page 113).

When queried, the `image_effect_params` (page 114) 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` (page 113) 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` (page 142) 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` (page 115) property returns the ISO setting of the camera, a value which represents the sensitivity of the camera to light[151]. 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 (by adjusting the `analog_gain` (page 105) and `digital_gain` (page 108)). 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.

On the V1 camera module, non-zero ISO values attempt to fix overall gain at various levels. For example, ISO 100 attempts to provide an overall gain of 1.0, ISO 200 attempts to provide overall gain of 2.0, etc. The algorithm prefers analog gain over digital gain to reduce noise.

On the V2 camera module, ISO 100 attempts to produce overall gain of ~1.84, and ISO 800 attempts to produce overall gain of ~14.72 (the V2 camera module was calibrated against the ISO film speed[152] standard).

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: Some users on the Pi camera forum have noted that higher ISO values than 800 (specifically up

[151] https://en.wikipedia.org/wiki/Film_speed#Digital
[152] https://en.wikipedia.org/wiki/Film_speed#Current_system:_ISO

to 1600) can be achieved in certain conditions with *exposure_mode* (page 110) set to 'sports' and *iso* (page 115) 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.

Note: Certain *exposure_mode* (page 110) values override the ISO setting. For example, 'off' fixes *analog_gain* (page 105) and *digital_gain* (page 108) entirely, preventing this property from adjusting them when set.

---

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[153] (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* (page 116) property returns the method by which the camera determines the exposure[154] 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[155] 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* (page 129) overlays.

[153] https://www.raspberrypi.org/documentation/configuration/config-txt.md
[154] https://en.wikipedia.org/wiki/Metering_mode
[155] https://www.raspberrypi.org/forums/viewtopic.php?p=565644#p565644

If no overlays are current active, `overlays` (page 116) will return an empty iterable. Otherwise, it will return an iterable of `PiRenderer` (page 129) 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` (page 129).

New in version 1.8.

preview

Retrieves the `PiRenderer` (page 129) displaying the camera preview.

If no preview is currently active, `preview` (page 117) will return `None`. Otherwise, it will return the instance of `PiRenderer` (page 129) which is currently connected to the camera's preview port for rendering what the camera sees. You can use the attributes of the `PiRenderer` (page 129) 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` (page 129) attribute of the `preview` (page 117) object instead.

preview_fullscreen

Retrieves or sets full-screen for the preview window.

Deprecated since version 1.8: Please use the `fullscreen` (page 130) attribute of the `preview` (page 117) object instead.

preview_layer

Retrieves or sets the layer of the preview window.

Deprecated since version 1.8: Please use the `layer` (page 130) attribute of the `preview` (page 117) object instead.

preview_window

Retrieves or sets the size of the preview window.

Deprecated since version 1.8: Please use the `window` (page 131) attribute of the `preview` (page 117) object instead.

previewing

Returns `True` if the `start_preview()` (page 103) method has been called, and no `stop_preview()` (page 104) call has been made yet.

Deprecated since version 1.8: Test whether `preview` (page 117) 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()` (page 103) method has been called, and no `stop_recording()` (page 104) 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` (page 117) 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()` (page 97) method will produce images at, and the resolution that `start_recording()` (page 103) 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[156] 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` (page 111), 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` (page 118) for more information.

The initial value of this property can be specified with the *resolution* parameter in the `PiCamera` (page 95) 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.

`revision`

Returns a string representing the revision of the Pi's camera module. At the time of writing, the string returned is 'ov5647' for the V1 module, and 'imx219' for the V2 module.

`rotation`

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

When queried, the `rotation` (page 118) 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` (page 118) 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` (page 117) and `framerate` (page 111). 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 *Sensor Modes* (page 75) 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

[156] https://en.wikipedia.org/wiki/Graphics_display_resolution

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` (page 95) 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` (page 119) 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` (page 119) 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` (page 110) 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` (page 111) 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` (page 119) 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()` (page 97) 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.

Warning: Enabling the still statistics pass will override fixed white balance[157] gains (set via `awb_gains` (page 106) and `awb_mode` (page 107)).

New in version 1.9.

[157] https://www.raspberrypi.org/forums/viewtopic.php?p=875772&sid=92fa4ea70d1fe24590a4cdfb4a10c489#p875772

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* (page 107) is `'raw'` the values returned by this attribute are comparable to those from the *frame* (page 111) *timestamp* (page 121) attribute.

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

When queried, the *vflip* (page 120) 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* (page 120) 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* (page 120) 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[158], not rotation.

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

When queried, the *zoom* (page 120) 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.

PiVideoFrameType

class picamera.**PiVideoFrameType**
This class simply defines constants used to represent the type of a frame in *PiVideoFrame.frame_type* (page 121). 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.

[158] https://www.raspberrypi.org/forums/viewtopic.php?p=342667&sid=ec7d95e887ab74a90ffaab87888c48cd#p342667

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

New in version 1.5.

PiVideoFrame

class picamera.**PiVideoFrame**(*index*, *frame_type*, *frame_size*, *video_size*, *split_size*, *timestamp*)
This class is a `namedtuple()`[159] 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* (page 120)). 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* (page 121) remains static. Query *complete* (page 121) to determine whether the frame has been completely output yet.

video_size
Returns the size in bytes of the entire video up to this 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()* (page 103) or *split_recording()* (page 102). 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 frame. This represents the point in time that the Pi received the first line of the frame from the camera.

The timestamp is measured in microseconds (millionths of a second). When the camera's clock mode is `'reset'` (the default), the timestamp is relative to the start of the video recording. When the camera's *clock_mode* (page 107) is `'raw'`, it is relative to the last system reboot. See *timestamp* (page 119) for more information.

> **Warning:** Currently, the camera 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. This happens for SPS header "frames", for example.

complete
Returns a bool indicating whether the current frame is complete or not. If the frame is complete then *frame_size* (page 121) will not increment any further, and will reset for the next frame.

Changed in version 1.5: Deprecated *header* (page 121) and *keyframe* (page 122) attributes and added the new *frame_type* (page 121) attribute instead.

Changed in version 1.9: Added the *complete* (page 121) attribute.

[159] https://docs.python.org/3.4/library/collections.html#collections.namedtuple

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* (page 121) to *PiVideoFrameType.sps_header* (page 120) 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* (page 121) to *PiVideoFrameType.key_frame* (page 120) instead.

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

PiResolution

class picamera.**PiResolution**(*width*, *height*)
A namedtuple()[160] derivative which represents a resolution with a *width* (page 122) and *height* (page 122).

width
The width of the resolution in pixels

height
The height of the resolution in pixels

New in version 1.11.

pad(*width=32*, *height=16*)
Returns the resolution padded up to the nearest multiple of *width* and *height* which default to 32 and 16 respectively (the camera's native block size for most operations). For example:

```
>>> PiResolution(1920, 1080).pad()
PiResolution(width=1920, height=1088)
>>> PiResolution(100, 100).pad(16, 16)
PiResolution(width=128, height=112)
>>> PiResolution(100, 100).pad(16, 16)
PiResolution(width=112, height=112)
```

transpose()
Returns the resolution with the width and height transposed. For example:

```
>>> PiResolution(1920, 1080).transpose()
PiResolution(width=1080, height=1920)
```

PiFramerateRange

class picamera.**PiFramerateRange**(*low*, *high*)
This class is a namedtuple()[161] derivative used to store the low and high limits of a range of framerates. It is recommended that you access the information stored by this class by attribute rather than position (for example: `camera.framerate_range.low` rather than `camera.framerate_range[0]`).

[160] https://docs.python.org/3.4/library/collections.html#collections.namedtuple
[161] https://docs.python.org/3.4/library/collections.html#collections.namedtuple

`low`

The lowest framerate that the camera is permitted to use (inclusive). When the *`framerate_range`* (page 113) attribute is queried, this value will always be returned as a `Fraction`[162].

`high`

The highest framerate that the camera is permitted to use (inclusive). When the *`framerate_range`* (page 113) attribute is queried, this value will always be returned as a `Fraction`[163].

New in version 1.13.

[162] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction
[163] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction

CHAPTER 10

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).

PiCameraCircularIO

class `picamera.`**`PiCameraCircularIO`**`(`*camera*, *size=None*, *seconds=None*, *bitrate=17000000*, *splitter_port=1*`)`

A derivative of `CircularIO` (page 126) which tracks camera frames.

PiCameraCircularIO provides an in-memory stream based on a ring buffer. It is a specialization of `CircularIO` (page 126) which associates video frame meta-data with the recorded stream, accessible from the `frames` (page 126) 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()` (page 126) method.

The *camera* parameter specifies the `PiCamera` (page 95) 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` (page 95)). 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()` (page 103). 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` (page 121) 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` (page 125) class documentation).

copy_to(*output*, *size=None*, *seconds=None*, *first_frame=PiVideoFrameType.sps_header*)

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` (page 120) 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.

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`[164] class. However, unlike `io.BytesIO`[165] its underlying storage is a ring buffer[166] 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()` (page 127), `tell()` (page 127), and `seek()` (page 127) methods all operate equivalently to those in `io.BytesIO`[167] whilst `write()` (page 127) only differs in the wrapping behaviour described above. A `read1()` (page 127) method is also provided for efficient reading of the underlying ring buffer in write-sized chunks (or less).

164 https://docs.python.org/3.4/library/io.html#io.BytesIO
165 https://docs.python.org/3.4/library/io.html#io.BytesIO
166 https://en.wikipedia.org/wiki/Circular_buffer
167 https://docs.python.org/3.4/library/io.html#io.BytesIO

A re-entrant threading lock guards all operations, and is accessible for external use via the `lock` (page 127) 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()` (page 127) 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` (page 126) 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()` (page 127) is particularly useful for efficient copying of the stream's content.

`readable``()`
Returns `True`, indicating that the stream supports `read()` (page 127).

`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()` (page 127) and `tell()` (page 127).

`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()` (page 127).

`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.

BufferIO

class `picamera.`**`BufferIO`**`(`*obj*`)`

A stream which uses a `memoryview`[168] 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`[169] 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()`* (page 128) 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()`* (page 128).

`readall``()`

Read and return all bytes from the buffer until EOF.

`readinto``(`*b*`)`

Read bytes into a pre-allocated, writable bytes-like object b, and return the number of bytes read.

`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()`* (page 128) and *`tell()`* (page 128).

`tell``()`

Return the current buffer position.

`truncate``(`*size=None*`)`

Raises `NotImplementedError`[170] as the underlying buffer cannot be resized.

`writable``()`

Returns `True`, indicating that the stream supports *`write()`* (page 128).

`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`[171].

`size`

Return the maximum size of the buffer in bytes.

[168] https://docs.python.org/3.4/library/stdtypes.html#memoryview
[169] https://docs.python.org/3.4/library/exceptions.html#IOError
[170] https://docs.python.org/3.4/library/exceptions.html#NotImplementedError
[171] https://docs.python.org/3.4/library/exceptions.html#IOError

CHAPTER 11

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()` (page 103) and `add_overlay()` (page 96) are generally used instead) but may find the attribute references for them useful.

PiRenderer

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

Wraps `MMALRenderer` (page 176) for use by PiCamera.

The *parent* parameter specifies the `PiCamera` (page 95) 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` (page 95), but the two derivatives defined below, `PiOverlayRenderer` (page 131) and `PiPreviewRenderer` (page 132), 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` (page 129) 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.

Note: If the renderer is being fed RGBA data (as in partially transparent overlays), the alpha property will be ignored.

crop

Retrieves or sets the area to read from the source.

The `crop` (page 130) 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` (page 130) and `window` (page 131).

Note: This property only affects the renderer; it has no bearing on image captures or recordings (unlike the `zoom` (page 120) property of the `PiCamera` (page 95) class).

fullscreen

Retrieves or sets whether the renderer appears full-screen.

The `fullscreen` (page 130) property is a bool which controls whether the renderer takes up the entire display or not. When set to `False`, the `window` (page 131) 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` (page 130) 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` (page 113) property of the `PiCamera` (page 95) class).

layer

Retrieves or sets the layer of the renderer.

The `layer` (page 130) 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` (page 130) 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` (page 118) property of the `PiCamera` (page 95) class).

vflip

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

When queried, the `vflip` (page 130) 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` (page 120) property of the `PiCamera` (page 95) class).

`window`

Retrieves or sets the size of the renderer.

When the `fullscreen` (page 130) property is set to `False`, the `window` (page 131) 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.

PiOverlayRenderer

class `picamera.`**`PiOverlayRenderer`**(*parent*, *source*, *resolution=None*, *format=None*, *layer=0*, *alpha=255*, *fullscreen=True*, *window=None*, *crop=None*, *rotation=0*, *vflip=False*, *hflip=False*)

Represents an `MMALRenderer` (page 176) with a static source for overlays.

This class descends from `PiRenderer` (page 129) and adds a static *source* for the `MMALRenderer` (page 176). The *source* must be an object that supports the buffer protocol[172] in one of the supported formats.

The optional *resolution* parameter specifies the size of the *source* 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` (page 117). The optional *format* parameter specifies the encoding of the *source*. This can be one of the unencoded formats: `'yuv'`, `'rgb'`, `'rgba'`, `'bgr'`, or `'bgra'`. If omitted or `None`, *format* will be guessed based on the size of *source* (assuming 3 bytes for RGB[173], and 4 bytes for RGBA[174]).

The length of *source* must take into account that widths are rounded up to the nearest multiple of 32, and heights to the nearest multiple of 16. For example, if *resolution* is `(1280, 720)`, and *format* is `'rgb'` 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)`, and *format* is `'rgb'` 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` (page 129).

Changed in version 1.13: Added *format* parameter

`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).

Note: If you repeatedly update an overlay renderer, you must make sure that you do so at a rate equal to, or slower than, the camera's framerate. Going faster will rapidly starve the renderer's pool of buffers leading to a runtime error.

[172] https://docs.python.org/3.4/c-api/buffer.html#bufferobjects
[173] https://en.wikipedia.org/wiki/RGB
[174] https://en.wikipedia.org/wiki/RGBA_color_space

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`* (page 176) which uses the camera's preview as a source.

This class descends from *`PiRenderer`* (page 129) and adds an *`MMALConnection`* (page 179) to connect the renderer to an MMAL port. The *source* parameter specifies the *`MMALPort`* (page 177) to connect to the renderer.

The *layer*, *alpha*, *fullscreen*, and *window* parameters are the same as in *`PiRenderer`* (page 129).

`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`* (page 132) 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`* (page 117) property of the *`PiCamera`* (page 95) class).

New in version 1.11.

PiNullSink

class `picamera.`**`PiNullSink`**(*parent*, *source*)

Implements an *`MMALNullSink`* (page 176) which can be used in place of a renderer.

The *parent* parameter specifies the *`PiCamera`* (page 95) instance which constructed this *`MMALNullSink`* (page 176). The *source* parameter specifies the *`MMALPort`* (page 177) which the null-sink should connect to its input.

The null-sink can act as a drop-in replacement for *`PiRenderer`* (page 129) 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`* (page 129) 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).

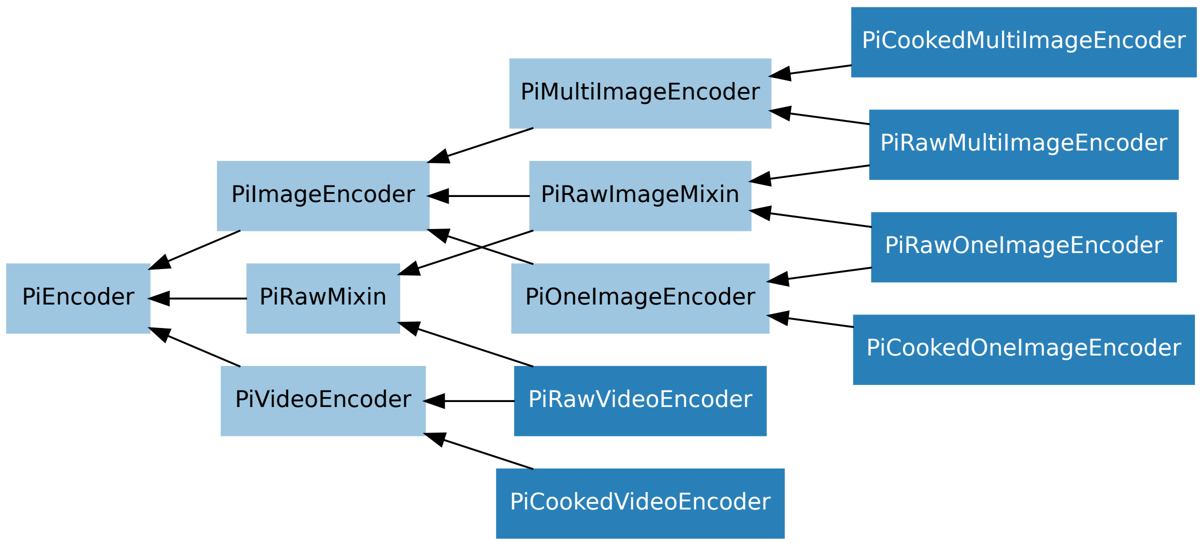
CHAPTER 12

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* (page 46).

Note: It is strongly recommended that you familiarize yourself with the `mmalobj` (page 159) layer before attempting to understand the encoder classes as they deal with several concepts native to that layer.

The inheritance diagram for the following classes is displayed below:



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` (page 95) 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` (page 178) 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` (page 175) 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` (page 138)).

`event`
A `threading.Event`[175] 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` (page 178) that the encoder should be connected to.

[175] https://docs.python.org/3.4/library/threading.html#threading.Event

output_port
The *MMALVideoPort* (page 178) 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* (page 135).

parent
The *PiCamera* (page 95) instance that created this PiEncoder instance.

pool
A pointer to a pool of MMAL buffers.

resizer
The *MMALResizer* (page 175) 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* (page 180) which can be used to obtain the data to write, along with meta-data about the current frame.

This method must set *event* (page 134) when the encoder has finished (and should set *exception* (page 134) if an exception occurred during encoding).

Developers wishing to write a custom encoder class may find it simpler to override the *_callback_write()* (page 135) method, rather than deal with these complexities.

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

This method is called by *_callback()* (page 135) to handle writing to an object in *outputs* (page 135) identified by *key*. The *buf* parameter is an *MMALBuffer* (page 180) 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=PiVideoFrameType.frame*)
Closes the output associated with *key* in *outputs* (page 135).

Closes the output object associated with the specified *key*, and removes it from the *outputs* (page 135) dictionary (if we didn't open the object then we attempt to flush it instead).

_create_encoder (*format*)
Creates and configures the *MMALEncoder* (page 175) component.

This method only constructs the encoder; it does not connect it to the input port. The method sets the *encoder* (page 134) attribute to the constructed encoder component, and the *output_port* (page 134) 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` (page 175) 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` (page 135) attribute to the constructed resizer component.

_open_output (*output*, *key=PiVideoFrameType.frame*)
Opens *output* and associates it with *key* in `outputs` (page 135).

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()` (page 135) 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` (page 128) stream to simplify writing).

The opened output is added to the `outputs` (page 135) 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.

PiVideoEncoder

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

This derivative of `PiEncoder` (page 133) configures itself for H.264 or MJPEG encoding. It also introduces a `split()` (page 137) method which is used by `split_recording()` (page 102) and `record_sequence()` (page 101) to redirect future output to a new filename or object. Finally, it also extends `PiEncoder.start()` (page 136) and `PiEncoder._callback_write()` (page 135) 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()` (page 137) is called.

_create_encoder (*format*, *bitrate=17000000*, *intra_period=None*, *profile='high'*, *level='4'*, *quantization=0*, *quality=0*, *inline_headers=True*, *sei=False*, *sps_timing=False*, *motion_output=None*, *intra_refresh=None*)
Extends the base `_create_encoder()` (page 135) 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()` (page 102) and `split()` (page 137) 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()` (page 102) and `record_sequence()` (page 101) to switch the encoder's `output` object to the *output* parameter (which can be a filename or a file-like object, as with `start()` (page 137)).

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

PiImageEncoder

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

This derivative of `PiEncoder` (page 133) extends the `_create_encoder()` (page 137) 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)*, *restart=0*)
Extends the base `_create_encoder()` (page 135) implementation to configure the image encoder for JPEG, PNG, etc.

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` (page 133), along with `_create_resizer()` and `_create_encoder()` (page 137) 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=PiVideoFrameType.frame*)
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.

PiCookedVideoEncoder

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

Video encoder for encoded recordings.

This class is a derivative of *PiVideoEncoder* (page 136) and only exists to provide naming symmetry with the image encoder classes.

PiRawVideoEncoder

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

Video encoder for unencoded recordings.

This class is a derivative of *PiVideoEncoder* (page 136) and the *PiRawMixin* (page 137) class intended for use with *start_recording()* (page 103) 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.

PiOneImageEncoder

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

Encoder for single image capture.

This class simply extends *_callback_write()* (page 135) to terminate capture at frame end (i.e. after a single frame has been received).

PiMultiImageEncoder

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

Encoder for multiple image capture.

This class extends *PiImageEncoder* (page 137) to handle an iterable of outputs instead of a single output. The *_callback_write()* (page 135) method is extended to terminate capture when the iterable is exhausted, while *PiEncoder._open_output()* (page 136) is overridden to begin iteration and rely on the new *_next_output()* (page 138) 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()* (page 136).

PiRawImageMixin

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

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

The *_callback_write()* (page 138) 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__()`).

PiCookedOneImageEncoder

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

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

This encoder extends *`PiOneImageEncoder`* (page 138) to include Exif tags in the output.

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`* (page 138) and the *`PiRawImageMixin`* (page 138) class intended for use with *`capture()`* (page 97) (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.

PiCookedMultiImageEncoder

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

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

This encoder descends from *`PiMultiImageEncoder`* (page 138) but includes no new functionality as video-port based encodes (which is all this class is used for) don't support Exif tag output.

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`* (page 138) and the *`PiRawImageMixin`* (page 138) class intended for use with *`capture_sequence()`* (page 100) 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.

CHAPTER 13

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` (page 142) derives from both `PiCameraError` (page 142) and `ValueError`[176]. 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')
```

Warnings

exception picamera.**PiCameraWarning**

Bases: `Warning`[177]

Base class for PiCamera warnings.

exception picamera.**PiCameraDeprecated**

Bases: `picamera.exc.PiCameraWarning`, `DeprecationWarning`[178]

Raised when deprecated functionality in picamera is used.

exception picamera.**PiCameraFallback**

Bases: `picamera.exc.PiCameraWarning`, `RuntimeWarning`[179]

Raised when picamera has to fallback on old functionality.

[176] https://docs.python.org/3.4/library/exceptions.html#ValueError
[177] https://docs.python.org/3.4/library/exceptions.html#Warning
[178] https://docs.python.org/3.4/library/exceptions.html#DeprecationWarning
[179] https://docs.python.org/3.4/library/exceptions.html#RuntimeWarning

exception picamera.**PiCameraResizerEncoding**
Bases: picamera.exc.PiCameraWarning, RuntimeWarning[180]

Raised when picamera uses a resizer purely for encoding purposes.

exception picamera.**PiCameraAlphaStripping**
Bases: picamera.exc.PiCameraWarning, RuntimeWarning[181]

Raised when picamera does alpha-byte stripping.

Exceptions

exception picamera.**PiCameraError**
Bases: Exception[182]

Base class for PiCamera errors.

exception picamera.**PiCameraValueError**
Bases: picamera.exc.PiCameraError, ValueError[183]

Raised when an invalid value is fed to a *PiCamera* (page 95) object.

exception picamera.**PiCameraRuntimeError**
Bases: picamera.exc.PiCameraError, RuntimeError[184]

Raised when an invalid sequence of operations is attempted with a *PiCamera* (page 95) object.

exception picamera.**PiCameraClosed**
Bases: picamera.exc.PiCameraRuntimeError

Raised when a method is called on a camera which has already been closed.

exception picamera.**PiCameraNotRecording**
Bases: picamera.exc.PiCameraRuntimeError

Raised when *stop_recording()* (page 104) or *split_recording()* (page 102) are called against a port which has no recording active.

exception picamera.**PiCameraAlreadyRecording**
Bases: picamera.exc.PiCameraRuntimeError

Raised when *start_recording()* (page 103) or *record_sequence()* (page 101) are called against a port which already has an active recording.

exception picamera.**PiCameraMMALError**(*status*, *prefix=''*)
Bases: picamera.exc.PiCameraError

Raised when an MMAL operation fails for whatever reason.

exception picamera.**PiCameraPortDisabled**(*msg*)
Bases: picamera.exc.PiCameraMMALError

Raised when attempting a buffer operation on a disabled port.

This exception is intended for the common use-case of attempting to get or send a buffer just when a component is shutting down (e.g. at script teardown) and simplifies the trivial response (ignore the error and shut down quietly). For example:

```python
def _callback(self, port, buf):
    try:
        buf = self.outputs[0].get_buffer(False)
```

[180] https://docs.python.org/3.4/library/exceptions.html#RuntimeWarning
[181] https://docs.python.org/3.4/library/exceptions.html#RuntimeWarning
[182] https://docs.python.org/3.4/library/exceptions.html#Exception
[183] https://docs.python.org/3.4/library/exceptions.html#ValueError
[184] https://docs.python.org/3.4/library/exceptions.html#RuntimeError

```
    except PiCameraPortDisabled:
        return True # shutting down
    # ...
```

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* (page 142) 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.

CHAPTER 14

API - Colors and Color Matching

The picamera library includes a comprehensive `Color` (page 145) 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.

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[185], Y'UV[186], Y'IQ[187], HLS[188], and HSV[189]. 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` (page 145) 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:

[185] https://en.wikipedia.org/wiki/RGB_color_space
[186] https://en.wikipedia.org/wiki/YUV
[187] https://en.wikipedia.org/wiki/YIQ
[188] https://en.wikipedia.org/wiki/HSL_and_HSV
[189] https://en.wikipedia.org/wiki/HSL_and_HSV

Style	Description
Single positional parameter	Equivalent to calling `Color.from_string()` (page 148).
Three positional parameters	Equivalent to calling `Color.from_rgb()` (page 148) if all three parameters are between 0.0 and 1.0, or `Color.from_rgb_bytes()` (page 148) 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()` (page 148) if *y* is between 0.0 and 1.0, *u* is between -0.436 and 0.436, and *v* is between -0.615 and 0.615, or `Color.from_yuv_bytes()` (page 148) otherwise.
Three named parameters: *y*, *i*, *q*	Equivalent to calling `Color.from_yiq()` (page 148).
Three named parameters: *h*, *l*, *s*	Equivalent to calling `Color.from_hls()` (page 147).
Three named parameters: *hue*, *lightness*, *saturation*	
Three named parameters: *h*, *s*, *v*	Equivalent to calling `Color.from_hsv()` (page 147)
Three named parameters: *hue*, *saturation*, *value*	
Three named parameters: *x*, *y*, *z*	Equivalent to calling `Color.from_cie_xyz()` (page 147)
Three named parameters: *l*, *a*, *b*	Equivalent to calling `Color.from_cie_lab()` (page 147)
Three named parameters: *l*, *u*, *v*	Equivalent to calling `Color.from_cie_luv()` (page 147)

If the constructor parameters do not conform to any of the variants in the table above, a `ValueError`[190] 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` (page 145) 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` (page 145) 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)
```

[190] https://docs.python.org/3.4/library/exceptions.html#ValueError

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` (page 145) instance is constructed from HLS (hue, lightness, saturation) values with a lightness of 0.5. This is multiplied by a `Lightness` (page 150) instance with a value of 0.8 which constructs a new `Color` (page 145) 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()`[191].

`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[192]. This is by far the fastest method, but also the least accurate in terms of human perception.
- 'cie1976' - Use the CIE 1976[193] formula for calculating the difference between two colors in CIE Lab space.
- 'cie1994g' - Use the CIE 1994[194] formula with the "graphic arts" bias for calculating the difference.
- 'cie1994t' - Use the CIE 1994[195] forumula with the "textiles" bias for calculating the difference.
- 'cie2000' - Use the CIEDE 2000[196] 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[197] value in which 2.3 is considered a just-noticeable difference[198] (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` (page 145) from (L*, a*, b*) float values representing a color in the CIE Lab color space[199]. The conversion assumes the sRGB working space with reference white D65.

classmethod `from_cie_luv`(*l*, *u*, *v*)

Construct a `Color` (page 145) from (L*, u*, v*) float values representing a color in the CIE Luv color space[200]. The conversion assumes the sRGB working space with reference white D65.

classmethod `from_cie_xyz`(*x*, *y*, *z*)

Construct a `Color` (page 145) from (X, Y, Z) float values representing a color in the CIE 1931 color space[201]. The conversion assumes the sRGB working space with reference white D65.

classmethod `from_hls`(*h*, *l*, *s*)

Construct a `Color` (page 145) from HLS[202] (hue, lightness, saturation) floats between 0.0 and 1.0.

classmethod `from_hsv`(*h*, *s*, *v*)

Construct a `Color` (page 145) from HSV[203] (hue, saturation, value) floats between 0.0 and 1.0.

[191] https://docs.python.org/3.4/library/functions.html#repr
[192] https://en.wikipedia.org/wiki/Euclidean_distance
[193] https://en.wikipedia.org/wiki/Color_difference#CIE76
[194] https://en.wikipedia.org/wiki/Color_difference#CIE94
[195] https://en.wikipedia.org/wiki/Color_difference#CIE94
[196] https://en.wikipedia.org/wiki/Color_difference#CIEDE2000
[197] https://en.wikipedia.org/wiki/Color_difference
[198] https://en.wikipedia.org/wiki/Just-noticeable_difference
[199] https://en.wikipedia.org/wiki/Lab_color_space
[200] https://en.wikipedia.org/wiki/CIELUV
[201] https://en.wikipedia.org/wiki/CIE_1931_color_space
[202] https://en.wikipedia.org/wiki/HSL_and_HSV
[203] https://en.wikipedia.org/wiki/HSL_and_HSV

classmethod **`from_rgb`**(*r*, *g*, *b*)
Construct a `Color` (page 145) from three RGB[204] float values between 0.0 and 1.0.

classmethod **`from_rgb_565`**(*n*)
Construct a `Color` (page 145) from an unsigned 16-bit integer number in RGB565 format.

classmethod **`from_rgb_bytes`**(*r*, *g*, *b*)
Construct a `Color` (page 145) from three RGB[205] byte values between 0 and 255.

classmethod **`from_string`**(*s*)
Construct a `Color` (page 145) 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[206]. Any other string format will result in a `ValueError`[207].

classmethod **`from_yiq`**(*y*, *i*, *q*)
Construct a `Color` (page 145) from three Y'IQ[208] 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` (page 145) from three Y'UV[209] 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` (page 145) from three Y'UV[210] 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` (page 149) instance which can be used in operations with other `Color` (page 145) instances.

`cie_lab`
Returns a 3-tuple of (L*, a*, b*) float values representing the color in the CIE Lab color space[211] with the D65 standard illuminant[212].

`cie_luv`
Returns a 3-tuple of (L*, u*, v*) float values representing the color in the CIE Luv color space[213] with the D65 standard illuminant[214].

`cie_xyz`
Returns a 3-tuple of (X, Y, Z) float values representing the color in the CIE 1931 color space[215]. The conversion assumes the sRGB working space, with reference white D65.

`green`
Returns the green component of the color as a `Green` (page 149) instance which can be used in operations with other `Color` (page 145) 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).

[204] https://en.wikipedia.org/wiki/RGB_color_space
[205] https://en.wikipedia.org/wiki/RGB_color_space
[206] http://www.w3.org/TR/css3-color/#svg-color
[207] https://docs.python.org/3.4/library/exceptions.html#ValueError
[208] https://en.wikipedia.org/wiki/YIQ
[209] https://en.wikipedia.org/wiki/YUV
[210] https://en.wikipedia.org/wiki/YUV
[211] https://en.wikipedia.org/wiki/Lab_color_space
[212] https://en.wikipedia.org/wiki/Illuminant_D65
[213] https://en.wikipedia.org/wiki/CIELUV
[214] https://en.wikipedia.org/wiki/Illuminant_D65
[215] https://en.wikipedia.org/wiki/CIE_1931_color_space

hue
Returns the hue of the color as a *Hue* (page 150) instance which can be used in operations with other *Color* (page 145) instances.

lightness
Returns the lightness of the color as a *Lightness* (page 150) instance which can be used in operations with other *Color* (page 145) instances.

red
Returns the red component of the color as a *Red* (page 149) instance which can be used in operations with other *Color* (page 145) 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* (page 150) instance which can be used in operations with other *Color* (page 145) 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.

Manipulation Classes

class picamera.**Red**
Represents the red component of a *Color* (page 145) for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the *Color.red* (page 149) attribute. Addition, subtraction, and multiplication are supported with *Color* (page 145) 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* (page 145) for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the *Color.green* (page 148) attribute. Addition, subtraction, and multiplication are supported with *Color* (page 145) 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` (page 145) for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the `Color.blue` (page 148) attribute. Addition, subtraction, and multiplication are supported with `Color` (page 145) 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 "#ffffe5">
```

class `picamera.`**`Hue`**

Represents the hue of a `Color` (page 145) 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[216]. 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` (page 148) attribute.

Addition, subtraction, and multiplication are supported with `Color` (page 145) 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` (page 150) 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` (page 145) for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the `Color.saturation` (page 149) attribute. Addition, subtraction, and multiplication are supported with `Color` (page 145) 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` (page 145) for use in transformations. Instances of this class can be constructed directly with a float value, or by querying the `Color.lightness` (page 149) attribute. Addition, subtraction, and multiplication are supported with `Color` (page 145) 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">
```

216 https://en.wikipedia.org/wiki/Hue

CHAPTER 15

API - Arrays

The picamera library provides a set of classes designed to aid in construction of n-dimensional numpy[217] arrays from camera output. In order to avoid adding a hard dependency on numpy to picamera, this module (`picamera.array` (page 151)) is not automatically imported by the main picamera package and must be explicitly imported, e.g.:

```
import picamera
import picamera.array
```

PiArrayOutput

class `picamera.array.`**`PiArrayOutput`**(*camera*, *size=None*)

Base class for capture arrays.

This class extends `io.BytesIO`[218] with a numpy[219] array which is intended to be filled when `flush()`[220] is called (i.e. at the end of capture).

`array`

After `flush()`[221] is called, this attribute contains the frame's data as a multi-dimensional numpy[222] 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()`[223] and `truncate()` (page 151) as one would with regular `BytesIO`[224] instances.

[217] http://www.numpy.org/
[218] https://docs.python.org/3.4/library/io.html#io.BytesIO
[219] http://www.numpy.org/
[220] https://docs.python.org/3.4/library/io.html#io.IOBase.flush
[221] https://docs.python.org/3.4/library/io.html#io.IOBase.flush
[222] http://www.numpy.org/
[223] https://docs.python.org/3.4/library/io.html#io.IOBase.seek
[224] https://docs.python.org/3.4/library/io.html#io.BytesIO

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`* (page 151) 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]))
```

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`* (page 151) 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[225] 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()` (page 97) 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]))
```

PiBayerArray

class `picamera.array.`**`PiBayerArray`**(*camera*, *output_dims=3*)

Produces a 3-dimensional RGB array from raw Bayer data.

This custom output class is intended to be used with the `capture()` (page 97) method, with the *bayer* parameter set to `True`, to include raw Bayer data in the JPEG output. The class strips out the raw data, and constructs a numpy array from it. The resulting data is accessed via the `array` (page 151) 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)
```

The *output_dims* parameter specifies whether the resulting array is three-dimensional (the default, or when *output_dims* is 3), or two-dimensional (when *output_dims* is 2). The three-dimensional data is already separated into the three color planes, whilst the two-dimensional variant is not (in which case you need to know the Bayer ordering to accurately deal with the results).

Note: Bayer data is *usually* full resolution, so the resulting array usually has the shape (1944, 2592, 3) with the V1 module, or (2464, 3280, 3) with the V2 module (if two-dimensional output is requested the 3-layered

[225] https://en.wikipedia.org/wiki/YCbCr#ITU-R_BT.601_conversion

color dimension is omitted). If the camera's `sensor_mode` (page 118) has been forced to something other than 0, then the output will be the native size for the requested sensor mode.

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[226] 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[227]). A trivial weighted-average demosaicing algorithm is provided in the `demosaic()` (page 154) 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).

Changed in version 1.13: This class now supports the V2 module properly, and handles flipped images, and forced sensor modes correctly.

demosaic()

Perform a rudimentary de-mosaic[228] of `self.array`, returning the result as a new array. The result of the demosaic is *always* three dimensional, with the last dimension being the color planes (see *output_dims* parameter on the constructor).

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()` (page 103) 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[229] (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` (page 151) 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[230] 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)
```

[226] https://en.wikipedia.org/wiki/Demosaicing
[227] https://en.wikipedia.org/wiki/Bayer_filter
[228] https://en.wikipedia.org/wiki/Demosaicing
[229] https://en.wikipedia.org/wiki/Macroblock
[230] https://en.wikipedia.org/wiki/Sum_of_absolute_differences

```
        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` (page 156) class instead.

PiAnalysisOutput

class `picamera.array.`**`PiAnalysisOutput`**(*camera*, *size=None*)

Base class for analysis outputs.

This class extends `io.IOBase`[231] with a stub `analyze()` (page 155) method which will be called for each frame output. In this base implementation the method simply raises `NotImplementedError`[232].

`analyse`(*array*)

Deprecated alias of `analyze()` (page 155).

`analyze`(*array*)

Stub method for users to override.

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()` (page 103) 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()` (page 155) method with the resulting array (this deliberately raises `NotImplementedError`[233] in this class; you must override it in your descendent class).

[231] https://docs.python.org/3.4/library/io.html#io.IOBase
[232] https://docs.python.org/3.4/library/exceptions.html#NotImplementedError
[233] https://docs.python.org/3.4/library/exceptions.html#NotImplementedError

Note: If your overridden `analyze()` (page 155) 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()` (page 155) 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'`).

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()` (page 103) 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()` (page 155) method with the resulting array (this deliberately raises `NotImplementedError`[234] in this class; you must override it in your descendent class).

Note: If your overridden `analyze()` (page 155) 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()` (page 155) 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.

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()` (page 103) method. While recording is in progress, the write method converts incoming motion data into numpy arrays and calls the stub `analyze()` (page 155) method with the resulting array (which deliberately raises `NotImplementedError`[235] in this class).

Note: If your overridden `analyze()` (page 155) 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()` (page 155) is organized as (rows, columns) where `rows` and `columns` are the number of rows and columns of macro-blocks[236] (16x16 pixel blocks) in the original frames. There is always one extra column of macro-blocks present in motion vector data.

[234] https://docs.python.org/3.4/library/exceptions.html#NotImplementedError
[235] https://docs.python.org/3.4/library/exceptions.html#NotImplementedError
[236] https://en.wikipedia.org/wiki/Macroblock

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[237] 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()` (page 103).

PiArrayTransform

class `picamera.array.`**`PiArrayTransform`**(*formats=('rgb', 'bgr', 'rgba', 'bgra')*)

A derivative of `MMALPythonComponent` (page 185) which eases the construction of custom MMAL transforms by representing buffer data as numpy arrays. The *formats* parameter specifies the accepted input formats as a sequence of strings (default: 'rgb', 'bgr', 'rgba', 'bgra').

Override the `transform()` (page 157) method to modify buffers sent to the component, then place it in your MMAL pipeline as you would a normal encoder.

`transform`(*source, target*)

This method will be called for every frame passing through the transform. The *source* and *target* parameters represent buffers from the input and output ports of the transform respectively. They will be derivatives of `MMALBuffer` (page 180) which return a 3-dimensional numpy array when used as context managers. For example:

```
def transform(self, source, target):
    with source as source_array, target as target_array:
        # Copy the source array data to the target
        target_array[...] = source_array
        # Draw a box around the edges
        target_array[0, :, :] = 0xff
        target_array[-1, :, :] = 0xff
        target_array[:, 0, :] = 0xff
        target_array[:, -1, :] = 0xff
        return False
```

The target buffer's meta-data starts out as a copy of the source buffer's meta-data, but the target buffer's data starts out uninitialized.

[237] https://en.wikipedia.org/wiki/Sum_of_absolute_differences

CHAPTER 16

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 idiosyncrasies in `libmmal`.

Warning: This part of the API is still experimental and subject to change in future versions. Backwards compatibility is not (yet) guaranteed.

The MMAL Tour

MMAL operates on the principle of pipelines:

- A pipeline consists of one or more MMAL components (*`MMALBaseComponent`* (page 174) and derivatives) connected together in series.
- A *`MMALBaseComponent`* (page 174) has one or more ports.
- A port (*`MMALControlPort`* (page 176) and derivatives) is either a control port, an input port or an output port (there are also clock ports but you generally don't need to deal with these as MMAL sets them up automatically):
  - Control ports are used to accept and receive commands, configuration parameters, and error messages. All MMAL components have a control port, but in picamera they're only used for component configuration.
  - Input ports receive data from upstream components.
  - Output ports send data onto downstream components (if they're connected), or to callback routines in the user's program (if they're not connected).
  - Input and output ports can be audio, video or sub-picture (subtitle) ports, but picamera only deals with video ports.
  - Ports have a *`format`* (page 178) which (in the case of video ports) dictates the format of image/frame accepted or generated by the port (YUV, RGB, JPEG, H.264, etc.)
  - Video ports have a *`framerate`* (page 178) which specifies the number of images expected to be received or sent per second.

  - Video ports also have a *framesize* (page 178) which specifies the resolution of images/frames accepted or generated by the port.
  - Finally, all ports (control, input and output) have *params* (page 177) which affect their operation.
- An output port can have a *MMALConnection* (page 179) to an input port. Connections ensure the two ports use compatible formats, and handle transferring data from output ports to input ports in an orderly fashion. A port cannot have more than one connection from/to it.
- Data is written to / read from ports via instances of *MMALBuffer* (page 180).
  - Buffers belong to a port and can't be passed arbitrarily between ports.
  - The size of a buffer is dictated by the format and frame-size of the port that owns the buffer. The memory allocation of a buffer (readable from *size* (page 181)) cannot be altered once the port is enabled, but the buffer can contain any amount of data up its allocation size. The actual length of data in a buffer is stored in *length* (page 181).
  - Likewise, the number of buffers belonging to a port is fixed and cannot be altered without disabling the port, reconfiguring it and re-enabling it. The more buffers a port has, the less likely it is that the pipeline will have to drop frames because a component has overrun, but the more GPU memory is required.
  - Buffers also have *flags* (page 181) which specify information about the data they contain (e.g. start of frame, end of frame, key frame, etc.)
  - When a connection exists between two ports, the connection continually requests a buffer from the output port, requests another buffer from the input port, copies the output buffer's data to the input buffer's data, then returns the buffers to their respective ports (this is a simplification; various tricks are pulled under the covers to minimize the amount of data copying that *actually* occurs, but as a mental model of what's going on it's reasonable).
  - Components take buffers from their input port(s), process them, and write the result into a buffer from the output port(s).

Components

Now we've got a mental model of what an MMAL pipeline consists of, let's build one. For the rest of the tour I strongly recommend using a Pi with a screen (so you can see preview output) but controlling it via an SSH session (so the preview doesn't cover your command line). Follow along, typing the examples into your remote Python session. And feel free to deviate from the examples if you're curious about things!

We'll start by importing the *mmalobj* (page 159) module with a convenient alias, then construct a *MMALCamera* (page 174) component, and a *MMALRenderer* (page 176) component.

```
>>> from picamera import mmal, mmalobj as mo
>>> camera = mo.MMALCamera()
>>> preview = mo.MMALRenderer()
```

Ports

Before going any further, let's have a look at the various ports on these components.

```
>>> len(camera.inputs)
0
>>> len(camera.outputs)
3
>>> len(preview.inputs)
1
>>> len(preview.outputs)
0
```

The fact the camera has three outputs should come as little surprise to those who have read the *Camera Hardware* (page 65) chapter (if you haven't already, you might want to skim it now). Let's examine the first output port of the camera and the input port of the renderer:

```
>>> camera.outputs[0]
<MMALVideoPort "vc.ril.camera:out:0": format=MMAL_FOURCC('I420')
buffers=1x7680 frames=320x240@0fps>
>>> preview.inputs[0]
<MMALVideoPort "vc.ril.video_render:in:0" format=MMAL_FOURCC('I420')
buffers=2x15360 frames=160x64@0fps>
```

Several things to note here:

- We can tell from the port name what sort of component it belongs to, what its index is, and whether it's an input or an output port
- Both ports are currently configured for the I420 format; this is MMAL's name for YUV420[238] (full resolution Y, quarter resolution UV).
- The ports have different frame-sizes (320x240 and 160x64 respectively), buffer counts (1 and 2 respectively) and buffer sizes (7680 and 15360 respectively).
- The buffer sizes look unrealistic. For example, 7680 bytes is nowhere near enough to hold 320 * 240 * 1.5 bytes (YUV420 requires 1.5 bytes per pixel).

Now we'll configure the camera's output port with a slightly higher resolution, and give it a frame-rate:

```
>>> camera.outputs[0].framesize = (640, 480)
>>> camera.outputs[0].framerate = 30
>>> camera.outputs[0].commit()
>>> camera.outputs[0]
<MMALVideoPort "vc.ril.camera:out:0(I420)": format=MMAL_FOURCC('I420')
buffers=1x460800 frames=640x480@30fps>
```

Note that the changes to the configuration won't actually take effect until the `commit()` (page 177) call. After the port is committed, note that the buffer size now looks reasonable: 640 * 480 * 1.5 = 460800.

Connections

Now we'll try connecting the renderer's input to the camera's output. Don't worry about the fact that the port configurations are different. One of the nice things about MMAL (and the `mmalobj` layer) is that connections try very hard to auto-configure things so that they "just work". Usually, auto-configuration is based upon the *output* port being connected so it's important to get that configuration right, but you don't generally need to worry about the input port.

The renderer is what `mmalobj` terms a "downstream component". This is a component with a single input that typically sits downstream from some feeder component (like a camera). All such components have the `connect()` (page 175) method which can be used to connect the sole input to a specified output:

```
>>> preview.connect(camera)
<MMALConnection "vc.ril.camera:out:0/vc.ril.video_render:in:0">
>>> preview.connection.enable()
```

Note that we've been quite lazy in the snippet above by simply calling `connect()` (page 175) with the `camera` component. In this case, a connection will be attempted between the first input port of the owner (`preview`) and the *first unconnected* output of the parameter (`camera`). However, this is not always what's wanted so you can specify the exact ports you wish to connect. In this case the example was equivalent to calling:

```
>>> preview.inputs[0].connect(camera.outputs[0])
<MMALConnection "vc.ril.camera:out:0/vc.ril.video_render:in:0">
>>> preview.inputs[0].connection.enable()
```

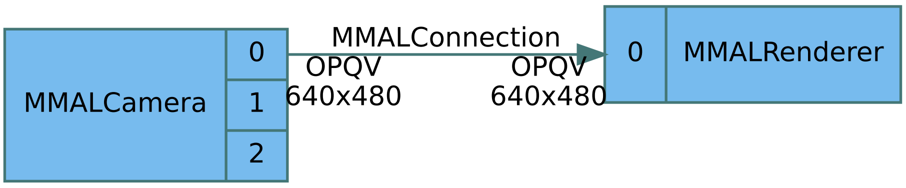
[238] https://en.wikipedia.org/wiki/YUV#Y.E2.80.B2UV420p_.28and_Y.E2.80.B2V12_or_YV12.29_to_RGB888_conversion

Note that the `connect()` (page 175) method returns the connection that was constructed but you can also retrieve this by querying the port's `connection` (page 178) attribute later.

As soon as the connection is enabled you should see the camera preview appear on the Pi's screen. Let's query the port configurations now:

```
>>> camera.outputs[0]
<MMALVideoPort "vc.ril.camera:out:0(OPQV)": format=MMAL_FOURCC('OPQV')
buffers=10x128 frames=640x480@30fps>
>>> preview.inputs[0]
<MMALVideoPort "vc.ril.video_render:in:0(OPQV)": format=MMAL_FOURCC('OPQV')
buffers=10x128 frames=640x480@30fps>
```

Note that the connection has implicitly reconfigured the camera's output port to use the OPAQUE ("OPQV") format. This is a special format used internally by the camera firmware which avoids passing complete frame data around, instead passing pointers to frame data around (this explains the tiny buffer size of 128 bytes as very little data is actually being shuttled between the components). Further, note that the connection has automatically copied the port format, frame size and frame-rate to the preview's input port.



Opaque Format

At this point it is worth exploring the differences between the camera's three output ports:

- Output 0 is the "preview" output. On this port, the OPAQUE format contains a pointer to a complete frame of data.
- Output 1 is the "video recording" output. On this port, the OPAQUE format contains a pointer to *two* complete frames of data. The dual-frame format enables the H.264 video encoder to calculate motion estimation without the encoder having to keep copies of prior frames itself (it can do this when something other than OPAQUE format is used, but dual-image OPAQUE is *much* more efficient).
- Output 2 is the "still image" output. On this port, the OPAQUE format contains a pointer to a strip of an image. The "strips" format is used by the JPEG encoder (not to be confused with the MJPEG encoder) to deal with high resolution images efficiently.

Generally, you don't need to worry about these differences. The `mmalobj` layer knows about them and negotiates the most efficient format it can for connections. However, they're worth bearing in mind if you're aiming to get the most out of the firmware or if you're confused about why a particular format has been selected for a connection.

Component Configuration

So far we've seen how to construct components, configure their ports, and connect them together in rudimentary pipelines. Now, let's see how to configure components via control port parameters:

```
>>> camera.control.params[mmal.MMAL_PARAMETER_SYSTEM_TIME]
177572014208
>>> camera.control.params[mmal.MMAL_PARAMETER_SYSTEM_TIME]
177574350658
>>> camera.control.params[mmal.MMAL_PARAMETER_BRIGHTNESS]
Fraction(1, 2)
>>> camera.control.params[mmal.MMAL_PARAMETER_BRIGHTNESS] = 0.75
>>> camera.control.params[mmal.MMAL_PARAMETER_BRIGHTNESS]
Fraction(3, 4)
```

```
>>> fx = camera.control.params[mmal.MMAL_PARAMETER_IMAGE_EFFECT]
>>> fx
<picamera.mmal.MMAL_PARAMETER_IMAGEFX_T object at 0x765b8440>
>>> dir(fx)
['__class__', '__ctypes_from_outparam__', '__delattr__', '__dict__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', '_b_base_', '_b_needsfree_', '_fields_', '_objects', 'hdr',
'value']
>>> fx.value
0
>>> mmal.MMAL_PARAM_IMAGEFX_NONE
0
>>> fx.value = mmal.MMAL_PARAM_IMAGEFX_EMBOSS
>>> camera.control.params[mmal.MMAL_PARAMETER_IMAGE_EFFECT] = fx
>>> camera.control.params[mmal.MMAL_PARAMETER_BRIGHTNESS] = 1/2
>>> camera.control.params[mmal.MMAL_PARAMETER_IMAGE_EFFECT] = mmal.MMAL_PARAM_
↪IMAGEFX_NONE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/pi/picamera/picamera/mmalobj.py", line 1109, in __setitem__
    assert mp.hdr.id == key
AttributeError: 'int' object has no attribute 'hdr'
>>> fx.value = mmal.MMAL_PARAM_IMAGEFX_NONE
>>> camera.control.params[mmal.MMAL_PARAMETER_IMAGE_EFFECT] = fx
>>> preview.disconnect()
```

Things to note:

- The parameter dictates the type of the value returned (and accepted, if the parameter is read-write).
- Many parameters accept a multitude of simple types like `int`[239], `float`[240], `Fraction`[241], `str`[242], etc. However, some parameters use `ctypes`[243] structures and such parameters only accept the relevant structure.
- The easiest way to use such "structured" parameters is to query them first, modify the resulting structure, then write it back to the parameter.

To find out what parameters are available for use with the camera component, have a look at the source for the *`PiCamera`* (page 95) class, especially property getters and setters.

File Output (RGB capture)

Let's see how we can produce some file output from the camera. First we'll perform a straight unencoded RGB capture from the still port (2). As this is unencoded output we don't need to construct anything else. All we need to do is configure the port for RGB encoding, select an appropriate resolution, then activate the output port:

```
>>> camera.outputs[2].format = mmal.MMAL_ENCODING_RGB24
>>> camera.outputs[2].framesize = (640, 480)
>>> camera.outputs[2].commit()
>>> camera.outputs[2]
<MMALVideoPort "vc.ril.camera:out:2(RGB3)": format=MMAL_FOURCC('RGB3')
buffers=1x921600 frames=640x480@0fps>
>>> camera.outputs[2].enable()
```

[239] https://docs.python.org/3.4/library/functions.html#int
[240] https://docs.python.org/3.4/library/functions.html#float
[241] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction
[242] https://docs.python.org/3.4/library/stdtypes.html#str
[243] https://docs.python.org/3.4/library/ctypes.html#module-ctypes

Unfortunately, that didn't seem to do much! An output port that is participating in a connection needs nothing more: it knows where its data is going. However, an output port *without* a connection requires a callback function to be assigned so that something can be done with the buffers of data it produces.

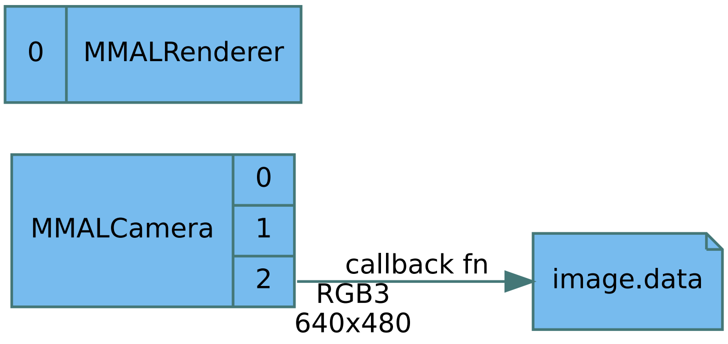
The callback will be given two parameters: the `MMALPort` (page 177) responsible for producing the data, and the `MMALBuffer` (page 180) containing the data. It is expected to return a `bool`[244] which will be `False` if further data is expected and `True` if no further data is expected. If `True` is returned, the callback will not be executed again. In our case we're going to write data out to a file we'll open before-hand, and we should return `True` when we see a buffer with the "frame end" flag set:

```
>>> camera.outputs[2].disable()
>>> import io
>>> output = io.open('image.data', 'wb')
>>> def image_callback(port, buf):
...     output.write(buf.data)
...     return bool(buf.flags & mmal.MMAL_BUFFER_HEADER_FLAG_FRAME_END)
...
>>> camera.outputs[2].enable(image_callback)
>>> output.tell()
0
```

At this stage you may note that while the file exists, nothing's been written to it. This is because output ports 1 and 2 (the video and still ports) won't produce any buffers until their "capture" parameter is enabled:

```
>>> camera.outputs[2].params[mmal.MMAL_PARAMETER_CAPTURE] = True
>>> camera.outputs[2].params[mmal.MMAL_PARAMETER_CAPTURE] = False
>>> output.tell()
921600
>>> camera.outputs[2].disable()
>>> output.close()
```

Congratulations! You've just captured your first image with the MMAL layer. Given we disconnected the preview above, the current state of the system looks something like this:



File Output (JPEG capture)

Whilst RGB is a useful format for processing we'd generally prefer something like JPEG for output. So, next we'll construct an MMAL JPEG encoder and use it to compress our RGB capture. Note that we're not going to connect the JPEG encoder to the camera yet; we're just going to construct it standalone and feed it data from our capture file, writing the output to another file:

```
>>> encoder = mo.MMALImageEncoder()
>>> encoder.inputs
(<MMALVideoPort "vc.ril.image_encode:in:0": format=MMAL_FOURCC('RGB2')
buffers=1x15360 frames=96x80@0fps>,)
>>> encoder.outputs
```

[244] https://docs.python.org/3.4/library/functions.html#bool

```
(<MMALVideoPort "vc.ril.image_encode:out:0": format=MMAL_FOURCC('GIF ')
buffers=1x81920 frames=0x0@0fps>,)
>>> encoder.inputs[0].format = mmal.MMAL_ENCODING_RGB24
>>> encoder.inputs[0].framesize = (640, 480)
>>> encoder.inputs[0].commit()
>>> encoder.outputs[0].copy_from(encoder.inputs[0])
>>> encoder.outputs[0]
<MMALVideoPort "vc.ril.image_encode:out:0": format=MMAL_FOURCC('RGB3')
buffers=1x81920 frames=640x480@0fps>
>>> encoder.outputs[0].format = mmal.MMAL_ENCODING_JPEG
>>> encoder.outputs[0].commit()
>>> encoder.outputs[0]
<MMALVideoPort "vc.ril.image_encode:out:0(JPEG)": format=MMAL_FOURCC('JPEG')
buffers=1x307200 frames=0x0@0fps>
>>> encoder.outputs[0].params[mmal.MMAL_PARAMETER_JPEG_Q_FACTOR] = 90
```

Just pausing for a moment, let's re-cap what we've got: an image encoder constructed, configured for 640x480 RGB input, and JPEG output with a quality factor of "90" (i.e. "very good" - don't try to read much more than this into JPEG quality settings!). Note that MMAL has set the buffer size at a size it thinks will be typical for the output. As JPEG is a lossy format this won't be precise and it's entirely possible that we may receive multiple callbacks for a single frame (if the compression overruns the expected buffer size).

Let's continue:

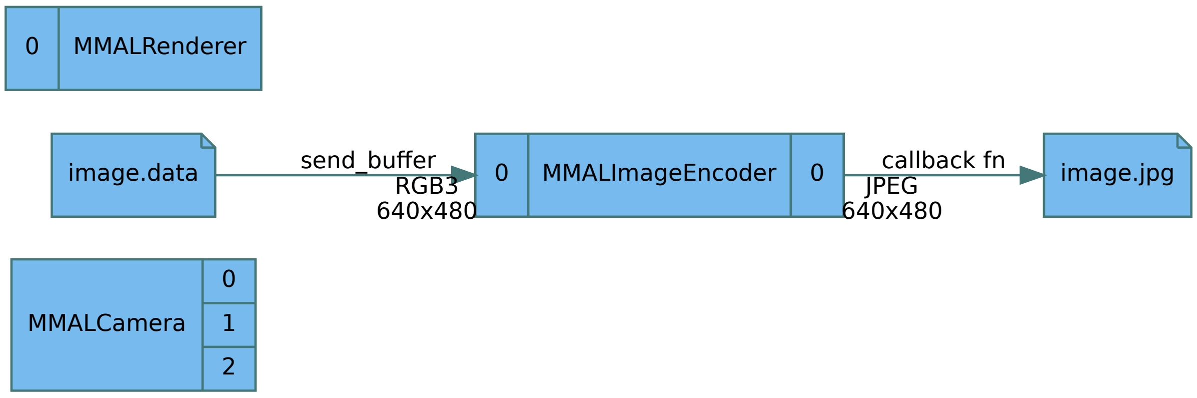
```
>>> rgb_data = io.open('image.data', 'rb')
>>> jpg_data = io.open('image.jpg', 'wb')
>>> def image_callback(port, buf):
...     jpg_data.write(buf.data)
...     return bool(buf.flags & mmal.MMAL_BUFFER_HEADER_FLAG_FRAME_END)
...
>>> encoder.outputs[0].enable(image_callback)
```

File Input (JPEG encoding)

How do we feed data to a component without a connection? We enable its input port with a dummy callback (we don't need to "do" anything on data input). Then we request buffers from its input port, fill them with data and send them back to the input port:

```
>>> encoder.inputs[0].enable(lamdba port, buf: True)
>>> buf = encoder.inputs[0].get_buffer()
>>> buf.data = rgb_data.read()
>>> encoder.inputs[0].send_buffer(buf)
>>> jpg_data.tell()
87830
>>> encoder.outputs[0].disable()
>>> encoder.inputs[0].disable()
>>> jpg_data.close()
>>> rgb_data.close()
```

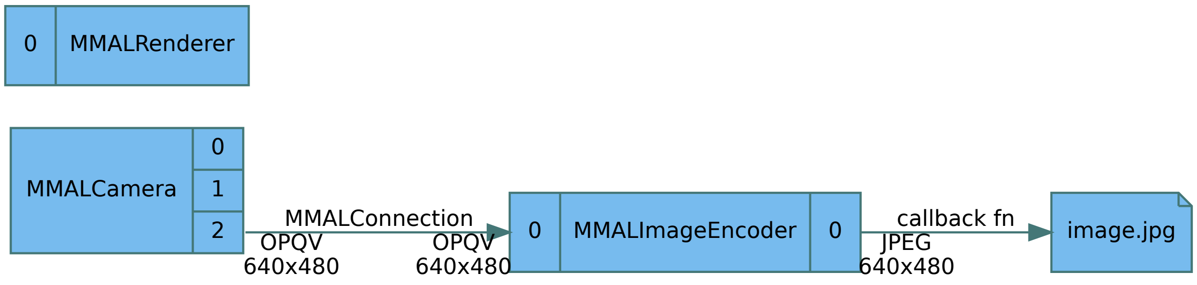
Congratulations again! You've just produced a hardware-accelerated JPEG encoding. The following illustrates the state of the system at the moment (note the camera and renderer still exist; they're just not connected to anything at the moment):



Now let's repeat the process but with the encoder attached to the still port on the camera directly. We can re-use our `image_callback` routine from earlier and just assign a different output file to `jpg_data`:

```
>>> encoder.connect(camera.outputs[2])
<MMALConnection "vc.ril.camera:out:2/vc.ril.image_encode:in:0">
>>> encoder.connection.enable()
>>> encoder.inputs[0]
<MMALVideoPort "vc.ril.image_encode:in:0(OPQV)": format=MMAL_FOURCC('OPQV')
buffers=10x128 frames=640x480@0fps>
>>> jpg_data = io.open('direct.jpg', 'wb')
>>> encoder.outputs[0].enable(image_callback)
>>> camera.outputs[2].params[mmal.MMAL_PARAMETER_CAPTURE] = True
>>> camera.outputs[2].params[mmal.MMAL_PARAMETER_CAPTURE] = False
>>> jpg_data.tell()
99328
>>> encoder.connection.disable()
>>> jpg_data.close()
```

Now the state of our system looks like this:



Threads & Synchronization

The one issue you may have noted is that `image_callback` is running in a background thread. If we were running our capture extremely fast our main thread might disable the capture before our callback had run. Ideally we want to activate capture, wait on some signal indicating that the callback has completed a single frame successfully, then disable capture. We can do this with the communications primitives from the standard `threading`[245] module:

```
>>> from threading import Event
>>> finished = Event()
>>> def image_callback(port, buf):
...     jpg_data.write(buf.data)
...     if buf.flags & mmal.MMAL_BUFFER_HEADER_FLAG_FRAME_END:
...         finished.set()
```

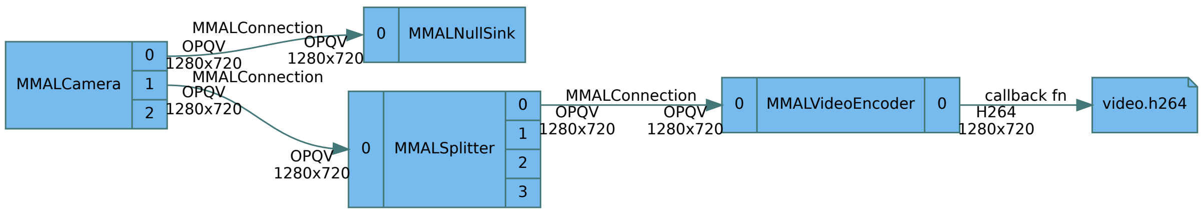
[245] https://docs.python.org/3.4/library/threading.html#module-threading

```
...         return True
...     return False
...
>>> def do_capture(filename='direct.jpg'):
...     global jpg_data
...     jpg_data = io.open(filename, 'wb')
...     finished.clear()
...     encoder.outputs[0].enable(image_callback)
...     camera.outputs[2].params[mmal.MMAL_PARAMETER_CAPTURE] = True
...     if not finished.wait(10):
...         raise Exception('capture timed out')
...     camera.outputs[2].params[mmal.MMAL_PARAMETER_CAPTURE] = False
...     encoder.outputs[0].disable()
...     jpg_data.close()
...
>>> do_capture()
```

The above example has several rough edges: globals, no proper clean-up in the case of an exception, etc. but by now you should be getting a pretty good idea of how picamera operates under the hood.

The major difference between picamera and a "typical" MMAL setup is that upon construction, the *PiCamera* (page 95) class constructs both a *MMALCamera* (page 174) (accessible as `PiCamera._camera`) *and* a *MMALSplitter* (page 175) (accessible as `PiCamera._splitter`). The splitter remains permanently attached to the camera's video port (output port 1). Furthermore, there's *always* something connected to the camera's preview port; by default it's a *MMALNullSink* (page 176) component which is switched with a *MMALRenderer* (page 176) when the preview is started.

Encoders are constructed and destroyed as required by calls to *capture()* (page 97), *start_recording()* (page 103), etc. The following illustrates a typical picamera pipeline whilst video recording without a preview:



Debugging Facilities

Before we move onto the pure Python components it's worth mentioning the debugging capabilities built into `mmalobj`. Firstly, most objects have useful `repr()`[246] outputs (in particular, it can be useful to simply evaluate a *MMALBuffer* (page 180) to see what flags it's got and how much data is stored in it). Also, there's the *print_pipeline()* (page 187) function. Give this a port and it'll dump a human-readable version of your pipeline leading up to that port:

```
>>> preview.inputs[0].enable(lambda port, buf: True)
>>> buf = preview.inputs[0].get_buffer()
>>> buf
<MMALBuffer object: flags=_____ length=0>
>>> buf.flags = mmal.MMAL_BUFFER_HEADER_FLAG_FRAME_END
>>> buf
<MMALBuffer object: flags=E____ length=0>
>>> buf.release()
>>> preview.inputs[0].disable()
>>> mo.print_pipeline(encoder.outputs[0])
 vc.ril.camera [2]                           [0] vc.ril.image_encode [0]
   encoding    OPQV-strips    -->    OPQV-strips      encoding       JPEG
      buf      10x128                     10x128         buf         1x307200
```

[246] https://docs.python.org/3.4/library/functions.html#repr

```
    bitrate     0bps                            0bps        bitrate         0bps
     frame      640x480@0fps            640x480@0fps         frame          0x0@0fps
```

Python Components

So far all the components we've looked at have been "real" MMAL components which is to say that they're implemented in C, and all talk to bits of the firmware running on the GPU. However, a frequent request has been to be able to modify frames from the camera before they reach the image or video encoder. The Python components are an attempt to make this request relatively simple to achieve from within Python.

The means by which this is achieved are inefficient (to say the least) so don't expect this to work with high resolutions or framerates. The `mmalobj` layer in picamera includes the concept of a "Python MMAL" component. To the user these components look a lot like the MMAL components you've been playing with above (*MMALCamera* (page 174), *MMALImageEncoder* (page 176), etc). They are instantiated in a similar manner, they have the same sort of ports, and they're connected using the same means as ordinary MMAL components.

Let's try this out by placing a transformation between the camera and a preview which will draw a cross over the frames going to the preview. For this we'll subclass *picamera.array.PiArrayTransform* (page 157). This derives from *MMALPythonComponent* (page 185) and provides the useful capability of providing the source and target buffers as numpy arrays containing RGB data:

```
>>> from picamera import array
>>> class Crosshair(array.PiArrayTransform):
...     def transform(self, source, target):
...         with source as sdata, target as tdata:
...             tdata[...] = sdata
...             tdata[240, :, :] = 0xff
...             tdata[:, 320, :] = 0xff
...         return False
...
>>> transform = Crosshair()
```

That's all there is to constructing a transform! This one is a bit crude in as much as the coordinates are hard-coded, and it's very simplistic, but it should illustrate the principle nicely. Let's connect it up between the camera and the renderer:

```
>>> transform.connect(camera)
<MMALPythonConnection "vc.ril.camera.out:0(RGB3)/py.component:in:0">
>>> preview.connect(transform)
<MMALPythonConnection "py.component:out:0/vc.ril.video_render:in:0(RGB3)">
>>> transform.connection.enable()
>>> preview.connection.enable()
>>> transform.enable()
```

At this point we should take a look at the pipeline to see what's been configured automatically:

```
>>> mo.print_pipeline(preview.inputs[0])
 vc.ril.camera [0]                             [0] py.transform [0]                    ␣
↪              [0] vc.ril.video_render
   encoding     RGB3            -->            RGB3    encoding    RGB3              -->
↪             RGB3      encoding
      buf       1x921600                   2x921600      buf       2x921600            ␣
↪        2x921600          buf
     frame      640x480@30fps         640x480@30fps     frame      640x480@30fps       ␣
↪    640x480@30fps         frame
```

Apparently the MMAL camera component is outputting RGB data (which is extremely large) to a "py.transform" component, which draws our cross-hair on the buffer and passes it onto the renderer again as RGB. This is part of the inefficiency alluded to above: RGB is a very large format (compared to I420 which is half its size, or OPQV which is tiny) so we're shuttling a *lot* of data around here. Expect this to drop frames at higher resolutions or framerates.

The other source of inefficiency isn't obvious from the debug output above which gives the impression that the "py.transform" component is actually part of the MMAL pipeline. In fact, this is a lie. Under the covers `mmalobj` installs an output callback on the camera's output port to feed data to the "py.transform" input port, uses a background thread to run the transform, then copies the results into buffers obtained from the preview's input port. In other words there's really *two* (very short) MMAL pipelines with a hunk of Python running in between them. If `mmalobj` does its job properly you shouldn't need to worry about this implementation detail but it's worth bearing in mind from the perspective of performance.

Performance Hints

Generally you want to your frame handlers to be *fast*. To avoid dropping frames they've got to run in less than a frame's time (e.g. 33ms at 30fps). Bear in mind that a significant amount of time is going to be spent shuttling the huge RGB frames around so you've actually got much less than 33ms available to you (how much will depend on the speed of your Pi, what resolution you're using, the framerate, etc).

Sometimes, performance can mean making unintuitive choices. For example, the Pillow library[247] (the main imaging library in Python these days) can construct images which share buffer memory (see `Image.frombuffer`), but only for the indexed (grayscale) and RGBA formats, not RGB. Hence, it can make sense to use RGBA (a format even larger than RGB) if only because it allows you to avoid copying any data when performing a composite.

Another trick is to realize that although YUV420 has different sized planes, it's often enough to manipulate the Y plane only. In that case you can treat the front of the buffer as an indexed image (remember that Pillow can share buffer memory with such images) and manipulate that directly. With tricks like these it's possible to perform multiple composites in realtime at 720p30 on a Pi3.

Here's a (heavily commented) variant of the cross-hair example above that uses the lower level *`MMALPythonComponent`* (page 185) class instead, and the Pillow library[248] to perform compositing on YUV420 in the manner just described:

```
from picamera import mmal, mmalobj as mo, PiCameraPortDisabled
from PIL import Image, ImageDraw
from signal import pause


class Crosshair(mo.MMALPythonComponent):
    def __init__(self):
        super(Crosshair, self).__init__(name='py.crosshair')
        self._crosshair = None
        self.inputs[0].supported_formats = mmal.MMAL_ENCODING_I420

    def _handle_frame(self, port, buf):
        # If we haven't drawn the crosshair yet, do it now and cache the
        # result so we don't bother doing it again
        if self._crosshair is None:
            self._crosshair = Image.new('L', port.framesize)
            draw = ImageDraw.Draw(self._crosshair)
            draw.line([
                (port.framesize.width // 2, 0),
                (port.framesize.width // 2, port.framesize.height)],
                fill=(255,), width=1)
            draw.line([
                (0, port.framesize.height // 2),
                (port.framesize.width , port.framesize.height // 2)],
                fill=(255,), width=1)
        # buf is the buffer containing the frame from our input port. First
        # we try and grab a buffer from our output port
```

[247] https://pillow.readthedocs.io/
[248] https://pillow.readthedocs.io/

```
        try:
            out = self.outputs[0].get_buffer(False)
        except PiCameraPortDisabled:
            # The port was disabled; that probably means we're shutting down so
            # return True to indicate we're all done and the component should
            # be disabled
            return True
        else:
            if out:
                # We've got a buffer (if we don't get a buffer here it most
                # likely means things are going too slow downstream so we'll
                # just have to skip this frame); copy the input buffer to the
                # output buffer
                out.copy_from(buf)
                # now grab a locked reference to the buffer's data by using
                # "with"
                with out as data:
                    # Construct a PIL Image over the Y plane at the front of
                    # the data and tell PIL the buffer is writeable
                    img = Image.frombuffer('L', port.framesize, data, 'raw', 'L',␣
↪0, 1)
                    img.readonly = False
                    img.paste(self._crosshair, (0, 0), mask=self._crosshair)
                # Send the output buffer back to the output port so it can
                # continue onward to whatever's downstream
                try:
                    self.outputs[0].send_buffer(out)
                except PiCameraPortDisabled:
                    # The port was disabled; same as before this probably means
                    # we're shutting down so return True to indicate we're done
                    return True
            # Return False to indicate that we want to continue processing
            # frames. If we returned True here, the component would be
            # disabled and no further buffers would be processed
            return False


camera = mo.MMALCamera()
preview = mo.MMALRenderer()
transform = Crosshair()

camera.outputs[0].framesize = '720p'
camera.outputs[0].framerate = 30
camera.outputs[0].commit()

transform.connect(camera)
preview.connect(transform)

transform.connection.enable()
preview.connection.enable()

preview.enable()
transform.enable()
camera.enable()

pause()
```

It's a sensible idea to perform any overlay rendering you want to do in a separate thread and then just handle compositing your overlay onto the frame in the `MMALPythonComponent._handle_frame()` (page 185) method. Anything you can do to avoid buffer copying is a bonus here.

Here's a final (rather large) demonstration that puts all these things together to construct a `MMALPythonComponent` (page 185) derivative with two purposes:

1. Render a partially transparent analogue clock in the top left of the frame.
2. Produces two equivalent I420 outputs; one for feeding to a preview renderer, and another to an encoder (we could use a proper MMAL splitter for this but this is a demonstration of how Python components can have multiple outputs too).

```
import io
import datetime as dt
from threading import Thread, Lock
from collections import namedtuple
from math import sin, cos, pi
from time import sleep

from picamera import mmal, mmalobj as mo, PiCameraPortDisabled
from PIL import Image, ImageDraw


class Coord(namedtuple('Coord', ('x', 'y'))):
    @classmethod
    def clock_arm(cls, radians):
        return Coord(sin(radians), -cos(radians))

    def __add__(self, other):
        try:
            return Coord(self.x + other[0], self.y + other[1])
        except TypeError:
            return Coord(self.x + other, self.y + other)

    def __sub__(self, other):
        try:
            return Coord(self.x - other[0], self.y - other[1])
        except TypeError:
            return Coord(self.x - other, self.y - other)

    def __mul__(self, other):
        try:
            return Coord(self.x * other[0], self.y * other[1])
        except TypeError:
            return Coord(self.x * other, self.y * other)

    def __floordiv__(self, other):
        try:
            return Coord(self.x // other[0], self.y // other[1])
        except TypeError:
            return Coord(self.x // other, self.y // other)

    # yeah, I could do the rest (truediv, radd, rsub, etc.) but there's no
    # need here...


class ClockSplitter(mo.MMALPythonComponent):
    def __init__(self):
        super(ClockSplitter, self).__init__(name='py.clock', outputs=2)
        self.inputs[0].supported_formats = {mmal.MMAL_ENCODING_I420}
        self._lock = Lock()
        self._clock_image = None
        self._clock_thread = None

    def enable(self):
        super(ClockSplitter, self).enable()
        self._clock_thread = Thread(target=self._clock_run)
        self._clock_thread.daemon = True
        self._clock_thread.start()
    def disable(self):
        super(ClockSplitter, self).disable()
        if self._clock_thread:
            self._clock_thread.join()
            self._clock_thread = None
            with self._lock:
                self._clock_image = None

    def _clock_run(self):
        # draw the clock face up front (no sense drawing that every time)
        origin = Coord(0, 0)
        size = Coord(100, 100)
        center = size // 2
        face = Image.new('L', size)
        draw = ImageDraw.Draw(face)
        draw.ellipse([origin, size - 1], outline=(255,))
        while self.enabled:
            # loop round rendering the clock hands on a copy of the face
            img = face.copy()
            draw = ImageDraw.Draw(img)
            now = dt.datetime.now()
            midnight = now.replace(
                hour=0, minute=0, second=0, microsecond=0)
            timestamp = (now - midnight).total_seconds()
            hour_pos = center + Coord.clock_arm(2 * pi * (timestamp % 43200 /␣
↪43200)) * 30
            minute_pos = center + Coord.clock_arm(2 * pi * (timestamp % 3600 /␣
↪3600)) * 45
            second_pos = center + Coord.clock_arm(2 * pi * (timestamp % 60 / 60))␣
↪* 45
            draw.line([center, hour_pos], fill=(200,), width=2)
            draw.line([center, minute_pos], fill=(200,), width=2)
            draw.line([center, second_pos], fill=(200,), width=1)
            # assign the rendered image to the internal variable
            with self._lock:
                self._clock_image = img
            sleep(0.2)

    def _handle_frame(self, port, buf):
        try:
            out1 = self.outputs[0].get_buffer(False)
            out2 = self.outputs[1].get_buffer(False)
        except PiCameraPortDisabled:
            return True
        if out1:
            # copy the input frame to the first output buffer
            out1.copy_from(buf)
            with out1 as data:
                # construct an Image using the Y plane of the output
                # buffer's data and tell PIL we can write to the buffer
                img = Image.frombuffer('L', port.framesize, data, 'raw', 'L', 0, 1)
                img.readonly = False
                with self._lock:
                    if self._clock_image:
                        img.paste(self._clock_image, (10, 10), self._clock_image)
            # if we've got a second output buffer replicate the first
            # buffer into it (note the difference between replicate and
            # copy_from)
            if out2:
                out2.replicate(out1)
            try:
                self.outputs[0].send_buffer(out1)
            except PiCameraPortDisabled:
                return True
        if out2:
            try:
                self.outputs[1].send_buffer(out2)
            except PiCameraPortDisabled:
                return True
        return False


def main(output_filename):
    camera = mo.MMALCamera()
    preview = mo.MMALRenderer()
    encoder = mo.MMALVideoEncoder()
    clock = ClockSplitter()
    target = mo.MMALPythonTarget(output_filename)

    # Configure camera output 0
    camera.outputs[0].framesize = (640, 480)
    camera.outputs[0].framerate = 24
    camera.outputs[0].commit()

    # Configure H.264 encoder
    encoder.outputs[0].format = mmal.MMAL_ENCODING_H264
    encoder.outputs[0].bitrate = 2000000
    encoder.outputs[0].commit()
    p = encoder.outputs[0].params[mmal.MMAL_PARAMETER_PROFILE]
    p.profile[0].profile = mmal.MMAL_VIDEO_PROFILE_H264_HIGH
    p.profile[0].level = mmal.MMAL_VIDEO_LEVEL_H264_41
    encoder.outputs[0].params[mmal.MMAL_PARAMETER_PROFILE] = p
    encoder.outputs[0].params[mmal.MMAL_PARAMETER_VIDEO_ENCODE_INLINE_HEADER] =␣
↪True
    encoder.outputs[0].params[mmal.MMAL_PARAMETER_INTRAPERIOD] = 30
    encoder.outputs[0].params[mmal.MMAL_PARAMETER_VIDEO_ENCODE_INITIAL_QUANT] = 22
    encoder.outputs[0].params[mmal.MMAL_PARAMETER_VIDEO_ENCODE_MAX_QUANT] = 22
    encoder.outputs[0].params[mmal.MMAL_PARAMETER_VIDEO_ENCODE_MIN_QUANT] = 22

    # Connect everything up and enable everything (no need to enable capture on
    # camera port 0)
    clock.inputs[0].connect(camera.outputs[0])
    preview.inputs[0].connect(clock.outputs[0])
    encoder.inputs[0].connect(clock.outputs[1])
    target.inputs[0].connect(encoder.outputs[0])
    target.connection.enable()
    encoder.connection.enable()
    preview.connection.enable()
    clock.connection.enable()
    target.enable()
    encoder.enable()
    preview.enable()
    clock.enable()
    try:
        sleep(10)
    finally:
        # Disable everything and tear down the pipeline
        target.disable()
        encoder.disable()
        preview.disable()
        clock.disable()
        target.inputs[0].disconnect()
        encoder.inputs[0].disconnect()
        preview.inputs[0].disconnect()
        clock.inputs[0].disconnect()
```

```
if __name__ == '__main__':
    main('output.h264')
```

IO Classes

The Python MMAL components include a couple of useful IO classes: `MMALSource` and `MMALTarget`. We could have used these instead of messing around with output callbacks in the sections above but it was worth exploring how those callbacks operated first (in order to comprehend how Python transforms operated).

Components

class `picamera.mmalobj.`**`MMALBaseComponent`**

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.

`disable()`

Disables the component.

`enable()`

Enable the component. When a component is enabled it will process data sent to its input port(s), sending the results to buffers on its output port(s). Components may be implicitly enabled by connections.

`control`

The *`MMALControlPort`* (page 176) control port of the component which can be used to configure most aspects of the component's behaviour.

`enabled`

Returns `True` if the component is currently enabled. Use *`enable()`* (page 174) and *`disable()`* (page 174) to control the component's state.

`inputs`

A sequence of *`MMALPort`* (page 177) objects representing the inputs of the component.

`outputs`

A sequence of *`MMALPort`* (page 177) objects representing the outputs of the component.

class `picamera.mmalobj.`**`MMALCamera`**

Bases: *`picamera.mmalobj.MMALBaseComponent`* (page 174)

Represents the MMAL camera component. This component has 0 input ports and 3 output ports. 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

Use the `MMAL_PARAMETER_CAMERA_CONFIG` parameter on the control port to obtain and manipulate the camera's configuration.

`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`* (page 174) class will pick the latest annotation structure supported by the current firmware but you can select older revisions with `annotate_rev` (page 174) for other purposes (e.g. testing).

class `picamera.mmalobj.`**`MMALCameraInfo`**

Bases: `picamera.mmalobj.MMALBaseComponent` (page 174)

Represents the MMAL camera-info component. Query the `MMAL_PARAMETER_CAMERA_INFO` parameter on the control port to obtain information about the connected camera module.

`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` (page 175) 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.`**`MMALComponent`**

Bases: `picamera.mmalobj.MMALBaseComponent` (page 174)

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.

`connect`(*source*, ***options*)

Connects the input port of this component to the specified *source* `MMALPort` (page 177) or `MMALPythonPort` (page 182). Alternatively, as a convenience (primarily intended for command line experimentation; don't use this in scripts), *source* can be another component in which case the first unconnected output port will be selected as *source*.

Keyword arguments will be passed along to the connection constructor. See `MMALConnection` (page 179) and `MMALPythonConnection` (page 186) for further information.

`disconnect`()

Destroy the connection between this component's input port and the upstream component.

`connection`

The `MMALConnection` (page 179) or `MMALPythonConnection` (page 186) object linking this component to the upstream component.

class `picamera.mmalobj.`**`MMALSplitter`**

Bases: `picamera.mmalobj.MMALComponent` (page 175)

Represents the MMAL splitter component. This component has 1 input port and 4 output ports which all generate duplicates of buffers passed to the input port.

class `picamera.mmalobj.`**`MMALResizer`**

Bases: `picamera.mmalobj.MMALComponent` (page 175)

Represents the MMAL VPU resizer component. This component has 1 input port and 1 output port. This supports resizing via the VPU. This is not as efficient as `MMALISPResizer` (page 175) but is available on all firmware verions. The output port can (and usually should) have a different frame size to the input port.

class `picamera.mmalobj.`**`MMALISPResizer`**

Bases: `picamera.mmalobj.MMALComponent` (page 175)

Represents the MMAL ISP resizer component. This component has 1 input port and 1 output port, and supports resizing via the VideoCore ISP, along with conversion of numerous formats into numerous other formats (e.g. OPAQUE to RGB, etc). This is more efficient than `MMALResizer` (page 175) but is only available on later firmware versions.

class `picamera.mmalobj.`**`MMALEncoder`**

Bases: `picamera.mmalobj.MMALComponent` (page 175)

Represents a generic MMAL encoder. This is an abstract base class.

class `picamera.mmalobj.`**`MMALVideoEncoder`**

Bases: `picamera.mmalobj.MMALEncoder` (page 175)

Represents the MMAL video encoder component. This component has 1 input port and 1 output port. The output port is usually configured with `MMAL_ENCODING_H264` or `MMAL_ENCODING_MJPEG`.

class `picamera.mmalobj.`**`MMALImageEncoder`**
Bases: *`picamera.mmalobj.MMALEncoder`* (page 175)

Represents the MMAL image encoder component. This component has 1 input port and 1 output port. The output port is typically configured with `MMAL_ENCODING_JPEG` but can also use `MMAL_ENCODING_PNG`, `MMAL_ENCODING_GIF`, etc.

class `picamera.mmalobj.`**`MMALDecoder`**
Bases: *`picamera.mmalobj.MMALComponent`* (page 175)

Represents a generic MMAL decoder. This is an abstract base class.

class `picamera.mmalobj.`**`MMALVideoDecoder`**
Bases: *`picamera.mmalobj.MMALDecoder`* (page 176)

Represents the MMAL video decoder component. This component has 1 input port and 1 output port. The input port is usually configured with `MMAL_ENCODING_H264` or `MMAL_ENCODING_MJPEG`.

class `picamera.mmalobj.`**`MMALImageDecoder`**
Bases: *`picamera.mmalobj.MMALDecoder`* (page 176)

Represents the MMAL iamge decoder component. This component has 1 input port and 1 output port. The input port is usually configured with `MMAL_ENCODING_JPEG`.

class `picamera.mmalobj.`**`MMALRenderer`**
Bases: *`picamera.mmalobj.MMALComponent`* (page 175)

Represents the MMAL renderer component. This component has 1 input port and 0 output ports. It is used to implement the camera preview and overlays.

class `picamera.mmalobj.`**`MMALNullSink`**
Bases: *`picamera.mmalobj.MMALComponent`* (page 175)

Represents the MMAL null-sink component. This component has 1 input port and 0 output ports. It is used to keep the preview port “alive” (and thus calculating white-balance and exposure) when the camera preview is not required.

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`* (page 176) (or descendent) and an *`MMALBuffer`* (page 180) instance. Any return value will be ignored.

`capabilities`
The capabilities of the port. A bitfield of the following:

- MMAL_PORT_CAPABILITY_PASSTHROUGH
- MMAL_PORT_CAPABILITY_ALLOCATION
- MMAL_PORT_CAPABILITY_SUPPORTS_EVENT_FORMAT_CHANGE

`enabled`
Returns a `bool`[249] indicating whether the port is currently enabled. Unlike other classes, this is a read-only property. Use *`enable()`* (page 176) and *`disable()`* (page 176) to modify the value.

[249] https://docs.python.org/3.4/library/functions.html#bool

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* (page 178) class.

type
The type of the port. One of:

- MMAL_PORT_TYPE_OUTPUT
- MMAL_PORT_TYPE_INPUT
- MMAL_PORT_TYPE_CONTROL
- MMAL_PORT_TYPE_CLOCK

class picamera.mmalobj.**MMALPort**(*port*, *opaque_subformat='OPQV'*)
Bases: *picamera.mmalobj.MMALControlPort* (page 176)

Represents an MMAL port with properties to configure and update the port's format. This is the base class of *MMALVideoPort* (page 178), *MMALAudioPort* (page 178), and *MMALSubPicturePort* (page 178).

commit()
Commits the port's configuration and automatically updates the number and size of associated buffers according to the recommendations of the MMAL library. This is typically called after adjusting the port's format and/or associated settings (like width and height for video ports).

connect(*other*, ***options*)
Connect this port to the *other* *MMALPort* (page 177) (or *MMALPythonPort* (page 182)). The type and configuration of the connection will be automatically selected.

Various connection *options* can be specified as keyword arguments. These will be passed onto the *MMALConnection* (page 179) or *MMALPythonConnection* (page 186) constructor that is called (see those classes for an explanation of the available options).

copy_from(*source*)
Copies the port's *format* (page 178) from the *source* *MMALControlPort* (page 176).

disable()
Disable the port.

disconnect()
Destroy the connection between this port and another 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* (page 176) (or descendent) and an *MMALBuffer* (page 180) 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.

flush()
Flush the port.

get_buffer(*block=True*, *timeout=None*)
Returns a *MMALBuffer* (page 180) from the associated *pool* (page 178). *block* and *timeout* act as they do in the corresponding *MMALPool.get_buffer()* (page 182).

send_buffer(*buf*)
Send *MMALBuffer* (page 180) *buf* to the port.

bitrate
Retrieves or sets the bitrate limit for the port's format.

buffer_count
The number of buffers allocated (or to be allocated) to the port. The `mmalobj` layer automatically configures this based on recommendations from the MMAL library.

buffer_size
The size of buffers allocated (or to be allocated) to the port. The size of buffers is typically dictated by the port's format. The `mmalobj` layer automatically configures this based on recommendations from the MMAL library.

connection
If this port is connected to another, this property holds the *MMALConnection* (page 179) or *MMALPythonConnection* (page 186) object which represents that connection. If this port is not connected, this property is `None`.

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()` (page 177) 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 (page 179) uses this information when negotiating formats for a connection between two ports.

pool
Returns the *MMALPool* (page 182) associated with the buffer, if any.

supported_formats
Retrieves a sequence of supported encodings on this port.

class `picamera.mmalobj.`**MMALVideoPort**(*port*, *opaque_subformat='OPQV'*)
Bases: *picamera.mmalobj.MMALPort* (page 177)

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()` (page 177) 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()` (page 177) to make the changes effective.

class `picamera.mmalobj.`**MMALSubPicturePort**(*port*, *opaque_subformat='OPQV'*)
Bases: *picamera.mmalobj.MMALPort* (page 177)

Represents an MMAL port used to pass sub-picture (caption) data.

class `picamera.mmalobj.`**MMALAudioPort**(*port*, *opaque_subformat='OPQV'*)
Bases: *picamera.mmalobj.MMALPort* (page 177)

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* (page 177) 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`[250] 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.

Connections

class `picamera.mmalobj.`**`MMALBaseConnection`**(*source*, *target*, *formats=()*)

Abstract base class for *`MMALConnection`* (page 179) and *`MMALPythonConnection`* (page 186). Handles weakrefs to the source and target ports, and format negotiation. All other connection details are handled by the descendent classes.

`source`

The source *`MMALPort`* (page 177) or *`MMALPythonPort`* (page 182) of the connection.

`target`

The target *`MMALPort`* (page 177) or *`MMALPythonPort`* (page 182) of the connection.

class `picamera.mmalobj.`**`MMALConnection`**(*source*, *target*, *formats=default_formats*, *callback=None*)

Bases: *`picamera.mmalobj.MMALBaseConnection`* (page 179)

Represents an MMAL internal connection between two components. The constructor accepts arguments providing the *source* *`MMALPort`* (page 177) and *target* *`MMALPort`* (page 177).

The *formats* parameter specifies an iterable of formats (in preference order) that the connection may attempt when negotiating formats between the two ports. If this is `None`, or an empty iterable, no negotiation will take place and the source port's format will simply be copied to the target port. Otherwise, the iterable will be worked through in order until a format acceptable to both ports is discovered.

Note: The default *formats* list starts with OPAQUE; the class understands the different OPAQUE sub-formats (see *MMAL* (page 78) for more information) and will only select OPAQUE if compatible sub-formats can be used on both ports.

The *callback* parameter can optionally specify a callable which will be executed for each buffer that traverses the connection (providing an opportunity to manipulate or drop that buffer). If specified, it must be a callable which accepts two parameters: the *`MMALConnection`* (page 179) object sending the data, and the *`MMALBuffer`* (page 180) object containing data. The callable may optionally manipulate the *`MMALBuffer`* (page 180) and return it to permit it to continue traversing the connection, or return `None` in which case the buffer will be released.

Note: There is a significant performance penalty for specifying a callback between MMAL components as it requires buffers to be copied from the GPU's memory to the CPU's memory and back again.

`default_formats` = **(MMAL_ENCODING_OPAQUE, MMAL_ENCODING_I420, MMAL_ENCODING_RGB24,**

Class attribute defining the default formats used to negotiate connections between MMAL components.

`disable`()

Disables the connection.

`enable`()

Enable the connection. When a connection is enabled, data is continually transferred from the output port of the source to the input port of the target component.

[250] https://docs.python.org/3.4/library/fractions.html#fractions.Fraction

enabled
Returns `True` if the connection is enabled. Use *enable()* (page 179) and *disable()* (page 179) to control the state of the connection.

Buffers

class picamera.mmalobj.**MMALBuffer**(*buf*)
Represents an MMAL buffer header. This is usually constructed from the buffer header pointer and is largely supplied to make working with the buffer's data a bit simpler. Using the buffer as a context manager implicitly locks the buffer's memory and returns the `ctypes`[251] buffer object itself:

```
def callback(port, buf):
    with buf as data:
        # data is a ctypes uint8 array with size entries
        print(len(data))
```

Alternatively you can use the *data* (page 181) property directly, which returns and modifies the buffer's data as a `bytes`[252] object (note this is generally slower than using the buffer object unless you are simply replacing the entire buffer):

```
def callback(port, buf):
    # the buffer contents as a byte-string
    print(buf.data)
```

acquire()
Acquire a reference to the buffer. This will prevent the buffer from being recycled until *release()* (page 180) is called. This method can be called multiple times in which case an equivalent number of calls to *release()* (page 180) must be made before the buffer will actually be released.

copy_from(*source*)
Copies all fields (including data) from the *source* *MMALBuffer* (page 180). This buffer must have sufficient *size* (page 181) to store *length* (page 181) bytes from the *source* buffer. This method implicitly sets *offset* (page 181) to zero, and *length* (page 181) to the number of bytes copied.

Note: This is fundamentally different to the operation of the *replicate()* (page 180) method. It is much slower, but afterward the copied buffer is entirely independent of the *source*.

copy_meta(*source*)
Copy meta-data from the *source* *MMALBuffer* (page 180); specifically this copies all buffer fields with the exception of *data* (page 181), *length* (page 181) and *offset* (page 181).

release()
Release a reference to the buffer. This is the opposing call to *acquire()* (page 180). Once all references have been released, the buffer will be recycled.

replicate(*source*)
Replicates the *source* *MMALBuffer* (page 180). This copies all fields from the *source* buffer, including the internal *data* (page 181) pointer. In other words, after replication this buffer and the *source* buffer will share the same block of memory for *data*.

The *source* buffer will also be referenced internally by this buffer and will only be recycled once this buffer is released.

Note: This is fundamentally different to the operation of the *copy_from()* (page 180) method. It is much faster, but imposes the burden that two buffers now share data (the *source* cannot be released

[251] https://docs.python.org/3.4/library/ctypes.html#module-ctypes
[252] https://docs.python.org/3.4/library/functions.html#bytes

until the replicant has been released).

reset()
Resets all buffer header fields to default values.

command
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
The data held in the buffer as a `bytes`[253] string. You can set this attribute to modify the data in the buffer. Acceptable values are anything that supports the buffer protocol, and which contains *size* (page 181) bytes or less. Setting this attribute implicitly modifies the *length* (page 181) attribute to the length of the specified value and sets *offset* (page 181) to zero.

Note: Accessing a buffer's data via this attribute is relatively slow (as it copies the buffer's data to/from Python objects). See the *MMALBuffer* (page 180) documentation for details of a faster (but more complex) method.

dts
The decoding timestamp (DTS) of the buffer, as an integer number of microseconds or `MMAL_TIME_UNKNOWN`.

flags
The flags set in the buffer's meta-data, returned as a bitmapped integer. Typical flags include:

- `MMAL_BUFFER_HEADER_FLAG_EOS` – end of stream
- `MMAL_BUFFER_HEADER_FLAG_FRAME_START` – start of frame data
- `MMAL_BUFFER_HEADER_FLAG_FRAME_END` – end of frame data
- `MMAL_BUFFER_HEADER_FLAG_KEYFRAME` – frame is a key-frame
- `MMAL_BUFFER_HEADER_FLAG_FRAME` – frame data
- `MMAL_BUFFER_HEADER_FLAG_CODECSIDEINFO` – motion estimatation data

length
The length of data held in the buffer. Must be less than or equal to the allocated size of data held in *size* (page 181) minus the data *offset* (page 181). This attribute can be used to effectively blank the buffer by setting it to zero.

offset
The offset from the start of the buffer at which the data actually begins. Defaults to 0. If this is set to a value which would force the current *length* (page 181) off the end of the buffer's *size* (page 181), then *length* (page 181) will be decreased automatically.

pts
The presentation timestamp (PTS) of the buffer, as an integer number of microseconds or `MMAL_TIME_UNKNOWN`.

size
Returns the length of the buffer's data area in bytes. This will be greater than or equal to *length* (page 181) and is fixed in value.

class picamera.mmalobj.**MMALQueue**(*queue*)
Represents an MMAL buffer queue. Buffers can be added to the queue with the *put()* (page 182) method, and retrieved from the queue (with optional wait timeout) with the *get()* (page 181) method.

get(*block=True*, *timeout=None*)
Get the next buffer from the queue. If *block* is `True` (the default) and *timeout* is `None` (the default) then the method will block until a buffer is available. Otherwise *timeout* is the maximum time to wait

[253] https://docs.python.org/3.4/library/functions.html#bytes

(in seconds) for a buffer to become available. If a buffer is not available before the timeout expires, the method returns `None`.

Likewise, if *block* is `False` and no buffer is immediately available then `None` is returned.

`put` (*buf*)
Place `MMALBuffer` (page 180) *buf* at the back of the queue.

`put_back` (*buf*)
Place `MMALBuffer` (page 180) *buf* at the front of the queue. This is used when a buffer was removed from the queue but needs to be put back at the front where it was originally taken from.

class `picamera.mmalobj.`**`MMALPool`** (*pool*)
Represents an MMAL pool containing `MMALBuffer` (page 180) objects. All active ports are associated with a pool of buffers, and a queue. Instances can be treated as a sequence of `MMALBuffer` (page 180) objects but this is only recommended for debugging purposes; otherwise, use the `get_buffer()` (page 182), `send_buffer()` (page 182), and `send_all_buffers()` (page 182) methods which work with the encapsulated `MMALQueue` (page 181).

`get_buffer` (*block=True*, *timeout=None*)
Get the next buffer from the pool's queue. See `MMALQueue.get()` (page 181) for the meaning of the parameters.

`resize` (*new_count*, *new_size*)
Resizes the pool to contain *new_count* buffers with *new_size* bytes allocated to each buffer.

new_count must be 1 or more (you cannot resize a pool to contain no headers). However, *new_size* can be 0 which causes all payload buffers to be released.

Warning: If the pool is associated with a port, the port must be disabled when resizing the pool.

`send_all_buffers` (*port*, *block=True*, *timeout=None*)
Send all buffers from the queue to *port*. *block* and *timeout* act as they do in `get_buffer()` (page 182). If no buffer is available (for the values of *block* and *timeout*, `PiCameraMMALError` (page 142) is raised).

`send_buffer` (*port*, *block=True*, *timeout=None*)
Get a buffer from the pool's queue and send it to *port*. *block* and *timeout* act as they do in `get_buffer()` (page 182). If no buffer is available (for the values of *block* and *timeout*, `PiCameraMMALError` (page 142) is raised).

`queue`
The `MMALQueue` (page 181) associated with the pool.

class `picamera.mmalobj.`**`MMALPortPool`** (*port*)
Bases: `picamera.mmalobj.MMALPool` (page 182)

Construct an MMAL pool for the number and size of buffers required by the `MMALPort` (page 177) *port*.

`send_all_buffers` (*port=None*, *block=True*, *timeout=None*)
Send all buffers from the pool to *port* (or the port the pool is associated with by default). *block* and *timeout* act as they do in `MMALPool.get_buffer()` (page 182).

`send_buffer` (*port=None*, *block=True*, *timeout=None*)
Get a buffer from the pool and send it to *port* (or the port the pool is associated with by default). *block* and *timeout* act as they do in `MMALPool.get_buffer()` (page 182).

Python Extensions

class `picamera.mmalobj.`**`MMALPythonPort`** (*owner*, *port_type*, *index*)
Implements ports for Python-based MMAL components.

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).

connect(*other*, ***options*)
Connect this port to the *other* `MMALPort` (page 177) (or `MMALPythonPort` (page 182)). The type and configuration of the connection will be automatically selected.

Various connection options can be specified as keyword arguments. These will be passed onto the `MMALConnection` (page 179) or `MMALPythonConnection` (page 186) constructor that is called (see those classes for an explanation of the available options).

copy_from(*source*)
Copies the port's `format` (page 183) from the *source* `MMALControlPort` (page 176).

disable()
Disable the port.

disconnect()
Destroy the connection between this port and another 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` (page 176) (or descendent) and an `MMALBuffer` (page 180) instance. Any return value will be ignored.

get_buffer(*block=True*, *timeout=None*)
Returns a `MMALBuffer` (page 180) from the associated `pool` (page 184). *block* and *timeout* act as they do in the corresponding `MMALPool.get_buffer()` (page 182).

send_buffer(*buf*)
Send `MMALBuffer` (page 180) *buf* to the port.

bitrate
Retrieves or sets the bitrate limit for the port's format.

buffer_count
The number of buffers allocated (or to be allocated) to the port. The default is 2 but more may be required in the case of long pipelines with replicated buffers.

buffer_size
The size of buffers allocated (or to be allocated) to the port. The size of buffers defaults to a value dictated by the port's format.

capabilities
The capabilities of the port. A bitfield of the following:

- MMAL_PORT_CAPABILITY_PASSTHROUGH
- MMAL_PORT_CAPABILITY_ALLOCATION
- MMAL_PORT_CAPABILITY_SUPPORTS_EVENT_FORMAT_CHANGE

connection
If this port is connected to another, this property holds the `MMALConnection` (page 179) or `MMALPythonConnection` (page 186) object which represents that connection. If this port is not connected, this property is `None`.

enabled
Returns a `bool`[254] indicating whether the port is currently enabled. Unlike other classes, this is a read-only property. Use `enable()` (page 183) and `disable()` (page 183) to modify the value.

[254] https://docs.python.org/3.4/library/functions.html#bool

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).

framerate
Retrieves or sets the framerate of the port's video frames in fps.

framesize
Retrieves or sets the size of the source'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).

index
Returns an integer indicating the port's position within its owning list (inputs, outputs, etc.)

pool
Returns the *MMALPool* (page 182) associated with the buffer, if any.

supported_formats
Retrieves or sets the set of valid formats for this port. The set must always contain at least one valid format. A single format can be specified; it will be converted implicitly to a singleton set.

If the current port *format* (page 183) is not a member of the new set, no error is raised. An error will be raised when *commit()* (page 182) is next called if *format* (page 183) is still not a member of the set.

type
The type of the port. One of:

- MMAL_PORT_TYPE_OUTPUT
- MMAL_PORT_TYPE_INPUT
- MMAL_PORT_TYPE_CONTROL
- MMAL_PORT_TYPE_CLOCK

class picamera.mmalobj.**MMALPythonBaseComponent**
Base class for Python-implemented MMAL components. This class provides the *_commit_port()* (page 184) method used by descendents to control their ports' behaviour, and the *enabled* (page 184) property. However, it is unlikely that users will want to sub-class this directly. See *MMALPythonComponent* (page 185) for a more useful starting point.

_commit_port(*port*)
Called by ports when their format is committed. Descendents may override this to reconfigure output ports when input ports are committed, or to raise errors if the new port configuration is unacceptable.

> **Warning:** This method must *not* reconfigure input ports when called; however it can reconfigure *output* ports when input ports are committed.

close()
Close the component and release all its resources. After this is called, most methods will raise exceptions if called.

disable()
Disables the component.

enable()
Enable the component. When a component is enabled it will process data sent to its input port(s), sending the results to buffers on its output port(s). Components may be implicitly enabled by connections.

control
The *MMALControlPort* (page 176) control port of the component which can be used to configure most aspects of the component's behaviour.

enabled
Returns `True` if the component is currently enabled. Use *`enable()`* (page 184) and *`disable()`* (page 184) to control the component's state.

inputs
A sequence of *`MMALPort`* (page 177) objects representing the inputs of the component.

outputs
A sequence of *`MMALPort`* (page 177) objects representing the outputs of the component.

class `picamera.mmalobj.`**`MMALPythonComponent`**(*name='py.component'*, *outputs=1*)
Bases: *`picamera.mmalobj.MMALPythonBaseComponent`* (page 184)

Provides a Python-based MMAL component with a *name*, a single input and the specified number of *outputs* (default 1). The *`connect()`* (page 186) and *`disconnect()`* (page 186) methods can be used to establish or break a connection from the input port to an upstream component.

Typically descendents will override the *`_handle_frame()`* (page 185) method to respond to buffers sent to the input port, and will set *`MMALPythonPort.supported_formats`* (page 184) in the constructor to define the formats that the component will work with.

`_commit_port`(*port*)
Overridden to to copy the input port's configuration to the output port(s), and to ensure that the output port(s)' format(s) match the input port's format.

`_handle_end_of_stream`(*port*, *buf*)
Handles end-of-stream notifications passed to the component (where *`MMALBuffer.command`* (page 181) is set to MMAL_EVENT_EOS).

The default implementation does nothing but return `True` (indicating that processing should halt). Override this in descendents to respond to the end of stream.

The *port* parameter is the port into which the event arrived.

`_handle_error`(*port*, *buf*)
Handles error notifications passed to the component (where *`MMALBuffer.command`* (page 181) is set to MMAL_EVENT_ERROR).

The default implementation does nothing but return `True` (indicating that processing should halt). Override this in descendents to respond to error events.

The *port* parameter is the port into which the event arrived.

`_handle_format_changed`(*port*, *buf*)
Handles format change events passed to the component (where *`MMALBuffer.command`* (page 181) is set to MMAL_EVENT_FORMAT_CHANGED).

The default implementation re-configures the input port of the component and emits the event on all output ports for downstream processing. Override this method if you wish to do something else in response to format change events.

The *port* parameter is the port into which the event arrived, and *buf* contains the event itself (a MMAL_EVENT_FORMAT_CHANGED_T structure). Use `mmal_event_format_changed_get` on the buffer's data to extract the event.

`_handle_frame`(*port*, *buf*)
Handles frame data buffers (where *`MMALBuffer.command`* (page 181) is set to 0).

Typically, if the component has output ports, the method is expected to fetch a buffer from the output port(s), write data into them, and send them back to their respective ports.

Return values are as for normal event handlers (`True` when no more buffers are expected, `False` otherwise).

`_handle_parameter_changed`(*port*, *buf*)
Handles parameter change events passed to the component (where *`MMALBuffer.command`* (page 181) is set to MMAL_EVENT_PARAMETER_CHANGED).

The default implementation does nothing but return `False` (indicating that processing should continue). Override this in descendents to respond to parameter changes.

The *port* parameter is the port into which the event arrived, and *buf* contains the event itself (a MMAL_EVENT_PARAMETER_CHANGED_T structure).

`connect`(*source*, ***options*)
Connects the input port of this component to the specified *source* `MMALPort` (page 177) or `MMALPythonPort` (page 182). Alternatively, as a convenience (primarily intended for command line experimentation; don't use this in scripts), *source* can be another component in which case the first unconnected output port will be selected as *source*.

Keyword arguments will be passed along to the connection constructor. See `MMALConnection` (page 179) and `MMALPythonConnection` (page 186) for further information.

`disconnect`()
Destroy the connection between this component's input port and the upstream component.

`connection`
The `MMALConnection` (page 179) or `MMALPythonConnection` (page 186) object linking this component to the upstream component.

class `picamera.mmalobj.`**`MMALPythonConnection`**(*source*, *target*, *formats=default_formats*, *callback=None*)
Bases: `picamera.mmalobj.MMALBaseConnection` (page 179)

Represents a connection between an `MMALPythonBaseComponent` (page 184) and a `MMALBaseComponent` (page 174) or another `MMALPythonBaseComponent` (page 184). The constructor accepts arguments providing the *source* `MMALPort` (page 177) (or `MMALPythonPort` (page 182)) and *target* `MMALPort` (page 177) (or `MMALPythonPort` (page 182)).

The *formats* parameter specifies an iterable of formats (in preference order) that the connection may attempt when negotiating formats between the two ports. If this is `None`, or an empty iterable, no negotiation will take place and the source port's format will simply be copied to the target port. Otherwise, the iterable will be worked through in order until a format acceptable to both ports is discovered.

The *callback* parameter can optionally specify a callable which will be executed for each buffer that traverses the connection (providing an opportunity to manipulate or drop that buffer). If specified, it must be a callable which accepts two parameters: the `MMALPythonConnection` (page 186) object sending the data, and the `MMALBuffer` (page 180) object containing data. The callable may optionally manipulate the `MMALBuffer` (page 180) and return it to permit it to continue traversing the connection, or return `None` in which case the buffer will be released.

`default_formats` = **(MMAL_ENCODING_I420, MMAL_ENCODING_RGB24, MMAL_ENCODING_BGR24, M**
Class attribute defining the default formats used to negotiate connections between Python and and MMAL components, in preference order. Note that OPAQUE is not present in contrast with the default formats in `MMALConnection` (page 179).

`disable`()
Disables the connection.

`enable`()
Enable the connection. When a connection is enabled, data is continually transferred from the output port of the source to the input port of the target component.

`enabled`
Returns `True` if the connection is enabled. Use `enable()` (page 186) and `disable()` (page 186) to control the state of the connection.

class `picamera.mmalobj.`**`MMALPythonSource`**(*input*)
Bases: `picamera.mmalobj.MMALPythonBaseComponent` (page 184)

Provides a source for other `MMALComponent` (page 175) instances. The specified *input* is read in chunks the size of the configured output buffer(s) until the input is exhausted. The `wait()` (page 187) method can be used to block until this occurs. If the output buffer is configured to use a full-frame unencoded format (like I420 or RGB), frame-end flags will be automatically generated by the source. When the input is exhausted an empty buffer with the End Of Stream (EOS) flag will be sent.

The component provides all picamera's usual IO-handling characteristics; if *input* is a string, a file with that name will be opened as the input and closed implicitly when the component is closed. Otherwise, the input will not be closed implicitly (the component did not open it, so the assumption is that closing *input* is the caller's responsibility). If *input* is an object with a `read` method it is assumed to be a file-like object and is used as is. Otherwise, *input* is assumed to be a readable object supporting the buffer protocol (which is wrapped in a `BufferIO` stream).

`wait` (*timeout=None*)
Wait for the source to send all bytes from the specified input. If *timeout* is specified, it is the number of seconds to wait for completion. The method returns `True` if the source completed within the specified timeout and `False` otherwise.

class `picamera.mmalobj.`**`MMALPythonTarget`** (*output*, *done=1*)
Bases: *`picamera.mmalobj.MMALPythonComponent`* (page 185)

Provides a simple component that writes all received buffers to the specified *output* until a frame with the *done* flag is seen (defaults to MMAL_BUFFER_HEADER_FLAG_EOS indicating End Of Stream).

The component provides all picamera's usual IO-handling characteristics; if *output* is a string, a file with that name will be opened as the output and closed implicitly when the component is closed. Otherwise, the output will not be closed implicitly (the component did not open it, so the assumption is that closing *output* is the caller's responsibility). If *output* is an object with a `write` method it is assumed to be a file-like object and is used as is. Otherwise, *output* is assumed to be a writeable object supporting the buffer protocol (which is wrapped in a `BufferIO` stream).

`wait` (*timeout=None*)
Wait for the output to be "complete" as defined by the constructor's *done* parameter. If *timeout* is specified it is the number of seconds to wait for completion. The method returns `True` if the target completed within the specified timeout and `False` otherwise.

Debugging

The following functions are useful for quickly dumping the state of a given MMAL pipeline:

`picamera.mmalobj.`**`debug_pipeline`** (*port*)
Given an *`MMALVideoPort`* (page 178) *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`* (page 178) (the specified output port)
- *`MMALEncoder`* (page 175) (the encoder which owns the output port)
- *`MMALVideoPort`* (page 178) (the encoder's input port)
- *`MMALConnection`* (page 179) (the connection between the splitter and encoder)
- *`MMALVideoPort`* (page 178) (the splitter's output port)
- *`MMALSplitter`* (page 175) (the splitter on the camera's video port)
- *`MMALVideoPort`* (page 178) (the splitter's input port)
- *`MMALConnection`* (page 179) (the connection between the splitter and camera)
- *`MMALVideoPort`* (page 178) (the camera's video port)
- *`MMALCamera`* (page 174) (the camera component)

`picamera.mmalobj.`**`print_pipeline`** (*port*)
Prints a human readable representation of the pipeline feeding the specified *`MMALVideoPort`* (page 178) *port*.

Note: It is also worth noting that most classes, in particular *MMALVideoPort* (page 178) and *MMALBuffer* (page 180) have useful `repr()`[255] outputs which can be extremely useful with simple `print()`[256] calls for debugging.

Utility Functions

The following functions are provided to ease certain common operations in the picamera library. Users of `mmalobj` may find them handy in various situations:

`picamera.mmalobj.`**`open_stream`**`(`*stream*, *output=True*, *buffering=65536*`)`

This is the core of picamera's IO-semantics. It returns a tuple of a file-like object and a bool indicating whether the stream requires closing once the caller is finished with it.

- If *stream* is a string, it is opened as a file object (with mode 'wb' if *output* is `True`, and the specified amount of *bufffering*). In this case the function returns `(stream, True)`.
- If *stream* is a stream with a `write` method, it is returned as `(stream, False)`.
- Otherwise *stream* is assumed to be a writeable buffer and is wrapped with `BufferIO`. The function returns `(stream, True)`.

`picamera.mmalobj.`**`close_stream`**`(`*stream*, *opened*`)`

If *opened* is `True`, then the `close` method of *stream* will be called. Otherwise, the function will attempt to call the `flush` method on *stream* (if one exists). This function essentially takes the output of *open_stream()* (page 188) and finalizes the result.

`picamera.mmalobj.`**`to_resolution`**`(`*value*`)`

Converts *value* which may be a (width, height) tuple or a string containing a representation of a resolution (e.g. "1024x768" or "1080p") to a (width, height) tuple.

`picamera.mmalobj.`**`to_rational`**`(`*value*`)`

Converts *value* (which can be anything accepted by `to_fraction()`) to an MMAL_RATIONAL_T structure.

`picamera.mmalobj.`**`buffer_bytes`**`(`*buf*`)`

Given an object which implements the buffer protocol[257], this function returns the size of the object in bytes. The object can be multi-dimensional or include items larger than byte-size.

[255] https://docs.python.org/3.4/library/functions.html#repr
[256] https://docs.python.org/3.4/library/functions.html#print
[257] https://docs.python.org/3.4/c-api/buffer.html#bufferobjects

CHAPTER 17

Change log

Release 1.13 (2017-02-25)

1.13 includes numerous bug fixes and several major enhancements, mostly in the `mmalobj` (page 159) layer:

- 10 second captures should now work with the V2 module as the default `CAPTURE_TIMEOUT` has been increased to 60 seconds (#284[258])
- A bug in `copy_to()` (page 126) caused it to copy nothing when it encountered "unknown" timestamps in the stream (#302[259], #319[260], #357[261])
- A silly typo in code used by `PiRGBArray` (page 152) was fixed (#321[262])
- A bug in `capture_continuous()` (page 98) which caused duplicate frames in the output was fixed (#311[263])
- Bitrate limits were removed on MJPEG, and full checking of H264 bitrates and macroblocks/s was implemented (#315[264])
- A bug was fixed in the `sensor_mode` (page 118) attribute which prevented it from being set after construction (#324[265])
- A bug in the custom encoders example was fixed (#337[266])
- Fixed a rare race condition that occurred when multiple splitter ports were in use (#344[267])
- Recording overlays is now possible, but currently requires using the lower level `mmalobj` (page 159) layer (#196[268])
- Capturing YUV arrays via `PiYUVArray` (page 152) is faster, thanks to GitHub user goosst (#308[269])

[258] https://github.com/waveform80/picamera/issues/284
[259] https://github.com/waveform80/picamera/issues/302
[260] https://github.com/waveform80/picamera/issues/319
[261] https://github.com/waveform80/picamera/issues/357
[262] https://github.com/waveform80/picamera/issues/321
[263] https://github.com/waveform80/picamera/issues/311
[264] https://github.com/waveform80/picamera/issues/315
[265] https://github.com/waveform80/picamera/issues/324
[266] https://github.com/waveform80/picamera/issues/337
[267] https://github.com/waveform80/picamera/issues/344
[268] https://github.com/waveform80/picamera/issues/196
[269] https://github.com/waveform80/picamera/issues/308

- Added the ability to specify a restart interval for JPEG encoding (#369[270])
- Added a property allowing users to manually specify a `framerate_range` (page 113) for the camera (#374[271])
- Added support for partially transparent overlays in RGBA format (#199[272])
- Improved MJPEG web-streaming recipe, many thanks to GitHub user BigNerd95! (#375[273])

Substantial work has also gone into improving the documentation. In particular:

- The *Advanced Recipes* (page 25) chapter has been thoroughly re-worked and I would encourage anyone using the camera for Computer Vision purposes to re-read that chapter
- The *Camera Hardware* (page 65) chapter has been extended to include a thorough introduction to the low level operation of the camera module. This is important for understanding the limitations and peculiarities of the system
- Anyone interested in using a lower level API to control the camera (which includes capabilities like manipulating frames before they hit the video encoder) should read the *API - mmalobj* (page 159) chapter
- Finally, some work was done on enhancing the PDF and EPub versions of the documentation. These should now be much more useable in hard-copy and on e-readers

Release 1.12 (2016-07-03)

1.12 is almost entirely a bug fix release:

- Fixed issue with unencoded captures in Python 3 (#297[274])
- Fixed several Python 3 bytes/unicode issues that were related to #297[275] (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` (page 145) class

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[276])
- Add `request_key_frame()` (page 102) method to permit manual request of an I-frame during H264 recording; this is now used implicitly by `split_recording()` (page 102) (#257[277])
- Added `timestamp` (page 119) attribute to query camera's clock (#212[278])
- Added `framerate_delta` (page 112) to permit small adjustments to the camera's framerate to be performed "live" (#279[279])
- Added `clear()` (page 126) and `copy_to()` (page 126) methods to `PiCameraCircularIO` (page 125) (#216[280])

[270] https://github.com/waveform80/picamera/issues/369
[271] https://github.com/waveform80/picamera/issues/374
[272] https://github.com/waveform80/picamera/issues/199
[273] https://github.com/waveform80/picamera/issues/375
[274] https://github.com/waveform80/picamera/issues/297
[275] https://github.com/waveform80/picamera/issues/297
[276] https://github.com/waveform80/picamera/issues/241
[277] https://github.com/waveform80/picamera/issues/257
[278] https://github.com/waveform80/picamera/issues/212
[279] https://github.com/waveform80/picamera/pull/279
[280] https://github.com/waveform80/picamera/issues/216

- Prevent setting attributes on the main `PiCamera` (page 95) class to ease debugging in educational settings (#240[281])
- 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` (page 132) separately from the camera's `resolution` (page 117) (required for maximum resolution previews on the V2 module - #296[282]).

There are also several bug fixes:

- Fixed basic stereoscopic operation on compute module (#218[283])
- Fixed accessing framerate as a tuple (#228[284])
- Fixed hang when invalid file format is specified (#236[285])
- Fixed multiple bayer captures with `capture_sequence()` (page 100) and `capture_continuous()` (page 98) (#264[286])
- Fixed usage of "falsy" custom outputs with `motion_output` (#281[287])

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!

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[288])
- A new *intra_refresh* attribute is added to the `start_recording()` (page 103) method permitting control of the intra-frame refresh method (#193[289])
- 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[290])
- 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` (page 145) class was introduced for representation and manipulation of colors (#203[291])
- Reverse enumeration of frames in `PiCameraCircularIO` (page 125) is now supported efficiently (without having to convert frames to a list first) (#204[292])
- Finally, the API documentation has been re-worked as it was getting too large to comfortably load on all platforms (no ticket)

Release 1.9 (2015-01-01)

1.9 consists mostly of bug fixes with a couple of minor new features:

[281] https://github.com/waveform80/picamera/issues/240
[282] https://github.com/waveform80/picamera/issues/296
[283] https://github.com/waveform80/picamera/issues/218
[284] https://github.com/waveform80/picamera/issues/228
[285] https://github.com/waveform80/picamera/issues/236
[286] https://github.com/waveform80/picamera/issues/264
[287] https://github.com/waveform80/picamera/issues/281
[288] https://github.com/waveform80/picamera/issues/184
[289] https://github.com/waveform80/picamera/issues/193
[290] https://github.com/waveform80/picamera/issues/198
[291] https://github.com/waveform80/picamera/issues/203
[292] https://github.com/waveform80/picamera/issues/204

- The camera's sensor mode can now be forced to a particular setting upon camera initialization with the new `sensor_mode` parameter to *PiCamera* (page 95) (#165[293])
- The camera's initial framerate and resolution can also be specified as keyword arguments to the *PiCamera* (page 95) initializer. This is primarily intended to reduce initialization time (#180[294])
- Added the *still_stats* (page 119) attribute which controls whether an extra statistics pass is made when capturing images from the still port (#166[295])
- Fixed the *led* (page 116) attribute so it should now work on the Raspberry Pi model B+ (#170[296])
- 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[297]) * Fixed a long standing issue with MJPEG recording which caused camera lockups when resolutions greater than VGA were used (#47[298] and #179[299])
- Fixed a bug with incorrect frame metadata in *PiCameraCircularIO* (page 125). 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* (page 121) attribute (#177[300])
- Fixed a bug which caused *PiCameraCircularIO* (page 125) to ignore the splitter port it was recording against (#176[301])
- Several documentation issues got fixed too (#167[302], #168[303], #171[304], #172[305], #182[306])

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!

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[307])
- 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[308])
- Text annotation functionality has been extended; up to 255 characters are now possible, and the new *annotate_frame_num* (page 106) attribute adds rendering of the current frame number. In addition, the new *annotate_background* (page 105) flag permits a dark background to be rendered behind all annotations for contrast (#160[309])
- Arbitrary image overlays can now be drawn on the preview using the new *add_overlay()* (page 96) 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* (page 117) attribute replaces them (#144[310])

[293] https://github.com/waveform80/picamera/issues/165
[294] https://github.com/waveform80/picamera/issues/180
[295] https://github.com/waveform80/picamera/issues/166
[296] https://github.com/waveform80/picamera/issues/170
[297] https://github.com/waveform80/picamera/issues/174
[298] https://github.com/waveform80/picamera/issues/47
[299] https://github.com/waveform80/picamera/pull/179
[300] https://github.com/waveform80/picamera/issues/177
[301] https://github.com/waveform80/picamera/pull/176
[302] https://github.com/waveform80/picamera/issues/167
[303] https://github.com/waveform80/picamera/issues/168
[304] https://github.com/waveform80/picamera/issues/171
[305] https://github.com/waveform80/picamera/pull/172
[306] https://github.com/waveform80/picamera/issues/182
[307] https://github.com/waveform80/picamera/issues/149
[308] https://github.com/waveform80/picamera/issues/153
[309] https://github.com/waveform80/picamera/issues/160
[310] https://github.com/waveform80/picamera/issues/144

- Image effect parameters can now be controlled via the new `image_effect_params` (page 114) attribute (#143[311])
- 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[312])
- 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[313])
- Finally, the confusingly named `crop` (page 108) attribute was changed to a deprecated alias for the new `zoom` (page 120) attribute (#146[314])

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[315])
- Support for more than one camera on the compute module has been added, but hasn't been tested yet (#84[316])
- The `exposure_mode` (page 110) `'off'` has been added to allow locking down the exposure time, along with some new recipes demonstrating this capability (#116[317])
- The valid values for various attributes including `awb_mode` (page 107), `meter_mode` (page 116), and `exposure_mode` (page 110) are now automatically included in the documentation (#130[318])
- Support for unencoded formats (YUV, RGB, etc.) has been added to the `start_recording()` (page 103) method (#132[319])
- A couple of analysis classes have been added to `picamera.array` (page 151) to support the new unencoded recording formats (#139[320])
- Several issues in the `PiBayerArray` class were fixed; this should now work correctly with Python 3, and the `demosaic()` method should operate correctly (#133[321], #134[322])
- A major issue with multi-resolution recordings which caused all recordings to stop prematurely was fixed (#136[323])
- Finally, an issue with the example in the documentation for custom encoders was fixed (#128[324])

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!

Release 1.6 (2014-07-21)

1.6 is half bug fixes, half new features:

[311] https://github.com/waveform80/picamera/issues/143
[312] https://github.com/waveform80/picamera/issues/135
[313] https://github.com/waveform80/picamera/issues/154
[314] https://github.com/waveform80/picamera/issues/146
[315] https://github.com/waveform80/picamera/issues/16
[316] https://github.com/waveform80/picamera/issues/84
[317] https://github.com/waveform80/picamera/issues/116
[318] https://github.com/waveform80/picamera/issues/130
[319] https://github.com/waveform80/picamera/issues/132
[320] https://github.com/waveform80/picamera/issues/139
[321] https://github.com/waveform80/picamera/issues/133
[322] https://github.com/waveform80/picamera/issues/134
[323] https://github.com/waveform80/picamera/issues/136
[324] https://github.com/waveform80/picamera/issues/128

- The `awb_gains` (page 106) attribute is no longer write-only; you can now read it to determine the red/blue balance that the camera is using (#98[325])
- The new read-only `exposure_speed` (page 110) 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` (page 119) (#98[326])
- The new read-only `analog_gain` (page 105) and `digital_gain` (page 108) 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[327])
- The new `drc_strength` (page 108) attribute can be used to query and set the amount of dynamic range compression the camera will apply to its output (#110[328])
- The *intra_period* parameter for `start_recording()` (page 103) can now be set to *0* (which means "produce one initial I-frame, then just P-frames") (#117[329])
- The *burst* parameter was added to the various `capture()` (page 97) methods; users are strongly advised to read the cautions in the docs before relying on this parameter (#115[330])
- 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` (page 125) (#109[331])
- Similarly, the `array extensions` (page 151) 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[332])
- A bug that caused picamera to fail when the display was disabled has been squashed (#120[333])

As always, many thanks to the community for another great set of bug reports!

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` (page 151) module provides a series of custom output classes which can be used to easily obtain numpy arrays from a variety of sources (#107[334])
- The *motion_output* parameter was added to `start_recording()` (page 103) 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[335])
- 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[336])
- An example recipe was added to the documentation covering loading and conversion of raw Bayer data (#95[337])

[325] https://github.com/waveform80/picamera/issues/98
[326] https://github.com/waveform80/picamera/issues/98
[327] https://github.com/waveform80/picamera/issues/98
[328] https://github.com/waveform80/picamera/issues/110
[329] https://github.com/waveform80/picamera/issues/117
[330] https://github.com/waveform80/picamera/issues/115
[331] https://github.com/waveform80/picamera/issues/109
[332] https://github.com/waveform80/picamera/issues/121
[333] https://github.com/waveform80/picamera/issues/120
[334] https://github.com/waveform80/picamera/issues/107
[335] https://github.com/waveform80/picamera/issues/94
[336] https://github.com/waveform80/picamera/issues/97
[337] https://github.com/waveform80/picamera/issues/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[338])
- An issue with out-of-order calls to `stop_recording()` (page 104) when multiple recordings were active was resolved (#105[339])
- 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[340])

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()` (page 103) to permit inclusion of "Supplemental Enhancement Information" in the output stream (#77[341])
- The `awb_gains` (page 106) attribute was added to permit manual control of the auto-white-balance red/blue gains (#74[342])
- A bug which cause `split_recording()` (page 102) to fail when low framerates were configured was fixed (#87[343])
- 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[344])
- A bug which caused the `frame` (page 111) attribute to fail when queried in Python 3 was fixed (#80[345])
- A bug which caused raw captures with "odd" resolutions (like 100x100) to fail was fixed (#83[346])

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[347])

Many thanks to the community for yet more excellent quality bug reports!

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[348])
- The `record_sequence()` (page 101) method was added to provide a cleaner interface for recording multiple consecutive video clips (#53[349])
- The *splitter_port* parameter was added to all capture methods and `start_recording()` (page 103) to permit recording multiple simultaneous video streams (presumably with different options, primarily *resize*) (#56[350])

[338] https://github.com/waveform80/picamera/issues/103
[339] https://github.com/waveform80/picamera/issues/105
[340] https://github.com/waveform80/picamera/issues/89
[341] https://github.com/waveform80/picamera/issues/77
[342] https://github.com/waveform80/picamera/issues/74
[343] https://github.com/waveform80/picamera/issues/87
[344] https://github.com/waveform80/picamera/issues/85
[345] https://github.com/waveform80/picamera/issues/80
[346] https://github.com/waveform80/picamera/issues/83
[347] https://github.com/waveform80/picamera/issues/73
[348] https://github.com/waveform80/picamera/issues/52
[349] https://github.com/waveform80/picamera/issues/53
[350] https://github.com/waveform80/picamera/issues/56

- The limits on the *framerate* (page 111) attribute were increased after firmware #656 introduced numerous new camera modes including 90fps recording (at lower resolutions) (#65[351])

And partly bug fixes:

- It was reported that Exif metadata (including thumbnails) wasn't fully recorded in JPEG output (#59[352])
- Raw captures with *capture_continuous()* (page 98) and *capture_sequence()* (page 100) were broken (#55[353])

Release 1.2 (2014-02-02)

1.2 was mostly a bug fix release:

- A bug introduced in 1.1 caused *split_recording()* (page 102) to fail if it was preceded by a video-port-based image capture (#49[354])
- 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[355])
- 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[356])

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[357])
- An assertion error raised when performing raw captures with an active resize parameter was fixed (#46[358])
- A couple of documentation enhancements made it in (#41[359] and #47[360])

Release 1.0 (2014-01-11)

In 1.0 the major features added were:

- Debian packaging! (#12[361])
- The new *frame* (page 111) attribute permits querying information about the frame last written to the output stream (number, timestamp, size, keyframe, etc.) (#34[362], #36[363])
- All capture methods (*capture()* (page 97) et al), and the *start_recording()* (page 103) method now accept a `resize` parameter which invokes a resizer prior to the encoding step (#21[364])
- A new *PiCameraCircularIO* (page 125) 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[365])

[351] https://github.com/waveform80/picamera/issues/65
[352] https://github.com/waveform80/picamera/issues/59
[353] https://github.com/waveform80/picamera/issues/55
[354] https://github.com/waveform80/picamera/issues/49
[355] https://github.com/waveform80/picamera/issues/23
[356] https://github.com/waveform80/picamera/issues/48
[357] https://github.com/waveform80/picamera/issues/40
[358] https://github.com/waveform80/picamera/issues/46
[359] https://github.com/waveform80/picamera/pull/41
[360] https://github.com/waveform80/picamera/issues/47
[361] https://github.com/waveform80/picamera/issues/12
[362] https://github.com/waveform80/picamera/issues/34
[363] https://github.com/waveform80/picamera/issues/36
[364] https://github.com/waveform80/picamera/issues/21
[365] https://github.com/waveform80/picamera/issues/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* (page 117) attribute is now deprecated (#32[366])

Some bugs were also fixed:

- GPIO.cleanup is no longer called on *close()* (page 101) (#35[367]), and GPIO set up is only done on first use of the *led* (page 116) 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[368])

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

- The continuous method was removed; this was replaced by *capture_continuous()* (page 98) in 0.5 (#7[369])

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()* (page 102) method, and requires that the *start_recording()* (page 103) 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)

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()* (page 103) method
- Fixed bugs in the *crop* (page 108) 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

[366] https://github.com/waveform80/picamera/issues/32
[367] https://github.com/waveform80/picamera/issues/35
[368] https://github.com/waveform80/picamera/issues/32
[369] https://github.com/waveform80/picamera/issues/7

Release 0.6 (2013-10-30)

In 0.6, the major features added were:

- New `'raw'` format added to all capture methods (`capture()` (page 97), `capture_continuous()` (page 98), and `capture_sequence()` (page 100)) to permit capturing of raw sensor data
- New `raw_format` (page 117) attribute to permit control of raw format (defaults to `'yuv'`, only other setting currently is `'rgb'`)
- New `shutter_speed` (page 119) 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

Release 0.5 (2013-10-21)

In 0.5, the major features added were:

- New `capture_sequence()` (page 100) method
- `continuous()` method renamed to `capture_continuous()` (page 98). Old method name retained for compatiblity until 1.0.
- *use_video_port* option for `capture_sequence()` (page 100) and `capture_continuous()` (page 98) to allow rapid capture of JPEGs via video port
- New `framerate` (page 111) attribute to control video and rapid-image capture frame rates
- Default value for `ISO` (page 105) changed from 400 to 0 (auto) which fixes `exposure_mode` (page 110) not working by default
- *intraperiod* and *profile* options for `start_recording()` (page 103)

In addition a few bugs were fixed:

- Byte strings not being accepted by `continuous()`
- Erroneous docs for `ISO` (page 105)

Many thanks to the community for the bug reports!

Release 0.4 (2013-10-11)

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

- `preview_alpha` (page 117)
- `preview_fullscreen` (page 117)
- `preview_window` (page 117)

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

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.

Release 0.2

The major change in 0.2 was support for video recording, along with the new *resolution* (page 117) property which replaced the separate `preview_resolution` and `stills_resolution` properties.

CHAPTER 18

License

---

[370] dave@waveform.org.uk

[371] https://en.wikipedia.org/wiki/File:Bayer_pattern_on_sensor.svg

The *YUV420 planar diagram* (page 26) in the documentation is Yuv420.svg[372] created by Geoff Richards (User:Qef) on Wikipedia, released into the public domain.

[372] https://en.wikipedia.org/wiki/File:Yuv420.svg

Python Module Index

Index